Exploiting Prescriptive Aspects: A Design Time Capability*

John A. Stankovic Prashant Nagaraddi Zhendong Yu Zhimin He

Department of Computer Science
University of Virginia
Charlottesville, VA 22904

{stankovic, pnn7f, zy2x, zh5f}@cs.virginia.edu

Brian Ellis

Boeing
8901 Airport Road
Berkeley, MO 63134

brian.j.ellis@boeing.com

ABSTRACT

Aspect oriented programming (AOP), when used well, has many advantages. Aspects are however, programming-time constructs, i.e., they relate to source code. Previously, we developed a tool called VEST that extended aspects to design time for embedded systems. Two types of design time aspects were identified which we labeled aspect checks and prescriptive aspects. In the original VEST tool several keys aspect checks and a simple form of prescriptive aspects were implemented. Prescriptive aspects are extremely powerful and result in many design time advantages and uses. This paper enhances and exploits the concept of prescriptive aspects well beyond its original purpose and results. A new prescriptive language is developed and implemented in the VEST tool. We also use prescriptive aspects in a case study for an avionics application and evaluate its benefits. The result is a tool with significant and new features for building distributed real-time embedded systems. It is shown in the case study that design time is shortened by 69%.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages.

Keywords

Aspects, prescriptive aspects, component-based design.

EMSOFT'04, September 27–29, 2004, Pisa, Italy.

1. INTRODUCTION

Aspects [12] are defined as those issues that cannot be cleanly encapsulated in a generalized procedure. For example, in a real-time embedded system, changing the code in one component may affect the overall end-to-end response time of an application task. Aspects, as defined in the literature, are at the programming language level. For example, AspectJ [12] provides syntax that permits the specification of aspects and a weaver that weaves the code specified in the aspect into the base Java code. In our work, as embodied in the VEST tool, we apply the concept of aspects as support for crosscutting dependencies at *design* time. This results in *language independent* aspects with many benefits. We have discovered that there are, at least, two types of language independent aspects. The first type we call *aspect checks*. Aspect checks address specific crosscutting dependencies, which are sometimes hidden from designers or are difficult to assess. For example, end-to-end real-time scheduling is one type of aspect check. The second type of aspect we call *prescriptive aspects*. In prescriptive aspects, a general set of advice is written and applied to the entire design. Note that this advice is applied to the design, not the source code. The application of this advice changes the reflective information associated with the affected components and their interactions. Prescriptive aspects, if deemed general enough, can be retained in a prescriptive aspect library for use in other similar projects. Compared with aspect oriented languages, language independent aspects reduce errors in the early stages of software design.

This paper briefly presents an overview of VEST [26] to set the context of this work (Section 2). Section 3 presents the significant benefits of prescriptive aspects in two major areas: for system design modifications and when used for expert advice. Section 4 describes the new prescriptive aspect language (VPAL) that was implemented in the VEST tool. Evaluations of the key benefits of prescriptive aspects are performed on an avionics case study (Section 5). The results show the value of prescriptive aspects both qualitatively and quantitatively. Section 6 presents the state of art including comparisons to component-based composition tools, aspect-based tools and AspectJ. Section 7 summarizes the main results.

* This work was supported, in part, by the DARPA PCES program under grant F33615-00-C-3048.

2. OVERVIEW OF VEST

Building distributed embedded system software is time-consuming and costly. The use of software components for constructing and tailoring these systems has promise. What are needed are tools to support program *composition* and *analysis* of component-based embedded systems. In these systems designs are instantiated largely by choosing pre-written components from libraries rather than by implementing the design from scratch. One major difficulty of embedded system composition is the crosscutting dependencies among components that are often hidden from the composers. Composition tools should support dependency checks across components boundaries and expose potential composition errors due to the crosscutting dependencies.

VEST provides an environment for constructing and analyzing component-based distributed real-time embedded systems. VEST helps designers select or create passive software components, compose them into a product, map the passive components onto active structures such as threads, map threads onto specific hardware, and perform dependency checks and non-functional analyses to offer as many guarantees as possible along many dimensions including real-time performance. Distributed embedded systems issues are explicitly addressed via the mapping of components to active threads and to hardware, the ability to include middleware as components, and the specification of a network and distributed nodes.

The VEST environment is composed of various domain specific component libraries, a prescriptive aspect language and library, an extensible set of aspect checks, and a GUI-based environment (shown in Figure 1) for composing and analyzing embedded products. VEST has been fully implemented and delivered to the Boeing corporation for further test and evaluation.

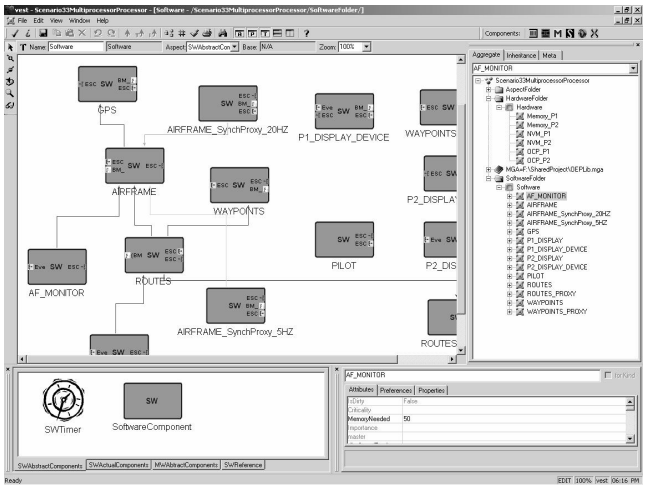

Figure 1

2.1.1 Component Libraries

Because VEST supports real-time distributed embedded systems, the VEST component libraries contain both software and descriptions of hardware components and networks. Sets of reflective information exist for each of these component types. The reflective information of a component includes its interface, requirements such as security, linking information, location of source code, worst-case execution time, memory footprint, and other reflective information needed to analyze crosscutting dependencies. The extent of the reflective information is one of the key features that distinguish VEST from other tools. To support the whole design process of embedded systems, VEST implements four domain specific component libraries: the application library, middleware library, OS library and a hardware library.

2.1.2 Prescriptive Aspects Language and Library

Prescriptive aspects (written in VPAL) are reusable programming language independent advice that may be applied to a design. For example, a designer can invoke a set of prescriptive aspects in the library to add a certain security mechanism *en masse* to an avionics product.

2.1.3 Aspect Checks

VEST implements both a set of simple intra- and inter-component aspect checks that crosscut component boundaries. A designer can apply these checks to a system design to discover errors caused by dependencies among components. One aspect check in VEST is the real-time schedulability analysis for both single-node and distributed embedded systems. See [26] for other aspect checks.

2.1.4 GUI Composition Environment

VEST provides a GUI-based environment that lets designers compose distributed embedded systems from components, perform dependency checks, and invoke prescriptive aspects on a design. For more details on VEST and its GUI which is based on GME [14], see [26].

3. PRESCRIPTIVE ASPECTS

The initial idea for prescriptive aspects was presented in [26]. However, only a brief description was given and limited evaluation of the concept was presented. This paper expands the prescriptive aspect concept, discusses important implications of prescriptive aspects, presents a new VPAL language, and evaluates prescriptive aspects on an avionics case study.

Prescriptive aspects have two major roles: as a system design modification tool, and as an application of expert advice obtained on previous domain specific implementations. In this section we consider each of these in turn. We then discuss the concept of hierarchies of prescriptive aspects which are useful for both types of prescriptive aspects.

3.1 System Design Modifications

Prescriptive aspects are *advice* that may be applied to a basic functional design. This encourages a designer to design in a functional manner and then consider the non-functional aspects. This separation of concerns makes design easier. For example, a designer might create the functional modules for navigation of an aircraft and then apply advice to support real-time performance and security. Overall, prescriptive aspects support a widespread global change in the design in a complete and consistent manner by simply defining new advice or using pre-declared advice and applying it to your design. This prevents bugs where (without this support) the changes required are only made in some of the requisite places. Also implied by this advantage is that re-applying different advice can be done simply and aspect checks and

schedulability analysis can be re-run automatically. This facilitates looking at multiple competing design options, thereby resulting in more effective final designs.

To change the system design, prescriptive aspects can adjust properties in the reflective information (e.g., change the priorities of a task or the replication levels of a software component). It can also add/delete components or interactions between components. When the properties of a component are changed, the associated code of this component is marked as inconsistent until it is changed to match the design.

To better understand the qualitative benefits of prescriptive aspects consider the following examples which are easy to implement with prescriptive aspects. After designing the basic system, one step towards achieving fault tolerance can be addressed by a prescriptive aspect that *makes 2 copies of all data of type waypoint_data and assigns those copies to different processors.* A designer might also want *all data of type pilot_actions to be logged.* In addition, it is easy to specify that *all data of type Y (no matter where it is in the system) should be encrypted with a particular encryption scheme.* Many other examples can be given for non-functional categories of modifications relating to security, persistence, locking, real-time and reliability.

Normally, prescriptive aspects are used to modify the basic design. However, since the prescriptive aspect language has a create statement, prescriptive aspects can, by themselves, implement the entire basic design plus changes to it. While we have not yet investigated this feature in detail, building a system this way would be very flexible since even the basic design would be easily re-done. With this feature it is also possible to construct a subsystem or infrastructure with the prescriptive aspect language and then import that subsystem or infrastructure. For example, the design of an OS for a set top box can be designed using prescriptive aspects, then that OS infrastructure could be added to a product simply by executing the prescriptive aspect.

3.2 Expert Advice

When advice is deemed important and potentially usable on more than one project, then that advice can be generalized and placed in a global (for a given application domain, e.g., avionics) prescriptive aspect library. VEST supports reusing such prescriptive aspects by organizing them into a prescriptive aspect library. Prescriptive aspects are not permitted into the prescriptive aspect library unless they meet with the approval of the library administrator. The requirements include that they are sufficiently general, can be parameterized, include a complete English description, meaningful constraints specified, and they relate to non-functional properties.

One way to use the expert advice is as a collection of ideas from previous projects that might be applicable. For example, a designer can walk through all the library advice and determine if they are appropriate. After designing a functional avionics product a designer may browse through these expert prescriptive aspects for security, real-time performance, fault tolerance, and persistence. For each category they can determine if any of the advice should be applied directly or that they need to create similar advice for their particular project. This browsing can aid in producing a more complete and tailored design and when specific advice is already in the library it is easy to apply.

Also, advice can be grouped in such a way to support implementing a wide reaching concept, such as improved computer security. For example, for general security advice there might exist a group of prescriptive aspects that relate to denial of service, encryption, and authentication. Applying the high level advice, applies the entire group.

3.3 Hierarchies of Advice

Regardless of how prescriptive aspects are added to a design there can be a need for hierarchies of advice. In some cases it may be necessary to apply to a design a set of seemingly "unrelated" aspects in some order. To support this feature, the designer has the capability to describe precedence constraints among the aspects. More importantly, the same mechanisms can be applied to create a "related" set of changes to effect a global change to the system (as described above for the security example). In order to make high level changes to a design (e.g., in regard to security, fault tolerance, reliability, and performance) it is usually necessary to make a set of "related" and more specific changes. For example, there can be a group of advice in the prescriptive library that supports a secure avionics system. This advice may encompass a collection of changes that includes encrypting certain types of communication, adding intrusion detection changes, adding modifications that prevent or minimize denial of service. The mechanisms in VEST support this type of design where the root of the hierarchy can imply changes needed for security, and the rest of the tree contains the specific modifications required.

4. VEST PRESCRIPTIVE ASPECT LANGUAGE

4.1 Design Philosophy

VPAL enables users of VEST to specify their prescriptive aspects. The syntax of VPAL is specific to the VEST entities that specify components, their attributes, and interactions between components. Ease-of-use and modification power are the driving forces behind VPAL's design. VPAL allows the specification of modifications using a simple yet powerful syntax. Consequently, VPAL is a language with no data type declarations, procedures, control flow, loops and classes. VPAL's syntax consists of just four key statements. It would take a few minutes for a novice programmer to understand VPAL and be able to write prescriptive aspects. The power of VPAL's syntax can only be fully realized through its use. The evaluation section presents concrete examples of the time saved by designers using prescriptive aspects written in VPAL.

VPAL is similar to SQL except that the data set being operated on is sets of components rather than sets of rows from a table. It is not a procedural, functional, object-oriented or even aspect-oriented programming language. It is intended to be specifically used in the VEST tool for easily creating prescriptive aspects.

4.2 Separation of Concerns

As mentioned earlier, prescriptive aspects change a design by adjusting properties in the reflective information of components and/or by adding/removing components from the design. VPAL

explicitly separates the concerns of collection, operation, addition and removal of components. Four key statements in the language, *Get*, *Set*, *Create* and *Delete* enable this separation of concerns. Each of these concerns plays an important role in fulfilling the objective of prescriptive aspects and they are described in detail below. The full BNF specification of the VPAL syntax is available in [27].

4.1.1 Collection

A Collection is defined as a set of components from a system design. A collection enables a designer to represent a cross section of the design based on the properties of components or the relationships between them. This is essentially the value of collection as it enables a designer to quickly and easily identify components to be modified which would have otherwise taken much manual search time. The *Get* statement in VPAL implements this feature. It assigns the collection to a variable for later use. For example, the GET statement

```
GET SWComps = (CT == SoftwareComponent);
```

finds all components whose component type (CT) is "SoftwareComponent" in the design and assigns this set to a variable called "SWComps". The right side of the statement specifies the search criteria. In this case, we used the component property of type as our search criteria, but in general, it can be any component property such as type, name or any of the extensive list of attribute values found in the reflective information of a component. Search criteria can also be combined into compound statements with boolean operations *AND*, *OR* and *NOT*.

4.1.2 Operation

An Operation involves changing a design on previously gathered collections. An operation enables the weaving of user-defined changes into a design. Operations on collections are performed with the *Set* statement that adjusts the properties in the reflective information of the collection. For example, the SET statement

```
SET SWComps.(PN = MemoryNeeded, PV = 0);
```

initializes the property (attribute) name (PN) "MemoryNeeded" of all components in the "SWComps" collection to a property value (PV) of zero.

4.1.3 Addition and Deletion

Addition and removal of components are self-explanatory. These commands enable users to weave changes into a design. Addition of components could also potentially be used to create large designs from scratch. The *Create* statement in VPAL adds a set of components to the design and assigns this set to a variable for later use. For example, the CREATE statement

```
CREATE DispComp = ($SW, Software,
     CT = SoftwareComponent,
     CN = MyDisplayComponent);
```

creates a software component in the parent model called "Software" in the software folder ($SW) with a component name (CN) of "MyDisplayComponent" and assigns it to variable "DispComp".

The *Delete* statement removes previously defined collections from the design. For example, the DELETE statement

```
DELETE DispComp;
```

deletes from the design the components defined in the "DispComp" collection.

4.3 Multi-line Semantics

VPAL supports multi-line semantics. This means that each prescriptive aspect can contain multiple lines of instructions. Each instruction is one of the four statements that were described above. The multi-line semantics of VPAL allows a user to define and operate on multiple collections within the same prescriptive aspect.

For example, suppose we wanted to apply the following prescriptive aspect to a distributed avionics system being designed in VEST:

> Double the memory needed for all device software components
> - and -
> change all display software components to use double buffering

Using the multi-line semantics of VPAL, we could specify this prescriptive aspect as

```
[1]  GET SwComp = (CT == SoftwareComponent);
[2]  GET DevComp = SWComp.(
       PN == componentType,
       PV == BM__DEVICE_COMPONENT);
[3]  GET DispComp = SWComp.(
       PN == componentType,
       PV == BM__DISPLAY_COMPONENT);
[4]  SET DevComp.(PN == MemoryNeeded,
       PV = PV * 2);
[5]  SET DispComp.(PN == DoubleBuffered,
       PV = 1);
```

This prescriptive aspect contains two different cross-sections of the design of interest to the designer. One contains all device components (line 2) and the other contains all display components (line 3). The designer then modifies each set according to the change desired (lines 4 and 5).

While VPAL is simple, the downside of simplicity is that the expressive power of the language is limited sometimes resulting in redundant code. For example, consider a design with a large number of software components that are sub-classified into many software component types. Suppose we wanted to write a prescriptive aspect to initialize several of the attributes of these software components to different values by type. The code would contain redundancy for a design with a large number of software component types. This redundancy could be eliminated with loops in VPAL. VPAL can be extended to allow loops and other programming language concepts such as control flow, procedures, inheritance, overriding, and so on but we have not found it necessary for embedded systems of small or moderate size.

5. CASE STUDY

In this section, we demonstrate the benefits of prescriptive aspects through a case study. We apply prescriptive aspects to the design of an avionics system, which is based on the Boeing Bold Stroke platform. We show how prescriptive aspects support system modification, provide expert advice, and save 69% of design time.

The baseline toolset for comparison includes Rational Rose [19] and Quantify which are both currently used in Boeing's product development. The UML models of all Bold Stroke components were available in Rational Rose before experimentation started. The worst-case execution times (WCET) of all components used were also available before experimentation began.

5.1 Design of an Avionics System

Appendix A shows the UML diagram of the software architecture of a typical avionics system on the Bold Stroke platform. This system corresponds to a navigation type function on an aircraft. The aircraft maintains a list of waypoints (points to fly the aircraft to). Waypoints are selected in groups to form routes (a series of points to fly the aircraft to, one after the other). The pilot can modify the waypoints to change the current route of the aircraft. In addition, GPS sends location information to the system periodically. The current waypoint and current aircraft position are displayed periodically.

This navigation system is a typical example of a distributed real-time embedded system with many crosscutting concerns. Such concerns include real-time schedulability as well as event channel, memory and buffer requirements. These and many other concerns are critical to the overall system. We use aspect checks to identify them and prescriptive aspects to modify them if they do not meet the system requirements.

5.1.1 Aspect Checks

Aspect checks verify certain properties of a real-time embedded system design. Aspect checks are explicit checks across components in a system. Usually an aspect check looks for hidden dependencies among components that are hard to directly identify by a designer. There are various kinds of "global" hidden dependencies in a system design. We focus on the most interesting checks to designers in this avionics application. In the domain of avionics systems, our aspect checks include a memory footprint check, an event channel check, a buffer size check, and schedulability analysis.

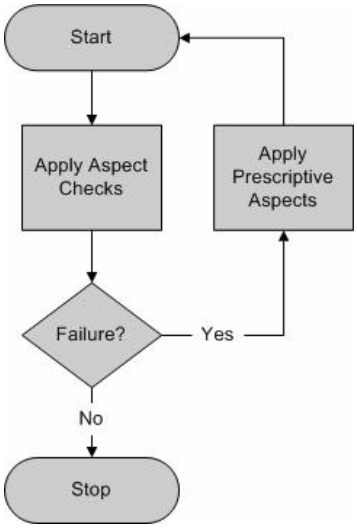

Figure 2

Aspect checks and prescriptive aspects work in a complementary way. Aspect checks examine the system for hidden crosscutting dependencies while prescriptive aspects are applied to modify the system as directed by the designer, e.g., if the aspect check determines a deficiency. Their relationship is described in Figure 2. Both aspect checks and prescriptive aspects are implemented as interpreters in VEST.

5.1.1.1 Memory Footprint Check

The memory footprint check is used to verify whether there is enough physical memory to support the system software. Insufficient memory can cause serious problems when the system is deployed. There are two parts to the check. The first part of the check is concerned with main memory. Here, a sum is done of the memory needed by all the software components, and the available physical memory (RAM) provided by the hardware. The check verifies whether there is enough physical memory in the system for the software components defined. The second part of the memory footprint check involves non-volatile memory (NVM). Similar to the first check, this check verifies that there is enough NVM available in hardware as needed by the software components of the system.

5.1.1.2 Event Check

In this particular avionics system, components communicate with each other by sending events through event channels or paths. The event check iterates through all components and makes sure that every event supplier has an event consumer corresponding to it and every event consumer has an event supplier corresponding to it. Mismatches in the event channel are automatically identified. Also, circular event dependencies can be checked by going through the event channel.

5.1.1.3 Buffer size check

The buffer size check is used to make sure that there are no buffer overflows during communication between software components. In our design, every component has a buffer to temporarily hold event messages received from other components before they are processed. The size of a buffer needed by a component to avoid overflow is based on four parameters – the number of event suppliers, event supplier's supply rate, event consumer's consume rate and the size of the event message. The event supply and consume rate vary among different components in the system. Also, different events have different message sizes. We can calculate the size of the buffer needed by a particular component by summing of the sizes of the buffers needed for the event messages it receives from each of its event suppliers. Each event buffer size is calculated as follows

$$\left(\frac{SupplierEventSupplyRate}{ConsumerEventConsumeRate}\right) \times EventMessageSize$$

5.1.1.4 Schedulability

An avionics system is a typical distributed real-time system. In such a system, every task is executed periodically, and it must complete its execution before its deadline. A system is schedulable if all of its tasks can meet their deadlines. Schedulability is an aspect check that crosscuts the system. A detailed explanation of schedulability checks in VEST can be found in a previous VEST paper [26]. In brief, VEST supports the standard rate monotonic and EDF scheduling policies. More importantly it also supports distributed real-time scheduling of

various types and distributed robust scheduling. In robust scheduling designers are informed not only of schedulability, but also how close the system is to missing deadlines.

5.1.1.5 Correctness

VEST does not support formal proof of correctness. Formal verification tools, while valuable, are not always used for various reasons. Rather, VEST aims to provide an extensible collection of key embedded and real-time systems checks to avoid many typical and difficult to find errors. The result is a practical tool that improves system design and analysis as demonstrated by various case studies in [26] and in Section 5.2. For a discussion of the underlying semantics of VEST see [26].

5.2 Experiment

Throughout our experiment, prescriptive aspects are used for two primary purposes: system modification and expert advice.

5.2.1.1 System Modification

First in this experiment, the designer started to design an avionics system in VEST based on product scenario 3.3 provided by Boeing. He composed the system using the components from the VEST component library. Afterward, he assigned different values to attributes such as memory size, buffer size, WCET, and period to the components accordingly.

After running the memory footprint aspect check however, the designer found out that the amount of memory allocated in hardware was smaller than required by the software components. Instead of modifying the attribute values (named MemoryNeeded) of the components manually, the designer decided to use a prescriptive aspect. He executed the following prescriptive aspect, which reduces by half the memory allocated to software components of type BM__DISPLAY_COMPONENT.

```
GET A = (CT == SoftwareComponent) AND
       (PN == componentType,
       PV == BM__DISPLAY_COMPONENT);
SET A.(PN=MemoryNeeded, PV=PV/2);
```

Then he re-ran the memory footprint check and it passed. This saves time over modifying the system parameters manually and is more accurate.

After checking the memory allocation, the designer checked the schedulability of the system design by running the schedulability aspect check. The check failed because in this case, event suppliers in the system were specified to have too high a WCET value that caused tasks in the system to miss their deadlines. In general, there can be several factors that cause a schedulability test to fail such as insufficient task period or high WCET value. Again, instead of modifying all these parameters manually, the designer modified the system design automatically by executing the following prescriptive aspect.

```
GET SW = (CT == SoftwareComponent);
GET ES = (CT == EventSupplied);
GET EC = (CT == EventConsumable);
GET ContES = SW[$ONEONE,$DR=$CONT]$ES;
GET ContEC = SW[$ONEONE,$DR=$CONT]$EC;
GET MappedES = $ES[$MANYONE,$DR=$CONN] EC;
SET MappedES.(PN=WCET, PV=10);
```

This prescriptive aspect collected the event supplier components that are *contained* in software components and are *connected* to event consumer components, and set their WCETs to 10ms. After making this modification, the schedulability test passed. Of course, these modified components must be reprogrammed to meet this new WCET. If this is not possible, then faster or more cpus must be added to the system and re-analyzed.

5.2.1.2 Expert Advice

Prescriptive aspect can be used to provide expert advice on the design of a system. Expert advice in this context is generic advice that applies to various scenarios sharing the same meta-model. Usually expert advice is stored in a library. Designers can retrieve the expert advice from the library and reuse the advice by applying them to every relevant scenario conforming to the same meta-model.

In this case study, we used assigning locking strategies to components as an example of expert advice. There are three kinds of locking strategies used by components in the Bold Stroke platform: internal, external and synchronous proxy. The internal locking strategy requires a component to lock itself when data is modified. An external locking strategy requires the user to explicitly acquire a component's lock before accessing its data and release the lock when finished. The synchronous proxy strategy requires the use of cached states. Knowledge of such locking strategies is generic and applies to all Boeing OEP product scenarios. Therefore, we put this particular prescriptive aspect into the general expert advice library. When a designer wants to apply this set of locking strategies to his design, he can choose the prescriptive aspect from the library and execute it.

The internal locking strategy:

```
GET SW = (CT == SoftwareComponent);
SET SW.(PN=lockingMode, PV=INTERNAL);
```

The external locking strategy:

```
GET PushPull =(CT==SoftwareComponent)
       AND (PN==componentType,
       PV==BM__PUSH_PULL_COMPONENT);
GET EC = (CT == EventConsumable);
GET PushPullMappedEC = $PushPull
       [$ONEMANY,$DR=$CONT] EC;
SET PushPullMappedEC.
       (PN=lockingMode,PV=EXTERNAL);
```

The synchronous-proxy locking strategy:

```
GET SW = (CT==SoftwareComponent);
GET EC = (CT==EventConsumable);
GET Timers = (CT==SWTimer);
GET SWMappedEC = $SW
      [$ONEMANY,$DR=$CONT] EC;
GET SWMappedTimer = $SW
      [$ONEONE,$DR=$CONT] Timers;
GET SynchProxy = $SWMappedEC
      [$ONEMANY,$DR=$CONN] SWMappedTimer;
SET SynchProxy.(PN=lockingMode,
      PV=SYNCH_PROXY_MASTER);
```

By default, we assume every software component uses internal locking. A "PushPull" software component is defined as one that updates its values (by pulling or getting data from its suppliers) when it receives an indication (through a push or set). According to our application rules, any PushPull software component that has one or more data suppliers must use external locking. This is what is coded in the external locking prescriptive aspect. Finally, any component that receives data from more than event channel, each running on different timers in the system should use synchronous-proxy locking as indicated by the last prescriptive aspect.

By applying this expert advice, we assign different locking strategies to all the software components in the system. This prescriptive aspect is stored as expert advice in the library. Using prescriptive aspects for expert advice saves the designer a lot of time by automating decision-making. This is especially useful when used in designs with a large number of components and where there are many interactions among the components.

5.2.1.3 Hierarchical Prescriptive Aspects

A simple prescriptive aspect is a self-contained entity of one or more VPAL statements. The previous sections illustrated some simple prescriptive aspects. In addition, VEST provides support for hierarchical prescriptive aspects.

Hierarchical prescriptive aspects are comprised of one or more simple prescriptive aspects with precedence constraint rules. This enables a designer to define several independent simple prescriptive aspects that can later be combined into a single compound prescriptive aspect. In addition, the designer can ensure that when the compound prescriptive aspect is executed, there is a guarantee over the order of execution of the constituent simple prescriptive aspects.

We used a hierarchical prescriptive aspect to perform system initiation in our experiment. We defined independent prescriptive aspects to initialize the memory requirements of the system, the buffer size allocation, real-time properties of components such as WCET and the locking strategies to be used by different components of the system. In the interest of space, we do not show these prescriptive aspects here. By combining these prescriptive aspects into a single hierarchical prescriptive aspect, we were able to precisely define how our design should be initialized before being deployed.

5.2.1.4 Experimental results

We performed an evaluation to measure the benefits of prescriptive aspects in composing distributed avionics systems. The performance metric is the time it takes to compose (including design, implementation via composition, and testing or analysis) an avionics product scenario to achieve end-to-end distributed real-time schedulability, memory allocation, buffer size assignment and locking strategy assignment. This experiment was accomplished in a very limited situation. An expert from Boeing performed the experiment using their current approach, and a researcher from the University of Virginia (UVa) performed the experiment using prescriptive aspects in VEST. For each person we timed the various steps involved with the experiment. Since this is a single experiment with many potential issues, the results are not definitive. However, we believe that the results are representative and are consistent with other tests performed earlier on other product scenarios [26].

The baseline comes from the time estimates for Boeing to build, analyze and validate an avionics system conforming to product scenario 3.3, while VEST uses prescriptive aspects to do the same work. A comparison between UVa and Boeing data is shown in Table 1.

Table 1

VEST			**Baseline**		
Step		Time (min)	Step		Time (min)
V.1.1	Design:	128	B.1.1	Design	280
	Memory check	1	B.1.2	Memory check	20
	Fixing memory problem using VPAL	20		Fixing memory problem	80
V.1.2	Scheduling check:	1	B.1.3	Timing Test	30
	Fixing scheduling using VPAL	15		Fixing scheduling	110
	Scheduling check	1		Timing Test	20
	Scheduling analysis: distributed	1		Test: distributed	20
	Assign locking strategies using VPAL	1		Assign locking strategies	30
	Implementation:	320		Implementation	960
Total Composition Time		488	Total Composition Time		1550

From the Table above all steps in the design process are faster with VEST. In particular, the time saved for the steps using VPAL show the value of prescriptive aspects. For example, using VPAL to fix the memory problem reduced the time from 80 to 20 minutes. Overall, the VEST approach saved 69% of the time needed to design and implement a (representative) distributed avionics system. Since the memory and real-time scheduling analysis are automatic, the VEST tool should save even more time both (i) when used for larger systems, and (ii) when designers wish to attempt multiple competing designs. For example, suppose a particular design solution, shown to meet the

requirements, had 3 processors, 1 MB of memory and various amounts of replication for different data types. The designer might consider removing a processor and modifying some of the replication and re-run the analysis. Re-running the analysis is very fast and each tradeoff-analysis cycle improves the time gains of using VEST. If the new system still meets the requirements, then the designer has competing solutions to choose among.

6. STATE OF THE ART

The work described in this paper builds upon and integrates research from three main areas: component based design tools, aspect-based design tools and aspect oriented programming. In section 6.1, we briefly discuss component based design and compare VEST to other such design tools. In sections 6.2 and 6.3, we more fully discuss the relationship of our work to other aspect-based tools and aspect-oriented languages such as AspectJ.

6.1 Component Based Design Tools

The software engineering field has worked on component based software development for a long time. Systems such as CORBA [23], COM [15], and DCOM [16] exist to facilitate object or component-based system development. These systems have many advantages including reusability of software and higher reliability since the components are written by domain experts [25]. However, none of these systems have adequate crosscutting analysis capabilities. One exception to this is KNIT. KNIT [20] is an interesting composition tool for general purpose operating systems. This system addresses a number of crosscutting concerns in composing operating systems. For example, it considers linking, initialization, and a few other dependencies. To date, it has not focused on real-time and embedded system concerns.

An excellent tool that matches our goals quite closely is MetaH [30]. MetaH consists of a collection of tools for the layout of the architecture of an embedded system and for its reliability and real-time analysis. MetaH begins with active tasks as components, assumes an underlying real-time OS, and has some dependency checking. Their work uses fixed priority scheduling. The MetaH work was done prior to aspect oriented languages. In contrast we elevate aspects to the central theme of VEST and focus on dependency checks. We also provide more general scheduling analysis support: including automatically collecting the task set characteristics and requirements from the design, matching the requirements with assumptions of various scheduling analyses, providing more than fixed priority scheduling, and supporting access to a commercial real-time scheduling tool.

Cadena [9] is an integrated environment for building and modeling CCM systems. It supports modeling of a system using specifications attached to IDL. Similar to VEST, Cadena can generate a configuration file for Boeing Bold Stroke configurator. Compared to VEST, Cadena has fewer analysis routines although it includes some simple analysis capabilities. In addition, Cadena does not support system modification using aspects. Finally, Cadena does not support GUI-based composition – the system diagram is generated from the specifications.

Automatic Integration of Reusable Embedded Software (AIRES) [7] is a tool to model and analyze embedded systems. In AIRES, both the real-time behavior of software controllers and the physical environment are modeled. AIRES focuses on the formal system analysis based on Timed Petri-Net theory. Unlike VEST, AIRES does not support the concept of aspects, nor does it support final code generation.

6.2 Comparison with Aspect-based Design Tools

In addition to general composition tools, there is much research work going on in the field of aspect-based tools. Like the aspect checks and VPAL in VEST, these aspect-based tools focus on the crosscutting issues in a system design and approaches to modify the system using aspects.

Time weaver [4] is a reusable component framework supporting aspects. There exist two distinct design *aspects* in Time Weaver: functional design and deployment design. Functional design deals with the application-specific logic, while deployment design deals with how various modules comprising the application-specific logic communicate with each other. Time Weaver aims at automating the deployment aspect, and thus improves software productivity. It is similar to VEST in its way of modifying systems. More specifically, the deployment aspect in Time Weaver is very comparable to the expert advice in VEST, by giving guidelines and advice to a specific non-functional concern crosscutting the system. However, Time weaver does not have the capability to do various analyses. All analyses are handled by external programs.

Constrain-Specification Aspect Weaver (C-SAW) [6] is a composition and analysis tool. Aspects used in it are quite similar to those in VEST. A weaver is specified using ECL (Embedded Constraints Language) formally, which is an extension to OCL. The system design can be modified according to the specification of the weaver. However, Aspect Weaver does not have a corresponding analysis counterpart to aspect checks as found in VEST.

6.3 Comparison with AspectJ

AspectJ is an aspect-oriented extension to the Java programming language that supports the modular implementation of cross-cutting concerns. The primary difference between VPAL and AspectJ is that VPAL provides support for design-time aspects whereas AspectJ provides support for run-time aspects. Thus, the crosscutting advice and language-support features of the two vary accordingly.

Both VPAL and AspectJ provide *property-based* crosscutting which is cross-cutting based on the properties of a design or program. However, only AspectJ provides language support for *control-flow* based crosscutting that allows crosscutting based on the control-flow relationships of a program. VPAL applies advice at design-time by weaving it into system designs. Advice is applied by changing the *reflective* information of components in a design. AspectJ on the other hand, applies advice at run-time by weaving it into the source code of Java programs. Here, advice is applied by specifying that certain code execute *before*, *after* or *around* each of the join points of a program. The design-time application of advice makes VPAL a language-independent means of creating and using aspects. AspectJ's operation at the source code level makes it a language-dependant framework for aspects.

Since VPAL and AspectJ operate at different stages in the software development cycle, they consequently provide different levels of language support for various features of aspect-based system development. Precedence among aspects is supported both in VPAL and AspectJ. Precedence in VPAL is not a language supported feature but there is a way to specify this in a separate aspect layer in a VEST design. AspectJ supports language-based specification of precedence. Parameterized advice is supported in AspectJ but not in VPAL. VPAL will provide support for this in future versions of VEST. Also, AspectJ provides support for inheritance and overriding among aspects whereas VPAL does not. Greater language support of the aforementioned features in AspectJ compared to VPAL can be attributed to the complexity of advice that each language has to support. The complexity of a system and its behavior increases when it moves from design to source-code level. Since VPAL supports aspects for design-time systems, its complexity is less than AspectJ. This is another reason why design-time aspects are useful. It keeps the aspect language simple to understand and implement.

7. CONCLUSION

When building embedded systems from components [3][9][22][28], those components must interoperate, satisfy various dependencies [6], and meet non-functional requirements. The VEST toolkit can substantially improve the development, implementation and evaluation of these systems as previously shown [26]. In this paper we focus on the prescriptive aspects capability of VEST. We have implemented a new prescriptive aspect language and discussed its advantages and implications for embedded systems. We evaluated prescriptive aspects on a case study. The case study (i) qualitatively demonstrates the benefits of prescriptive aspects and (ii) includes quantitative data that show a savings of over 69% in design and analysis time. Currently, a version of VEST has been delivered to Boeing.

8. REFERENCES

[1] Audsley, N. C. (1991) Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times, Tech. Report YCS 164, University of York, York, England.

[2] Bettati, R., (1994) End-to-End Scheduling to Meet Deadlines in Distributed Systems, PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] Booch G. (1987) Software Components with Ada: Structures, Tools and Subsystems. Benjamin-Cummings, Redwood City, CA.

[4] De Niz D., Rajkumar R., (2003) Time weaver: a software-through-models framework for embedded real-time systems, *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, California, USA

[5] Gill, C., Levine, D. and Schmidt, D. (2000) The Design and Performance of a Real-Time CORBA Scheduling Service, *Real-Time Systems*, 20(2), Kluwer.

[6] Gray, J., Bapty, T., Neema, S., and Tuck, J. (2001), Handling Crosscutting Constraints In Domain Specific Modeling, *CACM*, Vol. 44, No. 10.

[7] Gu Z., and Shin K., (2003) An Integrated Approach to Modeling and Analysis of Embedded Real-Time Systems Based on Timed Petri Nets*, International Conference on Distributed Computing Systems*, Providence, RI.

[8] Harrison T., Levine, D. and Schmidt, D. (1997), The Design and Performance of a Real-time CORBA Event Service, *Proceedings of OOPSLA '97*, ACM, Atlanta, GA.

[9] Hatcliff, J., et. al. (2003) Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems, *ICSE 2003*, Portland, Oregon.

[10] Humphrey, M. and Stankovic, J., (1996) CAISARTS: A Tool for Real-Time Scheduling Assistance, *IEEE Real-Time Technology and Applications Symposium*.

[11] Kao, B., and Garcia-Molina, H., (1994) Subtask Deadline Assignment for Complex Distributed Soft Real-time Tasks, *IEEE International Conference on Distributed Computing Systems*.

[12] Kiczales, G., Hilsdale, E., Hugunin , J., Kersten , M., Palm , J., and Griswold, W. (2001) Getting Started With AspectJ, *CACM*, 44(10).

[13] Klein, M., Ralya, T., Pollak, B., Obenza, R., Harbour, M. G. (1993) *A Practitioner's Handbook for Real-Time Analysis – Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers.

[14] Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason IV C., Nordstrom G., Sprinkle J., Volgyesi P. (2001) The Generic Modeling Environment, *Workshop on Intelligent Signal Processing*, Budapest, Hungary.

[15] Microsoft Corporation and Digital Equipment Corporation (1995) The Component Object Model Specification. Redmond, Washington.

[16] Microsoft Corporation (1998) Distributed Component Object Model Protocol, version 1.0. Redmond, Washington.

[17] Object Management Group (1997) The Common Object Request Broker: Architecture and Specification, Revision 2.0, formal document 97-02-25 (http://www.omg.org).

[18] Palencia, J. and Gonzlez Harbour, M. (1998) Schedulability Analysis for Tasks with Static and Dynamic Offsets, *Real-Time Systems Symposium*.

[19] Rational Software Corporation, Model Driven Development Using UML: Rational Rose http://www.rational.com/media/products/rose/D185F_Rose.pdf.

[20] Reid, A., M. Flatt, L. Stoller, J. Lepreau, and E. Eide. (2000) Knit: Component Composition for Systems Software*. OSDI 2000*, San Diego, Calif., pp. 347-360.

[21] Santarini, M, (2003) Cadence Says Platform Halves Verification Time, *EEdesign*.

[22] Short K. (1997) Component Based Development and Object Modeling. Sterling Software (http://www.cool.sterling.com).

[23] Siegel J. (1998), OMG Overview: Corba and OMA in Enterprise Computing, *CACM*, Vol. 41, No. 10.

[24] Schmidt, D., Levine, D., and Mungee, S. (1998) The Design of the TAO Real-Time Object Request Broker,

Computer Communications, Special Issue on Building Quality of Service into Distributed Systems, 21(4).

[25] Sharp, D. (1998) Reducing Avionics Software Cost Through Component Based Product Line Development, *Software Technology Conference*.

[26] Stankovic, J., Zhu, R., Poornalingham, R., Lu, C., Yu, Z., Humphrey, M., and Ellis, B., (2003) VEST: An Aspect-Based Composition Tool for Real-Time Systems, *Real-Time Applications Symposium*.

[27] Stankovic J., Nagaraddi P., Yu Z., He Z., (2004) VEST User's Manual, *UVa Technical Report TR-CS-2004-10*.

[28] Szyperski C. (1998) Component Software Beyond Object-Oriented Programming. Addison-Wesley, ACM Press, New York.

[29] Tindell, K. (1994) Adding Time-Offsets to Schedulability Analysis, Technical Report YCS 221, Dept. of Computer Science, University of York.

[30] Vestal, S. (1997) MetaH Support for Real-Time Multi-Processor Avionics, *Real-Time Systems Symposium.*

9. APPENDIX A