VEST: An Aspect-Based Composition Tool for Real-Time Systems *

John A. Stankovic Ruiqing Zhu Ram Poornalingam Chenyang Lu
Zhendong Yu Marty Humphrey Brian Ellis

Abstract

Building distributed embedded systems from scratch is not cost-effective. Instead, designing and building these systems by using domain specific components has promise. However, in using components, the most difficult issues are ensuring that hidden dependencies won't cause failures and that non-functional properties such as real-time performance are being met. We have built the VEST toolkit whose aim is to provide a rich set of dependency checks based on the concept of aspects to support distributed embedded system development via components. We describe the toolkit and its novelty. We also use VEST on two case studies of a CORBA-based middleware for avionics. Data collected shows that VEST can significantly reduce the time it takes to build a distributed real-time embedded system by over 50%. Key "lessons learned" from our experience with using VEST on these case studies are also highlighted.

1. Introduction

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.

Our work focuses on the development of effective composition mechanisms, and the associated dependency and nonfunctional analyses for real-time embedded systems. Our solution is based on extending the notion of *aspects*. Aspects [10] are defined as those issues that cannot be cleanly encapsulated in a generalized procedure. They usually include issues that affect the performance or semantics of components. This includes many real-time, concurrency, synchronization, and reliability issues. Aspects, to date, have largely been language dependent in that aspects are implemented as language constructs. A major contribution of our work is that we extend the concept of aspects to language independent notions and apply them at design time. We identify two types of language-independent aspects referred to as *aspect checks* and *prescriptive aspects.* Together these permit the benefits of aspects to be exercised early in the composition process rather than in the implementation phase. Our solutions are embodied within a toolkit called VEST (Virginia Embedded Systems Toolkit). VEST itself is not a complete requirements, design and implementation tool; rather it currently focuses on the specific composition and analysis tasks.

2. Overview of VEST

VEST provides an environment for constructing and analyzing component-based distributed real-time embedded systems. VEST helps developers 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 and reliability. Distributed embedded systems issues are explicitly addressed via the mapping

* Stankovic, Zhu, Poornalingam, Yu, and Humphrey are with Department of Computer Science, University of Virginia, Charlottesville, VA 22903. Lu is with Department of Computer Science and Engineering, Washington University in St Louis, St Louis, MO 63130. Ellis is with the Boeing Company. E-mail of Stankovic (corresponding author): stankovic@cs.virginia.edu. This work was supported, in part, by the DARPA PCES program under grant F33615-00-C-3048 and by Microsoft Research.

Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)
1080-1812/03 $17.00 © 2003 IEEE

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2003		**00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
VEST: An Aspect-Based Composition Tool for Real-Time Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Virginia,Department of Computer Science,151 Engineer's Way,Cahrlottesville,VA,22094-4740**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**12**	

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 five libraries, a set of aspect checks, and a GUI-based environment for composing and analyzing embedded products.

- **4 Component Libraries**: Because VEST supports real-time distributed embedded systems, the VEST component libraries contain both software and descriptions of hardware components and networks. VEST components can be *abstract* or *actual*. An abstract component is a design entity that represents the requirements, e.g., a timer with certain requirements or a generic processor is an abstract component. An actual component is the implementation or description of a reusable entity. A specific timer module written in C and a Motorola MPC7455 are examples of actual components. Sets of reflective information exist for each of these component types. The reflective information of an abstract component includes its interface and requirements such as for security. The reflective information for actual components includes categories such as 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 and its extensibility are some of the key features that distinguish VEST from many other tools. To support the whole design process of embedded systems, VEST implements four component libraries: the application library, middleware library, OS library and a hardware library.
- **Prescriptive Aspects Library:** Prescriptive aspects are reusable programming language independent advice that may be applied to a design. For example, a developer can invoke a set of prescriptive aspects in the library to add a certain security mechanism *en masse* to an avionics product.
- **Aspect Checks:** VEST implements both a set of simple intra- and inter-component aspect checks that crosscut component boundaries. A developer 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.
- **Composition Environment:** VEST provides a GUI-based environment that lets developers compose distributed embedded systems from components, perform dependency checks, and invoke prescriptive aspects on a design. For details on the GUI see [24].

3. Language Independent Aspects

Aspects [10] are defined as those issues that cannot be cleanly encapsulated in a generalized procedure. For example, changing one component may affect the end-to-end response time of many components that are working together. Security aspects of a system also involve multiple correlated components. Aspects, as defined in the literature, are at the programming language level. For example, AspectJ [10] 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 VEST, we apply the concept of aspects as crosscutting dependencies at *design* time. This results in *language independent* aspects. We have discovered that there are, at least, two types of language independent aspects. The first type we call *prescriptive aspects*. In prescriptive aspects, a general set of advice is programmed and retained in the prescriptive aspect library. This advice can then be applied to the design, not source code. The application of this advice changes the reflective information associated with the affected components and their interactions (section 3.1). The second type of aspect we call *aspect checks*. Aspect checks look for specific crosscutting dependencies, which are often hidden from developers (sections 3.2 and 4). Language independent aspects help developers handle crosscutting dependencies among components at the design stage. Compared with aspect oriented languages, language independent aspects reduce errors in the early stages of software design lifecycles, which lead to shorter time to market. Language independent aspects can achieve the benefit of aspects in embedded systems even when general purpose languages (e.g., C++, C, and Java) are used for implementation.

3.1. Prescriptive Aspects

Prescriptive aspects are *advice* that may be applied to a design. The advice is written in a simple VEST Prescriptive Aspect Language (VPAL). Prescriptive aspects are independent of programming languages because they apply to the system design, and the resultant new design can be implemented in any programming language. 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. An English language description may also be associated with each aspect. This permits an

explanation of why this advice is in the library and how and when to use it.

Specification and Examples

We have examined specific prescriptive aspects related to the distributed avionics domain via Boeing's Bold Stroke middleware. The following are examples of prescriptive aspects organized in categories. Each of these examples only list the (parameterized) advice; they do not show the accompanying English language description and constraints. These examples demonstrate that prescriptive aspects can be a powerful tool in real-time embedded system design.

1) **Security:**
   a) for all pilot to ground communication encrypt it with RSA;
   b) for all data of type X encrypt with technique Y;
   c) for security level of any data of type X change to Y;
   d) move all data with security levels above X to physical store Y;
2) **Persistence:**
   a) for all log data in the navigation subsystem make it persistent;
   b) for all data of type X make it persistent;
   c) for all objects of type X make the save rate Y;
3) **Redundancy:**
   a) make X copies of all data of type Y;
   b) update all backups for data X with rate Y;
4) **Locking:**
   a) for data of type X lock all/fields of data;
   b) for all data of type X change to spin lock;
5) **Events:**
   a) for threads of priority higher than X modify the conditions in which to fire events;
   b) for all events of a type X make them also wait for condition Y;
   c) for all components that are critical have them fire a new event called X whenever they execute;
   d) for all components that filter their own events make a change to remove that filter and use the event channel filter.

Applying Prescriptive Aspects

The developer can apply a prescriptive aspect to a design by running a VPAL interpreter on its specification. The interpreter modifies the reflective information of design components. Since the code itself would no longer reflect the new design change, the interpreter marks the actual source code associated with that change as *"inconsistent and needing changes"* to meet the new design. Currently VEST does not support automatic code generation/modification, and the developer needs to implement the code change manually. Once the new code is created and linked to the component then the inconsistency indication is removed. Currently, we are implementing a tool to automatically convert a system design in VEST to a Bold Stroke configuration (implementation) through an XML configuration interface provided by Boeing.

Prescriptive Aspect Library

Prescriptive aspects should be general enough to be used in different products. VEST supports reusing prescriptive aspects by organizing them into the prescriptive aspect library. Prescriptive aspects will not be permitted into the prescriptive aspect library unless it meets with the approval of the system administrator. The requirements include sufficiently general, parameterized, complete English description, meaningful constraints specified, and relating to non-functional properties.

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 developer 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. In order to make high level changes to a design (e.g., in regard to security, fault tolerance, reliability, performance, etc.) 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, etc. 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. Future work will exploit this novel view supported by VEST.

The Value of Prescriptive Aspects

There are many ways in which prescriptive aspects have shown to be valuable. First, by using prescriptive aspects a developer is encouraged to design in a functional manner and then to apply non-functional updates to the design. This separation of concerns makes design easier. Second, prescriptive aspects can be thought of as general advice for changing a design in a global manner. The advice is domain specific. In this case, a developer can walk through all the library advice categories and determine if they are appropriate. For example, after

designing a functional avionics product a developer may browse through prescriptive aspects for security, real-time performance, fault tolerance, persistence, etc. For each category they can determine if any of the advice should be applied. 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 it. Third, advice can be grouped in such a way to support implementing a wide reaching concept, such as improved computer security. Under this general advice notion there might exist a group of prescriptive changes that relate to denial of service, encryption, and authentication. Applying the high level advice, applies the entire group. Fourth, prescriptive aspects support a widespread global change in the design by simply defining new advice or using pre-declared advice and applying it to your design. This prevents bugs where 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 dependency checks and schedulability analysis can be re-run automatically. This facilitates looking at multiple competing design options and making modifications easy.

3.2. Aspect Checking

One goal of VEST is to provide support for various types of dependency checking among components during the composition process. Dependency checks are invoked to establish certain properties of the composed system. This is a critical part of real-time embedded system design and implementation. Some dependency checks are simple and have been understood for a long time. We call these intra- and inter-component dependency checks. Other dependencies are very difficult and even insidious. We refer to these as crosscutting dependencies or *aspect checks*. Aspect checking is an explicit check across components that exist in the current product configuration. We have identified many aspect checks that would help a developer avoid difficult to find errors when creating embedded systems from components. In many cases the important thing is identifying the check required and implementing it so that it is automatic. Although the implementation of some checks may be simple, when these checks are combined with all the other features of VEST, the result is a powerful tool. To illustrate these concepts we discuss one of the most important aspect checks, end-to-end real-time scheduling, in the next section.

4. End-to-End RT Scheduling Aspect

An important check for real-time embedded systems is the schedulability analysis that validates whether all tasks can make their deadlines. Note that while designing and implementing a system that most changes made will affect the real-time properties of the system. This makes real-time scheduling a global cross cutting dependency. While many different schedulability analysis techniques exist [1, 2, 4, 9, 11, 16, 26], they differ in their assumptions on the task set and none of the existing analysis is applicable to all real-time embedded systems. The compatibility between schedulability analyses and the characteristics of the designed system is a typical crosscutting dependency that is "hidden" from the designer. Using an incompatible analysis on a system can lead to timing violations even when the schedulability analysis itself is correct. To handle different types of embedded systems, VEST provides a flexible scheduling tool that provides aspect checks on the compatibility between existing schedulability analyses and the system. This tool is composed of a set of schedulability analysis routines, an assumption table, and a reflective information collector. The assumption table lists the assumptions of each schedulability analysis routine. The current list of assumptions includes the applicability of various scheduling analysis with respect to periodicity, distribution, importance, blocking and precedence constraints. For example, the assumptions of the Rate Monotonic analysis are that all tasks are periodic. The Rate Monotonic with Priority Ceiling protocol's assumptions are (periodic, blocking). The VEST scheduling tool is extensible and new scheduling techniques can be added to the tool together with their assumptions.

Developers can assess the schedulability of the current design by running the scheduling tool from the GUI. The reflective information collector scans the software, hardware and network components of the design and produces a platform/task set information file that includes a list of the characteristics and the timing information of the task set. The tool selects an analysis whose assumptions match the characteristics of the system. This ensures that proper analysis and scheduling policy is applied. For example, for a system with all independent periodic tasks on a single processor, the Rate Monotonic check or MUF will be applied to the system. However, if the same task set is designed on a distributed platform, the DM/Offset analysis described below will be applied.

4.1. Deadline Monotonic with Phase Offset

Currently the VEST scheduling tool implements the basic Rate Monotonic check, the Maximum Urgency First algorithm [4], and a more sophisticated end-to-end analysis for distributed systems. In applying the tool to a Boeing's distributed avionics case study we found that RMA and MUF were not sufficient because such systems often run on a distributed platform. Avionics based on real-time CORBA (e.g., Bold Stroke and TAO) [6, 15, 22] requires support for the following distributed scheduling problem.

A periodic task T_i consists of multiple subtasks $\{T_{ij}\}$ on different processors. The set of subtasks have the same period P_i, and the task deadline $D_i = P_i$. Figure 1 shows a task T_1 having three subtasks connected by arrows (consider these T_{11}, T_{12} T_{13} not labeled in the figure). After completion of the first subtask T_{11}, an event is pushed to the second subtask T_{12}, and similarly for the third subtask T_{13}. The set of three subtasks of T_1 has a single deadline and period $P_1=D_1$. In this example, this task T_1 is physically placed on three distinct processors connected via a bus or a LAN. This example explains a single task. The system is then composed of multiple such tasks, each task T_i composed of one or more subtasks placed on one or more physical processors, and with communications proceeding in possibly "different" directions among the processors.

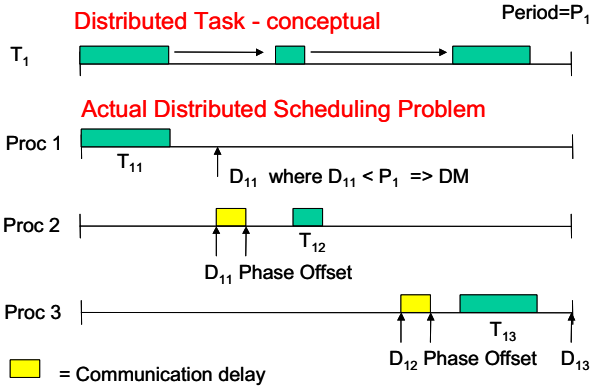

Figure 1. Schedulability Analysis for Deadline Monotonic with Phase Offset

This distributed scheduling problem can be modeled as an end-to-end scheduling problem. To provide scheduling support for the above distributed scheduling problem, VEST implements a scheduling analysis that we call *Deadline Monotonic with phase Offset* (*DM/Offset*). The assumptions of DM/Offset are (periodic, distributed).

If the design matches the its assumptions, DM/Offset assigns intermediate deadlines $\{D_{ij}\}$ (e.g., D_{11}, D_{12} and D_{13} in Figure 1) for the subtasks $\{T_{ij}\}$ of each task T_i, and accounts for the worst-case network delay t_c. The first subtask T_{i1} has a start time at the beginning of its period and a deadline less than its period; the subsequent subtask have a static phased offset equal to the deadline of its previous component plus t_c. (The static offset requires delaying the release of a subtask T_{ij} if its predecessor T_{ij-1} finishes earlier than its deadline.) The deadline of the last subtask equals the deadline of the whole task. If every subtask T_{ij} meets its intermediate deadline, the whole task meets its deadline D_i. Consequently, the distributed schedulability analysis is reduced to the analysis of each node independently with phased offset. This phase offset policy is similar to the Modified Phase Modification Protocol [2] and a protocol described in [26].

For the schedulability analysis on each node, we employ Audsley's priority assignment and analysis algorithm found in [1]. The Audsley algorithm provides an optimal priority assignment and feasibility test algorithm for static priority tasks with arbitrary start times (phase offsets) on a single node. It is different from Rate Monotonic and Deadline Monotonic priority assignment schemes, which assumes that tasks must be released simultaneously, i.e., without considering the start times (phase offsets). The current DM/Offset analysis takes a simple approach that evenly divides the deadline of each task as the intermediate deadlines of its subtasks.

In our system task and hardware models are automatically determined by VEST itself. Other tools such as CAISARTS [8] are not linked to a design and analysis system, and commercial tools such as TimeWiz use simulation to analyze most types of distributed real-time programs.

5. Semantics and Correctness

The VEST tool does not support formal proof of correctness. Rather, its goal is to apply key checks and analysis to avoid many common and insidious cross cutting problems that might otherwise exist. However, VEST is based on various underlying semantics that support these various types of checks and analysis.

First, VEST is built with GME [12]. GME has explicit semantics for components, interfaces, relationships, dependencies, and constraints. Subsequent analysis and checks rely on this underlying semantics.

Second, VEST specifically implements the semantics of the Bold Stroke middleware in regards to tasks and events. This is the same as the ACE/TAO semantics [22]. VEST is application domain dependent so implementing domain specific semantics is necessary. For example, the current implementation described in

this paper focuses on the avionics domain as supported by Bold Stroke middleware [23]. VEST is also extensible and can easily implement other semantics from other application domains.

Third, aspect checks and prescriptive aspects collect data from the underlying model and apply interpreters to that data. For example, precise event semantics (of ACE/TAO) permit VEST to collect all suppliers and consumers of events and then perform checks such as determining if any events have no suppliers or consumers, or determine if cycles exist. Consequently, the correct implementation of the interpreters rely on the underlying semantics of GME and Bold Stroke.

Fourth, the VEST semantics for threads, hardware specification, events, resource assignments, etc. permit automatic creation of precise system-wide task set and hardware requirements including real-time requirements. The associated analysis supplied by VEST uses this specification and matches it with appropriate scheduling techniques. This provides correct schedulability analysis.

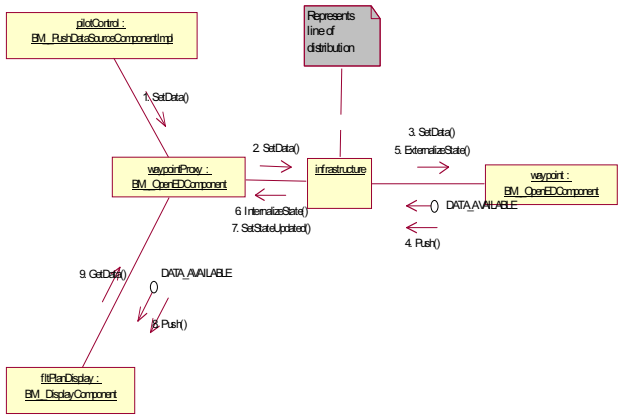

Figure 2. UML Diagram of a Pilot Control Subsystem

6. Case Study I: Composition and Analysis Scenario

The purpose of this case study is to demonstrate the effectiveness of the ideas incorporated in VEST. To do this we applied VEST to the design and composition of a portion of a distributed avionics system that is based on the Bold Stroke middleware. In this avionics system, a pilot control component measures coordinate data periodically, then sends its coordinate data to a waypoint control component. Upon receiving coordinate data, the waypoint control component calculates a new route for the plan, updates its database, and sends that new route to a display component. This avionic control system is a typical example of a distributed real-time embedded system with many crosscutting concerns. In fact, this example scenario is posted by Boeing as a good scenario for evaluating design and analysis tools. Figure 2 shows the UML diagram of the avionic system's software architecture.

To better understand the case study additional details about the application are provided: The system is composed of four first level components: pilotControl, waypointProxy, waypoint, and fltPlanDisplay. They run on the Bold Stroke middleware. The pilotControl component is an event supplier. It supplies coordinate data to the waypointProxy component at a specified frequency. WaypointProxy is a proxy representing the waypoint component and it runs on another processor. Communication is supported by the middleware service known as an event channel. Via the event channel, data originating in the pilotControl component is forwarded to the waypoint component. Likewise, the waypoint component sends the newly calculated route back to waypointProxy. Finally, the fltPlanDisplay component gets the new route information and displays it.

6.1. Design the Pilot Control Subsystem

In this case study, the developer first creates the system using abstract components. After the abstract specification has been performed, the system design might look as shown in Figure 3.

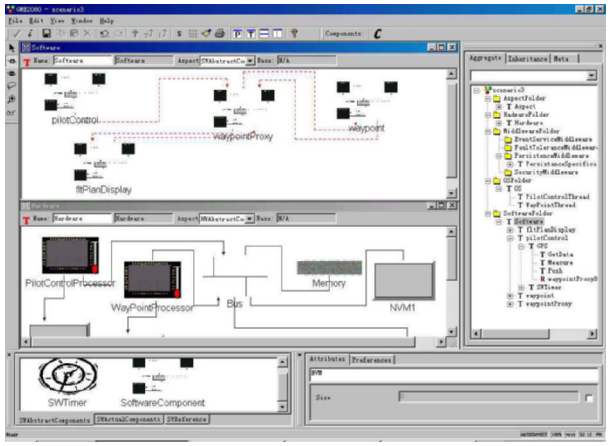

Figure 3. VEST model of a pilot control system

In the above diagram there are two layers shown. One layer is the software layer. This layer (see the top panel of the figure) has basic four components: pilot control, waypointProxy, waypoint, and fltPlanDisplay. The high-level interaction of these components are shown by the dashed lines. By high-level interaction we mean that if there is any event propagation from one component to another then these components are connected by an arc.

Direction of a connection shows the flow of events. A second canvas in the picture shows the hardware layer. In this example, the system is deployed in a distributed environment. It contains two processors: a pilot processor and a waypoint processor. They are connected via a bus interconnect. Also, the system has two non-volatile memory units and one volatile memory unit. What are not shown in the diagram are the OS, Aspect, and middleware layers. The components of these layers can be viewed from the browser menu shown on the right-hand side of the figure. In the OS layer, we have two threads: a Waypoint thread and a PilotControl thread. The waypoint thread is mapped to the waypoint processor and the pilot control thread is mapped to the pilot control processor. The components that run on the waypoint thread are Pilot Control, WaypointProxy and fltPlanDisplay.

The persistence service of Bold Stroke is one focus of this case study. Every application component that needs to maintain persistent data needs to create a persistence adapter that set the follow attributes of the persistence service: save_rate, is_double buffered, and track_dirtiness. The save rate specifies the frequency of the persistence thread. Is_double_buffered identifies whether the state should be saved twice or not. Track_dirtiness is a boolean variable; if true, this parameter causes the state to be persistent if the persistent object is dirty.

Double clicking on the software components shows the methods and member variables modeled in this component. An event graph is specified at this view (VEST models systems at the method level). The specification of the method-calling graph helps in completely characterizing the systems execution and thereby provides needed data for VEST to perform interface checking and schedulability analysis. After performing these operations the developer chooses actual components from the libraries and maps them to these abstract components. After modeling, the VEST developer makes various checks to boost his confidence in the correctness of the system.

6.2. Memory Footprint Check

In this case study, the first checks performed are intra-component checks. For instance, enough memory is vital for the system's performance. A memory footprint check is available in VEST. The first part of the memory footprint check is concerned with main memory. It sums the memory needed by all the components in the system, and all the available physical memory (RAM) provided by the hardware, and check if there is enough physical memory in the system. In the case study, the developer initially specified the system as shown in the following table:

	Pilot Control	Waypoint Proxy	Waypoint	fltPlan Display
max foot - print	50M	100M	300M	100M

However, the hardware memory is only of size 500M. Considering the system overhead, the memory check informs the developer of insufficient memory. The developer either adds more memory, or reduces memory consumption by modifying application components.

The second part of the memory check deals with NVRAM (e.g., EEPROM). Bold Stroke allows application programs to specify a set of data in some components to be persistent, so that important data in the system survives power failures. For the system to function correctly, sufficient NVRAM for persistent components should be provided. Our check assures the developer when there is enough non-volatile memory to meet the system's requirement, or gives warning when not enough NVRAM is provided. In this case study, the system has two NVRAMs with a total capacity of 300 MB. The sum of the persistent objects' size is 200 MB. The persistent object is originally configured as double-buffered, which doubles the needed capacity of NVRAM to 400 MB. When invoked, the memory footprint check warns that there is insufficient NVRAM. In this case study, the designer now reconfigures the persistence adapter to single-buffered mode, and the memory check returns successful confirmation. While these checks are trivial, they are useful and demonstrate a simple cross cutting constraint.

6.3. End-to-End Schedulability Aspect Check

The developer may then proceed to make additional checks that are more sophisticated. VEST provides an automatic schedulability analysis. After the designer completes the design of the model, he runs the schedulability analysis to check the model. This analysis requires the DM/Offset analysis because the software components are mapped to multiple interconnected processors in the model. However, the output of the schedulability analysis shows that the model is not schedulable, as depicted in the following chart. The output contains a list of methods including the period and WCET of the methods in the CORBA components. Based on the event graph, multiple interacting methods on a same processor are grouped into a subtask, which is mapped to a thread. The second part of the output is the subtask list on each processor and its schedulability

analysis results. The subtask list includes the period, WCET, and the intermediate deadline and offset of each subtask. For the initial design with a period 400 ms, the analysis shows that processor 2 is schedulable, but processor 1 is not. Therefore, the design should be changed.

List of methods
MethodName MeasureLocation Processor
Processor1 Period 400 WCET 67
MethodName Push Processor
Processor1 Period 400 WCET 2
MethodName Push Processor
Processor2 Period 400 WCET 4
... ...

Subtasks on Processor2
Subtask Push Processor 2 Period 400 WCET 4
Deadline 160 Startime 81
Subtask CalculateRoute Processor 2 Period 400
WCET 2 Deadline 320 Startime 241
Priority level 2 has been assigned to Push.
Priority level 1 has been assigned to CalculateRoute.
Schedulability test on Processor2 passed.

Subtasks on Processor1
Subtask MesureLocation+Push+GetData Processor 1
Period 400 WCET 102 Deadline 80 Startime 0
Subtask DataReadyPush+Push+GetData+Display
Processor 1 Period 400 WCET 11 Deadline 240
Startime 161Subtask GetData Processor 1 Period 400
WCET 2 Deadline 400 Startime 321
Couldn't assign, try next
Priority level 3 has been assigned to
DataReadyPush+Push+GetData+Display.
Couldn't assign, try next
Priority level 2 has been assigned to GetData.
Couldn't assign, try next
Schedulability test on Processor1 failed.

Output of the schedulability check on the original pilot control subsystem with a period of 400 ms

6.4. Prescriptive Aspects

As part of the VEST tool, the designer can use a prescriptive aspect to change the design. To make the system schedulable, the developer applies the following prescriptive aspect to relax the period of each component from 400 ms to 600 ms.

```
for *.Period=400
  change *.Period=600
```

After applying the prescriptive aspect (assuming that this change is compatible with the semantics of the application), the designer runs the schedulability analysis again, which as it turns out succeeds on both processors this time. The output is not shown due to space limitations.

As another example, assume that every component has a notion of importance, whose value is [high, medium, low]. For the sake of fault tolerance, the developer would like to double buffer as many important components as possible. In order to do that, he uses a prescriptive aspect. The developer drags a prescriptive aspect into the system.

```
for SoftwareComponent.importance=*.[medium,high]
  change PersistanceAdapter.isdoublebuffered=true
```

Applying the above prescriptive aspect initiates a search for all software components. In the list of components, the prescriptive aspect interpreter looks for a property of type importance. The interpreter then tries to match the importance attribute of components to either medium or high, and if there is a success then it changes the persistent adapter associated with that component's property is_double_buffered to true.

Obviously this prescriptive aspect crosscuts multiple facets of the system. The developer wants to make sure that this aspect does not violate other specification for the system. He runs the interpreter to execute the prescriptive aspect, and the changes are made to related components. Then he runs the memory footprint check. It turns out that there are two persistent components of high importance (each has a size of 50M), and one persistence component of medium importance (which as a size of 100M). If all of these are double buffered then the memory requirements are now 400M. The physical non-volatile memory in the system is only 300M, so there is not enough non-persistent memory to meet the requirement of the prescriptive aspect he enabled. Upon receiving a warning from VEST concerning the lack of memory availability, the developer changes the prescriptive aspect to only double buffer the highly important components.

```
for Software.importance=high
  change PersistanceAdapter.isdoublebuffered=true
```

The above syntax is similar to the previous one except that the range of components to change is now narrower, limiting itself to only high importance components. The developer then re-executes the prescriptive aspect, and applies the non-volatile memory check again. The check passes successfully, since the changes in the software and middleware layers are in harmony with the hardware layer of the system.

This case study shows that VEST has many advantages. First, it provides a way to speed up code-test-debug cycle through various checks it implements. Second, it

makes components more reusable across multiple projects, because it allows configurablility within components (at middleware layer and software layer) so that they are easily reusable, and it provides a library for a user to navigate and choose components. Third, VEST also gives users a higher confidence in the correctness of system operation. Checks such as buffer sizing, memory sizing, and schedulability analysis reduce a user's effort to achieve confidence. Fourth, the use of prescriptive aspects in a system makes it easy to extend/contract a system's capabilities with global wide changes being performed automatically, avoiding errors of forgetting to change one or more locations.

7. Case Study II: Measurement of Composition Time

We performed a second case study to measure the benefits of VEST 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. This experiment was accomplished in a very limited situation. One expert from Boeing performed the experiment using their current approach, and one grad student performed the experiment using VEST. For each person we timed the various steps involved with this 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 discuss how they might generalize to a larger experiment.

7.1. Experimental set-up

The experiments used Boeing's MOBIES OEP2.1 Product Scenario 3.2 (PS 3.2) as a target system to be composed. The scenario represents that portion of an avionics system that displays waypoint and radar data and is published by Boeing as a typical subsystem to facilitate research that is applicable to real world problems. The waypoint data can be changed by the pilot and the radar data is produced at a 5hz rate by the radar device. The sensor coordinator notifies each logical sensor of when its data should be updated.

This scenario is initially triggered by an interval timeout that is consumed by the pilotControl component. Upon receipt of this event, the pilotControl pushes data to the waypointProxy via the Set operations in the proxy's facet. The waypointProxy then forwards this call via the Infrastructure component to the waypoint component. The waypoint then updates its state and issues a *Data Available* event. This event causes the Replication Service to extract the state from the waypoint and send it to the waypointProxy. The waypointProxy internalizes this state and issues its own *Data Available* event. The proxy's event is consumed by the fltPlanDisplay component that gets the data from the proxy and displays it.

The baseline toolset for comparison includes Rational Rose [17] and Quantify both of which are currently used in Boeing's product development. The UML models of all Bold Stroke components were available in Rational Rose before the experiment started. The worst-case execution times (WCET's) of all used components were also available in the library before the experiment started. An expert at Boeing used the following process to compose PS 3.2:

1. Design PS 3.2 by integrating the UML models of existing components in Rational Rose.
2. Implementation: Program the design by connecting existing Bold Stroke components in C++ through the Bold Stroke event service.
3. Testing: Run the implemented system to check for timing violations. If any timing violations are detected, go back to step 1; Otherwise, the composition is completed.

At UVA, a graduate student familiar with VEST used VEST to compose the same product scenario. The VEST experiment included the following steps:

1. Design PS 3.2 in VEST using component libraries.
2. Scheduling analysis: Run the VEST scheduling tool to assess the schedulability of the design (without implementing the system). If the analysis shows that the design is not schedulable, go back to step 1. Otherwise, go to step 3.
3. Implementation: Program the VEST design.

Both VEST and the baseline experiments included two iterations of composition. Initially, the system was designed on a single-processor platform. Since the single-processor design turned out to be unschedulable, a new composition was needed. A new processor was added to the system and a distributed version of PS 3.2 was composed by moving several components to the new processor. The distributed version was found to be schedulable. The VEST scheduling tool can automatically identify the applicable scheduling analysis that matches the system characteristics. Maximum Urgency First (MUF) scheduling analysis was automatically invoked for the single-processor design,

and the DM/Offset scheduling analysis was automatically invoked for the distributed design.

7.2. Experimental Results

Table 1: Measured Time with VEST and Baseline in Case Study II

VEST			Baseline		
Step		Time (min)	Step		Time (min)
V.1.1	Design: single processor	40	B.1.1	Design: single processor	25
			B.1.2	Implement: single processor	75
V.1.2	Scheduling analysis: single processor	1	B.1.3	Test: single processor	30
V.2.1	Design: distributed	25	B.2.1	Design: distributed	90
			B.2.2	Implementation: distributed	105
V.2.2	Scheduling analysis: distributed	1	B.2.3	Test: distributed	20
	Implementation: distributed	105			
Total Composition Time		172	Total Composition Time		345

We measured the total composition time as well as the time that each step took in both experiments. The results are summarized in table 1. We used *X.i.k* to represent the k^{th} step in the i^{th} iteration of the *X* experiment, where *X=V* refers to the VEST experiment and *X=B* refers to the baseline experiment.

Our measurement showed that VEST effectively reduced the total composition time of PS 3.2 by 50%. Analyses on the time spent on each step shows two key advantages of VEST compared to the baseline:

- Reduce the rounds of implementation: Scheduling analysis enables VEST to drop wrong (unschedulable) designs without implementing the system. In this case study, the scheduling analysis showed that the single-processor design was unschedulable. Hence, the VEST user avoided implementing the single-processor composition (Step B.1.2) and saved 75 min. Compared to the baseline, this reduced the total composition time by 22%. Note also that in VEST the scheduling is a rigorous analysis and in the standard approach it is only done via testing which is more error prone.
- Replace time-consuming testing with quicker analyses: Two schedulability analyses (Steps V.1.2 and V.2.2) in VEST took a total of only 2 minutes, compared to a total testing time of 50 minutes (Steps B.1.3 and B.2.3) in the baseline experiment. This saved the VEST user 48 min and reduced the composition time by 14% compared to the baseline.

While this case study focused on the scheduling part of VEST, we should note that both of the above benefits are also applicable to other aspect checks of VEST.

8. Additional Lessons Learned

One set of lessons learned deals with the modeling experience. Graphical modeling of complex systems, while very useful, still results in many human errors. In general, a model stabilizes only after a number of re-designs. This includes many changes to high-level model components as well as fine tuning the properties of the components. Navigating and applying changes to these models are increasingly difficult as the models grow more complex. What is required are active components that make consistent and global changes. This is one value to our prescriptive aspects.

We also learned some additional things from the use of prescriptive aspects. The language that was initially designed could operate on properties of a set of objects. This was very useful in applying global and consistent changes to a class of objects based on types, properties or value. However, more capabilities are necessary. In particular, it is necessary to manipulate components based on their "relationships". One motivation for this is that it is sometimes necessary to adjust certain properties or location of components based on their relationship with other components. For example, one very practical use we found was that if a certain set of components were mapped to a periodic thread we found that prescriptive aspects based on relationships could propagate any changes in the period down the calling chain. This means that the designer only needs to deal with the thread and not all the components in that task.

We also found that automated dependency checks, even if simple, are very helpful. For example, making sure that there is enough memory and that all events have suppliers and consumers are simple, but necessary checks.

Another set of lessons involve the schedulability framework. The schedulability framework in VEST is not meant to implement a novel schedulability analysis, but rather to ensure that the proper analysis is used and to automatically identify the task model from the system

description. These features improve accuracy and reduce design time. In fact, we have found that it is possible to automate the many of the time consuming steps including many parts of the schedulability analysis and just as importantly re-analysis. For example, it is often necessary to re-design when one processor does not have sufficient capacity to meet deadlines. This requires new analysis and distributed task allocation. New analysis often requires a new algorithm as well because many of the uni-processor algorithms don't work for distributed systems. We also found that while there are a good number of solutions for distributed real-time scheduling, they are often (i) not supported in a tool, (ii) are not known by designers, and (iii) do not meet the requirements of the specific distributed programs actually being run. We also found that in some tools the task set characteristics are input directly, i.e., separately from the program design. Again, deriving the task set characteristics automatically, as in VEST, saves time and is more accurate.

Finally, it is obvious that designers of embedded real-time systems face many difficult problems. By working through various product scenarios with avionics designers we were able to identify the actual time it takes to perform various steps in the design process (for simple designs) so we can concentrate on automating the most time consuming parts [19]. In this paper we have shown that in one case study using VEST the designer saved over 50% of the time to design a system due to automation. We have also added an automatic task allocation module to automate another part of the re-analysis. The designers now saved over 67% of their time (this data was not reported in this paper due to space limitations).

9. State of the Art

The work described in this paper builds upon and integrates research from three main areas: component based design, aspect oriented programming, and design tools.

The software engineering field has worked on component based solutions for a long time. Systems such as CORBA [21], COM [13], and DCOM [14] exist to facilitate object or component composition. These systems have many advantages including reusability of software and higher reliability since the components are written by domain experts [23]. However, none of these systems have adequate crosscutting analysis capabilities. One exception is KNIT. KNIT [18] is an interesting composition tool for general purpose operating systems. This system is addressing a number of crosscutting concerns in composing operating systems. For example, they consider linking, initialization, and a few other dependencies. To date, it has not focused on real-time and embedded system concerns.

A promising line of research is Aspects Oriented Languages [10]. This work attempts to address complex crosscutting dependencies at the source code level. As mentioned above, we are extending this work to design time where errors can be found early.

An excellent tool that matches our goals quite closely is MetaH [27]. 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.

10. Conclusion

When building embedded systems from components [3, 7, 20, 25], those components must interoperate, satisfy various dependencies [5], and meet non-functional requirements. The VEST toolkit can substantially improve the development, implementation and evaluation of these systems. The toolkit focuses on using language independent notions of aspects to deal with non-functional properties, and is geared to distributed embedded system issues that include application domain specific code, middleware, the OS, prescriptive aspects, and the hardware platform. The VEST tool has been implemented and used on three case studies, two of which are described in this paper. The case studies (i) qualitatively demonstrate the benefits of our tool and (ii) include quantitative data that show a savings of over 50% in design and analysis time. Overall, a main advantage of our tool is that it has the potential to address the most difficult parts of component composition, the hidden crosscutting dependencies including overall, distributed real-time analysis. The next step is to incorporate VEST into a top-to-bottom requirements specification and design methodology. This step is underway. Currently, a beta version of VEST has been delivered to Boeing. The plans are to make VEST (including a user's manual) available by the summer 2003.

Acknowledgments

We thank Dave Sharp and Mark Shulte of Boeing for their help in understanding the case study and for their comments and suggestions on the features found in VEST. We also thank Akos Ledeczi and others at Vanderbilt University for making GME available for this research.

11. 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] 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.

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

[6] 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.

[7] Hatcliff, J., et. al. (2003) Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems, ICSE, to appear.

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

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

[10] Kiczales, G., Hilsdale, E., Hugunin , J., Kersten , M., Palm , J., and Griswold, W. (2001) Getting Started With ASPECTJ, *Communications of the ACM*, 44(10).

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

[19] Santarini, M, Cadence Says Platform Halves Verification Time, EEdesign, Feb. 24, 2003.

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

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

[22] 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).

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

[24] Stankovic, J., Zhu, R., Poornalingham, R., Lu, C., Yu, Z., Humphrey, M., and Ellis, B., VEST: An Aspect-Based Real-Time Composition Tool, University of Virginia TR-CS-2003-07, March 2003.

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

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

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