Software architecture classification for estimating the cost of COTS integration.

Daniil Yakimovich
University of Maryland
Computer Science Department
University of Maryland
College Park, MD 20742, USA
1-(301)-405-2721
dyak@cs.umd.edu

James M. Bieman
Colorado State University
Computer Science Department
Colorado State University
Fort Collins, CO 80523, USA
1-(970)-491-7096
bieman@cs.colostate.edu

Victor R. Basili
University of Maryland
Computer Science Department
University of Maryland
College Park, MD 20742, USA
1-(301)-405-2668
basili@cs.umd.edu

ABSTRACT[i]

The use of commercial-off-the-shelf (COTS) products creates a software integration problem, whether a single COTS software component is being integrated into a software system, or the whole system is being built primarily from COTS products. This integration may require considerable effort and affect system quality. A good estimate of integration cost can help in the decision of whether or not to use a COTS solution, the selection of the best COTS products, and determine the amount and type of glueware that needs to be built. In this paper, we introduce a set of variables that have the potential to estimate the integration cost. We present a classification scheme of software architectures with respect to the integration of COTS products. The scheme is based on inter-component interactions within software architectures. The classification scheme allows the comparison of integration costs of different COTS products relative to different software architectures.

Keywords

COTS integration, software architectures, cost estimation.

1. INTRODUCTION.

Commercial-off-the-shelf (COTS) software products are widely used now in software development [9], and their usage should increase quality of the product and reduce the time of its development. However, COTS products often require significant effort for their integration into a system [4]. Software system architecture is one factor that affects integration cost [5]

The architecture of a software system is defined by its components and interactions between them, where components are things such as servers, databases, filters, etc. [10]. We can also consider computational units at a lower level such as procedures, objects, modules to be components. A COTS product can consist of one or many components. Interactions between the components can be simple, such as calling procedures or shared variables, or complicated, such as data-base protocols [10]. Besides conventional architectural styles such as main program and subroutines, OO systems, interpreters, etc. [10], a number of architectures have appeared recently to facilitate the integration of externally created components. Example architectures include industrial standards such as the Component Object Request Broker Architecture (CORBA) [1], Common Object Model (COM) [3], and experimental architectures such C2 [7].

When a software component is integrated into a system, it must support the style of the interactions of the system's architecture in order to work together with other components. If a COTS product has another style of interactions, programmers must write integration software to allow this product to interact with other components of the system. An overview of integration techniques can be found in [9]. Most of the techniques either change the component being integrated, or create wrappers or adapters, which are special software that support interactions between the component and its environment. Since most COTS products can not be changed by users because of absence of source code and other reasons, the integration of COTS products is usually performed by glueware. Glueware is integration software; it

provides the proper interface for a component being integrated and serves as a mediator for its interactions with other components.

Integration work increases the cost and development time of the system. Moreover, changing the COTS components, or adding glueware can lower system quality. An architectural style and COTS products with a close match can minimize integration work. In this study we use a set of variables to estimate the distance between architectures and components. These distance variables are ordinals; they provide relative distances between a COTS product and system requirements. Distance variables may be independent with independent and possibly conflicting orderings. This distance can be used to estimate integration effort from an architecture description and a COTS product specification.

In section 2 we discuss the problem of interactions between components and their environment and the scope of our study. Section 3 gives the dimensions of interaction assumptions. Section 4 discusses types of interactions for different architectural styles and their classification. Section 5 describes the approach for estimating the integration cost based on the architecture classification. Section 6 contains the conclusions.

2. INTERACTIONS AS THE ORIGIN OF THE INTEGRATION PROBLEM.

Integration problems arise when a component depends on certain assumptions concerning its interactions with its environment, but is to be placed into a system that is based on different assumptions. The result is interaction protocol mismatches. We define four types of interactions:

Component-platform interactions. A component must be executed somewhere. It can be either a real processor and an operating system for binary executables, or a virtual one. If an executable program was compiled for one type of CPU, it will need an emulator or a code converter in order to run it on another CPU.

Component-hardware interactions. A component can interact directly with hardware writing-reading from ports. If the port's numbers are different from what is expected by the component, the component must undergo some modification.

Component-user interactions. A component's user interface requirements may also change. For example, a component can have its messages in one language, when the system requires another language.

Component-software interactions. A component almost always interacts with other software components, and there can be mismatches between the components. A set of possible mismatches between components [9]: representation, communication, packaging, synchronization, semantics, control, etc.

Although all four types of interactions can cause problems for a component reuse and must be overcome, the main concern of this study is component-software interactions.

3. ASSUMPTIONS ABOUT INTERACTIONS BETWEEN SOFTWARE COMPONENTS.

Consider the integration of COTS components on the architecture level and their interactions with other components of the system. Every component is designed with assumptions concerning its interactions, and the assumptions strongly depend on the particular architecture. For example, some architectures have only one control thread, e.g., main program and subroutines, and their components depend on the control structure. In contrast, distributed architectures such as client-servers have components that are independent processes. When a component to be integrated has different control assumptions from the system's, development effort is required to overcome the differences in the interaction protocols.

The following classification of architectural assumptions that can cause mismatches in inter-component interactions is given in [6]:

1. Assumptions about the nature of the components

1.1 infrastructure - the substrate on which the component is built;
1.2 control model - assumptions about which component controls the sequencing of computations;
1.3 data model - assumptions about the way the environment will manipulate data managed by a component.
2. Assumptions about the nature of the connectors
2.1 protocols - assumptions about the patterns of interaction characterized by a connector;
2.2 data model - assumptions about the kind of the data that is communicated.
3. Assumptions about the global architectural structure. These include the particular topology of a system.
4. Assumptions about the construction process, in what order pieces are instantiated and put together.

We use the following variables to represent assumptions about inter-component interactions: component packaging, type of control, type of information flow, synchronization and binding. Other issues that are difficult to measure but can have a considerable impact on integration are syntax and semantic of interaction protocols, the topology of the system, and whether COTS products and the system can be modified or not. We also have to compare different values of these variables with respect to compatibility of the assumptions.

Trivially, if two components have the same assumptions on an aspect of interaction, the values of the respective variables are equal. When the assumptions are different, two situations are possible. First, one assumption may still be compatible with another. For example, with respect to packaging, a component can be implemented in ANSI C, and another component can be implemented in Borland C, which is an extension of ANSI C [2]. In this case, the first type of packaging is compatible with the second one, because all Borland C compatible compilers will compile programs written in ANSI C. Thus, if a system is developed in Borland C, and a procedure is written in ANSI C, it is safe to use this procedure. If the system is developed in ANSI C, and the procedure uses some Borland C specific functions, then the whole system must be recompiled with a Borland C compiler. In this case, the value of the packaging variable of the Borland C component is greater than the value of the ANSI C component's

packaging variable. We treat all assumptions in this way: if assumption A is compatible with assumption B, then the respective variables are comparable, and A ≤ B. Some assumptions are incompatible, for example data flow interactions and control flow interactions. In this case the values representing these assumptions are incomparable. So the values of the variable have partial orders. Figures 1 through 5 depict orderings of the variables with arrows going from greater values to lesser ones.

Component Packaging: how a component is packaged for integrating into a system. Two big classes of packaging types can be defined: linkable and independent components. These classifications represent the values of the component packaging variable:

Linkable components, e.g., object modules, class libraries, function libraries, etc. These components must be linked together with other system components in one executable program. They can be represented either in source code or as object files.
Independent components, e.g., overlays, dynamic linkage libraries, independent programs, etc., that need not to be linked with other components. The exact ordering of the values mainly depend on the compatibility between different language dialects and object file formats. The values for independent components are greater than values of linkable components, because it is easier to convert a linkable component into an independent one. For example, it is possible to create an independent component from a library component by just putting it into a wrapper program, or it is possible to convert an object module into a dynamic linkage library.
The values for independent stand-alone programs are greater than values for other independent components such as overlays and dynamic linkage libraries because the latter can be put into wrapper programs and thus converted into independent programs (Fig. 1).

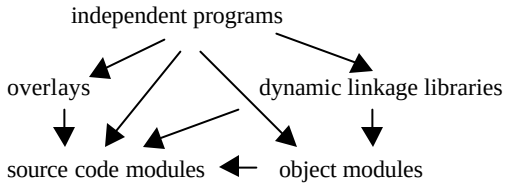

Figure 1: Ordering of types of packaging

Type of Control: how a system provides control flow to its components.
Centralized control. Components in systems such as single programs written in programming languages without concurrency must assume the existence of only one control thread, which is passed from one component to another.
Decentralized control. Components in systems with concurrent processes or/and multiple threads inside of one program can have their own control.
No control assumption. Some components, such as libraries, do not make any assumptions on the control and can be used in systems with any type of control.
Decentralized control is more general than centralized control. A component from a centralized control system may be wrapped inside of an independent process and used in a system with decentralized control. But it is difficult, if possible, to deprive an independent process of its control structure putting it into a system with centralized control (Fig. 2).

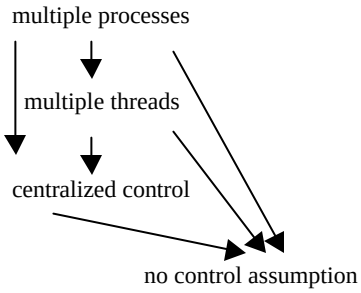

Figure 2: Ordering of types of control

Information Flow: what type of information flows between components: data, control, or mixed.
Control flow interactions that invoke some routines of components such as procedure calling, remote procedure calling, etc.
Data flow interactions that use exchange data between components (shared memory, message passing, etc).
Mixed data-control interactions that can accept data from one component and convert it into control for another component, and vise versa.

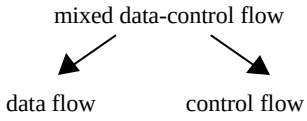

Figure 3: Ordering of types of information flow

Synchronization: whether or not a component blocks when waiting for a response during an interaction with another one.
Synchronous, a component suspends its execution after sending a request until receiving response.
Asynchronous, a component proceeds further after sending a request without blocking itself.
Asynchronous calling can easily simulate synchronous by using a loop which waits for a response, but it is much more difficult to do the opposite. So, asynchronous interactions are compatible with synchronous ones (Fig. 4).

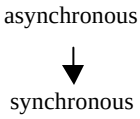

Figure 4: Ordering of the types of synchronization

Component Binding: how components are attached to connectors, and how the participants of an interaction are determined.
Static binding. When a component initiates an interaction it will interact with the same component every time. Programs written in procedural languages use this type of binding between the components.
Dynamic or late binding. The target component of an interaction does not depend on the component that initiates the interaction.

During compile-time it can depend on other factors, such as parameters of the procedure call (object-oriented systems), or the current system's topology (pipes-and-filters systems). In some architectures such as CORBA and SOM, a component itself can find during execution time, a specific component to interact with.
Ordering of the values depends on compatibility between particular types of bindings (e.g., between CORBA and SOM), but generally dynamic binding is compatible with static binding. It is possible to have several types of binding (*mixed binding*) and be compatible, e.g., a component can support static binding, dynamic binding C++, and binding of CORBA (Fig. 5).

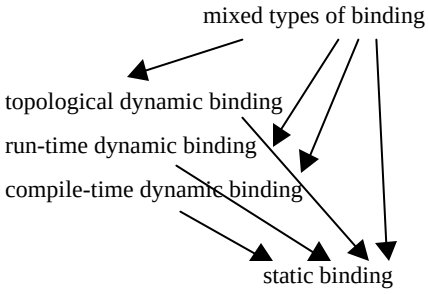

Figure 5: Ordering of the types of binding

There are other issues, such as syntax, semantic, system topology and permissible changes of components, which are not measured in our approach, although they can affect component integration.

4. A SOFTWARE SYSTEM CLASSIFICATION.

In the previous section we established a set of variables to characterize component interactions. Architectural styles and COTS products that have some common assumptions for their components, thus these variables can describe them, but not only single components. Here are some of common architectural styles [10] with their characteristics according to the set of variables. Often the exact value of a variable is not relevant for an architectural style, because it depends on a particular implementation. For example, depending on the programming language, operating system, and compiler, a call-and-return system can have only linkable components (a single executable program), or it may have independent components such as overlays or dynamic linkage libraries (DLL).

Pipes and Filters. The components (filters) have inputs and outputs, they read data from their inputs, transform it and send through the outputs to other filters. The connectors (pipes) connect outputs and inputs of filters.
Packaging: not relevant, independent UNIX processes that can be pipelined together, although modules in one program can use data flows.
Control: not relevant (e.g., UNIX pipeline work sequentially, but generally the filters can be concurrent).
Information flow: data.
Synchronization: not relevant.
Binding: dynamic (determined by the particular system's topology).

Main Program and Subroutine. The components are procedures and functions, and the connectors are calls between them. Most programs written in procedural programming languages fall into this category.
Packaging: not relevant, usually all routines are linked together in one executable program, but overlays and dynamic linkage libraries are stored separately.
Control: centralized.
Information flow: control.
Synchronization: synchronous.
Binding: static.

OO Systems. The components are objects. The connectors are invocations of methods or member functions of the objects.
Packaging: not relevant, usually all routines are linked together in one executable program, but overlays and dynamic linkage libraries are stored separately.
Control: usually centralized.
Information flow: control.
Synchronization: synchronous.
Binding: dynamic, depending on the exact type of objects, which is usually resolved using virtual function tables.

Communicating Processes. The components are independent processes, the connectors are either messages they exchange or other means of process communication (shared memory, remote procedure calls, etc.)
Packaging: not relevant, a single program can have several threads of control, or several independent programs can be executed as concurrent processes.
Control: decentralized.
Information flow: not relevant, remote procedure calling can be used for control flow, and message passing can serve for data exchange.
Synchronization: not relevant.
Binding: not relevant.

Event Systems. Unlike the conventional communicating processes, in event systems the components interact via implicit invocations. A component can register for an event and assign a procedure to be called in case of its announcement, and when the event is actually announced in the system, all assigned procedures are called for all components registered for this event.
Packaging: not relevant.
Control: decentralized.
Information flow: control (implicitly invoked routines).
Synchronization: not relevant.
Binding: dynamic.

Blackboards. The components are the blackboard that stores the current state of the system and other components that have access to it. Actions of these components are triggered by specific states of the blackboard.
Packaging: not relevant.
Control: not relevant.
Information flow: data.
Synchronization: not relevant.
Binding: static, everything is attached to the blackboard.

Besides these common architectural styles, there are some specific architectures, specially designed for integrating components,

Component Object Model (COM), Common Object Request Broker Architecture (CORBA), Chiron-2 (C2), etc. Below we characterize some of them.

C2 Architectural Style [7]. C2 is actually an OO framework, but unlike many frameworks it depends on fewer assumptions due to its system of wrappers and domain translators. It does not assume component homogeneity (some of frameworks can integrate components written on a certain programming language), message synchronization (many frameworks support only synchronous message passing, with waiting for the reply), shared address space (no global variables needed), single thread control (the components can have their own control thread, that differentiates C2 from, maybe, all other frameworks). So the C2-style architecture has fewer limitations than typical OO frameworks. Still, C2 requires message-based communication (which is semantically rich) for its components, and also provides a wrapper when a component does not support this style of interactions. The C2 architectural style was specially designed for reusing COTS products, which is why it makes so few assumptions about component interactions. A limitation of C2 is its layered structure, and although a component in C2 is not aware of components below it, it sends requests to the components above it, and all components must fit into the overall layered structure. C2 can be considered a layered data-flow architecture.
Packaging: depends on whether the language is supported by C2.
Control: all types.
Information flow: data (message passing)
Synchronization: all types.
Binding: dynamic, defined by the system's topology

Common Object Request Broker Architecture (CORBA) of Object Management Group [1] is an industrial standard for distributed object-oriented architectures. CORBA allows objects created in different languages and platforms to send messages each other, and it has some additional capabilities.
Packaging: depends on whether the language is supported by CORBA.
Control: decentralized.
Information flow: control (remote procedure call).
Synchronization: all types.
Binding: run-time dynamic, the Naming Service of CORBA allows clients to find objects based on their names, the Trading Service allows clients to find objects based on their properties.

Component Object Model (COM) of Microsoft Corporation [3] is a binary and network standard and a supporting system allows objects to make remote procedure calls. Objects using COM can be implemented in any language and they can have multiple interfaces. COM helps an object-client to establish connections with an interface of object-server.
Packaging: depends on whether the language is supported by COM.
Control: decentralized.
Information flow: control.
Synchronization: all types.
Binding: run-time dynamic, COM function QueryInterface asks an object whether it has a specific interface, after that a decision on interaction can be based on the result of QueryInterface. This allows it to establish interaction only with a component that has a desired interface.

5. ESTIMATING INTEGRATION COST.

Interaction assumptions of single components and software architectures can be represented by an interaction vector V = (P,C,I,S,B) where the variables P,C,I,S,B are respectively packaging, control, information flow, synchronization and binding. The values of these variables and their orderings are as discussed above. Now we compare two interaction vectors V1 and V2; this comparison reflects mismatches between the architectural assumptions with these vectors:

If some components of V1 are incomparable with the respective components of V2 then V1 and V2 are incomparable (V1~V2).
If all components of V1 are equal to the respective components of V2 then the two vectors are equal (V1=V2).
If all components of V1 are greater than or equal to then the respective components of V2 then V1 is greater than or equal to V2 (V1≥V2).
If all components of V2 are greater than or equal to the respective components of V1 then V2 is greater than or equal to V1 (V2≥V1), or V1 is less than or equal to V2 (V1≤V2).
If some components of V1 are greater than those of V2 and some components of V1 are less than those of V2 then V1 and V2 are incomparable (V1~V2).

To estimate the integration costs of a system and a COTS product we suggest the following procedure. First, the interaction vector Vs of the system architecture and the interaction vector Vp of the COTS product are found using the above classification. Then we compare Vs and Vp. Several outcomes of the comparison are possible:

Vs=Vp. The architectural assumptions of the system and the product match, so integration must consider only syntax and semantic of their interactions. Therefore integration is respectively cheap.
Vs≥Vp. Some assumptions are different, but those of the COTS product are still compatible with the assumptions of the system. Like in the previous case not much integration is needed. Since the COTS product will work under architectural assumptions of the system after being integrated in it, this direction of compatibility is acceptable.
Vs≤Vp or Vs~Vp. Some assumptions of the COTS product are not compatible with the assumptions of the system. For example a COTS Borland C procedure may be used in a program written and compiled for ANSI C. In this situation integration work must be done to overcome the architectural mismatches. Here, much will depend on whether changes are permitted for the COTS products. If yes, then it might be possible to modify the COTS product so that it becomes fully compatible with the system. In this case the cost of the COTS product modification must be estimated. If no changes are allowed, then either the architecture of the system must changed, or some glueware for the COTS product integration must be written. To estimate the integration cost find an architecture such that both the system and the product are compatible with it. To achieve this find a vector Vc such that Vc≥Vs and Vc≥Vp. The vector Vc is a common upper element of the vectors Vs and Vp. Apparently, if Vs≤Vp then it is possible that Vc=Vp. This will guarantee that all components of the system and the product can interact inside of the architecture with the interaction vector Vc. If the whole system architecture is modified, then we must estimate the cost of the modification according to the distance between Vc and Vs and the distance

between Vc and Vp. However, it can be sufficient to write glueware rather than changing the whole architecture. Again, the glueware must have properties defined by Vc and the costs depend on the distance between Vc and Vs and the distance between Vc and Vp. To minimize the integration cost, it might be necessary to find the minimal common upper element Vc' for Vs and Vp in the space of interaction vectors. This means that no Vc'' exists such that Vc''≥Vs and Vc''≥Vp and Vc'≥Vc'', except for Vc''=Vc'.

We demonstrate our approach using an example. Consider a real-time software system consisting of independent processes, and the processes are programs implemented in Ada which interact with themselves by sending asynchronous messages. The processes know each other by fixed names, so their binding is static. So, the system has the communicating processes architecture. The problem is to attach a 3-D graphics engine to this system. A COTS object-oriented library, such as QuickDraw 3D (QD3D) [11] has been selected for the engine. Its functions can be called from C/C++ only and it can not be used directly in the processes (different packaging assumptions of the processes and the library). The integration cost of the library is estimated here using the proposed approach. The interaction vectors of the system, the library and their minimal common upper element are given in the table below (Table 1).

Variables	**The System**	**QuickDraw 3D**	**The Minimal Common Upper Element**
Packaging	Independent programs	C++ class library	Independent programs
Control	Multiple processes	No assumption	Multiple processes
Information flow	Data, message passing	Control, method invocation	Mixed control-data
Synchronization	Asynchronous	Synchronous	Asynchronous
Binding	Static	Dynamic	Dynamic

Table 1: The interaction vectors of a real-time Ada system, QuickDraw3D and their minimal common upper element.

Variables	**Processes**	**OpenGL**	**The Minimal Common Upper Element**
Packaging	Ada programs	Ada function library	Ada programs
Control	Centralized	No assumption	Centralized
Information flow	Control, procedure calling	Control, procedure calling	Control, procedure calling
Synchronization	Synchronous	Synchronous	Synchronous
Binding	Static	Static	Static

Table 2: The interaction vectors of Ada programs, OpenGL, and their minimal common upper element.

We define as follows the minimal common upper element:

- *Packaging*. Since it is impossible to link the library to the processes; it must be put into a separate wrapper-program that will work essentially as a driver of the library.
- *Control*. The library driver can work as an independent process in the system, it can have a closed loop of control, in which it reads messages from processes-clients, translates them into calls for the library objects and sends the results back.
- *Information flow*. The information flow type in the resulting system must support both message passing for the processes and method invocation for the library. The processes can send messages to the driver in some data form with the object name, method name and parameters of the invocation encoded in a predefined format. The driver will translate this message and execute the proper method invocation. It will then send back the result again in the data format.
- *Synchronization*. The interactions in the system are generally asynchronous, meanwhile the library assumes that calls to its objects are synchronous. This conflict is not a problem, because asynchronous interactions of the system are more general than the synchronous ones of the library.
- *Binding*. Although the processes know each other by their names, the objects of the library still require dynamic binding. Consider a process that creates several windows for its output using the library. Whenever the process wants to use one of these windows it must send a message to the library driver, which still must contain a descriptor of the particular window. The descriptor can be a pointer to the window object or another label through which the library driver can find the object. Dynamic binding can complicate the messages sent to the library driver and can be a special concern for the system developers.

Thus the main problems the developers have to overcome in this example are making a wrapper for the library (the library driver) and implementing a special protocol to interact with it. The protocol must allow the processes to reference particular objects of the library. These issues can be evaluated further in order to obtain more precise effort estimation, for example a number of different types of messages to the driver according to the types of library functions, etc.

Another option for the developers is to take OpenGL [11] instead of QD3D. OpenGL is a function library, it has no object orientation but it can be called directly from Ada. That is why it is possible to integrate OpenGL not on the level of the whole system, like QD3D, but on the lower level of Ada programs, that has the main program and subroutines architecture. In this case we take the interaction vector of programs, and compare it with the interaction vector of OpenGL. (Table 2). The minimal common upper element of them coincides with the interaction vector of programs. That means that there is no architectural mismatch between the programs and OpenGL, and it can be easily integrated into the system.

Although the proposed approach does not directly give the amount of effort in units, such as staff hours, this example shows how it can point out major problems for the component integration and suggest solutions. Moreover, the information used in this approach can be easily obtained, making the approach relatively easy to apply.

6. CONCLUSIONS.

Although there is prior work on classifications of software architecture styles and architectural mismatches [6], [9], [10], we do not know of any scheme that connects particular architectures to particular problems with software component integration. In this work we tried to fill this gap and to create a method that would support estimation of the cost of integration of COTS products in various architectures.

Five variables were used for describing assumptions of component interactions. Their possible values were compared with respect to ease of mutual integration of components with these assumptions. A number of software architectures were characterized using these variables. The obtained partially ordered set can serve as a basis for estimating integration costs in different architectures by the relations of the set. Architectures that can easily integrate more types of components are higher in this order set. Architectures that can integrate fewer types of components because of their strict assumptions are lower. Moreover, these variables can help to estimate cost of integration of particular components into a software system by comparing their assumptions about interaction with those of the system's architecture. In turn, knowledge of the differences between the component and the system would allow the system developers to figure out what wrapper must be created for successful integration of the component. However, we do not claim that the suggested set of variables or their values used in this paper are exhaustive, they can be refined in the future.

Future research will attempt to validate this classification scheme and the estimation method empirically, using data from real projects with COTS product integration. The empirical data also will help to improve the method. Another direction of study might be adding other architectural styles to this classification. Finally, this classification scheme might serve as the foundation for a COTS integration methodology, which would be useful for software developers.

REFERENCES:

[1] Baker, S., "CORBA distributed objects using Orbix", 1997, Addison-Wesley.

[2] "Borland C++, Version 2.0 User's Guide", 1991, Borland International Inc.

[3] Box, D., "Essential COM", 1998, Addison-Wesley.

[4] Carney, D.J., Oberndorf, P.A., "The Commandments of COTS: Still in Search of the Promised Land", Crosstalk, May, 1997, pp. 25 – 30.

[5] Davis, M.J., Williams, R.B., Software Architecture Characterization, Proceedings of the 1997 Symposium on Software Reusability (SSR'97), Boston, USA, May, 1997, pp. 30-38.

[6] Garlan, D., Allen, R., Ockerbloom, J., "Architectural Mismatch or Why it's hard to build systems out of existing parts", Proceedings of International Conference on Software Engineering, 1995, Seattle, WA, USA, pp. 179 – 185.

[7] Medvidovic, N., Oreizy, P., Taylor, R.N., Reuse of Off-the-Shelf Components in C2-Style Architectures, Proceedings of the 1997 Symposium on Software Reusability (SSR'97), Boston, USA, May, 1997, pp. 190-198.

[8] Parra, A., Seaman, C.B., Basili, V.R., Kraft, S., Condon, S., Burke, S., Yakimovich, D. The Package-Based Development Process in the Flight Dynamic Division, The twenty-second Software Engineering Workshop, NASA/Goddard Space Flight Center Software Engineering Laboratory (SEL), Greenbelt, MD, December 1997, pp. 21-56.

[9] Shaw, M., Architectural Issues in Software Reuse: It's Not Just the Functionality, It's Packaging, Proceedings of the Symposium on Software Reusability, 1995, Seattle, WA, USA, pp. 3-6.

[10] Shaw, M., Garlan, D., "Software Architecture. Perspectives on an Emerging Discipline", 1996, Prentice Hall, Upper Saddle River, NJ.

[11] Thompson, T., "Must-See 3-D Engines", Byte, June 1996, pp. 137 – 144.

[i] This work has been supported by NASA grant NCC5170.