

Algebra of Systems: A Metalanguage for Model Synthesis and Evaluation

Benjamin H. Y. Koo, Willard L. Simmons, *Member, IEEE*, and Edward F. Crawley

Abstract—This paper represents system models as algebraic entities and formulates model transformation activities as algebraic operations. We call this modeling framework “Algebra of Systems” (AoS). To show that AoS can automate complex model reasoning tasks in system design projects, we implemented the abstract algebraic specification as an executable metalanguage named Object-Process Network, which serves as a tool for automatic model transformation, enumeration, and evaluation. A case study of the Apollo lunar landing mission design is developed using this algebraic modeling approach.

Index Terms—Algebraic reasoning, metalanguage, metamodeling, systems architecture, systems engineering.

I. INTRODUCTION

WHEN reasoning about a complex system, it is often necessary to decompose the system into smaller units of abstraction and then represent interactions among the units. The objective of this paper is to present a mathematical framework called Algebra of Systems (AoS) that provides a formal structure to reason about systems of elements and interactions in terms of algebraic operands and operators. To automate the algebraic manipulation of system models, we also implement an executable modeling language, called Object-Process Network (OPN), that follows the formalisms defined in AoS.

A. Background

An algebra is a mathematical structure made up of two kinds of things: an operand set and an operator set. The operators for an algebra must also be closed under the operand set, meaning that the application of operators to any element in the operand

set always results in elements in the same set. An algebra A can be written as a tuple

$$A = \langle \{\text{Operands}\}, \{\text{Operators}\} \rangle .$$

where A is an algebra if and only if arbitrary compositions of $\{\text{Operators}\}$ on elements in $\{\text{Operands}\}$ always result in elements that are also in $\{\text{Operands}\}$. This is called the *closure* property of an algebra.

Algebras are of interest to engineers because they provide a representationally efficient language to reason about systems with desirable level of information resolution. *Operators* in an algebra provide the means to rewrite algebraic expressions based on formal properties such as equality, associativity, or commutativity. Creating operators that operate in the system model domain allows us to transform, simplify, and reveal hidden qualities of the system model and therefore improve our understanding of the system of interest. Similarly, *Operands* in an algebra provide the means to encode the information content of algebraic expressions using abbreviated variable names or mathematically defined constant values that decompose the system into manageable subsets of qualitative and quantitative properties.

Together, these subsets of qualitative and quantitative properties form a language that can illustrate both functions and forms of systems. It also provides a consistent way to modularize the complexity of a system model into discrete elements. For example, segments of complex algebraic expressions can be simplified using a new and unique variable name without distorting the original meaning. For example, in an algebra of real numbers, several operators can be simplified into one operator, such as the operator multiplication (\times), which is a way to concatenate multiple operations of addition ($+$). In other words, the operands and operators of an algebra provide a robust instrument to describe systems in distinctive modules that help manage the varying scopes and levels of detail of system representation.

A less known feature of algebra is its procedural nature. The definition of operators naturally provides the procedural mechanism to interpret or transform algebraic expressions that can be realized as computer programs. In many cases, the mathematical definition of an algebra inherently provides a significant portion of the specification content of executable modeling languages.

B. Literature Review

The power of algebraic reasoning in engineering design has been articulated by Hoare *et al.* [2]. In the world of information

Manuscript received January 24, 2007; revised August 24, 2008. Current version published April 17, 2009. This work was supported in part by Charles Stark Draper Laboratory, by the National Aeronautics and Space Administration, and by the Natural Science Foundation of China under Project 70771059. An earlier version of this paper was presented at the 2007 International Conference on Systems Engineering and Modeling [1]. This paper was recommended by Associate Editor L. Fang.

B. H. Y. Koo is with the Department of Industrial Engineering, Tsinghua University, Beijing 100084, China (e-mail: benkoo@tsinghua.edu.cn).

W. L. Simmons was with the Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA 02139 USA. He is now with DataXu, Inc., Cambridge, MA 02142 USA (e-mail: wsimmons@alum.mit.edu).

E. F. Crawley is with the Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: crawley@mit.edu).

Color versions of one or more of the figures in this paper are available online at <http://ieeexplore.ieee.org>.

Digital Object Identifier 10.1109/TSMCA.2009.2014546

systems, the invention and application of relational algebra have significantly influenced the way data-intensive systems are designed and implemented [3], [4]. In the world of physical devices, certain safety-critical digitally controlled real-time systems are made feasible with the support of algebraic modeling techniques [5], [6]. To reason about the design of systems in algebraic terms, Cousot and Cousot demonstrated a system analysis technique called abstract interpretation [7] which uses pairs of fixed-point operators such as “narrowing/widening” defined on a domain of lattices.¹ Similar to relational algebra, whose operators are applied to the domain of relations, abstract interpretation uses lattices as mathematical structures that approximate engineering design models, and uses the aforementioned operators to perform computations on the domain of lattices. The resulting lattices can be translated back to a representation of more refined engineering design models.

In [8], Suh illustrated that designs can be expressed in terms of axiomatic requirement statements. He also proposed that these statements and the associated design parameters could be formulated into a matrix-based computational framework to assess the likelihood of satisfying the requirements. The likelihood metric is calculated as an entropy-like objective function and is used as a design quality filter that helps designers select design alternatives under this computational framework. A limitation of this approach is that assessing the overall likelihood of design success may not be easily formulated into the prescribed matrix format. Moreover, computing the numeric value of likelihood could be a major technical challenge.

Baldwin and Clark [9] suggested that certain socio-technical properties of product development can be represented using a small set of transformation operators on designed structures. In contrast to the common perception that engineering resources are often expensed on solving domain-specific (and often isolated) specific design problems, Baldwin and Clark argued for the strategic advantages of exploring the design spaces as a unified whole, which is composed of many subdesigns, tasks, and contracts. They point out that in a dynamic marketplace, it is inevitable that the artifacts under design must also evolve with the inevitable changes in their environment. Therefore, it is advantageous for design teams to systematically introduce the concept of “real options” [10] into a product’s architecture to allow some flexibility. A challenge in this approach is that one must provide scalable and flexible representational languages as well as the “valuation technologies” [9] to assess the abstract design space. Many modeling and valuation techniques have been developed since then. For instance, the work in aspect-oriented programming [11] and dynamic analysis [12] provides methods that enable technologies to represent and evaluate the properties of an abstract and evolving design space. In 1993, Carlson [13] provided an algebraic formulation of design spaces and presented a programming language, *Grammatica*, as a symbolic representation of design spaces. Due to the limited computational power at that time, *Grammatica* has only been applied to small-scaled geometrical configuration problems. However, it demonstrated that an algebraic formulation of design spaces can be supported in the form of a programming language.

¹Lattice is a kind of algebra.

Another related design technique is called “Configuration Design.” In [14], Wielinga and Schreiber stated that certain design problems could be stated as an assembly of selected components that satisfies a set of requirements and constraints and that the ranking order of alternative designs could be established using an optimality criterion. Many configuration design problems can be treated as some kind of constraint-based programming problem [15], solved using rule-based languages [16]. However, one could imagine that when the number of components becomes very large, the complexity of model management can be a major bottleneck in practicing Configuration Design.

The amount of information that must be managed in a modern product development environment is also making design activities more challenging. Today, designers not only need to formulate and assess design spaces based on one’s own knowledge about the design issue but also they need to capture a significant amount of scattered piece-meal knowledge of many different designers in a design committee. It is also expected that designers must consider many remotely relevant regulatory rules. Distributed and context-aware computing systems can also use certain algebraic approaches to automatically merge and partition certain context-sensitive design constraints [17].

These prior arts led us to work on an algebraic language to describe and evaluate the design space of a wide range of engineering systems. Simple and precise syntax is necessary, so that we may apply formal techniques to analyze the representational correctness and efficiency of the system models written in this language. Moreover, the language should also possess the computational qualities that we may interpret or compile the system model to conduct simulation or performance metric evaluation. Moreover, we show that this algebraic formulation allows designers to symbolically track the design tasks that involve human interventions as a set of model manipulation instructions. When appropriate, the algebraic formulation also enables certain model construction and reasoning tasks to be computationally automated.

C. Motivation

Ideally, system designers need an expressive information platform, which can support a broad range of design activities, and also follow a concise set of design principles, so that designers can have some intuitive sense of how to proceed with the design exploration activities. We argue that these kinds of conceptual guides could come from a small set of algebraic rules, realized in an executable programming language. With such a language, designers could manage the logical relationships of their knowledge by using computers to help them reason about the compositional structures of design alternatives. To help people see and seek the values in the design landscape, we need a unifying algebraic framework to guide the implementation of a general-purpose computable design language.

D. Synopsis

In this paper, we first present an overview of the development of the AoS, followed by a description of the supporting tool called OPN Integrated Development Environment (OPN-IDE).

TABLE I
 AoS ALGEBRAIC DOMAINS

Domain	Operands	Operators
Algebra of Systems Domain, AoS	$\langle P, B, C \rangle$	{ <i>enc, enum, eval</i> }
Properties Domain, P	{ $\langle key, value \rangle^*$ }	{ <i>merge, substitute, interp, delete</i> }
Boolean Domain, B	{ $TRUE, FALSE$ }	{ <i>and, or, negate, interp</i> }
Composition Domain, C	$\langle \{Obj\}, \{Proc\}, \{pre\}, \{post\} \rangle$	{ <i>union, subtract</i> }

To demonstrate the utility of this approach, we present a concrete example derived from the mission-mode selection studies for the Apollo Moon mission. These sections are followed by a discussion and conclusions.

II. SYSTEM ABSTRACTIONS

When we describe a system, one often needs to encode two aspects of system properties: what the system *is* and what the system *does*. Therefore, it is common to characterize modeling languages into two generic classes of orientation: data abstraction and functional abstraction. Data abstraction modeling languages describe what the system “is” by encoding systems parametrically as a collection of values. Functional abstraction modeling languages describe what the system “does” by encoding a system’s transformative behavior.

However, for practical modeling of systems, it is often necessary to incorporate both data and functional abstractions in *one* language. The challenge lies in creating a language with the best mix of abstraction features. To avoid bias, our approach is to provide a metalanguage that supports a basic set of abstraction features so that users can customize a modeling language for their specific application needs. In other words, in this approach, system models are encoded as formal language specifications. By treating models as instances of customized languages, one may systematically apply language manipulation techniques, such as interpretation and compilation, to analyze the models.

The AoS is a *many-sorted algebra* [18]. It is an algebraic domain made up of subalgebraic domains. Table I gives the domain names and their respective operands and operators. Section II-A illustrates how AoS uses its primitive data abstractions to symbolically parameterize the structural and behavioral properties of a system of interest. The data and functional abstractions are captured in domains, P , B , and C . Next, Section II-B defines the AoS domain and shows how it uses a set of algebraic model transformation operators to approximate the model manipulation processes for system design activities.

A. Algebraic Domains for Data and Functional Abstraction

Design knowledge about the design space might be stored in many formats. One role of formal languages is to provide appropriate data structures to encode different kinds of knowledge. AoS uses a composite data structure, a triple $\langle P, B, C \rangle$, as its operand domain. In this triple, P denotes the quantitative and qualitative properties, B denotes the Boolean value status, and C denotes the compositional structure of a system. These data types are also operands for their respective subalgebras. They are explained hereinafter.

Domain Definition 1 (P : Properties Domain): When encoding a set of properties for a certain system in a formal data structure, we organize them as a list of uniquely labeled values. It is also useful to think of P as a domain of 2-tuples. It is a set of tuples with two elements, where the data content of the keys is nonrepetitive. In other words, $P = \{\langle key, value \rangle^*\}$. The properties domain P is similar to an equation-like language. When expressed in a textual format, it can be written as a collection of key-value pairs in the format $\{key1 = value1; key2 = value2; \dots\}$.

When we treat P as a textual language, its formal syntax can be specified in the Extended Backus Naur Form.

```

<P> ::= { } | {<PROP>*}
<PROP> ::= <VAR> = <EXPR>;
<EXPR> ::= <CNST> | <VAR> | <EXPR> <OPER> <EXPR> | <FUNC>
<VAR> ::= <LETTER> [<DIGIT>|<LETTER>]*
<OPER> ::= <|>|<=|>|=|<and|or|+|-|"*|/|...
<FUNC> ::= <VAR>" (" LIST ") "
<LIST> ::= { (<EXPR> ", ")*<EXPR>
<CNST> ::= <VAR> | <NUMBER>
<NUMBER> ::= <DIGIT>*|. <DIGIT>*
<LETTER> ::= [a~z] | [A~Z]
<DIGIT> ::= 0~9
    
```

By design, all the operators in P are closed under P . This implies that the “language” P can be considered an algebra. Four closure operators of interest defined in this paper are as follows: *merge*, *substitute*, *delete*, and *interp*. One might notice that this syntactical specification allows one to define n -ary functions or multiargument functions. The definitions of these “functions” should be supplied by different users.² To illustrate these operators, a set of examples is given hereinafter.

Given a data point $p \in P : p = \{x = x + 1; y = 2x; \}$

$$merge(p, \{z = 1\}) = \{x = x + 1; y = 2x; z = 1\}$$

$$substitute(p, \{x = 1\}) = \{x = 2; y = 2\}$$

$$delete(p, \{y = 2x\}) = \{x = 1\}$$

$$interp(p) = \{x = x + 1; y = 2x + 2\}.$$

The first three operators take two arguments and generate a new operand as a result. The *merge* operator is like a union operator of two sets of key-value pairs. Unlike the *union* operator for set theory, *merge* is not commutative, since the second argument always overwrites the data content in the first argument. The definitions of these operators are shown as follows.

Operator Definition 1 ($merge(\cdot, \cdot)$): *merge* is a binary operator that uses the information content in the second operand

²Our current implementation allows users to define these n -ary functions using popular scripting languages such as Python or Jython (for more details, see [19]).

to overwrite the information content in the first operand. For example

$$\begin{aligned} \text{merge}(\{x = 3\}, \{y = 3\}) &= \{x = 1, y = 3\} \\ \text{merge}(\{y = 1\}, \{y = 3\}) &= \{y = 3\}. \end{aligned}$$

Operator Definition 2 ($\text{substitute}(\cdot, \cdot)$): The *substitute* operator is used to replace variables with some specific values or expressions. For instance

$$\begin{aligned} \text{substitute}(\{x = x + 1; z = x\}, \{x = y\}) \\ = \{x = y + 1; z = y\}. \end{aligned}$$

This operator takes “knowledge” encoded in the second argument to rewrite the expressions in the first argument. Note that the substitution only applies to the right-hand-side expression, not the variable names on the left-hand side.

Operator Definition 3 ($\text{delete}(\cdot, \cdot)$): The *delete* operator is designed to help reduce the information content of a data point in P . For instance

$$\text{delete}(\{x = x + 1; z = x\}, \{x = x + 1\}) = \{z = x\}.$$

This helps to remove some information content when it is deemed to be irrelevant. *delete* helps reduce memory and processing overhead when AoS is implemented on a computer.

Operator Definition 4 ($\text{interp}(\cdot)$): The *interp* operator is an arithmetic expression simplifier. For instance

$$\begin{aligned} \text{interp}(\{x = z + 3y; y = 2; z = y\}) \\ = \{x = 8; y = 2; z = 2\}. \end{aligned}$$

This operator provides the interpretive power to convert an arithmetic expression into a simpler expression or a numeric value. By converting certain arithmetic expressions into numeric values, the *interp* operator can be thought of as a calculation engine that translates arithmetic expressions into numerical quantities. There are certain cases where a set of key-value pairs might involve circular references. This is handled by always stopping the expression rewrite process when a key-value pair presents the fixed-point format ($x = f(x)$). For instance, whenever $x = 3x$, the expression rewrite procedure would stop, since this key-value pair can be interpreted as a fixed-point equation.

These four operators, *merge*, *substitute*, *delete*, and *interp*, are all closed under the domain P . When applied to certain data point in P sequentially, one may use the compositions of these four operators to define other useful operations. For example, one may construct a new binary operator $\text{combine}(p, q) = \text{interp}(\text{merge}(\text{substitute}(p, q), q))$. The ability to construct new operators based on a small set of primitive operators is an important feature of P .

Domain Definition 2 (B : Boolean Domain): Whether or not an AoS model satisfies certain constraints is encoded in a Boolean expression such as “*prob* > 0.5 and *cost* < 300.” In this example, the values of the variables *prob* and *cost* are supplied by the information content in the domain P . The operators in this domain are Boolean operators such as *and*, *or*, *negate*, and *interp*. The first three operators’ functions are well-known Boolean operators, and the *interp* operator is



Fig. 1. Composition domain C depicted as a bipartite graph.

simply a function that translates an expression into a Boolean value or a simpler Boolean expression. The *interp* operator in the Boolean domain is equivalent to the *interp* operator in the Properties domain. An example of *interp* in the Boolean domain is

$$\text{interp}((x - x > 1) \text{ or } (z < 3)) = z < 3.$$

Domain Definition 3 (C : Composition Domain): The composition domain C is the third element in AoS’s triple. It is a bipartite graph data structure. An element in C encodes a system as a collection of smaller building blocks and a set of relationships between them. The bipartite graph structure enforces modelers to distinguish between two kinds of building blocks of a system that represent data and functional units of abstraction. The data and functional units are named *Objects* and *Processes*. When a point $p = \{x = 2; y = 3\}$ in P is associated with an *Object*, it is interpreted as a set of parameters that describes the *Object*. When p is associated with a *Process*, it is interpreted as a set of functional operations that assigns a defined value to variables named x and y . To encode the structure of interactions between multiple building blocks, the relationships between *Objects* and *Processes* can be graphically represented as directed arcs between them. Each directed arc is associated with a Boolean expression, and its applicability is constrained by the result of the Boolean expression.

The bipartite formalism forbids arcs to directly connect two *Objects* or two *Processes*. This bipartite formalism ensures that all system abstractions can always be mechanically transformed among one of three possible cases: pure data abstraction, pure functional abstraction, or a composition of the two. The bipartite formalism also makes it easier to utilize computational models that are also based on bipartite graphs such as Petri net [20], Bayesian Belief Network, and System Dynamics. By using graph manipulation operators, such as *union* and *subtract*, the structure and information content of graphs can be combined and divided. Examples of these bipartite graphs are shown in Figs. 1, 2, and 4.

B. Algebraic Domain for Design Process Abstraction

Domain Definition 4 (AoS : AoS Domain): The three domains, in AoS’s triple $\langle P, B, C \rangle$, are three kinds of algebraic structures. As a triple, they are considered a data point in the AoS operand domain. The AoS domain is a composite algebraic structure, also known as a many-sorted algebra. One may use the operators in the individual data domains to compose a set of macrolevel operators that directly apply to instances of AoS triples. We now propose three operators $\{\text{encl}, \text{enum}, \text{eval}\}$ that capture the essence of our model refinement framework as a many-sorted algebra. AoS can be formally written as

$$\text{AoS} = \langle \{P, B, C\}, \{\text{encl}, \text{enum}, \text{eval}\} \rangle.$$

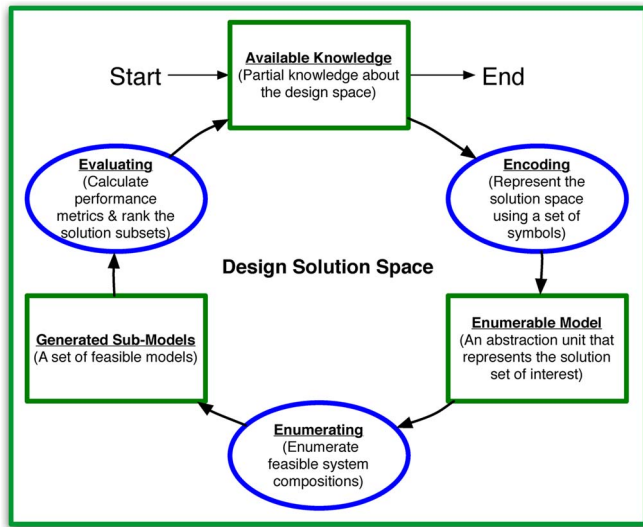


Fig. 2. Methodology overview.

The operators *encl*, *enum*, and *eval* are described in detail hereinafter. Table I lists the set of operators and the signature of the operands for the three domains *P*, *B*, and *C* as well as the set of macrooperators and signature of the macrooperands for the macrodomain *AoS*.

Similar to a source-code compilation process, knowledge about system composition can be processed in several stages. The overall process is to transform a set of abstract modeling concepts into a set of concrete system design models. This process can be thought of as a knowledge compilation process, where the inputs and outputs of this process are sets of system models. As shown in Fig. 2, we first assume that the *Design Solution Space* (everything inside the outer box) is a universe of knowledge elements that might be encoded in the *AoS* language. Fig. 2 shows an iterative knowledge compilation process that converts *Available Knowledge* elements and *Enumerable Model* elements into *Generated Submodels* and back into *Available Knowledge*. The design cycle ends when the *Available Knowledge* domain contains sufficient information to implement a design.

As shown in Fig. 2, the process of system model transformation can be decomposed into three subprocesses. In this figure, the rectangles represent *Objects* that store sets of transformed models, and the ellipses represent model transformation *Processes*. Each subprocess produces a particular kind of system model that can be iteratively refined during the overall design process. All three subprocesses represent algorithms or data manipulation operators that can be conceptualized as three individual algebraic operators. Their functional roles are described hereinafter.

Operator Definition 5 (Encoding: (*encl*)): The *Encoding* process is a system modeling process that often involves a combination of human intervention and machine verification. The process takes *Available Knowledge* into the form of *Enumerable Model*. Enumerable Models are constructed using appropriate combinations of the operators for *P*, *B*, and *C*. These operators may be applied to the individual domains in the *AoS* triple, and the end result is always an *AoS* triple with

different content. By using the operators, existing submodels can be combined with new human-edited new model content. We argue that any type of information content in *Available Knowledge* can always be formally expressed in the *AoS* domain by manipulating the content in their properties, Boolean, or compositional domains.

At the start of the design process, when no existing models are available, the *Encoding* process is like a bootstrapping activity that takes amorphous knowledge content into the three value domains of *AoS*. This creates an initial *AoS* model that brings the design process into the algebraic *AoS* domain. The essence of the *Encoding* process is to provide a symbolic representation for each element of explicitly available knowledge so that we can employ the algebraic operators. Section IV will provide additional treatment on this subject.

Operator Definition 6 (Enumerating: (*enum*)): Once the system abstraction knowledge is formulated as an *Enumerable Model*, it can be submitted to the *Enumerating* process. The enumeration algorithm can utilize the model’s information content to generate all subgraphs of the initial bipartite graph. These subgraphs represent elements in a set of *Generated Submodels*. All model enumeration algorithms can be decomposed into individual steps that utilize operators in the three data and functional abstraction domains *P*, *B*, and *C*. Counting the original model, the enumeration procedure must be able to generate one or more models that are distinctively different. Each generated model instance is a description of a subset (or partition) of the overall solution space. All the enumerated models are members in the object *Generated Submodels*. As new submodels are generated, it triggers the *Evaluating* process to perform model evaluation. By providing implicit constraints such as counting the loops in the graph only once, this model enumeration operation will only generate a finite number of submodels.

Operator Definition 7 (Evaluating: (*eval*)): Each of the elements in the *Generated Submodels* is a combinatorial variation of the original model. Consequently, each generated model will have a different subset of specification rules encoded in the domain *P* as a by-product of the model enumeration process. These rule sets can be executed or interpreted to infer additional knowledge about each of the *Generated Submodels*. Specifically, rules written in *P* may be algebraically manipulated to derive quantitative information about certain metrics of the models. In addition, these models may be sorted as part of the *Evaluating* process and organized into a partially ordered set (poset) of *Generated Submodels*. The ranked positions of these generated models can be used as the selection criteria for a final design solution.

The *Evaluating* process is a way to computationally derive new knowledge about system design from a collection of subsystem knowledge. Through computation, it changes the information content in the generated and evaluated models. It may also enrich the *Available Knowledge* about the solution space. The newly acquired knowledge may include the number of generated submodels, the calculated metric values, and the ranking order of the calculated metrics.

The three operators *encl*, *enum*, and *eval* computationally derive elements of knowledge. The elements of knowledge can

be recursively applied to these three operators to generate more knowledge. This recursive process ends on one of the three following cases: 1) when no new models can be generated by iteratively and exhaustively applying these operators; 2) when the set of generated models is “good enough” for practical use; and 3) when available resources or time runs out.

This concludes the presentation of the three model refinement operators that directly operate on the AoS domain. Conceptually speaking, the three operators provide a generic functional abstraction to categorize the dynamic nature of design activities.

III. COMPUTATIONAL TOOL FOR AoS

We implemented an executable programming language, OPN [21], based on the mathematical specification of AoS. The idea of representing objects and processes in one system description language is derived from Object-Process Methodology [22]. As an executable language, OPN is very similar to Petri net; the detailed comparison between Petri net and OPN can be found in [21]. The computer program that we created to edit and execute OPN models is called the OPN-IDE.

OPN-IDE as a modeling tool serves two purposes. First, it provides a human-machine interface to display and edit the information content of system models. Second, it also provides an execution engine that performs automated model composition and model transformation tasks. The main purpose of OPN is to streamline experimental modeling activities that are traditionally done manually.

A. Language in Multiple Views

Recalling in Section II, the abstract data specification of AoS includes three different domains of information content. The tool supports the editing and displaying of all three types of information content using different editable views. Each editable view visualizes the information content in the modeling language in a different form, such as graph, tree, and matrix. We created a bipartite graph editor to display and change the composition domain. The bipartite graph provides the compositional overview of the subsystems. Subsystems are categorized as processes and objects, which are represented by ellipses and rectangles, respectively. When users click on a “process,” a dialog box pops up and allows users to enter the transformation rules. When users click on an “object,” a dialog box would display the current information content captured in the “object.” When users click on the directed arcs that link objects and processes, a dialog box opens and allows users to edit the logical expressions. The screenshot in Fig. 3 shows the key user interface elements of the OPN-IDE.

B. Algebraic Properties of OPN

The essence of the AoS approach is to treat models as algebraic entities. The recursive application of operators on models must always yield models encoded in the same domain of models. To facilitate this closure property of model manipulation, OPN-IDE’s editable views are laid out to support this model production concept.

In Fig. 3, the diagrammatic view of the *metamodel* is shown in the upper-left corner. A metamodel is the original specification that describes a design solution space (see Fig. 2). The specification can be visualized in the *tree view* or the *matrix view* located in the lower-left corner. The middle section is a list of model names, which represents the set of *Generated Submodels* (or object models) of the metamodel. During the enumeration phase of model processing, new models appear in this selection list as they are generated. Users can choose one of the *Generated Submodels* by clicking on the list. Detailed information of the selected model is shown on the right-hand-side panel. The right-hand-side panel also contains the graph view, tree view, and matrix view. It displays the selected model using the same user interface elements as the generating model, because they all contain the same data structures. In our implementation, the enumeration process also performs the evaluation tasks, so that the generated models also contain the evaluation results. Users can inspect the evaluation results by inspecting the tree view or the associated customized views for different properties of the generated models.

The tool also allows users to save any one of the *Generated Submodels* as an independent model file. Then, the submodel can be edited and executed to generate additional submodels. The recursive nature of model processing is a natural by-product of an algebraic language. In programming language literature, it is like a graphical functional language, since the language interpreter produces outputs that are also interpretable graphical models. In our implementation, the layout of the original model is preserved, so that users can easily identify the locations of objects and processes via visual inspection.

C. Implementation Details for OPN

The specific implementation details for an earlier version of OPN are available in [21]. However, the implementation details concerning the implementation of this AoS framework in the OPN-IDE are the subject of a planned future publication.

IV. APPLICATION: APOLLO MISSION-MODE STUDY

The National Aeronautics and Space Administration’s (NASA) Apollo program in the 1960s to “land a man on the Moon and return him safely to Earth” was arguably the most ambitious engineering project that had ever been proposed. The transportation system to accomplish this task, including rockets and spacecraft, was the central aspect of the design. A review of the written history [23]–[26] and conversations with the Associate Administrator of NASA at the time, Dr. Robert C. Seamans [27], reveal that one of the most critical design decisions in the lunar program was to select the mission mode.³ This decision directly and indirectly influenced the design, task, and contract structures of the overall project. The main challenge was to perform a thorough and technically sound assessment of alternative designs. Even today, this combinatorially complex problem is still considered as a highly

³In aerospace engineering, a mission mode is defined as the number, types, destinations, and interactions of vehicles for a space mission [28].

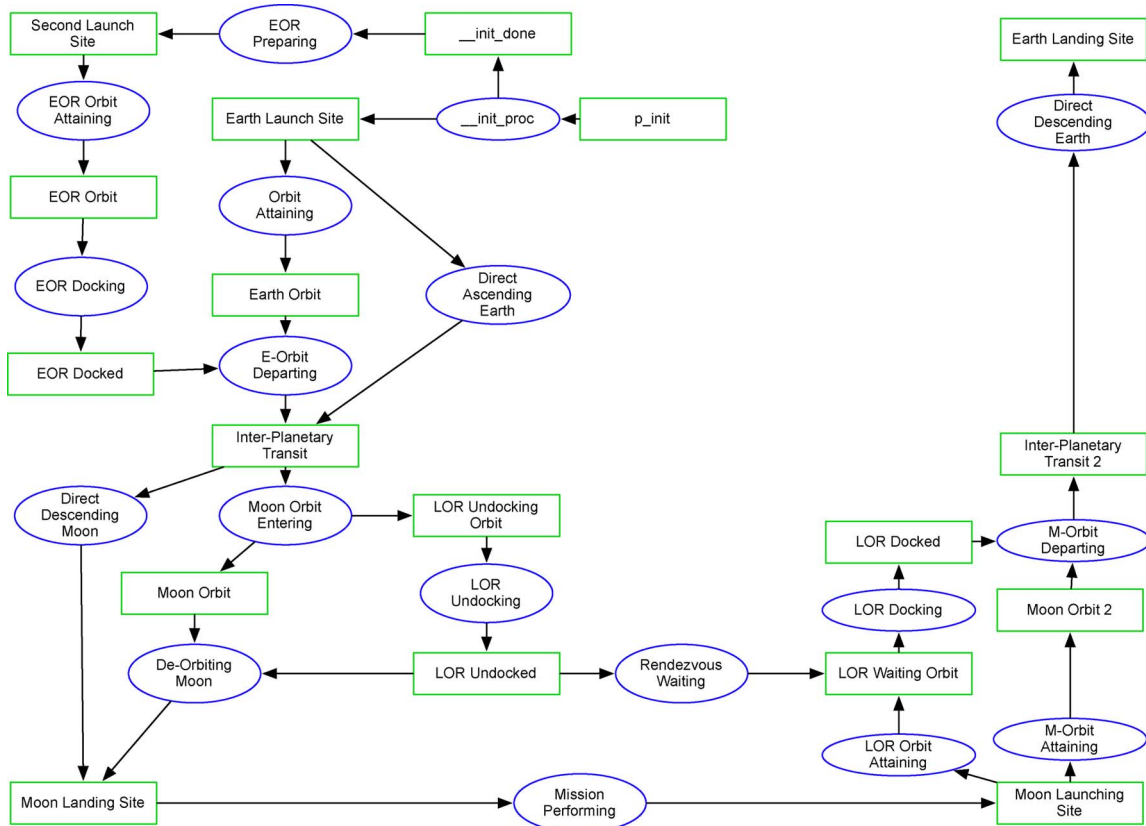


Fig. 4. Model representing mission modes.

useful. Without using algebraic rules to simplify the constraints during simulation, the sizes of state spaces can easily become intractable.

A. Encoding the Solution Space

The first step in the process shown in Fig. 2 is to “Encode” the abstract knowledge about the system design into the composition, properties, and Boolean domains of the AoS language.

Mission-Mode Composition Domain: The domain of compositions (domain C in Table I) is made up of “objects” and “processes” connected by “preconditions” and “postconditions.” For the mission-mode selection problem, objects are used to represent the different steady-state phases of the missions. Some examples of these phases are Earth launch site, Earth orbit, interplanetary transit, Moon orbit, and Moon landing site. These steady-state time phases can be characterized parametrically in terms of position or velocity and therefore can be treated as the units of data abstraction. Processes in the mission-mode study are finite-time transitions between steady-state phases, such as deorbiting, ascending from Earth, and waiting for rendezvous. The pre- and postconditions in this composition domain represent the rules that allow a process to follow an object. For example, descent to the surface can be accomplished either by directly descending, or by entering orbit, orbiting, then descending from orbit.

We categorize these mission-mode elements as units of functional abstractions because they consistently play the roles of transforming the objects from their previous data states to

one or more new states. Having divided the mission-mode elements into two separate categories, we may connect any two consecutive mission-mode elements using directed arcs representing the pre- and postconditions. Note that objects may only connect to processes and vice versa, since this abstraction scheme of mission modes⁴ enforces the bipartite graph formalism. The overall mission-mode possibility space can be therefore represented as a bipartite graph, as shown in Fig. 4. By using this modeling approach, a complete mission mode is a composition of many mission-mode elements.

Mission-Mode Properties Domain: The properties domain P of the model encodes the systems attributes or behaviors. Following common aerospace engineering practice, we recursively used the rocket equation to calculate vehicle masses, depending on the velocity increment they must supply, their payload, and their ordering in the mission mode. Mass is considered a good proxy for cost of development and operations of space hardware. The rocket equation provides a framework for deriving mass-based metrics [33]. The equation is given as follows:

$$m_f = m_i \exp\left(\frac{-\Delta V}{g_0 \cdot I_{sp}}\right)$$

where ΔV is the difference in velocity over entire period of the maneuver, g_0 is the gravitational constant, I_{sp} is the specific impulse of the propulsion system, m_f is the final mass after the

⁴Frazzoli’s work on Maneuver Automata helped inspire our abstraction scheme [32].

TABLE II
 PROBABILITY TABLE

Reference Risk		Higher than Reference Risk	
to Earth orbit	0.98	to Earth orbit (second launch)	0.95
ascend to lunar orbit	0.98	rendezvous in Earth orbit	0.95
		rendezvous in lunar orbit	0.95
		descend to lunar surface from orbit	0.95
		direct Earth arrival	0.95
Lower than Reference Risk		Much Higher than Reference Risk	
departure from Earth orbit	0.99	direct ascend from Earth	0.90
lunar orbit entry	0.99	direct descend to lunar surface	0.90
		ascend from lunar surface	0.90

maneuver, and m_i is the initial mass before the maneuver. The two mass terms m_f and m_i can be broken down as follows:

$$m_f = m_{bo} + m_{pl}$$

$$m_i = m_{bo} + m_{pl} + m_{prop}$$

where m_{bo} is the burnout (structure-only) mass, m_{pl} is the payload mass, and m_{prop} is the propellant (fuel) mass. For a multistage rocket system, the rocket equation can be applied recursively for each maneuver. If the “payload” of a stage is actually another rocket with its own fuel and payload, then m_{pl} becomes the initial mass for the next application of the equation.

In our study, values of constants such as the structural mass ratios, propulsion characteristics, and models for crew compartment sizes were taken from a combination of historic data and the assumptions used in the contemporary 1961 Houbolt Report [34]. These specific details and the governing rocket equation were entered in the P domain.

In addition to lift-off mass, the other major factor in selection of mission mode was operational risk. A simple model of the relative risk of various operations, shown in Table II, was derived based on expert opinion of lunar exploration vehicle designers. This table represents relative risk, not absolute risk. The risks of the various processes were encoded in the model. We present numeric values here for historical reasons and to help the reader visualize the language’s numerical ability to compute quantitative values. If the probability measures had not been known at the time of model execution, they could have been encoded as symbolic unknown variables. The interpreter for the properties domain P would have automatically constructed algebraic expressions and simplified them according to the known and unknown quantities associated with different mission modes.

Given these properties about individual mission-mode elements, we can assign these properties to transformation rules in different processes. For example, the two rules regarding the mission-mode risk and rocket mass can be written as two separate rules using domain P

$$\{ \text{prob} = \text{prob} * 0.95; \\ \text{massR} = \text{massR} * \text{staged}(dV, \text{Isp}, 3); \}$$

where massR is the ratio of final to initial mass, prob is the probability based on relative risk, and $\text{staged}()$ is a multi-argument function for calculating staged rocket performance. The recursive definition of massR should be interpreted as a

previous massR value multiplied by the function $\text{staged}()$. This statement is similar to a typical assignment operation in a procedure programming language.

By using OPN-IDE, the function $\text{staged}()$; can be implemented in any procedural language, such as Python or Java. The function signature of $\text{staged}()$ in domain P is considered an algebraic entry. When all its arguments are available, the numerical value will be calculated and the value will replace this function signature in the expression. Because OPN’s interpreter employs the concept of partial evaluation [35], when some argument values are unknown, the function signature will carry on as a part of the arithmetic expression to be determined later.

Mission-Mode Boolean Domain: The Boolean domain B is made up of Boolean expressions that define the conditions for feasible models. For example, one may write the expression $\text{prob} > 0.1$ to impose a rule that the probability of success must be greater than 10%. These Boolean expressions are associated with the incoming and outgoing arcs connected to *Processes*. In our OPN engine implementation, as the *Enumerating* operator executes, it also evaluates these parameters whenever possible, so that models that fail to satisfy these Boolean conditions will be eliminated. The result is that only models that are feasible under all of the logical conditions would reach the “Earth Landing Site.” This modeling approach effectively transforms the feasibility problem of space exploration missions into a reachability problem of a model enumeration process.

B. Enumerating Models

The second process in Fig. 2 is *Enumerating*. This process translates the *Enumerable Model* into a set of *Generated Submodels*. Based on a breadth-first traversal algorithm of the OPN graph, the different mission modes are composed through the combination of mission elements proposed by domain experts. The enumerator operation basically lists all the variants of the original model. The detail of the enumeration algorithm is documented in [21]. Intuitively speaking, each generated submodel is like a travel itinerary of the astronauts. It documents the sequence of using particular “Processes” to arrive at specific mission-mode “Objects” and the conditions they encountered at the arrival time. Since the enumeration process produces composed models that are also in the AoS model domain, the enumeration process can also be characterized as an algebraic operator. The enumeration process is an algebraic operator with respect to the AoS, because its input is a set of models (in most cases, just one element in that set), and the output is also a set of system models. It fulfills the closure condition of algebras.

C. Evaluating Models

The third step of Fig. 2 is *Evaluating*, which depends on system metrics recorded by the designer during the *Encoding* process and itineraries developed during the *Enumerating* process. Fig. 5 shows an example of an evaluation. The mission mode m at the Earth launch site has an initial probability and mass ratio. As it is transformed by the process, the probability is decreased, and the initial mass needed is increased (by this

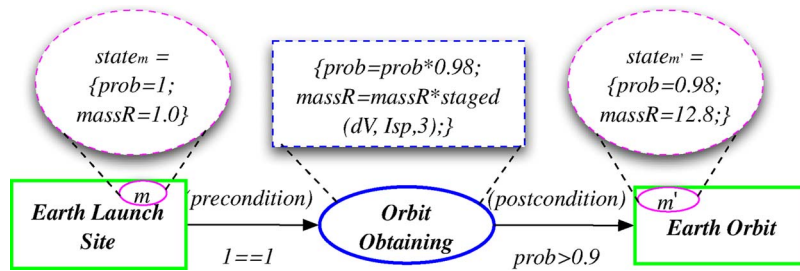


Fig. 5. Evaluation example.

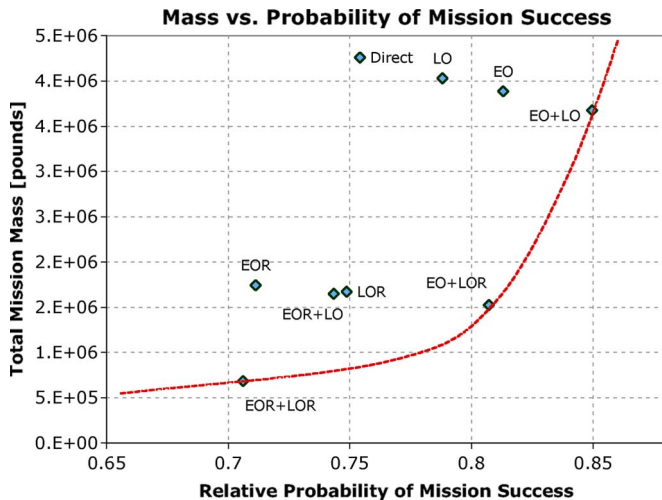


Fig. 6. Scatter plot representation of the mission modes.

bookkeeping scheme). The postcondition would eliminate any mission mode with probability less than or equal to 0.9 at this transition. As shown in the diagram, the properties can be treated as data that indicate the current state of a model and also can be used as transformation rules to evaluate transformed states.

D. Mission-Mode Example Summary

The Apollo mission-mode model generates and evaluates nine unique mission modes for sending humans from the Earth to the Moon and returning them. Fig. 6 shows a summary chart, showing the cumulative probability of success and initial mass of the nine mission modes. The “utopia point” is in the lower right—a mission mode having low mass and high probability of success.

In Fig. 6, we can see the three major mission modes for Apollo: Earth orbit rendezvous (EOR), lunar orbit rendezvous (LOR), and Direct (not stopping in either Earth or lunar orbit). Variants of these modes are possible, including EO+LO, stopping in both Earth and lunar orbit, but with no rendezvous (the mode originally favored by NASA). The mission mode involving Earth orbit then lunar orbit rendezvous (denoted EO+LOR) was the mission mode eventually chosen for the Apollo mission in June 1962. According to our study, the choice of EO+LOR was a good compromise between vehicle weight and relative risk and was the closest to the “utopia point” in the design space. By visual inspection, the historical choice is the middle one of the three options that locate on the “Pareto front”

of the tradeoff curve [36]. The Pareto front is a collection of points approximating the set of nondominated solutions. It is shown with a red line in Fig. 6.

This study showed that a model manipulation tool such as OPN-IDE could be used to automatically construct models based on a collection of explicitly stated assumptions and feasibility rules. Without OPN, each of the mission modes must be manually enumerated, and the respective formulas for the “total mission mass,” and “relative probability of mission success,” must be manually constructed. Then, some computer program or calculator procedures must be manually typed into the computer to conduct the evaluation. The algebraic nature of the OPN language made it particularly easy to change the assumptions of these calculations. For example, one may easily change the value of the specific impulse of the fuel, denoted as I_{sp} , and it would immediately produce a new set of numerical results of all the mission modes, respectively.

Of the three steps in the outline, the *Enumerating* and *Evaluating* steps can be realized as mechanical operations, so they are done automatically by the OPN kernel. The *Encoding* operation requires human intervention. The human modeling activities are realized by adding or removing elements in the bipartite graph, or rule entries written in L and logical expressions. These human-initiated model refinement tasks can be mechanically recorded as union and subtraction operations on the three value domains of AoS. The fact that all models can be encoded and manipulated in value domains of one functional programming language ensures the algebraic nature of this approach.

E. Other AoS Applications

The algebraic principles in AoS and the modeling language OPN have been successfully applied to several other more complex applications. It was used to study the varying compositional structures of different designs and assess the interactions between many types of variables. Other published examples include the following.

- 1) Study of Moon and Mars exploration architectures for the NASA Vision for Space Exploration [28]. In this study, over a thousand alternative feasible mission modes were generated and compared for human travel to the Moon or Mars.
- 2) Study of electronics pods for a military aircraft [21]. This study demonstrates OPN’s ability to reason about the possibility space of physical configurations under incomplete information.

- 3) Study of space-shuttle-derived cargo launch vehicles [37]. This study generated and evaluated hundreds of models for evolving space shuttle hardware into a new launch vehicle. This project also uses AoS/OPN as a method and the tool to perform the trade-space study on the physical design configuration of the system.
- 4) Study of NASA stakeholder network models for architecture evaluation [38]. This study used the OPN-IDE to build a system for discriminating between architectures based on a stakeholder-benefit model.
- 5) Decision-support framework for systems architecting [39]–[41]. Reference [41] describes the decision-support framework called architecture decision graph (ADG) that was developed using the principles of AoS and the tools provided by the OPN-IDE. ADG was applied to study the configuration of human lunar outpost architectures.

Additional examples of the application of AoS and OPN are the subject of future planned publications. These include a study of oil exploration systems in extreme environments and a study of the long-term plan for Earth observation for NASA.

V. DISCUSSION

The goal of this language design effort is to provide a machine-executable framework to support the process of model composition, reasoning, and selection. Via the three value domains, AoS allows us to represent submodel composition tasks as a dynamic process, express qualitative and quantitative inference rules as propositional statements, and capture the model selection/feasibility criteria as logical statements. This three-tiered language enables system designers and modelers to describe, share, and track the evolutionary paths of the modeling process; simulate system properties computationally; and filter the variations of models based on feasibility rules. Therefore, systems designers are better equipped to accumulate knowledge about the overall system design.

Enumerating models is usually considered as an intractable computational task. By using a symbolic programming language, such as OPN, it is possible to subjectively control the number of symbolically distinguishable models. For example, designers could choose to represent certain variables in the design as a set of symbolic variables. These variables can be subjectively assigned with different symbolic values that are distinguishable from each other, such as SMALL, MEDIUM, and LARGE. Therefore, the maximum number of algebraically distinguishable models can be subjectively determined *a priori*. This approach is inspired by partial evaluation [42], which enables designers to first tackle potentially intractable computing problems as a program compilation problem. It first processes the symbolic content of the model, without fully executing it. When more information becomes available, it then picks a “compiled” or symbolically specialized version of the model as a candidate for further computation or repetitive refinements. In the process of design exploration, where few variables should have decisive values, a lot of exploratory computational tasks could be avoided using this “partial evaluation” technique. In our implementation, we allow designers to use algebraic rules to distinguish between models of design spaces based on both

symbolic and numerical values. Without the ability to process numeric and symbolic values interchangeably in an executable model manipulation language, it is difficult to approach these kinds of model enumeration problems.

A notable movement called model-driven engineering (MDE) [43] is also directly related to our work. Under the MDE paradigm, models are not just artifacts manipulated by human hands and minds, they should be generated, analyzed, manipulated, and verified by computing processes whenever possible. Executable specification languages such as Executable UML [44] are being developed and used by systems engineers. However, these methodologies and their tools usually focus on specifying a particular instance of engineering system design. At the time this paper was written, rather few features in the existing Executable-UML tools are designed to help designers explore the design space at the early and fuzzy front end of design where creating and experimenting with different kinds of models may be most appropriate. We did not overlook the fact that most explorative design activities may involve an intractable amount of possible models. By using our algebraic approach, one could use symbolic variables to divide the space of design possibilities into a manageable number of model subspaces without overflowing the computational capacity of our model analysis tools. Clearly, this would require some advanced modeling skills and domain-specific insights in the modeling process. In any case, our algebraic modeling framework provides a symbolic basis to organize this type of complexity.

In contrast with others, our framework emphasizes that a model should be treated as a representation of a set of design possibilities. Each model should be considered a localized language for a local design space. With an algebraic approach to abstraction management, we argue that our model generation and solution filtering technique better addresses the modeling needs in model-driven engineering.

VI. FUTURE DEVELOPMENT

The algebra presented here is only one flavor of many possible algebras. The goal is to present a basic set of algebraic operands and operators so that we may use them as a metalanguage to describe other instances of system description languages. For example, there may exist other model enumeration schemes that are better than the Petri-net-like approach presented in this paper. Under the framework of AoS, different enumeration schemes may be treated as variations of the *Enumerating* operator as long as they satisfy the closure condition of the algebra. Finding some other enumeration schemes and more efficient model enumeration and filtering mechanisms is an important area of our future development.

To make it more convenient for designers to express their domain-specific knowledge, such as the statistical performance data of certain devices, we plan to create an OPN kernel that includes Bayesian Belief Network [45] as part of the calculation engine. The probabilistic belief structures will be encoded in the bipartite graph data structure. These new language features will make the OPN language more expressive. As a modeling language, when applied to distributed and context-aware computing systems, such as the intelligent workspace described in

[17], synchronizing different subsystems' behavior could be a major challenge. To enable more convenient representation of synchronized process interactions, formalized barrier and lease mechanisms should also be included in the OPN language. In [46], the authors demonstrated that a data-driven synchronization mechanism can be introduced into the OPN language. Our future work is to formalize these synchronization semantics as a part of the AoS specification.

One key feature of the AoS approach is the *Enumerating* operator, which maps the description of a solution set with the set of solution instances. Clearly, it would be extremely helpful when highly efficient model enumeration algorithms are available for enumerating solution instances in more complex design spaces. Therefore, integrating algorithms that can effectively enumerate feasible solution instances or quickly count the number of solution instances would be of immediate interest to the follow-on research. For identifying feasible solutions quickly, mature and high-performance constraint satisfaction checkers, often based on binary decision diagram [47], [48] manipulation algorithms, should be utilized. We have identified the execution engine KodKod of the language Alloy [49] as a potential alternative for efficient solution enumeration.

From a designer's viewpoint, a great modeling tool should enable designers to make sense of the calculated results, save time in the modeling process, and quickly leverage a wide range of sophisticated mathematical algorithms. At the time of this writing, a prototypical version of OPN kernel has been implemented in Mathematica [50]. We have also looked into implementing OPN kernel using other metamodeling environment, such as the generic modeling environment [51] by Vanderbilt University's Institute for Software. These tools will allow OPN users to have ready access to a larger collection of mathematical algorithms, interactive visualization gadgets, and code refactoring tools. However, without some rigorous benchmark modeling problems in place, it is hard to distinguish what are the driving factors in making better design models. As the software tools mature, and more application experience of the tools is accumulated, it would be more intuitive to summarize and compare the pros and cons of this approach in the format of case studies.

VII. CONCLUSION

This paper shows that algebraic techniques can be implemented as a computational method to manage the abstraction process of complex system designs. We showed that it is possible to use a set of objects and processes defined by domain experts that approximates operands and operators in a problem-specific algebra. This algebraic model can be iteratively submitted to a generic algebraic interpretation engine, such as the OPN interpreter, to generate many generations of more specialized algebraic models and perform model refinement calculations. This technique treats system specification models as instances of algebraic languages and recursively applies language refinement rules to identify for desirable system abstractions in the domain of algebraic languages. Clearly, systematically choosing the most effective language becomes a central question that needs to be answered. For

computing scientists, searching for an effective design in an explosive, often denoted as the NP-hard, design space can be a futile exercise. This algebraic approach partially alleviates the NP-hard issue by allowing users to partition the design space into mutually exclusive subdesign spaces, each covered by one specialized design language. Therefore, it significantly reduces the size of the problem from a potentially infinite number of concrete design plans into a finite number of "languages" that each describes a subset of the infinite number of design plans. The ability to incrementally narrow the design space using machine-generated algebraic specifications is the main reason that we chose "languages" or "algebras" as the medium to describe system designs.

In [2], Hoare *et al.* questioned whether a small set of algebraic laws can be directly useful in a practical engineering design problem. The absence of a tool that can bridge the cognitive gap between mathematical abstractions and engineering problems may have been the main reason for their conservative attitude. We hope that by showing our example and tool, designers can see that a small set of algebraic operations on models is a representationally efficient way to reason about certain complex design problems.

ACKNOWLEDGMENT

The authors would like to thank D. Dori, P. Carlile, and E. Qiu for their inputs on this paper and also D. Rayside and R. Seater for their insightful discussions and technical inputs that made this paper more technically accurate and readable.

REFERENCES

- [1] B. H. Y. Koo, W. L. Simmons, and E. F. Crawley, "Algebra of systems: An executable framework for model synthesis and evaluation," in *Proc. Int. Conf. Syst. Eng. Model.*, 2007, pp. 42–49.
- [2] C. A. R. Hoare, I. J. Hayes, J. He, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin, "Laws of programming," *Commun. ACM*, vol. 30, no. 8, pp. 672–686, Aug. 1987.
- [3] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.
- [4] P. Chen, "The entity-relationship model—Toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976.
- [5] P. Cousot, "Integrating physical systems in the static analysis of embedded control software," in *Proc. 3rd Asian Symp. Program. Languages Syst.*, K. Yi, Ed., 2005, pp. 135–138.
- [6] B. C. Williams, "A theory of interactions: Unifying qualitative and quantitative algebraic reasoning," *Artif. Intell.*, vol. 51, no. 1–3, pp. 39–94, 1991.
- [7] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points," in *Proc. Conf. Rec. 4th Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, Los Angeles, CA, 1977, pp. 238–252.
- [8] N. P. Suh, *The Principles of Design*. London, U.K.: Oxford Univ. Press, 1990.
- [9] C. Y. Baldwin and K. B. Clark, *Design Rules*, vol. 1. Cambridge, MA: MIT Press, Mar. 2000.
- [10] R. de Neufville, *Applied Systems Analysis: Engineering Planning and Technology Management*. New York: McGraw-Hill, 1990.
- [11] W. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular software design with crosscutting interfaces," *IEEE Softw.*, vol. 23, no. 1, pp. 51–60, Jan./Feb. 2006.
- [12] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
- [13] C. Carlson, "Grammatical programming: An algebraic approach to the description of design spaces," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1993.

[14] B. Wielinga and G. Schreiber, "Configuration-design problem solving," *IEEE Expert: Intell. Syst. Their Appl.*, vol. 12, no. 2, pp. 49–56, Mar. 1997. [Online]. Available: <http://www.cs.vu.nl/~guus/papers/Wielinga97a.pdf>

[15] R. Dechter, *Constraint Processing*. San Mateo, CA: Morgan Kaufmann, 2003.

[16] T. Darr, M. Klein, and D. L. McGuinness, "Special issue: Configuration design," *Artif. Intell. Eng. Des., Anal. Manuf.*, vol. 12, no. 4, pp. 293–294, Sep. 1998.

[17] A. Padovitz, S. W. Loke, and A. Zaslavsky, "Multiple-agent perspectives in reasoning about situations for context-aware pervasive computing systems," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 38, no. 4, pp. 729–742, Jul. 2008.

[18] S. Burris and H. P. Sankappanavar, *A Course in Universal Algebra*. New York: Springer-Verlag, 1981.

[19] S. Pedroni and N. Rapping, *Jython Essentials*. Sebastopol, CA: O'Reilly, 2002.

[20] C. A. Petri, "Kommunikation mit automaten," Ph.D. dissertation, Univ. Bonn, Bonn, Germany, 1962.

[21] B. H. Y. Koo, "A meta-language for systems architecting," Ph.D. dissertation, MIT, Cambridge, MA, 2005.

[22] D. Dori, *Object-Process Methodology: A Holistic Systems Paradigm*. New York: Springer-Verlag, 2002.

[23] C. G. Brooks, J. M. Grimwood, and L. S. Swenson, *Chariots for Apollo: A History of Manned Lunar Spacecraft*. Washington DC: NASA: Sci. Tech. Inf. Branch, 1979.

[24] J. R. Hansen, *Spaceflight Revolution: NASA Langley Research Center From Sputnik to Apollo*, ser. The NASA History Series. Washington, DC: Nat. Aeronautics Space Admin., 1995. NASA SP-4308.

[25] C. Murray and C. B. Cox, *Apollo*. Burkittsville, MD: South Mountain Books, 2004.

[26] R. C. Seamans, Jr., *Project Apollo: The Tough Decisions*, ser. Monographs in Aerospace History Number 37. Washington, DC: Nat. Aeronautics Space Admin., 2005. NASA SP-2005-4537.

[27] W. Simmons, M. Kinnunen, and B. H. Y. Koo, Apr. 1, 2005, Beverly, MA. Interview with Robert C. Seamans.

[28] W. L. Simmons, B. H. Y. Koo, and E. F. Crawley, "Architecture generation for Moon-Mars exploration using an executable meta-language," in *Proc. AIAA Space*, Long Beach, CA, Aug. 30–Sep. 1, 2005.

[29] B. H. Y. Koo, W. L. Simmons, and E. F. Crawley, "A valuation technology for product development options using an executable meta-modeling language," in *Complex Systems Concurrent Engineering: Collaboration, Technology Innovation and Sustainability*. New York: Springer-Verlag, 2007, pp. 107–115.

[30] M. Amram and N. Kulatilaka, *Real Options: Managing Strategic Investment in an Uncertain World*. London, U.K.: Oxford Univ. Press, 1998.

[31] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2006.

[32] E. Frazzoli, "Robust hybrid control for autonomous vehicle motion planning," Ph.D. dissertation, MIT, Cambridge, MA, Jun. 2001.

[33] V. A. Chobotov, Ed., *Orbital Mechanics*, 2nd ed. Washington, DC: AIAA, 1996.

[34] J. C. Houbolt, "Manned lunar-landing through use of lunar-orbit rendezvous," NASA, Washington, DC, Tech. Rep. NASA-TM-74736, 1961.

[35] Y. Futamura, "Partial evaluation of computation process; An approach to a compiler-compiler," *Higher-Order Symbolic Comput.*, vol. 12, pp. 381–391, 1999.

[36] V. Pareto, *Manual of Political Economy*. New York: A. M. Kelley Publisher, 1971.

[37] W. L. Simmons, B. H. Y. Koo, and E. F. Crawley, "Space systems architecting using meta-languages," presented at the 56th Int. Astronautical Congr., Fukuoka, Japan, 2005, Paper IAC-05-D1.P.03.

[38] B. G. Cameron, S. Catanzaro, and E. F. Crawley, "Value based architecture selection," presented at the 57th Int. Astronautical Congr., Valencia, Spain, 2006, Paper IAC-06-D3.1.03.

[39] W. L. Simmons, B. H. Y. Koo, and E. F. Crawley, "A computational method for mapping the decision space of the lunar exploration program," presented at the 57th Int. Astronautical Congr., Valencia, Spain, 2006, Paper IAC-06-D1.3.07.

[40] W. L. Simmons, B. H. Y. Koo, and E. F. Crawley, "Decision-making for systems architecting using meta-languages," in *Proc. IEEE Int. Conf. SMC*, Montréal, QC, Canada, Oct. 2007, pp. 703–707.

[41] W. L. Simmons, "A framework for decision-support in systems architecting," Ph.D. dissertation, MIT, Cambridge, MA, Feb. 2008.

[42] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[43] S. Kent, "Model driven engineering," in *Proc. IFM*. Berlin, Germany: Springer-Verlag, 2002, vol. 2335, pp. 286–298.

[44] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA: Addison-Wesley, 2002.

[45] K. Murphy, *Bayes Net Toolbox for MATLAB*. [Online]. Available: <http://www.cs.ubc.ca/~murphyk/Bayes/bnsoft.html>

[46] K. Huang, Z. Li, and B. H. Y. Koo, "Process abstraction and its applications in transportation systems," in *Proc. IEEE Int. Conf. Ind. Inf.*, Jul. 2008, pp. 227–232.

[47] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978.

[48] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, Sep. 1992.

[49] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, 2006.

[50] Wolfram Research, *Mathematica*. [Online]. Available: <http://www.wolfram.com/products/mathematica/index.html>

[51] Inst. Software and Integrated Syst., Vanderbilt Univ., *GME: Generic Modeling Environment*. [Online]. Available: <http://www.isis.vanderbilt.edu/projects/gme/>



Benjamin H. Y. Koo received the B.E. degree in mechanical engineering and the M.S. degree in industrial engineering and operations research from the University of Minnesota, Minneapolis, in 1992 and 1995, respectively, and the M.S. degree in systems design and management and the Ph.D. degree in engineering systems from the Massachusetts Institute of Technology, Cambridge, in 2002 and 2005, respectively.

He has worked in the information technology industry for over ten years and has served various clients in different industry sectors, particularly in the financial and telecommunication industries. He is currently an Associate Professor of industrial engineering with the Department of Industrial Engineering, Tsinghua University, Beijing, China.



Willard L. Simmons (M'07) received the B.S. degree in mechanical engineering from the University of New Hampshire, Durham, in 1997, the M.S. degree in aerospace engineering sciences from the University of Colorado, Boulder, in 1999, the M.S.E. degree in mechanical and aerospace engineering from Princeton University, Princeton, NJ, in 2005, and the Ph.D. degree from the Massachusetts Institute of Technology, Cambridge, in 2008.

From 1999 to 2001, he was with Lockheed Martin as a Guidance, Navigation, and Controls Engineer for the Atlas Launch Vehicles Program. From 2001 to 2002, he was with T.R.W. (now part of Northrop Grumman) as a Controls and Propulsion Engineer for the Chandra Space Telescope. He is currently the Vice-President of Technology at DataXu, Inc., Cambridge, MA.

Dr. Simmons is a member of the American Institute of Aeronautics and Astronautics.



Edward F. Crawley received the B.S. and M.S. degrees in aeronautics and astronautics and the Sc.D. degree in aerospace structures from the Massachusetts Institute of Technology (MIT), Cambridge, in 1976, 1978, and 1980, respectively.

He was formerly the Executive Director of the Cambridge-MIT Institute (CMI). Before working with CMI, he was the Head of the Department of Aeronautics and Astronautics, MIT, where he is currently the Ford Professor of engineering. He is also currently a Professor of engineering systems with MIT. He is the author of over 50 journal publications in the *AIAA Journal*, the *ASME Journal*, the *Journal of Composite Materials*, and *Acta Astronautica*.

Dr. Crawley is a Fellow of the Royal Aeronautical Society (U.K.), the Royal Swedish Academy of Engineering Science, and the American Institute of Aeronautics and Astronautics. He is a member of the National Academy of Engineering. He has been awarded the NASA Public Service Medal.