Proceedings of the 1995 *International Symposium and Workshop on Systems Engineering of Computer Based Systems*, Tucson, Arizona

Domain Modeling in Engineering Computer-Based Systems[1]

Don Batory
Dept. of Computer Sciences
University of Texas
Austin, Texas 78712
batory@cs.utexas.edu

David McAllester
The AI Laboratory
MIT
545 Technology Square
Cambridge, MA 02139
dam@ai.mit.edu

Lou Coglianese and Will Tracz
Loral Federal Systems
1801 State Route 17C, MD 210
Owego, New York 13827-1298
{ lou, tracz }@lfs.loral.com

Abstract

Domain modeling is believed to be a key factor in developing an economical and scalable means for constructing families of related software systems. In this paper, we review the current state of domain modeling, and present some of our work on the ADAGE project, an integrated environment that relies heavily on domain models for generating real-time avionics applications. Specifically, we explain how we detect errors in the design of avionics systems that are expressed in terms of compositions of components. We also offer insights on how domain modeling can benefit the engineering of computer-based systems in other domains.

1 Introduction

Design techniques and methodologies for large scale software systems have traditionally focussed on one-of-a-kind products. Leveraging of software designs, and maybe even code, from previously built systems has largely been done in an ad hoc manner. For many years, this approach was adequate: software systems were relatively simple; it was cost effective to write software from scratch and to use ad hoc methods of reuse. This is no longer true. It is widely recognized that traditional methods of software construction are not cost effective in realizing large systems. Further, because software complexity is growing rapidly, there is a critical need for economical and scalable methods of software construction.

1. This research was sponsored, in part, by the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Air Force Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

Conventional software design methodologies fail to recognize that software systems are rarely one-of-a-kind. There is often a long history of similar and competing designs and implementations that give rise to next generation products. Conventional methodologies fail to amortize costs of design and implementation for creating families of similar systems. Herein lies the key to next generation software construction.

Domain modeling is the name given to an emerging class of software design methodologies that formally define families of related systems. A domain model identifies the essential features, components, capabilities, interfaces, abstractions, etc. of a *family* (or *domain*) of systems. A domain model explains precisely the differences and similarities of distinct members of the domain, and provides system designers with language(s) that allow them to specify, evaluate, and possibly generate target family members. Reuse of design and code across multiple members is both an essential and expected benefit.

ARPA's *Domain Specific Software Architecture* (DSSA) project was created to foster innovative approaches for generating control systems. The goal is to use domain models and advances in nonlinear control and hierarchical control theory to generate avionics, command and control, and vehicle management applications with an order of magnitude improvement in productivity and quality. ADAGE (Avionics Domain Application Generation Environment) is a DSSA project for avionics [Cog92-93, Goo92a-b]. The premise of ADAGE is that many of the problems in navigation, guidance, and flight director software are well-understood. For any new avionics system, several features will require new and innovative

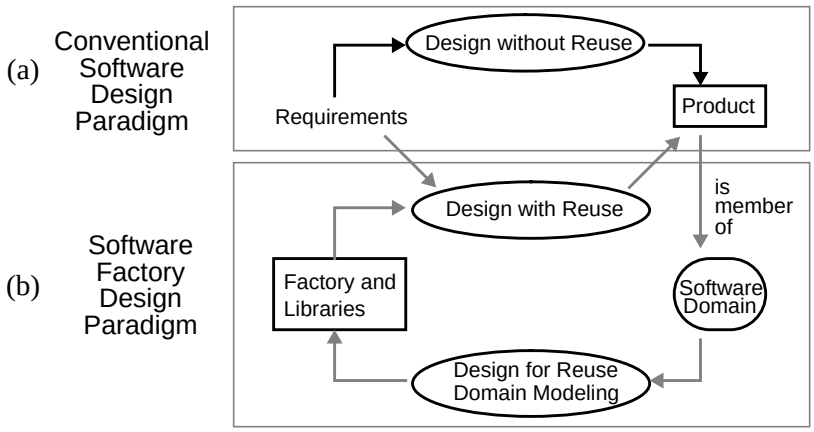

Figure 1. A Software Factory Paradigm

software, but much of a new system can be built by combining and adapting existing components.

We believe that domain modeling has applications beyond systems that are exclusively software-based. In fact, avionics systems are a tightly-coupled meld of hardware and software, so the domain modeling techniques used in ADAGE are definitely not limited to software. In this paper, we review the current state of domain modeling, and explain how a particular ADAGE tool automatically detects errors in avionics systems that have been specified as compositions of components. We also offer insights on the role of domain modeling in engineering computer-based systems.

2 Current State of Domain Modeling

Domain modeling is an integral part of a "software factory" paradigm. Rather than going directly from requirements to a design and implementation phase to produce a product (i.e., the conventional software design paradigm - Figure 1a), the software factory paradigm relies on generators and libraries of reusable components to produce a target system. *Domain modeling* is an essential intermediate step which outlines the design of a factory (or generator) and libraries from which to assemble new products (Figure 1b). That is, a software product is recognized to be a member of a domain of similar software systems. A domain model captures this domain by differentiating the features of different systems, and prescribing the libraries and generators needed to automatically construct domain members.

Two distinct kinds of domain models have emerged in recent years: integrative and generative. Integrative models define target systems from a problem-oriented feature perspective; their goal is to automatically produce specifications for target systems. Generative models define target systems from a solution-oriented system constraint perspective; their goal is to actually produce the software for targeted systems. Integrative models are the most common and best understood.

An *integrative model* is integrated set of distinct submodels that are used to develop specifications for target systems. Submodels provide different views (i.e., information) about a target system. Typically submodels include generalization/specialization hierarchies, state transition diagrams, aggregation hierarchies, and features. Generalization/specialization submodels define the fundamental objects of a domain and their specializations. Aggregation hierarchy submodels express complex aggregate object types in terms of more primitive types. State transition submodels define the execution semantics of objects in terms of finite state machines. As a general rule, the generalization, aggregation, and state transition submodels embody direct extensions to conventional software design methodologies; the basic extension is the ability to include or exclude optional objects, relationships, and states when specifying a target system.

The most innovative submodel is the feature submodel. A *feature* is a prominent or distinctive, user-visible quality or characteristic of a system. A *feature submodel* is the set of features that characterize and distinguish the members of a family of systems. System designers initially define their target system by specifying its set of features. In general, not all combinations of features are meaningful. Thus, an important part of a feature submodel is a rule base/knowledge base that constrains the selection of features to only reasonable (or supportable) combinations. Once the set of features for a target system have been defined, CASE tools propagate the selected features to their representations in the other submodels (e.g., generalization submodel, etc.) to make all submodels consistent. It is using the information in these consistent, integrated submodels that specifications for the target system can be generated automatically.

Well-known examples of integrative models are the Evolutionary Domain Life Cycle (EDLC) Model [Gom94], Feature Oriented Domain Analysis (FODA) [Coh94], and Organization Domain Modeling (ODM) [Sim94].

Integrative models do not generate code for the target system; they rely on software system generators for the target domain. *Generative models* or *reference architecture models* are the second kind of domain model that serve as blueprints for domain-specific software generators. Generative models primarily deal with software architecture issues: identifying the fundamental programming abstractions of a domain, creating libraries of plug-compatible and interoperable software building blocks for the domain, defining knowledge representation languages for stating which combinations of building blocks are meaningful (and which are illegal), and classifying different types of “glue” (i.e. communication protocols) that can be used to interconnect legal combinations of components.

Examples of generative models are DRACO [Nei89], DTRE [Bla91], and GenVoca [Bat92]. Special cases of the basic ideas behind generative models include object-oriented frameworks and design patterns [Gam94].

Generative models capture much of the same information as integrative models, but present only a limited, implementation-oriented view of target systems (i.e., in terms of software components, their compositions, and communication protocols). Ideally, a generative model should be another submodel of an integrated model. However, generative and integrative have evolved independently and their unification has not yet been achieved.

3 The ADAGE Generative Model

ADAGE uses integrative and generative domain models to produce avionics systems to produce avionics systems (consisting Navigation, Guidance, and Flight Director subsystems) that can support programs as JAST (Joint Advanced Strike Technology) and others [Cog93]. The development and experimentation with ADAGE is ongoing; recent descriptions of the ADAGE environment and its support for specific avionics applications are presented in [Tra95, Bat95]. In this paper, we will focus on the generative domain model of ADAGE, and in particular, an important problem that arises in its generators.

ADAGE relies on GenVoca [Bat92] to express its generative model. Briefly, avionics software can be represented by a collection of primitive, plug-compatible building blocks called *components*. All components export and import standardized interfaces, so components “snap” together like legos. Libraries of components that implement the same standardized interface are *realms*. A realm, in effect, is a library of plug-compatible and interchangeable components. An avionics software system is defined as a composition of components (legos); the family of all avionics systems that can be generated is defined by the set of all possible combinations of realm components. In general, the number of such compositions is very large. In the case that needed capabilities are not available, new components and even new realms can be added.

This approach to modeling software imposes a layered view of avionics systems. Figure 2 depicts the general layering of avionics subsystems and

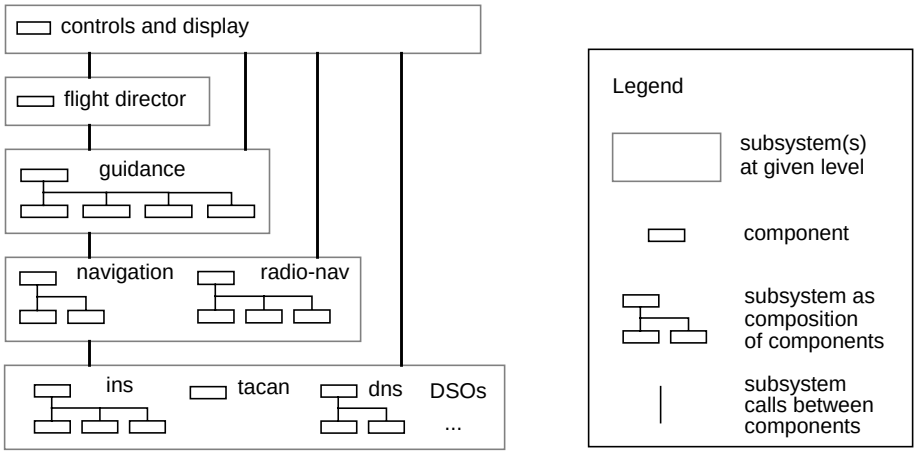

Figure 2. A Layered Architecture for Avionics Software

suggests that each major subsystem is a composition of primitive components. The bottom-most subsystems are *data source objects* (DSOs), i.e., sensors. Examples include inertial navigation sensors (INS), VHF omnirange sensors (VOR), and global positioning sensors (GPS). DSO subsystems report their raw sensor values to the *navigation* and *radio navigation* subsystems, which estimate the aircraft's position relative to earth coordinates or a fixed-location radio beacon. The *guidance* subsystem determines the difference between mission objectives and the current aircraft state (position). The *flight director* subsystem converts guidance errors into pilot control cues or autopilot commands.

The ADAGE generative model identifies over 40 different realms containing over 350 components. An avionics system is modeled by a composition of components, called a *type equation*, that specifies how components/layers are stacked. Type equations for even simple avionics systems tend to be non-trivial. Equations often reference more than 50 distinct components that are stacked 15 layers deep. The key contribution of this approach is that it is possible to reason about avionics software at a high level; instead of examining the details of code and ad hoc models of software design, properties of avionics software systems can be deduced from the properties of its components. Because components are "standardized", composition tools, generators, analysis tools, etc. can be created to support the development of families of avionics software systems quickly and inexpensively. It is this integrated suite of tools that defines the ADAGE environment [Cog93].

An important problem that arises in modeling avionics software in ADAGE (and GenVoca) is that there are syntactically correct type equations (i.e., compositions of components) that are not semantically correct. That is, the specified composition of components simply will not produce a correctly functioning avionics system. A critical capability of the ADAGE tool set is to detect composition errors automatically. This capability is defined by Variational Attribute Grammars (VAGs).

4 Variational Attribute Grammars

VAG is a functional programming language designed for the development of constraint-based CAD systems such as ADAGE. There are two kinds of users of the VAG system: the VAG programmer and the CAD system user. The VAG programmer defines a set of VAG programs which are then used as design primitives in a CAD system. Each VAG program models the constraints associated with a particular (ADAGE) software component. VAG programs are parameterized, much like their ADAGE component counterparts. When an avionics system is modeled by a composition of components, the corresponding composition of

VAG programs models the inter-component constraints that must be satisfied by that composition. CAD systems that are built/modeled in this way are called *VAGCAD systems*.

For example, in the avionics domain a natural component is a high pass filter. Natural parameters of a high pass filter are the frequency at which the filter starts to pass the signal and the "sharpness" of the step in the frequency response. Different values of the sharpness parameter can lead to quite different filter designs. The VAG program representing this component is a function which takes various parameters and returns a data structure representing the filter. The VAGCAD system is designed to manipulate and reason about designs, i.e., designs in which some parameters and components are represented by variables whose value has not been specified.

The term "VAG" is an acronym for Variational Attribute Grammar. Traditional attribute grammars play an important role in software development tools such as compilers and editors. A computer program is an expression satisfying the syntactic requirements embodied in a grammar which describes the language in which the program is written. Traditional attribute grammars augment traditional context free grammars with equations that define "attributes" of programs and program fragments [Rep85, Der88]. A set of VAG programs can be viewed as an attribute grammar and a VAG programmer can be viewed as a grammar designer. Actually, it is the type information associated with defined functions that plays the role of a grammar. Type information specifies a set of well typed expressions in much the same way that a traditional grammar specifies a set of grammatical expressions. The relationship between types and grammars has been widely studied in the context of natural language syntax [Adj35, Bac88]. A design can be viewed as an attributed grammatical (or well typed) expression. However, the analogy with attribute grammars hides the fact that the VAG language is also a general purpose functional programming language. Both views of VAG --- the grammar view and the functional programming view --- are important and useful.

.

Primitive	**Type Declaration**
`+, *, -, /`	`number` $\times$ `number` $\rightarrow$ `number`
`<, <=, >, >=`	`number` $\times$ `number` $\rightarrow$ `boolean`
`and, or`	`boolean` $\times$ `boolean` $\rightarrow$ `boolean`
`not`	`boolean` $\rightarrow$ `boolean`
`true, false`	$\rightarrow$ `boolean`
`=`	$\sigma \times \sigma \rightarrow$ `boolean`
`if`	`boolean` $\times \sigma \times \sigma \rightarrow \sigma$
`cons`	$\sigma \times$ `(list-of` σ`)` $\rightarrow \sigma$
`car`	`(list-of` σ`)` $\rightarrow$ `(list-of` σ`)`
`cdr`	`(list-of` σ`)` $\rightarrow$ `(list-of` σ`)`
`nil`	$\rightarrow$ `(list-of anything)`
`null?`	$\sigma \times$ `(list-of` σ`)` $\rightarrow$ `boolean`
`member?`	$\sigma \times$ `(list-of` σ`)` $\rightarrow$ `boolean`
`append`	`(list-of` σ`)` $\times$ `(list-of` σ`)` $\rightarrow$ `(list-of` σ`)`
`map`	`(`$\sigma \rightarrow \tau$`)` $\times$ `(list-of` σ`)` $\rightarrow$ `(list-of` τ`)`

TABLE 1. The Primitives of VAG

Formally, VAG is best defined as a functional programming language. It uses Lisp syntax (or lack thereof) but unlike Lisp, VAG is strongly typed. The language consists of the primitive functions shown in Table 1 together with the ability to extend the language with recursive (and non-recursive) definitions and structure declarations. Structure declarations are given with Common Lisp syntax as in the following example

```
(defstruct drawer
   (drawer-height number)
   (drawer-width number)
   (drawer-depth number))
```

The above declaration introduces **drawer** as a type symbol. It also introduces four functions **make-drawer**, **drawer-height**, **drawer-width** and **drawer-depth**. The function **make-drawer** takes three arguments, one for each field of the drawer data structure. For example, the value of **(make-drawer 10 15 25)** is a drawer data structure with a height of **10** a width of **15** and depth of **25**. The functions **drawer-height**, **drawer-width** and **drawer-depth** each take a drawer data structure as an argument and return a number representing the height, width, or depth of that drawer respectively.

Both structure declarations and function definitions can associate constraints with structure types and functions respectively. Figure 3 shows an example of a constraint associated with a structure type. It is a conjunction of two primitive constraints: the first states that the sum of the drawer heights is 10 units less than the height of the dresser, and the second states that the widths of all drawers equals the width of the dresser minus 5 units. This constraint involves the defined functions **sum** and **every**, each having simple recursive definitions. Now suppose that we ask the system to consider the following partial design:

```
(defvar w  number)
(defvar h1 number)
(defvar w1 number)
(defvar d  number)
(defvar w2 number)
(make-dresser w 50
    (list (make-drawer h1 w1 d)
          (make-drawer 15 w2 d)
          (make-drawer 15 30 d)))
```

The above design states that the height of a dresser is 50 units; the height of the bottom two drawers is 15 units, and the width of the bottom drawer is 30 units. The constraints generated by this expression will be partially evaluated to yield the following:

```
(and (= (+ h1 (+ 15 15)) (- 50 10))
     (and (= w1 (- w 5))
          (and (= w2 (- w 5))
               (and (= 30 (- w 5))
                    (true)))))
```

Given these formulas, VAGCAD's boolean and numerical reasoning will derive that **h1** is 10, **w** is 35, and **w1** and **w2** are both 30. Note that in this example, the design is incomplete. That is, parameter **d**, the depth of the dresser, cannot be inferred from constraints and remains to be defined.

Traditional attribute grammars make a distinction between inherited attributes whose values are

```
(defstruct (dresser
             (with-constraint
               (and (= (sum (map (lambda (x) (drawer-height x))
                                 (dresser-drawers self)))
                       (- (dresser-height self) 10))
                    (every (map (lambda (drawer)
                                  (= (drawer-width drawer)
                                     (- (dresser-width self) 5)))
                                (dresser-drawers self))))))
   (dresser-width number)
   (dresser-height number)
   (dresser-drawers (list-of drawer)))
```

Figure 3. A VAG Constraint

derived from context and synthesized attributes whose values are computed from subexpressions. In a VAG expression, attributes are related by constraints which allow multidirectional flow of information. Variational attribute grammars are similar to, but formally simpler than, relational attribute grammars as described in [Cou88]. Although the value of a VAG attribute can be inferred using multidirectional constraint flow, at a semantic level the VAG language has only synthesized attributes. An attribute **`width`** of an object *x*, denoted by **`(width`** *x***`)`**, is represented by the term which is semantically no different from any other term in a functional programming language. The value of the term **`(width`** *x***`)`** is derived by applying the function **`width`** to the object *x*. However, if *x* is a variable, or contains free variables, then the value of **`(width`** *x***`)`** can not be determined in this way. It might be determined, however, by global constraints on the design from which one can infer that the term **`(width`** *x***`)`** must have value **`3`**. In this way multidirectional constraints can be used to derive the value of the attribute of the object *x*.

Although VAG is formally a functional programming language, VAG programming is quite different from programming in traditional functional languages. VAG programs are intended to be used as parts of partial designs. In particular, the VAG interpreter is designed to operate on partial designs. In a VAGCAD system, a partial design is a VAG expression in which some parts are not given. These yet to be designed parts are represented by variables. In order to create useful VAG primitives the VAG programmer must understand the constraint based reasoning mechanisms that will be available to the VAGCAD system user. The VAG functions created by the VAG programmer should be such that the VAGCAD reasoning mechanisms can reason about them effectively.

The use of variables to represent yet to be designed components is formally similar to the use of logic variables in constraint logic programming. The main difference is the intended degree of interaction between the system and a human designer. In a VAGCAD system, logic variables range over yet to be designed aspects of a system. The VAGCAD system performs as much constraint based reasoning as is feasible but the system relies on a human designer to make major decisions in filling in a design.

5 Engineering of Computer-Based Systems

Designing and building systems (largely) from prefabricated components has long been a common engineering discipline. Standards and criteria for acceptable designs have been established; common sets of analyses can be applied to interrogate specific capabilities of designs and to identify potential weaknesses. The design environments for computer software have not matched their counterparts of computer hardware. How one designs large software systems from existing components is only now becoming the focus of extensive research [Gar93].

The benefits of creating (domain) models that define families of related systems are clear [Par79]; the cost of creating multiple systems is amortized over the lifetime of a domain model. (Reports that a payback is achieved within the first few systems uses of the domain model are common [Bar94]).

Much effort has been invested in studying the practices of other (non-software) disciplines in order to discern how to build software better. These efforts have largely failed; it is not obvious how lessons learned from, say, the mechanics of building bridges or designing homes can be applied to software. We believe that projects like ADAGE and its reliance on VAGS to detect errors in designing systems as compositions of components will ultimately lead to practical paradigms for mass-producing customized software. Only after such projects have matured will correlations with other engineering disciplines be truly valuable and meaningful.

6 References

[Adj35] K. Adjuciewicz, “Die Syntaktische Konnexitat”, *Studia Philophica*, Vol #1 pp. 1-27, 1935. Translated as “Sytactic Connection”

in Strolls McCall (ed), *Polish Logic: 1920-1939*, Oxford University Press, 1967.

[Bac88] E. Bach, "Categorial Grammars as Theories of Language", in *Categorial Grammars and Natural Language Structures*, pp. 17-34, R. Oehrle and E. Bach (ed), D. Reidel, 1988.

[Bar94] D. Barstow, "Domain-Specific Architectures and Software Customization: A Case Study", key note presentation at the *3rd International Conference on Software Reuse*, November, 1994.

[Bat92a] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Trans. Software Engineering and Methodology*, October 1992.

[Cog92] Lou Coglianese, et al., "An Avionics Domain-Specific Software Architecture," *ARPA PI Conference*, 1992. Also in *CrossTalk*, October 1992, and IBM Owego T.R. ADAGE-IBM-92-07, April 1992.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proceedings of AGARD 1993*. Also, IBM Owego T.R. ADAGE-IBM-93-04, May 1993.

[Coh91] S.G. Cohen, J.L. Stanley, A.S. Peterson, R.W. Krut, "Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain", Software Engineering Institute, Carnegie Mellon University, CMU/SEI-91-TR-28.

[Cou88] B. Courcelle and P. Deransart, "Proofs of Partial Correctness for Attribute Grammars with Applications to Recursive Procedures and Logic Programming", *Information and Computation*, Vol. #78, pp. 1-55, 1988.

[Der88] P. Deransart, M. Jourdan, and B. Lorho, "Attribute Grammars: Defintions, Systems and Bibliography", Springer-Verlag 1988, in *Lecture Notes in Computer Science* Vol. 323.

[Gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gar93] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Adances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.

[Gom94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli, "A Prototype Domain Modeling Environment for Reusable Software Architectures", *International Conference on Software Reuse*, 1994.

[Goo92a] M. Goodwin and L. Coglianese, "Dictionary for the Avionics Domain Architecture Generation Environment of the Domain-Specific Software Architecture Project", ADAGE-IBM-92-04.

[Goo92b] M. Goodwin and M. Kushner, "Domain Analysis for the Avionics Domain Architecture Generation Environment of Domain Specific Software Architecture", ADAGE-IBM-92-11, November 1992.

[Nei89] J.M. Neighbors, "DRACO: A Method for Engineering Reusable Software Systems", in *Software Reusability*, T. Biggerstaff and A. Perlis, ed., ACM Press Frontier Series, Addison Wesley, 1989.

[Par79] D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979.

[Reps85] T. Reps and T. Teitelbaum, "The Synthesizer Generator Manual", Cornell University, August 1985.

[Sim94] M.A. Simos, "An Introduction to Organization Domain Modeling", tutorial notes, *International Conference on Software Reuse*, 1994.