To appear in the *Symposium on Software Reusability 1995*, Seattle, Washington.

Creating Reference Architectures: An Example from Avionics[1]

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

Lou Coglianese, Mark Goodwin, and Steve Shafer
Loral Federal Systems Company
Owego, New York 13827
{ lou, goodwin, reve }@lfs.loral.com

Abstract

ADAGE is a project to define and build a domain-specific software architecture (DSSA) environment for assisting the development of avionics software. A central concept of DSSA is the use of software system generators to implement component-based models of software synthesis in the target domain [SEI90].

In this paper, we present the ADAGE component-based model (or reference architecture) for avionics software synthesis. We explain the modeling procedures used, review our initial goals, show how component reuse is achieved, and examine what we were (and were not) able to accomplish. The contributions of our paper are the avionics reference architecture and the lessons that we learned; both may be beneficial to others in future modeling efforts.

1 Introduction

ARPA's Domain-Specific Software Architectures (DSSA) program was established in 1990 to create innovative approaches for generating control systems [SEI90]. The goal is to use formal descriptions of software architectures and advances in non-linear 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. A DSSA not only provides a framework for reusable software components, but it also organizes design rationale and structures adaptability. 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 software, but much of the new system can be built by combining and adapting existing components. Therefore, a domain analysis can be used to identify components and constraints inherent in the avionics domain. A product of domain analysis, called a *reference architecture*, is a blueprint for an avionics software system generator.

A critical step in creating a DSSA is the definition of a reference architecture. Domain analysis techniques are still immature [Ara93, War92]; there are no commonly accepted modeling processes or meta-modeling constructs that are used to define reference architectures. Of critical importance is that whatever constructs or processes are chosen must be domain-independent; repeatability across multiple domains is an essential requirement of a successful DSSA methodology.

We have chosen to develop the ADAGE reference architecture in terms of the GenVoca model [Bat92a]. This model is suited for software system generation; it relies on a particular style of organizing hierarchical software systems in terms of standardized sets of parameterized, plug-compatible, and reusable layers called components. Using components and rules for their composition, large families of software systems can be defined.

In this paper, we present the ADAGE reference architecture. Although our work focuses on avionics software, the emphasis of this paper is on creating reference architectures, using the avionics domain as an example. We begin by explaining the role of domain modeling and reference architecture modeling in ADAGE (and we believe DSSA software system generators, in general). We then review GenVoca modeling

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.

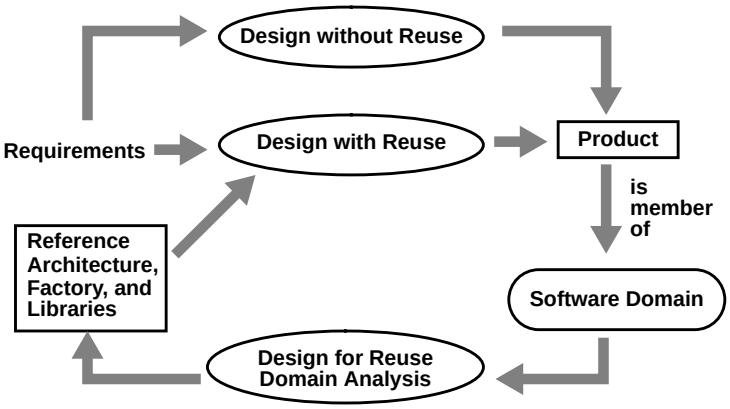

Figure 1. A Software Factory Paradigm

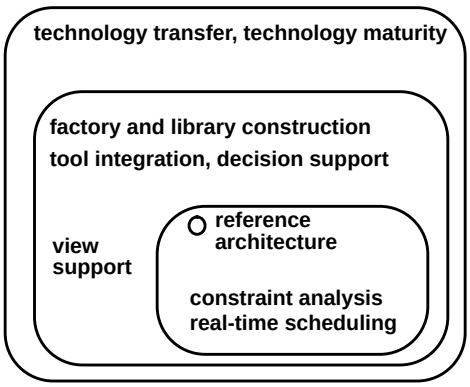

Figure 2. ADAGE Scopes of Activities

constructs and present the ADAGE reference architecture couched in its terms. The contributions of our paper are an avionics reference architecture and the lessons that we learned in applying GenVoca to the avionics domain; we believe that both may be beneficial to others in future modeling efforts.

2 Context and Modeling Objectives

ADAGE embraces a reuse factory paradigm (Figure 1). Instead of moving directly from requirements to a one-of-a-kind avionics software system (a process called *design without reuse*), we recognize that target avionics systems belong to a *family* [Par76] or *domain* of similar software products. By analyzing the avionics domain (a process called *domain analysis*), it is possible to define libraries of primitive components and a factory for assembling these components into target systems. The blueprint for the libraries and factory is called the *reference architecture*. Using design requirements and the factory to produce a target system is called *design with reuse*.

We believe the key to improved avionics software productivity is an integrated environment for exploring, evaluating, and synthesizing different avionics software architectures. As Figure 2 indicates, reference architecture modeling is a rather small part of the scopes of activities of ADAGE. Satisfying strict timing constraints, modeling performance based on various scheduling paradigms (e.g., rate monotonic, earliest deadline, cyclic), estimating execution times accurately, determining aircraft-specific performance tuning constants, presenting different views of an architecture (e.g., functional, data flow, object-oriented), etc. all contribute to the enormous difficulty of avionics software engineering. Nevertheless, reference architecture modeling is critical because the software generator is the centerpiece for integrating performance requirements and analysis tools, software documentation and design rationale tools, etc. The integration of these tools in ADAGE is discussed in [Cog92-93].

We identified four objectives when we began our modeling process:

1. To identify primitive, reusable components of avionics software.
2. To explain how components fit together to form *scalable* avionics systems and subsystems.
3. To explain variations in avionics software as different combinations of components.
4. To outline features of a plausible avionics system generator (i.e., factory).

The GenVoca model [Bat92a] was instrumental in formulating our reference architecture (points #1-#3). This model, which we review in the next section, relies on hierarchical decompositions of avionics software systems into layers that import and export standardized interfaces. The use of standardized interfaces is key to component composibility, component reusability, and software system scalability.

The most critical objective was #4; it reflects the obvious fact that any reference architecture is a theory that postulates how to generate software systems of the target domain. Until the theory is validated through extensive experimentation and implementation, it (the reference architecture) is suspect. Our initial goal was

to make the model plausible to avionics system engineers; we have since verified parts of the model in a series of progressively more complex prototypes.

3 The GenVoca Model

GenVoca is a domain-independent model for defining scalable families of hierarchical systems as compositions of reusable components. GenVoca is the distillation of the designs of two independently-conceived software system generators for the domains of databases and communications protocols [Bat92a]. Other recent software system generators, such as Ficus (distributed file systems [Hei90]), Predator (data structures [Sir93, Bat93b-94]), and Starburst (database systems [Haa90]), can also be recognized as examples of GenVoca organizations. The theoretical foundation for GenVoca has its roots in Parnas' families of systems [Par76], Habermann's FAMOS project [Hab78], and Goguen's model of parameterized programming [Gog86, Tra93]. The features that distinguish GenVoca are realms (or libraries) of plug-compatible components, symmetric components, and type equations.

Components and Realms. A hierarchical software system is defined by a series of progressively more abstract virtual machines. A *component* or layer is an implementation of a virtual machine. The set of components that implement the same virtual machine is called a *realm*. A realm is, in effect, a library of plug-compatible and interchangeable components.

Generally, the membership of a realm can be enumerated. Consider realms **R** and **S**:

```
R = { a, b, c }

S = { d[ R ], e[ R ], f[ R ] }
```

The above notation means that realm **R** has three members, namely components **a**, **b**, and **c**, that are different implementations of the interface (i.e., virtual machine) of **R**. Realm **S** also has three members, each representing a distinct and alternative implementation of the interface of **S**.

Parameters. Components may be parameterized. All components of realm **S**, for example, have a single parameter of realm **R**.[2] Each component of **S**, say **d**, exports the virtual machine interface of **S** and imports virtual machine interface of **R**. Component **d** can be understood as a *transformation* that maps objects and operations of virtual machine **S** to objects and operations of virtual machine **R**. The key idea is that the translation performed by **d** *does not depend on how the interface of* **R** *is implemented*; in effect, component **d** encapsulates a complex mapping between the interfaces of **R** and **S**.

A critical aspect to the composition (and indeed, correct functioning) of components is that they are all designed to work cooperatively without violating their encapsulations. For example, all components of **R** may get much of their information via calls to the virtual machine of **R**, but inherent in component designs is other common knowledge about global variables, etc., and the side effects that their updates might trigger. Much of this information can be made explicit as component parameters; however the key point is that components — whether or not they have parameters — meet system constraints and can exploit knowledge that is known system-wide.

Type Equations, Families of Systems, and Scalability. Software systems are modeled as *type equations*. Consider the following two systems:

```
System_1 = d[ b ];
System_2 = f[ a ];
```

System_1 is a composition of component **d** with **b**; **System_2** composes **f** with **a**. Note that both systems are equations of type **S**. This means that both systems implement the same virtual machine and hence, **System_1** and **System_2** are interchangeable implementations of the interface of **S**.[3]

Realms and their components define a grammar whose sentences (component compositions) are software systems. Just as the set of all sentences defines a language, the set of all component compositions defines a Parnas *family of systems*. Adding a new component to a realm is akin to adding a new rule to a grammar; the family of systems automatically enlarges. Because large families of systems can be built using relatively few components, GenVoca is a *scalable* model of software system construction.

---

2. Parameterizations that we examine in this paper are simple enough to dispense with formal parameter names.

3. Note that composing components can be interpreted as stacking layers in hierarchical software systems. We use the terms component and layer interchangeably, although our use of the term 'layer' is different from its typical ad hoc usage.

Design Rules. In principle, any component of realm **R** can instantiate the parameter of a component of realm **S**. However, there are always certain combinations of components that semantically incorrect, even though their compositions are type correct. Additional domain-specific constraints called *design rules* are needed to preclude illegal component combinations. Attribute grammars appear to be a unifying formalism that can be used to define realms, their components, and design rules [Bat92b, McA93, Bat95].

Symmetry. Just as recursion is fundamental to grammars, recursion in the form of symmetric components is fundamental to GenVoca. Symmetric components have the unusual property that they can be composed in arbitrary ways. More specifically, a component is *symmetric* if it exports the same interface that it imports (i.e., a symmetric component of realm **W** has at least one parameter of type **W**). In the realm below, components **n** and **m** are symmetric whereas **p** is not.

```
W = { n[ W ], m[ W ], p, ... }
```

Because **n** and **m** are symmetric, compositions **n[m[p]]**, **m[n[p]]**, **n[n[p]]** and **m[m[p]]** are possible, the latter two showing that a component can be composed with itself. In general, the order in which components are composed can significantly affect the semantics, performance, and behavior of the resulting system.

4 The ADAGE Reference Architecture

Avionics software systems have been long understood as layered subsystems. A typical layering is shown in Figure 3. The bottom layer is formed by *data source objects (DSOs)*, i.e., sensor subsystems. Examples include subsystems for inertial navigation sensors (**ins**), tactical air navigation (**tacan)**, and doppler navigation sensors (**dns**).

DSO subsystems report their raw sensor values to the *navigation* and *radio navigation* subsystems. Navigation (**nav**) software estimates the aircraft's position relative to earth coordinates; radio navigation (**rnav**) software estimates the aircraft's position relative to a (typically) fixed-position radio beacon. Navigation and radio navigation software operates by integrating diverse DSO estimates into a single estimate, as well as dynamically adapting its calculations according to currently available data sources (i.e., DSOs can fail in flight), filters, gains, and earth and atmospheric models.

The *guidance* subsystem determines the difference between mission objectives and the current aircraft state (e.g., position). It calculates a desired flight profile, estimates errors in heading, speed and/or altitude, and assures smooth transitions between guidance modes. The guidance subsystem selects the guidance modes (i.e., algorithms for computing flight profiles), filters and gains, and specifies mode preconditions such as data quality, capture criteria, and mode conflicts.

The *flight director* subsystem converts guidance errors into pilot control cues or autopilot commands. Its primary function is to develop cues based on errors, aircraft performance models and pilot models. As designed, this subsystem can accommodate fixed-wing or rotary-wing aircraft parameters, varying aircraft flight models and pilot models, and different sets of control laws and gains.

The *controls and display* subsystem presents the interface to pilots. Results computed by any subsystem can be displayed at flight time; pilots use this capability to assess manually the reliability of sensor outputs and their impact on subsequent stages of transformations performed by the navigation, guidance, and flight director subsystems.

The basic unit of data that is exchanged between subsystems is a *state vector*, which contains values that define the state of the aircraft (i.e., position, heading, etc.). At execution time, avionics software emulates a pipeline where state vectors flow upward from data source objects toward the controls and display subsystem, while commands flow downward. Each execution cycle initiates another wave of state vector transmissions and command propagations.

ADAGE addresses a core portion of avionics software, namely the flight director to the DSOs. Our task was to decompose each of these major subsystems into their constituent symmetric and nonsymmetric components and to show how their compositions described existing and envisioned subsystems. We relied on the relative independence of these major subsystems, starting with DSO subsystems, identifying their realms and components, and then progressing to navigation subsystems, and higher level subsystems.

In the following sections, we explain a portion of the model that we have developed. The full model is discussed in [Bat93a]. Here, we concentrate on explaining how specific features and designs of avionics software can be expressed in terms of components and type equations. As mentioned earlier, there are many differ-

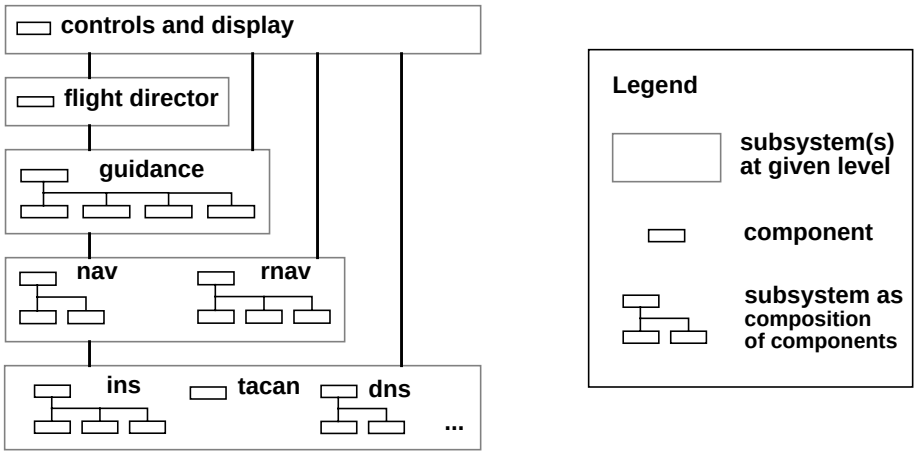

Figure 3. A Layered Architecture for Avionics Software

ent views or models of avionics software; the model that we present here is central to software system generation in ADAGE.

4.1 Data Source Objects

We identified over twenty distinct data source objects, including air data computers (**adc**), doppler navigation subsystems (**dns**), radar altimeters (**radalt**), and global positioning subsystems (**gps**). We rejected the possibility of creating a single virtual machine that was to be exported by all sensors because of the wide array of sensor functionality we were facing. Instead, proposing a standard virtual machine interface for all **adc** sensors, another interface for all **gps** sensors, etc. was much easier and more readily acceptable by avionics engineers [Bri81]. This decision led to a realm of components for each distinct DSO type.

The realm of inertial navigation sensors (**ins**) is a typical example:

```
ins = { // equipment list

        ins_asn_141, ins_asn_143,
        ins_asn_143rlg, ins_asn_90,
        ins_ln_15j, ins_ltn_51,
        ins_skn_2443,

        // simulators, filters, etc.

        ins_simulator, ins_filter[ins],
        ins_select[{ins}],
        ins_average[{ins}], ...
      }
```

An **ins** equipment list is the list of actual inertial navigation sensors that are available from different manufacturers. Each **ins** component is actually a physical sensor and a device driver which translates the standard **ins** interface into device-specific actions. Simulator components can be swapped with equipment components during test phases of system designs. Filter components smooth state vector outputs.

A traditional software design in DSO subsystems is to present a single sensor image to navigation from a collection of redundant sensors. This view is accomplished using the symmetric components **`ins_select`** and **`ins_average`**. Both are parameterized by a set of **ins** components, denoted by parameter **`{ins}`**. The **`ins_select`** operates as a multiplexor by reading the state vectors of its input components and reporting the state vector of the unit that was selected by the pilot. The **`ins_average`** combines its input state vectors to produce a single **ins** state vector that is more reliable than that from any single **ins** sensor.

DSO subsystems are rather flat hierarchies; it is not uncommon for a DSO subsystem to be modeled by a single component. A more complex **ins** subsystem, for example, might have a pair of **`ins`** sensors combined by a selector. Such a system would be specified as an equation of type **ins**:

```
multiple_ins =  ins_select[ {
                    ins_skn_2443,
                    ins_skn_2443 } ];
```

4.2 Navigation Software

Navigation subsystems are highly layered. They are compositions of DSO subsystems and components from five realms: **earth_model**, **earth_geometry**, **atmos**, **inav**, and **nav**. The figure below offers a simple overview the relationships among realm components. Each node represents a realm; an arc from node **A** to node **B** means that components of realm **A** may be parameterized by components from realm **B**. A loop represents the presence of symmetric components. Note that there is a cycle in this graph: **inav** components can be parameterized by **earth_geometry** components and vice versa. We will see later that such cycles are critical to navigation software.

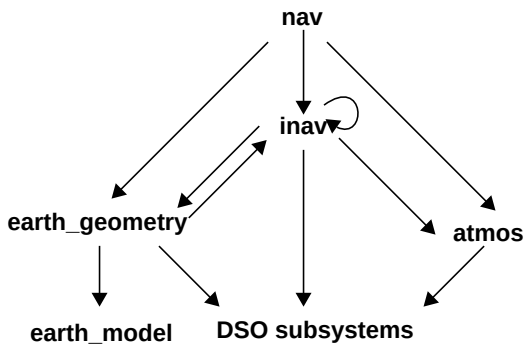

Figure 4. Realm Dependency Graph for Navigation Subsystems

As a brief summary of realm semantics, **inav** components encapsulate navigation modes, filters, and selectors. A navigation subsystem is an equation of type **nav**. The state vector of a **nav** component includes all the values of its input **inav** state vector, plus derived values (possibly computed from **earth_geometry** and **atmos** component state vectors).

An **earth_model** component encapsulates a set of equations that estimate statistics about the earth's geometry. An **earth_geometry** component converts **earth_model** outputs into estimates of the earth's radius, true heading, magnetic heading, etc. An **atmos** component (or atmosphere model) computes dynamic and static air pressure, barometric altitude, true air speed, and calibrated air speed, among other values from air data DSO subsystems.

In the following section, we examine the **inav** realm in detail because **inav** subsystems (equations) contain some of the most interesting and complex features of avionics software.

4.2.1 Internal Navigation Realm

inav is a realm of components whose interface is exposed only to other components within a **nav** subsystem. The software for basic navigation modes can be encapsulated as individual components in the **inav** realm; so too can the software for "combined" modes which transform state vector inputs from multiple DSO subsystems. These are the only nonsymmetric components in **inav**.

The most interesting **inav** components are symmetric. Navigation software typically has multiple navigation mode (components); each mode produces an **inav** state vector that contains estimates of the current position of the aircraft. State vectors from different modes will vary in accuracy because of the DSO sources from which they were derived. Mode control software reads each of these state vectors and outputs a single **inav** state vector which contains the most accurate value for each vector field. Mode control software can be encapsulated as a symmetric component. Current avionics technology imposes a static ranking per field (e.g., **`static_nav_mode_control[{inav}]`**), while more advanced techniques permit dynamic rankings (**`dynamic_nav_mode_control[{inav}]`**).

Estimates of aircraft positions can change abruptly when input sources (chosen by mode control components) change. First or second order washout filters are needed to smooth these transitions; such filters are symmetric (e.g., **`high_pass_filter[inav]`**) because they import (unsmoothed) and export (smoothed) **inav** state vectors and **inav** interfaces.

The partial membership of the **inav** realm is:

```
inav = { // basic navigation modes

        addr_nav_mode [atmosphere_model],
        gps_nav_mode [gps],
        ins_nav_mode [ins],
        trn_nav_mode [ins], ...

        // combined mode components

        gps_ins_nav_mode [gps, ins],
        ins_dns_nav_mode [ins, dns,
                  earth_geometry], ...

        // filters

        high_pass_filter [inav],
        low_pass_filter [inav],
        washout_filter_inav_dns [inav,
                  dns, earth_geometry],
```

```
        // Mode-controllers and selectors

       static_nav_mode_control [{inav}],
       dynamic_nav_mode_control [{inav}],
       inav_select [{inav}], ...
}
```

Aiding is the process by which data from one or more different types of DSOs are combined to produce results that are better than any single sensor [Goo92b]. A important design trade-off that arises in navigation subsystem designs is whether to perform aiding before selection. This trade-off is captured in our model by different compositions of DSO and **inav** components. Figure 5a depicts selection before aiding; Figure 5b shows aiding before selection.

Feedback loops are a fundamental requirement of navigation software. The state vector outputs of DSO subsystems are essentially derivatives of aircraft states (e.g., changes in position, speed, direction, etc.). In order to compute the state of an aircraft at time *i*, the state at time *i-1* must be available. Systems with feedback loops are expressed in GenVoca by *recursive equations*. **inav** subsystems are represented by equations of the form (see Figure 6):

```
s_inav = inav_subsystem[ s_inav ];
```

On each execution cycle, the state vector of the **inav** subsystem is input to an **earth_geometry** component which is internal to that subsystem. This is the feedback loop that occurs in navigation software.

4.3 Other Realms

The basic structure of realms with parameterized components was repeated for radio navigation, guidance, and flight director subsystems [Bat93]. Overall, our model identified 20 realms containing over 350 primitive components. These components allow us to define a vast family of avionics software systems.

In general, type equations provide a very compact way to express and reason about the architectural designs of avionics software systems. We found that a simple avionics system could be expressed in 20 equations that referenced approximately 50 components; more complex systems would easily double the number of equations and components.

5 Lessons Learned

The GenVoca model was a consequence of distilling a common design paradigm from two independently-conceived software system generators. The challenge that we faced was that no methodology had been created for applying GenVoca to new domains. Many of the lessons that we learned and discuss below illuminate key features that should be part of such a methodology.

Look for transformations. As mentioned in Section 3, components are transformations or mappings between virtual machine interfaces. We recognized early that the basic unit of exchange between components was a state vector, and therefore the virtual machine interfaces for all realms should export operations on state vectors, such as **initialize_vector**, **read_vector**, **write_vector**, and **terminate**. Thus, our approach to decomposing the DSO, navigation, guidance, etc. subsystems was to look for primitive transformations on state vectors, such as filtering, combining, etc. These primitive transformations became our components. We knew that whenever input (lower-level) state vectors were different from output (higher-level) state vectors, a new realm and virtual machine had to be created. Furthermore, a component/transformation was parameterized by the state vectors that it read as input.

Distinguishing “primitive” from “nonprimitive” transformations was subjective. We did not look for further decompositions of a transformation when (1) system and software engineers felt that the mapping it performed was a basic unit of computation in avionics systems, and (2) no further decomposition of the transformation was obvious. *Our goal was to make each component match a basic term or processing step that avionics engineers used in describing the computations in avionics software. Thus, type equations would be precise specifications of these verbal descriptions*. Overall, we feel this decomposition approach worked quite well.

Exemplar systems. Reference architecture modeling requires access to information on multiple systems in a domain in order to discern their similarities and differences. In previous modeling efforts, over ten different systems were analyzed in creating a reference architecture [Bat88]. We faced a different situation where we had access to detailed knowledge on only two, but very different, avionics systems: a special operations helicopter and a fixed-wing transport. The lack of exemplar systems was offset by a considerable expertise at Loral Federal Systems Company on avionics software sys-

tems. Thus, having access to a few systems supplemented by domain expertise was sufficient for our modeling effort.

Information gathering. Given the right information, creating a reference architecture isn't difficult. Unfortunately, we were often surprised at how hard it was to obtain (what would seem to be) relatively elementary information. For example, finding the inputs to guidance mode components required us to find the right system engineer(s) to talk to. Quite often, no such person could be found. System engineering expertise tended to be highly focused on a particular project coupled with diffuse information (or rumors) on what was done on other aspects of the same project or on other projects. We often had to rely on educated guesswork as the primary means of deducing the parameterizations of components within certain realms. Overall, collecting information was the most difficult aspect of our modeling effort.

Object-oriented and functional designs. As mentioned earlier, the key to our modeling approach is recognizing the primitive transformations in avionics software. Modeling transformations is really a process of modeling algorithms and how algorithms fit together in a layered manner. This approach is different than object-oriented design methods where objects and their operations are the central focus. Our approach is actually closer to functional decompositions, although noted earlier, a transformation does not deal with just a single function/operation, but an encapsulated suite of interrelated functions (e.g., **initialize_vector**, **read_vector**, etc.). Our method of software decomposition might be characterized as a meld of functional and OO design methods.

Don't look at code. Our domain model concentrated on components and their compositions. There is nothing in our model that addressed the internal construction/organization of a component's software. From our experience in building and validating several reference architectures [Bat88, Sir93, Bat93b-94], we know that the similarity of components within a realm is considerable. Exploiting this similarity is important in creating a software system generator.

We made a brief foray into navigation software to better understand component organizations and to compare the implementation of (what appeared to be) the same component in two different avionics systems. This experiment failed in that we learned nothing on the similarity of components, but learned a lot about their dissimilarity. Navigation components, like most avionics software, are defined by a set of mathematical equations, where each equation has multiple terms. There is general agreement in the avionics community on what these equations are supposed to do — this is the information that we were able to glean at the component level. However, the number of ways of coding the same set of equations is immense. Different terms of the same equation would be given different (variable) names and would appear in completely different places in different systems, making it virtually impossible to verify that the same equations/component were in fact reused. The phrase "individual coding preferences" kept cropping up in explanations as to why source codes for the same component looked so different.

We concluded that, in general, the transformations that we identified as primitive components are not encapsulated in existing systems. Consequently, it is clear why very little code reuse in avionics software has been achieved in the past.[4] Another conclusion is that domain (reference architecture) modeling must be done

4. In contrast, requirements, equations, models and test data are often reused in avionics system development. The modularization that is imposed by our model can make avionics code and designs more reusable.

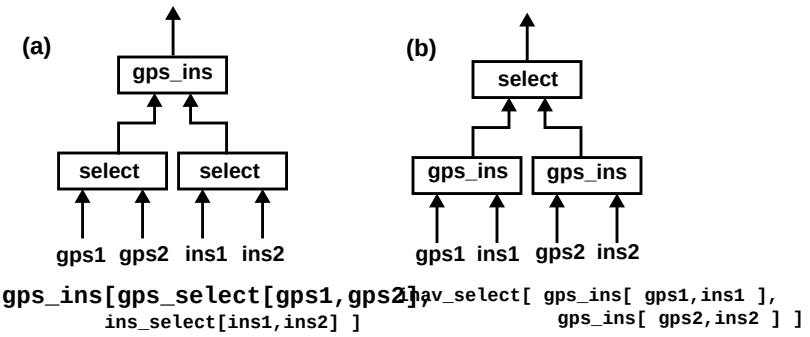

Figure 5. Common Configurations of Selection and Aiding

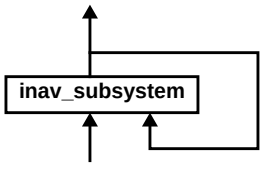

Figure 6. Feedback Loops in Internal Navigation

at the conceptual, not code, level. The critical issue of modeling is identifying reusable abstractions. Studying code is often too detailed to identify such abstractions.

Triad of expertise. Creating a reference architecture requires expertise in software engineering, domain modeling, and domain applications. In previous instances of systems organized by GenVoca concepts, a single person assumed all three roles. This was definitely not the case here. We created a modeling team with different individuals that covered the required triad of expertise.

Normally, a domain modeling expert consults with the domain expert (i.e., a system engineer or engineers) to create a model. Systems engineers are not always software engineers, and thus are not the best candidates to provide the reality checks that are essential in creating a viable model. Thus, a software engineer (who has experience in building software in the domain) plays an important role in keeping both the system engineer and domain modeling expert focussed on the key issues as well as maintaining their lines of communication. One should not assume that system engineers understand (reusable) software and that domain modelers understand the domain (e.g., avionics). We found that a person capable of bridging both worlds was very important.

Interface design. During the initial phases of modeling, it is sufficient to know approximately what operations would define a virtual machine interface (and that system engineers are confident that a standard could be realized). Defining a standardized virtual machine requires building lots of components for that particular realm [Bri81]. Each time a new component is added, the standard may evolve to accommodate its particular features (without exposing implementation detail). Time and experience will eventually lead to a steady-state virtual machine interface for that realm. Predeclaring or hypothesizing interface "standards" prior to extensive implementation and testing is doomed to failure.

Tools are essential. As our approach to decomposing software systems hierarchically is not standard, there were no tools to support and express our model. One of our big wins was writing a simple compiler (or syntax checker), called **`le`** (for *l*ayout *e*ditor), in which realms and their components could be declared and compositions of components could be specified. The syntax that we have used in this paper for declaring realms and systems is the syntax of **`le`**. We found **`le`** clarified discussions and understandings with people who were learning to use the model. It caught the errors that commonly arose component modeling: referencing undefined realms and components, identifying ambiguous or incorrect type expressions, undefined variables, etc. Modeling software systems is an art and any precision that can be interjected into a modeling process is always beneficial. **`le`** is available via anonymous **`ftp`** at **`cs.utexas.edu`** in directory **`/pub/predator`**.

Participation of others. As mentioned earlier, a triad of expertise is needed to create a reference architecture. There is also the obvious and fundamental need for people (other than the domain modeler) to understand the domain model that is being created. The involvement of other system engineers and domain experts provides (a) critical feedback on versions of the model as it develops, and (b) opportunities to apply the model to describe systems that are being built or that might be built. Such interaction exposes misconceptions (both on the domain modeler's part and others) and is generally a good education opportunity for all.

Current software engineering practices. Successful domain modeling requires a global understanding of how software works in a domain. Most individuals who have worked on representative software systems have an understanding of a small corner of the domain and rarely have the insights that are needed to extrapolate accurately beyond their experiences. Engineers will gladly offer conflicting opinions on how things work, thereby contributing to the overall confusion.

It seems that current software engineering practices of partitioning expertise actually hinders if not outright precludes a successful domain modeling effort and, consequently, reduces the probability of realizing a software generator for that domain. It is worth noting that a study of different software projects conducted by MCC revealed that successful projects always had one or two individuals (not necessarily the project leaders) who understood their entire system [Cur88]; these are the individuals who programmers turn to when all other information sources for answering their questions had been exhausted. Projects that did not have such individuals were destined for failure (or were guaranteed to be expensive). Thus, there appears to be something fundamental about understanding and designing software systems and software system families: for a project to succeed, there must be individuals with a global understanding of the system or domain. For exactly the same reason, it comes as no surprise that individuals with global understandings are critical in creating reference architectures and software system generators.

Composition is the key. Reuse libraries should *not* be populated with randomly-harvested components. The difficulties encountered with this approach are legion:

components are typically not interoperable (because they are based on conflicting assumptions), they are not composable because they have ad-hoc interfaces, and they can be customized only with great difficulty (and considerable hacking) [Gar95]. Instead, reuse libraries for generators must have much more restrictive admission requirements. The ability to compose components to build systems quickly and cheaply is the key to creating successful generators. We also believe that the composibility of components is also the key to enhancing their reuse.

It takes time. It is very easy to underestimate the effort to design a reference architecture and to build a generator that validates the model. Four months were originally allocated for the process of creating the reference architecture for ADAGE; it took us over a year-and-a-half to finalize the model. The outlines of the model were actually created quickly, in a few months. However, some investigations resulted in dead-ends (e.g., our "looking at code" experience described above), and other aspects (e.g., learning a new domain, building tools, generalizing existing modeling concepts) also took time. Granted with more experience, the effort needed to develop reference architectures for other domains will certainly shrink. Overall, however, our experiences seem common to most domain modeling/ reference architecture modeling efforts.

In general, a good rule of thumb would be to assume that it is twice as difficult and costly to develop reference architectures and generators than it would be to design and develop a single system. There are two reasons for this. First, it is hard enough to design a single software system. But in addition to all the usual difficulties, designers must also deal with the constraints that the resulting components must be "reusable", i.e., they must be plug-compatible, interoperable, and interchangeable. This often doubles the design time.

Second, one typically has to build many more components than that needed for a single system. In principle, one typically can't demonstrate convincingly the generation of a family of related systems without a fairly well-stocked library of components to begin with; 50% more components than that needed by a single system seems to be a lower limit. The lesson to be learned here is that researchers who wish to follow our lead (or those of others) should be prepared to expend the resources and time to maximize the chances of a successful outcome.

6 Conclusion

In this paper, we presented a building-blocks view of the reference architecture for avionics software generation in ADAGE. We defined the reference architecture in terms of the GenVoca model, a domain-independent model that is aimed at software system generation. We explained how GenVoca concepts were applied to real-time avionics software and important design configurations and alternatives were captured as different compositions of primitive components. We also presented lessons that we learned, in the hope that others will find them useful in future DSSA modeling efforts.

There are other important views of avionics software besides the one we presented in this paper: execution models (or implementation styles), a dataflow/control flow view, composition constraints and requirements/ rationale linkage are other critical features of our reference architecture [Cog92]. Currently, ADAGE is refining formalisms and prototype tools that combine these separate views into executable avionics programs.

In the end, the success of ADAGE depends largely on the ease of specifying and combining components to rapidly create new applications and prototypes. The reference architecture — a high level guide to application development — provides us with a major step toward greatly improving productivity and quality for avionics development.

Acknowledgments. We thank Paul Clements and Vivek Singhal for their valuable comments on earlier drafts of this paper.

7 References

[Ara93] Guillermo Arango. Domain analysis methods. In *Software Reusability*, W. Schafer and R.Prieto-Diaz, editors, Ellis Horwood Publishers, 1993.

[Bat88] D. Batory, "Concepts for a Database System Synthesizer", *Proc. ACM Principles of Database Systems*, 1988.

[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.

[Bat92b] D. Batory and J. Barnett, "DaTE: The Genesis DBMS Software Layout Editor", in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos and R. Zicari, editors, Wiley, 1992.

[Bat93a] D. Batory and S. Shafer, “A Domain Model for Avionics Software”, IBM Owego T.R. ADAGE-UT-93-03, May 1993.

[Bat93b] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *Proceedings of ACM SIGSOFT 1993*.

[Bat94] D. Batory, J. Thomas, and M. Sirkin, “Reengineering a Complex Application Using a Scalable Data Structure Compiler”, *Proceedings of ACM SIGSOFT 1994*.

[Bat95] D. Batory and B. Geraci, “Validating Component Compositions in Software System Generators”, Dept. Computer Sciences, University of Texas at Austin, TR-95-03.

[Bri81] K.H. Britton, R.A. Parker, and D.L. Parnas, “A Procedure for Designing Abstract Interfaces for Device Interface Modules”, *Proceedings of ICSE* 1981.

[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.

[Cur88] B. Curtis, H. Krasner, and N. Iscoe, “A Field Study of the Software Design Process for Large Systems”, *Communications of the ACM*, November 1988.

[Gar95] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural Mismatch or Why it’s Hard to Build Systems out of Existing Parts”, *International Conference on Software Engineering*, Seattle, Washington, 1995.

[Gog86] J. Goguen, “Reusing and Interconnecting Software Components”, *IEEE Computer*, February 1986.

[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.

[Haa90] L. Haas, et al., “Starburst Mid-Flight: As the Dust Clears”, *IEEE Trans. on Knowledge and Data Engineering*, March 1990.

[Hab78] A. Habermann, “Modularization and Hierarchy in a Family of Operating Systems”, Carnegie Mellon University Tech. Report CS-78-101, February 1978.

[Hei90] J. Heidemann and G. Popek, “An Extensible, Stackable Method of File System Development”, TR CSD-910007, UCLA, December 1990.

[Hut91] N. Hutchinson and L. Peterson, “The x-kernel: an Architecture for Implementing Network Protocols”, *IEEE Trans. Software Engineering*, January 1991.

[Luc85] D. Luckham and F.von Henke, “An Overview of ANNA, a Specification Language for ADA”, *IEEE Software*, March 1985.

[McA93] D. McAllester, “DSSA-ADAGE Avionics/Architecture Knowledge Representation Language (Draft Report), ADAGE-MIT-93-01, 1993.

[Par76] D.L. Parnas, “On the Design and Development of Program Families”, *IEEE Trans. Software Engineering*, March 1976.

[Tra93] W. Tracz, “LILEANNA: A Parameterized Programming Language”, *Proc. 2nd International Workshop on Software Reuse*, March 1993.

[Sir93] M. Sirkin, D. Batory, and V. Singhal, “Software Components in a Data Structure Precompiler”, *Proc. 15th International Conf. on Software Engineering*, May 1993.

[SEI90] Software Engineering Institute, *Proc. Workshop on Domain-Specific Software Architectures*, Hidden-Valley, Pennsylvania, 1990.

[War92] S. Wartik and R. Prieto-Diaz, “Criteria for Comparing Reuse-Oriented Domain Analysis Approaches”, *International Journal of Software Engineering and Knowledge Engineering*, September 1992.