To appear in the AIAA Computing in Aerospace 10 Conference, San Antonio, 1995.

The ADAGE Avionics Reference Architecture[1]

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

Lou Coglianese, Steve Shafer, and Will Tracz
Loral Federal Systems
Owego, New York 13827
{ lou, reve, tracz }@lfs.loral.com

Abstract

ADAGE is a project to define and build a domain-specific software architecture (DSSA) environment for avionics. A central concept of ADAGE is the use of generators to implement scalable, component-based models of avionics software. In this paper, we review the ADAGE model (or reference architecture) of avionics software and describe techniques for avionics software synthesis.

Keywords: software reuse, software scalability, avionics domain modeling, domain-specific software architectures, GenVoca, LILEANNA.

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 was 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, an analysis of existing avionics software can identify components and constraints inherent in the avionics domain. A product of domain analysis, called a *reference architecture*, is a model or blueprint for an avionics software system generator.

A central task in creating a DSSA is the definition of a reference architecture. Domain analysis techniques are still immature [Ara93]; 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 that are chosen to represent a reference architecture, they must be domain-independent; the applicability of a DSSA methodology across multiple domains is an essential requirement.

We developed the ADAGE reference architecture in terms of the GenVoca model [Bat92]. This model is suited for software system generation; a software domain is expressed in terms of standardized sets of parameterized, plug-compatible, and reusable layers called *components*. A composition of components defines a software system or subsystem of the domain. A key advantage of GenVoca models is *scalability*: a small number of GenVoca components can be combined in different ways to describe vast families of distinct software systems.

In this paper, we review the ADAGE reference architecture and explain how software for data source object and navigation subsystems can be synthesized from prefabricated components. We also examine tools in the ADAGE environment that support our approach to component-based avionics 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.

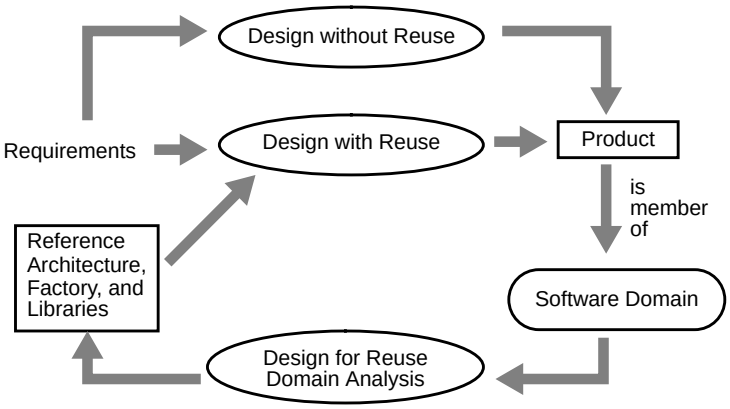

Figure 1. A Software Factory Paradigm

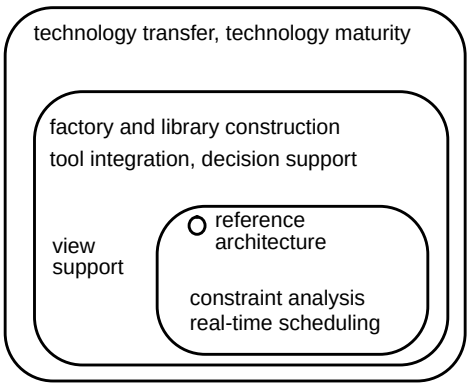

Figure 2. ADAGE Scopes of Activities

2 Context and Objectives

ADAGE embraces a reuse factory paradigm (see 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* or *domain* of similar software products [Par76]. 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. [Cog92-93].

3 The Reference Architecture

Avionics software has long been understood in terms of layered subsystems (see Figure 3). The bottom-most subsystems are *data source objects* (DSOs), i.e., sensor subsystems. 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 GenVoca model of software construction postulates that complex subsystems, such as avionics subsystems, can be explained as compositions of primitive, plug-compatible, and interchangeable components. For this to be possible, it is necessary to define standardized programming interfaces to fundamental domain abstractions. Standardized interfaces to DSO, navigation, radio navigation, guidance, and flight director software can indeed be devised.

An important consequence of interface standardization is subsystem plug-compatibility and interchangeability. Another consequence is that there is an enormous number of subsystems that share the same interface. Navigation subsystems, for example, can vary greatly in terms of the navigation modes offered, the means by which aircraft position values are filtered, combined, etc. The ability to describe vast families of different subsystem implementations compactly is critical for a

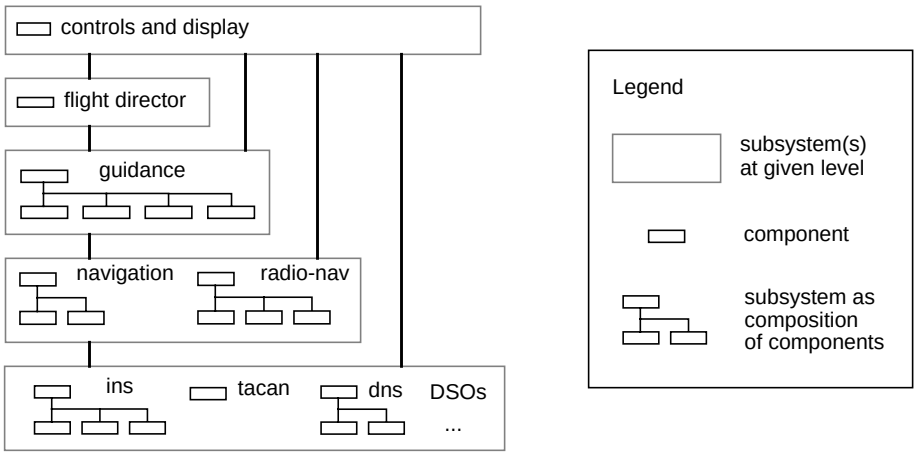

Figure 3. A Layered Architecture for Avionics Software

practical model of avionics software generation. The GenVoca approach decomposes subsystems into primitive components that export and import standardized interfaces. The combination of components that defines a system or subsystem is called a *type equation*. By combining (i.e., layering) components in different ways, vast families of subsystems (type equations) can be defined. We illustrate these ideas in the following paragraphs.

GenVoca organizes components that export the same interface into a *realm* (i.e., a library of plug-compatible and interchangeable components). As an example, consider the realm **`ins`** of inertial navigation sensors. The components of this realm can be enumerated, as shown below:

```
ins = { // equipment list
        ins_asn_141, ins_asn_143,
        ins_skn_2443,
        ...

        // simulators and filters
        ins_simulator,
        ins_filter[x:ins],
        ...

        // selectors and averagers
        ins_select[x:{ins}],
        ins_ave[x:{ins}],
        ...
      }
```

There are components that encapsulate different inertial navigation sensors (**`ins_asn_141`**, **`ins_asn_143`**, **`ins_skn_2443`**), components that encapsulate inertial navigation simulators and filters (**`ins_simulator`**, **`ins_filter[x:ins]`**), and components that encapsulate multiplexors (**`ins_select[x:{ins}]`**) and ways of combining outputs of different INS sensors into a single estimate (**`ins_ave[x:{ins}]`**). All of these components implement the standard interface for inertial navigation systems (denoted by the realm name "**`ins`**"). This means that all **`ins`** components are plug-compatible and interchangeable.

Combinations of **`ins`** components define INS subsystems. An INS subsystem that has a pair of INS sensors combined by a selector would be specified as an equation of type **`ins`**:

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

The way to understand this equation is that both INS sensor components submit their readings to the INS selector; the selector, in turn, outputs only one of these readings. (The reading that is output is chosen dynamically at run-time by the pilot). In general, a component operates as a transducer: its output is generated from the data that it receives from its inputs. In the case of avionics software, subsystem executions tend to be bottom-up, i.e., the lowest-level components in a subsystem execute first; the top-most components execute last.

The **`ins`** realm illustrates two basic characteristics of GenVoca components. First, components are parameterized. The **`ins_filter[x:ins]`** component, for example, has a single parameter **`x`** of type **`ins`**. What this means is that **`ins_filter`** filters the state vector output of a subsystem **`x`** of type **`ins`**. Second, compo-

nents are *symmetric* if they export the same interface that they import. The **ins_filter[x:ins]** component is symmetric: it exports the **ins** interface (because it belongs to realm **ins**) and imports the **ins** interface (because it has a parameter of type **ins**). Symmetric components are unusual in that they can be composed in arbitrary ways. Consider the following equations that define two distinct INS subsystems. Both are similar in that they have pairs of INS sensors that are combined by a component that averages their outputs; they differ only in the placement of a filtering component:

```
subsystem1 = ins_filter[
         ins_ave[ { ins_skn_2443,
                    ins_skn_2443 } ] ];

subsystem2 = ins_ave[ {
         ins_filter[ ins_skn_2443 ],
         ins_filter[ ins_skn_2443] }];
```

subsytem1 filters the average of the outputs of two INS sensors, whereas **subsystem2** filters the output of INS sensors directly before averaging them. Although both subsystems are composed from exactly the same components, their outputs can be quite different. This example illustrates the scalability of GenVoca models: a small number of components can be composed in many ways to describe large families of implementations.

Realms for other DSOs, such as global positioning sensors (**gps**) and VHF omnirange sensors (**vor**), are described similarly. There are over 20 distinct DSO realms in the ADAGE reference architecture containing over 100 distinct components.

Additional realms are needed to define navigation subsystems. The internal navigation (**inav**) realm contains the most diverse set of components. It includes components that encapsulate primitive navigation modes (i.e., modes that examine the output of a single DSO subsystem), combined navigation modes (i.e., modes that combine the outputs of multiple DSO subsystems), filters and mode-control.[2] Equations of type **inav** define navigation subsystems.

```
inav = { // primitive navigation modes
         ins_nav_mode[x:ins],
         gps_nav_mode[x:gps],
           ...

         // combined navigation modes
         gps_ins_nav_mode[x:gps,y:ins],
         ins_dns_nav_mode[x:ins,y:dns,
                     z:earth_geometry]
         ...

         // filters
         high_pass_filter[x:inav],
         washout_filter_dns[x:inav,
               y:dns, z:earth_geometry],
          ...

          // mode control components
          static_mode_ranking[x:{inav}],
          dynamic_mode_ranking[x:{inav}],
           ...
       }
```

Overall, the ADAGE reference architecture defines over 40 different realms with a total of over 350 distinct components. 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. Further discussions and experiences with the ADAGE domain model are discussed in [Bat94a].

There are two other topics worth mentioning about GenVoca reference architectures. First, our model of component parameters is actually more complicated than indicated above. Until now, the only parameters of components that we have discussed have had realm types. In reality, components can have many other non-realm-type parameters. Called *configuration parameters*, these parameters include aircraft-class-specific and aircraft-specific tuning constants, performance constraints, functions, and data types. Equations where both realm parameters and configuration parameters are defined for every component are called *GenVoca Well-Formed Expressions* (GWFEs). (The type equation of a GWFE is formed by removing configuration parameters from components of the GWFE.) The grammar for a GWFE is given in Figure 4.

The second topic arises from the observation that not all type-correct equations are semantically meaningful. Some components work correctly only in the presence (or absence) of other components. Domain-specific constraints, called *design rules*, are needed to identify (and thus preclude) illegal component combinations. A GenVoca reference architecture is defined by both the realms of components and the constraints (design rules) of component compositions. *Design rule checking* of

2. **dns** denotes the realm of doppler navigation system components, **gps** denotes the realm of global positioning sensor components, and **earth_geometry** denotes the realm of components that compute statistics about the earth's geometry.

```
GWFEs : GWFE ';'
      | GWFEs GWFE ';'
      ;

GWFE  : IDENTIFIER '=' ModuleInstantiation
      ;

ModuleInstantiation : IDENTIFIER
      | IDENTIFIER '(' ConfigParams ')'
      | IDENTIFIER '[' ModuleParams ']'
      | IDENTIFIER '[' ModuleParams ']' '(' ConfigParams ')'
      ;

ModuleParams : ModuleInstantiation
      | '{' ModuleParams '}'
      | ModuleParams ',' ModuleParams
      ;

ConfigParams : Parameter
      | ConfigParams ',' Parameter
      ;

Parameter : IDENTIFIER '=' IDENTIFIER
      | IDENTIFIER '=' REAL_LITERAL
      | IDENTIFIER '=' STRING_LITERAL
      ;
```

Figure 4. Grammar for GenVoca Well-Formed Expressions

type equations and GWFEs is discussed in [McA93, Bat95].

4 The ADAGE Environment

ADAGE is an integrated environment for avionics software system specification and synthesis. The elements of ADAGE that are relevant to our discussions are depicted in Figure 5. ADAGE users define avionics software systems using GLUE (Graphical Layout User Environment) [Hig94]. GLUE graphically enables users to select components from realms, to define type equations by instantiating component parameters, to specify component configuration parameters, to record justifications for design choices, and to store, retrieve, and edit previously defined equations. GLUE has two additional responsibilities. First, GLUE performs design rule checking to make sure that user-defined systems do not violate preconditions of components or global constraints imposed on the target system [McA93]. Second, when software generation is to occur, GLUE outputs a transcription of a specification in the form of GWFEs.

LILEANNA (Library Interconnection Language Extended by Annotated Ada) and MEGEN (Module Expression GENerator) are tools used by ADAGE to generate avionics software in Ada. LILEANNA is a general-purpose language for composing, customizing, and generating Ada software [Tra93a]. It has the capabilities of editing and composing prewritten Ada packages automatically by following a script of high-level source-code modification statements, called *module expressions*. Module expressions are used by ADAGE to customize exemplar software, such as a universal device driver, to produce drivers for specific devices.

LILEANNA receives its module expression scripts from MEGEN, a tool that understands avionics software and GWFEs [Tra93b]. From the GWFEs output from GLUE, MEGEN generates module expressions that (1) instantiate the Ada package of every component referenced in the GWFEs, and (2) customize each

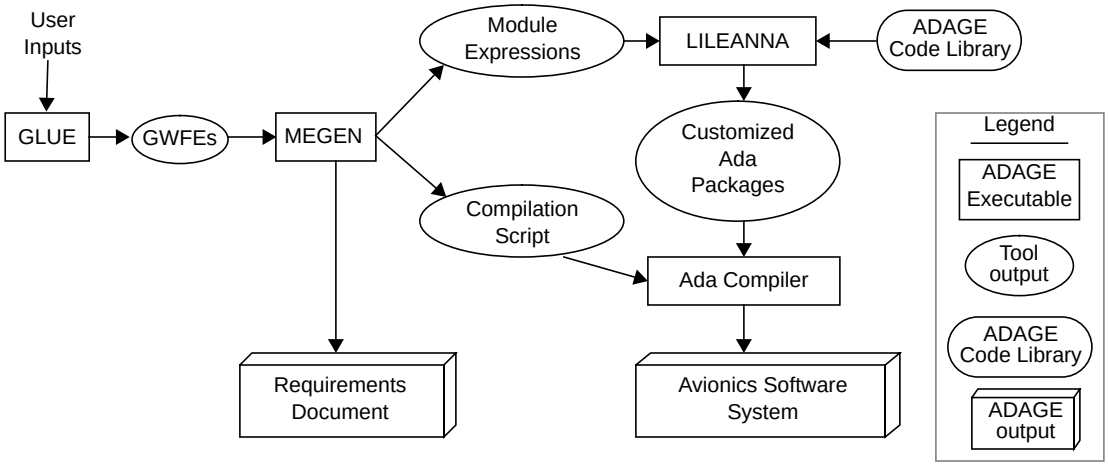

Figure 5. ADAGE Tools for Avionics Software System Synthesis

instantiated package according to the given configuration parameters and the particular call structure that encodes the component layering of the GWFEs. This produces the source modules for the target avionics system. MEGEN also generates a compilation script of these source modules for the Ada compiler in order for it to produce a target system executable. Another capability of MEGEN is to produce a high-level requirements document from the input GWFEs. This document details the configuration choices made by the user.[3]

In the following sections, we review the design of Ada packages and explain further the capabilities of LILEANNA and MEGEN to illustrate their use in avionics software generation.

4.1 LILEANNA

LIL (Library Interconnect Language) is a general-purpose module interconnection language for designing, structuring, composing, and generating software systems in the Ada programming language [Gog83]. LILEANNA is an implementation of LIL that combines the power of ANNA (Annotated Ada) [Luc85] and is a language for formally specifying and generating Ada packages [Tra93a].

LILEANNA supports parameterized programming in Ada by introducing two entities, theories and views, and enhancing a third, package specifications. A *LILEANNA package*, with semantics specified either formally or informally, represents a template for generating a family of Ada packages. A *theory* is a higher-level abstraction that describes the syntax and semantic interface of a package and the properties of its parameters. A *view* is a mapping between types, operations, and exceptions. Views are used to instantiate parameterized (generic) packages. LILEANNA goes beyond the Ada instantiation capability in that generic packages can be composed to create new parameterized packages without themselves being instantiated. Partial instantiations are also possible.

Two different forms of parameterization are distinguished: horizontal and vertical. Vertical parameterization corresponds to the notion of layering (component composition) in GenVoca, and is supported in LILEANNA by the **`needs`** statement, which expresses import dependencies. Horizontal parameterization corresponds to the customization of software via instantiation of configuration parameters in GenVoca. LILEANNA supports horizontal parameterization by the **`import`**, **`protect`**, and **`extend`** statements - three forms of inheritance[4] - and the **`include`** statement, a subtyping construct [Tra93a].

The power of programming in LILEANNA centers on its ability to generate new instances of LILEANNA packages and to compose them. Existing packages can be combined (by merging their operations and types); types, operations, and/or exceptions can be added, exchanged, or removed; operations and types can be renamed; and so on. The following module expressions (i.e., **`make`** statements) illustrate some of the capabilities of LILEANNA:

- A new package can be a renamed copy of an existing package:

```
make <new package name> is
     <old package name>
end;
```

- Add an import clause to an existing package:

```
make <new package name> is
     <old package name>
     *(add with <package name>)
end;
```

- Replace the import clause of an existing package:

```
make <new package name> is
     <old package name>
     needs ( <old with name> =>
             <new with name> )
end;
```

- Add another value to an enumerated type:

```
make <new package name> is
     <old package name>
     *(add literal to <type>
         (<literal>))
end;
```

- Add a procedure invocation as the last statement of a given subprogram:

3. The current version of MEGEN only generates a LaTeX file, however future versions could be integrated with a skeleton Systems/Software Requirements Specification (SRS) document.

4. Since there is no inheritance in versions of Ada prior to Ada 9X, compositions that use inheritance need either to import all modules in the inheritance hierarchy (being careful to rename those that might result in ambiguity) or to include all necessary functionality directly in the implementation (i.e., the package body).

```
make <new package name> is
     <old package name>
     *(add call to <subprogram name>
         (<procedure invocation>))
end;
```

In general, the result of evaluating a LILEANNA **make** statement is an executable Ada package specification and body. We will later show an example that illustrates how the above statements are used by MEGEN.

4.2 Ada Package Design

Generated avionics code arises from three different sources: realm code, glue code, and support code. Each is described below.

Ada packages that define components of ADAGE realms are called *realm code*. Generally, a package for each individual ADAGE component is hand-written and is entered into the ADAGE code library. However, not all components are represented in this manner.

As an example, ADAGE presently contains a package for only one sensor driver. When more than one sensor is to be used in an avionics system, LILEANNA must create the Ada packages for these other sensors. This is done through the use of *exemplars*. Exemplars are LILEANNA packages that are transformed into Ada packages. Exemplars are used both as part of a system under construction and as a foundation from which other packages will be built. In the above example, the lone sensor driver defines an exemplar "universal device driver". LILEANNA is instructed by MEGEN to copy this package, and then to modify certain statements to produce a package that is specific for a given sensor.

Instantiating the packages for each component of a type equation does not produce all the code that is needed for a complete and compilable system. This is where the other two categories of code come into play. *Glue code* can be thought of as the code that binds components together; it introduces the call bindings that realize the specific component layering of a type equation. The current method of introducing bindings is through the LILEANNA **make** statements that add procedure invocations to subprograms. MEGEN generates these statements. As in the case of exemplar device driver packages, MEGEN has deep knowledge about the semantics of components and how their packages have been coded. To keep such knowledge to a minimum, realm code packages are structured to conform to precise coding and organization standards, so that adding procedure invocations as the last statement of subprograms (via a LILEANNA **make**), for example, will correctly bind packages together.[5]

Support code is code that virtually all avionics software systems will share. This includes standard type definition packages, global data (abstract data type) packages, some reuse code, and so on. For the most part, support code does not need to be modified by LILEANNA, except perhaps to change a constant value in a package specification.

4.3 MEGEN

As is evident from the previous sections, MEGEN (Module Expression GENerator) serves a key role in generating avionics software [Tra93b]. Unlike LILEANNA, MEGEN alone contains knowledge of the avionics domain (e.g., realms of components) and code-level details of Ada packages that implement ADAGE components. Given the GWFEs of a target system, MEGEN generates command scripts (module expressions) that LILEANNA uses to produce actual Ada source.

MEGEN supports three composition and integration constructs: (1) direct selection of required Ada packages (with no customization or configuration), (2) creation of new Ada packages from exemplar packages with specific parametric configurations, and (3) integration of selected, configured and/or created packages with other packages through importation of scope.

In general, producing a script of module expressions from a GWFE can be quite involved. As a flavor of the scripts output by MEGEN, consider the simple problem where a GWFE of an avionics system references a GPS sensor. There is no package for a GPS sensor in the ADAGE code library. MEGEN instructs LILEANNA to perform transformations on realm code in order to produce a package for this sensor and to integrate this

5. In future versions of ADAGE, glue code will also determine the execution model of the target avionics system. That is, the target system might follow a tasking model, or an executive model, etc. depending on the glue code used. The appropriate set of glue code directives to include in the system is determined by the coding style annotation in the GWFE that is output from GLUE.

package into the target source code. (Additions of other sensors would be analogous to this example.)

Specifically, three transformations are needed: (1) create the new sensor package, (2) add the new sensor to the enumerated type in support code package **ConfiguredSensors**, and (3) add with statements and call statements to the new sensor in support code package **SampleSensors**, which is responsible for periodically polling the sensors of the target system.

The first step is to create the new sensor package. This involves copying an exemplar sensor package (**INS1**) and modifying its import clause to import the **GPS_Sensor** package. MEGEN generates the following LILEANNA **make** statement to accomplish this:

```
make GPS1_Sensor is INS1_Sensor
     needs (INS_Sensor => GPS_Sensor)
end;
```

The second step is to modify an enumerated type. The ADAGE support code package **ConfiguredSensors** maintains an enumerated type (called **List**) that defines the identifiers of sensors that are supported in a target system. By convention, MEGEN generates the name for this value by concatenating the name of the device ("**GPS**") with an internally generated device number ("**1**"). (In this way, replicated devices are given distinct names - "**GPS1**", "**GPS2**", etc.) MEGEN generates the following **make** statement to perform the second step:

```
make ConfiguredSensors is
 ConfiguredSensors
     *( add literal to List ( GPS1 ) )
end;
```

The third step is to integrate the package created in Step 1 into the target system source code. The support code package **SampleSensors** periodically polls the sensors of a target system. This package must be modified so that it includes a call to the package created in Step 1. MEGEN generates the following **make** statement to accomplish this:

```
make SampleSensors is SampleSensors
     *( add with GPS1_Sensor )
     *( add call to SampleSensors
           (GPS1_Sensor.SampleState)
end;
```

This simple example points out valuable lessons that we have learned in building ADAGE. Readers may have noted that the above **make** statements perform elementary transformations on existing source code. These same transformations could have been done by hand using an editor. LILEANNA offers significant advantages over hand-editing: the transformations are done automatically, quickly, and correctly. Such transformations are tedious, slow, and error-prone if done by hand. LILEANNA performs transformations directly on the syntax trees of Ada source. Thus, it has access to additional contextual information to verify the correctness of transformations automatically.

Another lesson that we have learned is that a small subset of program transformations (**make** statements) is adequate to compose and configure complex systems correctly and efficiently. This was possible because of the (considerable) investment we made in the design of ADAGE components. Not only is it important for ADAGE components to implement the standardized abstractions of their realms, but it is also important that the code of packages that implement components must be "regular" so that tools like MEGEN can transform them easily. We believe these are the keys for achieving reuse and generation of avionics software.

5 Conclusions

ADAGE is an experiment in software architectures and software system generation. It is indeed possible for domains of well-understood software systems to be standardized in order to create libraries (realms) of reusable software components. Generators exploit the regularities of standardized decompositions/descriptions of systems by assembling these systems from prewritten components. The GenVoca model has been successfully applied to the domains of database systems, communication protocols, data structures, and distributed file systems [Bat94b]. We found that GenVoca was also well suited for defining reference architectures and generators of avionics software, thereby reinforcing the belief that its concepts are truly domain independent. We did find that extensions to GenVoca were needed (e.g., to capture configuration parameters) in order to address adequately the complexities of avionics system synthesis.

We learned similar lessons about LILEANNA. LILEANNA was designed to be a general-purpose module composition language that offered a basic set of program transformations (module expressions/**make** statements) as its means for reusing and customizing software. It was gratifying that using LILEANNA to generate avionics software required adding only a few make statements to the original set. This is an indica-

tion of LILEANNA's utility for generating software in other domains.

The development and experimentation of ADAGE is ongoing. As more production code is generated, the system's user interface and generation capabilities are being refined and extended. In addition, we plan to demonstrate the GenVoca model on new domains and investigate LILEANNA transformations on other programming languages (i.e., FORTRAN and C++).

Acknowledgments. We thank Dinesh Das for his clarifying insights on drafts of this paper.

6 References

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

[Bat92] 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.

[Bat94a] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", submitted for publication, 1994.

[Bat94b] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca Model of Software-System Generators", *IEEE Software*, September 1994.

[Bat95] D. Batory and B. Geraci, "Validating Component Compositions in Software System Generators", in preparation, 1995.

[Cog92] Lou Coglianese, et al., "An Avionics Domain-Specific Software Architecture," *ARPA PI Conference*, 1992. Also in *CrossTalk*, October 1992, and Loral Federal Systems Owego TR. 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, Loral Federal Systems Owego TR. ADAGE-IBM-93-04, May 1993.

[Gog83] J. Goguen and K. Levitt, editors, *Report on Program Libraries Workshop*, SRI International, Menlo Park, California, 1983.

[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.

[Hig94] J. Higgins, "ADAGE Layout Editor (LE) User's Manual", Loral Federal Systems Owego, TR. ADAGE-LOR-94-04, May 1994.

[Luc85] D.C. Luckham and F.W. Von Henke, "An Overview of ANNA: A Specification Langauge for Ada", *IEEE Software*, March 1995.

[McA93] D. McAllester, "DSSA-ADAGE Avionics/ Architecture Knowledge Representation Language", Loral Federal Systems Owego TR. ADAGE-MIT-91-01.

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

[Tra93a] W. Tracz, "LILEANNA: A Parameterized Programming Language", *Proc. 2nd International Workshop on Software Reuse*, March 1993.

[Tra93b] W. Tracz and L. Coglianese, "An Adapatable Software Architecture for Integrated Avionics", in *Proceedings of NAECON*, 1993.

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