Connecting Research and Practice: An Experience Report on Research Infusion with SAVE

Mikael Lindvall[1], William C. Stratton[2], Deane E. Sibol[2], Christopher Ackermann[1], W. Mark Reid[3], Dharmalingam Ganesan[1], David McComas[4], Maureen Bartholomew[4], Sally Godfrey[4]

1: Fraunhofer Center for Experimental Software Engineering (FC-MD), College Park, Maryland

2: Johns Hopkins University Applied Physics Laboratory (JHU/APL) Space Department
Information Systems Branch Ground Applications Group (SIG), Laurel, MD

3: Johns Hopkins University Applied Physics Laboratory (JHU/APL) Space Department
Information Systems Branch Embedded Application Group (SIE) , Laurel, MD

4: NASA Goddard Space Flight Center Software Engineering Division, Greenbelt, Maryland

MLindvall@fc-md.umd.edu,
William.Stratton@jhuapl.edu, Deane.Sibol@jhuapl.edu, CAckermann@fc-md.umd.edu,
Mark.Reid@jhuapl.edu, dganesan@fc-md.umd.edu, David.C.Mccomas@ nasa.gov,
Maureen.O.Bartholomew@nasa.gov,
Sara.H.Godfrey@nasa.gov

Introduction

NASA systems need to be highly dependable to avoid catastrophic mission failures. This calls for rigorous engineering processes including meticulous validation and verification. However, NASA systems are often highly distributed and overwhelmingly complex, making the software portion of these systems challenging to understand, maintain, change, reuse, and test. NASA's systems are long-lived and the software maintenance process typically constitutes 60-80% of the total cost of the entire lifecycle [9]. Thus, in addition to the technical challenges of ensuring high life-time quality of NASA's systems, the post-development phase also presents a significant financial burden.

Some of NASA's software-related challenges could potentially be addressed by some of the many powerful technologies that are being developed in software research laboratories. Many of these research technologies seek to facilitate maintenance and evolution by for example architecting, designing and modeling for quality, flexibility, and reuse. Other technologies attempt to detect and remove defects and other quality issues by various forms of automated defect detection, architecture analysis, and various forms of sophisticated simulation and testing. However promising, most such research technologies nevertheless do not make the transition from the research lab to the software lab.

One reason the transition from research to practice seldom occurs is that research infusion and technology transfer is difficult. For example, factors related to the technology are sometimes overshadowed by other types of factors such as reluctance to change [8] and therefore prohibits the technology from sticking. Successful infusion

might also take very long time. One famous study showed that the discrepancy between the conception of the idea and its practical use was 18 years plus or minus three [7].

Nevertheless, infusing new technology is possible. We have found that it takes special circumstances for such research infusion to succeed: 1) there must be evidence that the technology works in the practitioner's particular domain, 2) there must be a potential for great improvements and enhanced competitive edge for the practitioner, 3) the practitioner has to have strong individual curiosity and continuous interest in trying out new technologies, 4) the practitioner has to have support on multiple levels (i.e. from the researchers, from management, from sponsors etc), and 5) to remain infused, the new technology has to be integrated into the practitioner's processes so that it becomes a natural part of the daily work.

NASA IV&V's Research Infusion initiative sponsored by NASA's Office of Safety & Mission Assurance (OSMA) through the Software Assurance Research Program (SARP), strives to overcome some of the problems related to research infusion. The Research Infusion initiative connects researchers with NASA projects who are willing to try new and promising technologies. Such research infusion projects are financially supported by NASA IV&V allowing both the project and the researchers to devote resources to infusing the new technology into the project. Such a project is a win-win for both the practitioners and the researchers because, for example, the practitioners get a solution to their problem, while the researchers get feedback on how well the technology fits the practitioners' needs.

In 2006, such a research infusion project involving Johns Hopkins University Applied Physics Laboratory (JHU/APL) and the Fraunhofer Center for Experimental Software Engineering Maryland (FC-MD), was successfully completed. The infused technology, Fraunhofer's Software Architecture Visualization and Evaluation (SAVE[1]) tool [2], facilitates documenting, communicating, and managing software architectures. The research infusion project was successful because it transitioned the SAVE technology to JHU/APL allowing the technology to be used as part of their regular software development process. It also helped improve JHU/APL's software architecture. In addition, the infusion project produced evidence that SAVE is applicable to software architecture problems in the aerospace domain, spawning a series of related research infusion projects. The project also led to the discovery of other needs that could not be addressed by current technologies and therefore spawned the research and development of a new technology that will be ready for infusion in the future. Thus, in this paper, we are able to describe two research infusion paths: 1) The *direct infusion path*, along which practitioners are increasingly willing to experiment with a new technology because of growing empirical indications that the technology is applicable and provide value in a certain domain; and 2) the *indirect infusion path* along which feedback and results from infusion attempts lead to the identification of needs that are not yet covered by the current technology. These needs can spawn new research projects that lead to new technology that can later be infused into other organizations and projects to address similar needs.

[1] http://www.theSAVEtool.com

The remainder of this paper describes the infusion projects and the two paths. We then describe the SAVE technology, which is the subject of the infusion, followed by the description of the infusion of SAVE at JHU/APL, which we can describe in more detail since the authors were very close to that project. We then describe how SAVE was applied to the other projects, which is done in less detail since it was applied without the researchers' direct involvement. The infusion projects are finally described by the newly started Dynamic SAVE project. Lessons learned throughout this project, a summary, and future work concludes the paper.

A summary of the infusion projects

We will now describe the various infusion projects as well as the two paths in more detail before we describe the infused technology and how it was applied.

The Direct Infusion Path

The initial infusion project

The initial infusion project was established between the Space Department Information Systems Branch Ground Applications Group (SIG) at JHU/APL and FC-MD. JHU/APL's Common Ground System (CGS), which at that time supported all of JHU/APL's NASA missions, was a 10 year old software system that had emerged from the work of at least three independent teams. To address the need to increase maintainability of CGS, Fraunhofer's SAVE technology was infused into JHU/APL's development and maintenance processes. Throughout the infusion project, FC-MD and SIG worked in close collaboration using SAVE to visualize, analyze, and suggest improvements of the CGS. The infusion team successfully achieved the stated goals for the project and documented their work in several papers and presentations.[10;11]. Advantages and disadvantages with the SAVE tool were noted, and the disadvantages resulted in ideas for the development of a new SAVE-related tool: Dynamic SAVE (DynSAVE), further described below as part of the indirect infusion path.

After the initial infusion project

After the infusion project was completed and the results were reported to NASA, SIG continued to use SAVE on their own. SIG, for example, applied SAVE to CGS to check the thread-safety of enhancements to existing CGS software [11]. During this analysis, FC-MD scientists were not involved, making this the first indication that the technology had indeed been transferred to SIG. Some of the reasons SIG could continue using SAVE on their own were 1) that there were enough "examples" developed during the infusion project that showed them how to use SAVE to address different questions regarding the CGS; 2) that the technology was mature and user-friendly enough to be used in a real setting, and that it provided enough "new" value to make it worthwhile to continue using it. For example, SAVE provides diagrams of the analyzed software that currently no other tool at JHU/APL can provide; 3) that the key people in the SIG were patient enough to continue on their own despite incomplete documentation and a somewhat unpolished technology.

SIG is now working on developing the ground system for the Radiation Belt Storm Probes (RBSP) mission and uses SAVE for that project also. SAVE has, for example, been applied to the first build of the RBSP ground software to ensure proper use of class interfaces. Actually, the SAVE analysis of CGS convinced JHU/APL that there was a need to rework the architecture to form a modern replacement for CGS implemented in Java, and the ground system for RBSP is the first step in towards that goal.

The technology spreads further at JHU/APL

Shortly after the initial infusion, JHU/APL's Information Systems Branch Embedded Application Group (SIE) started experimenting with SAVE. The SIE heard about the technology from their colleagues, for example at JHU/APL lunch seminars on new technologies, and got interested in using it for their purposes. SIE used a different strategy for infusion: they let a summer intern apply SAVE to some of their software in order to determine its applicability and value. The intern started using SAVE without FC-MD's involvement, and with limited involvement from SIG. Instead of consulting other people knowledgeable in SAVE, the intern mainly used the SAVE user manual as a guide for the work. As a matter of fact, a gap or incompleteness in the user manual was identified by the intern, which the intern addressed by adding a section to the user guide, which was fortunately shared with FC-MD.

The goal of the intern's work was to study how JHU/APL's flight software architecture could be engineered to use a framework for flight software provided by NASA Goddard called the *Core Flight Executive* (cFE) and the *Core Flight System* (cFS). The result of the SAVE analysis was presented at JHU/APL's First Flight Software Workshop (FSW) [6]. It should be noted that this project had mixed results from using SAVE in their context. This was probably due to various factors such as the lack of support from the researchers that stemmed from the intern's choice to use SAVE in an independent fashion. Another reason is probably that the technology was, at that time, not mature enough to provide the user with polished functionality. Since the researchers were not directly involved, they could not help the intern in finding ways around usability problems that occurred. The reasons SIE still could use SAVE independently were 1) that enough knowledge about how to use SAVE had been transferred to JHU/APL; 2) that the technology, despite encountered problems, was mature and intuitive enough, and that the documentation was instructive enough that SAVE could be used without much support; and 3) that the intern received strong support from SIE management to complete the evaluation. It should be noted that the feedback to FC-MD regarding the experiences with applying SAVE in this context was appreciated and very useful in improving the technology.

The technology spreads to Goddard

At JHU/APL's First Flights Software Workshop (FSW), the architect of the cFE/cFS gave a presentation on the cFE/cFS system [4]. The presentation on applying SAVE to JHU/APL's flight software, which actually included cFS, sparked some interest and prompted the cFE/cFS architect to ask questions about SAVE and its applicability to the cFE/cFS software. After some discussions, it was decided that a new infusion project

would be started with the goal of studying the architecture of cFE/cFS. This analysis was again made possible through the research infusion effort supported by NASA IV&V. This experience indicates that once a technology has been applied to a specific domain and good enough results from that application have been provided, other projects in that domain are also willing to experiment with it. The results from the cFE/cFS analysis was presented at the second FSW at JHU/APL in 2008 [1], which again spawned interest from other potential users of the technology. We will report on those experiences in a future paper.

The Indirect Infusion Path

New research

In parallel with infusing SAVE, as described above, we have started working on the research and development of a new technology: Dynamic SAVE (DynSAVE). While the existing (static) SAVE technology is useful for analyzing single software applications, additional information is necessary to allow for analysis of systems of systems in which such software applications connect to and communicate with each other in run-time. Such run-time based connections are typically not detectable through static analysis but require dynamic analysis, i.e. the analysis of a system during execution. DynSAVE, which addresses these issues, is now a SARP project supported by the NASA IV&V center. The new technology has already been applied to several of JHU/APL's software systems and will be ready for research infusion into other projects in the near future. We think this development illustrates well how the feedback from research infusion can be used to identify new needs that lead to new research, which again can be infused.

SAVE

Let us now describe the Software Architecture Visualization and Evaluation (SAVE) technology, before we discuss in more detail how SAVE was applied in these infusion projects.

SAVE addresses the problem that software systems often are difficult to understand, maintain, and evolve. Since the software maintenance phase typically lasts an average of 10 years [tam92], significant costs are associated with the post-development phase. SAVE, which was developed in collaboration between Fraunhofer IESE in Kaiserslautern, Germany, and FC-MD, addresses such problems by allowing software architects to document, communicate, and manage their software architectures. These higher goals are supported by features allowing software architects to navigate, visualize, analyze, compare, and evaluate their software systems from an architecture viewpoint. SAVE is often used to analyze legacy systems and can be used to analyze, for example, existing C/C++ and Java code. Using external parsers and importing the results to SAVE, it can also be used to analyze software implemented in programming languages such as ADA and Fortran. SAVE can then be used to identify violations of interactions between components/modules of the software architecture in the implementation (the actual architecture) as compared to the planned (or intended) architecture. SAVE can also be used to develop a new target architecture, which, for example, can be compared to the

implemented architecture to measure the distance between the target and the actual architecture. Another usage of SAVE is to analyze the product-line potential and deviations of the software through the detection of architectural commonalities and differences between different software products. SAVE also allows the architect to visualize and navigate the architecture of an existing system on different levels of abstraction, for example, in order to understand the architecture, or to investigate potential architecture violations. By identifying and removing architectural violations, the architect can prevent the architecture from degenerating ensuring that the architecture is kept flexible despite software change and evolution. Thus, this technology complements and makes reviews more efficient because it provides high level views of the software and can be used to identify areas of the software needing more attention. Most software systems would benefit from using SAVE. However, larger software systems that will live and evolve over long time especially benefit from using SAVE because of the fact that it can help assure that their software will continue to conform to its architectural specifications over time. Product-line architectures can also benefit from using SAVE for the same reasons because SAVE can help ensure that important variability points that provide product flexibility are kept intact.

The SAVE Tool

The SAVE Tool is a collection of components each supporting a different aspect of the architecture analysis task including components for extraction, visualization, graphical editing, rule editing, mapping, and evaluation (see Figure 1). The *extraction component* is used to extract the software architecture from the source code. The extracted architecture is called the *actual architecture* because it represents the architecture that is actually implemented in the source code. The actual architecture can be visualized using SAVE's *visualization component,* which includes various features for navigating and browsing the architecture. Integrated with the visualization component is the *graphical editor*, which allows the user to manipulate and refactor the actual architecture as well as to create a planned architecture graphically including laying out the expected connections between the components in the system creating connection rules. The *rule editor* allows the user to define architecture rules that complement the rules that were defined using the graphical editor. Once a planned architecture has been created using the graphical editor, the user uses the *mapping component* to map entities in the actual architecture to entities in the planned. Based on this mapping, SAVE can assess the adherence of the actual architecture to architecture and design rules using the *evaluation component.* SAVE accomplishes the evaluation by comparing the actual architecture to the planned architecture design and/or by applying the architecture rules. We will now discuss each component in detail.

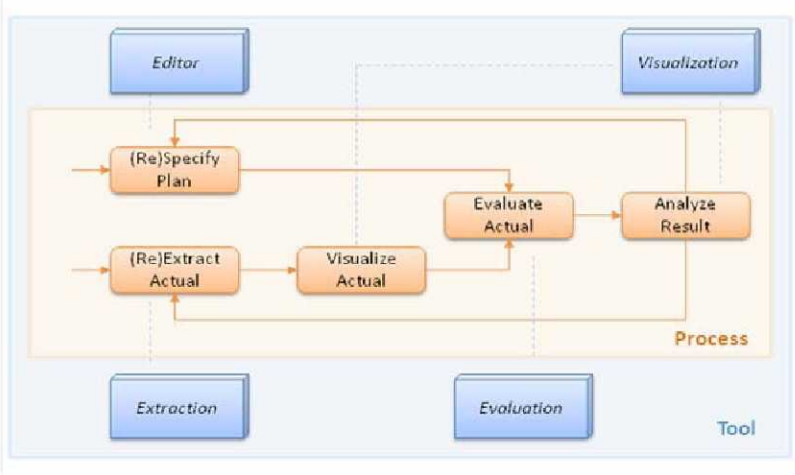

Figure 1: SAVE components

The Extraction Component

Given the source code of a software system, the SAVE tool can extract the architecture in terms of the source code structure. It does so by using advanced language specific parsers, which parse and extract an architecture model of the source code. Most systems, to which SAVE has been applied, were developed in C/C++ and Java. However, SAVE also supports ADA, Delphi, Fortran, and Simulink by using external parsers and importing the results into SAVE. A feature that has proven to be beneficial for many projects is the ability to extract the architecture from the source code even if it does not compile. Especially in the domains of embedded systems and aerospace, software systems often operate in environments that are difficult to reproduce on a local computer. The SAVE tool's forgiving parsing strategy ignores dependencies to source code files of the system that are not present and extracts all dependencies that can be resolved. Source code that cannot be parsed is skipped in such a way that even uncompileable code can be analyzed. In this way, SAVE extracts information about static couplings from the source code that are used to model dependencies between source code entities. Source code entities are files and folders, while static couplings are, for example, inter and intra class method calls and variable accesses as well as import and inheritance relationships.

The Visualization Component

Few software systems are small enough that their architecture can be captured by one picture on a computer screen. Large numbers of components, deep hierarchies, and complex dependencies disqualify a single view from supporting effective analysis and evaluation of many software architectures. To address this problem, SAVE visualizes the software architecture in diagrams, similar to UML class diagrams, in which the user can expand and collapse components to visualize, analyze, and illustrate the most important aspects of the architecture. To manage complexity, SAVE starts by creating a view of the highest architecture level showing only the top level components, i.e. components that are not contained by other components. The user can then focus on certain parts of the architecture or on certain kind of information by highlighting some details while hiding others. For example, the user can create new views visualizing only the aspects that the

user finds important. Such views can be created by hiding components, or by selecting one or more components and creating a view including the selected components and/or their neighbor components. The user can also create a diagram that illustrates a certain point, for example by arranging components in a certain way or by mixing manual and auto-layout of components. In order to further handle complexity, dependencies from one component to another are initially shown in an aggregated form as an arrow where the thickness of the arrow indicates the number of underlying dependencies. Each such aggregated set of dependencies can be expanded showing the underlying dependencies, one for each type, e.g. call, data access, inheritance etc. The thickness of these individual dependencies indicate the number of underlying dependencies and each can be expanded to show concrete dependencies, e.g. a call from a method in one class to a method in another class. In order to further manage complexity, the user can hide certain types of dependencies while highlighting others. Since the actual architecture is extracted from source code, the user can, at any point, click on a dependency and list all underlying details including the type of each dependency. A call dependency is, for example, listed as *from component*, *from class*, and *from method*, *to component*, *to class*, and *to method*. In projects that use eclipse as a development platform (e.g. Java and C++), the user can navigate to the source code of a particular dependency in order to investigate what caused the specific dependency to occur in the diagram. Once it has been determined that the dependency is indeed undesired and be removed, the source code can immediately be altered.

The Graphical Editor Component

The graphical editor is integrated with the visualization component and is used to create the planned architecture as well as to manipulate and refactor the actual architecture. The planned architecture is created by selecting a component type, for example, *system*, *sub system*, *layer*, or *design pattern*, and placing components on the drawing canvas and naming them. The user can then add dependencies between any two components. Several types of dependencies are available. The most general dependency is used to represent all other dependency types while named dependency types are used to represent specific types of dependencies such as *call dependency*, *data access dependency*, *inheritance dependency*, and *import dependency*. The user can, for example, initially model the architecture in an abstract way by using the general dependency type to model the planned architecture. Later, when more details of the architecture are known, the user can refine the model by changing general dependencies to specific ones. Components can be created in a hierarchical fashion meaning that one component can contain another component, thus creating an implicit containment dependency. Such implicit containment dependencies can be visualized explicitly by altering a configuration setting which causes components to be placed next to each other instead of inside each other. In this view, the containment dependency is drawn as a dependency between the super component and the sub component. All features described in the visualization section above can be used to create a planned architecture illustrating the most important points of the architecture. The editor can also be used to manipulate and refactor the actual architecture. For example, in order to experiment with how the architecture can be made clearer, the user can refactor it by, for example, adding a new component to the actual architecture. Existing components can then be moved into the new component, which then can be

collapsed. In such a way, new layers or new aggregated components can be added to the actual architecture.

The Rule Editor Component

Using the graphical editor, the user can define planned architectures using components and dependencies. A dependency between two components implies that the user expects that such a dependency does exist in the actual architecture. The direction as well as the type of the dependency are significant, meaning that any deviation from this "specification" will be reported. For example, if the type of dependency from component A to component B in the planned architecture is "Call dependency" and the dependency from component A to B in the actual is a "Data access" dependency, then a deviation will be reported. The reason is that the actual dependency is of a different type than the planned one. This form of graphically defined rules applies to many situations, however, there are situations where more explicit rules are easier to use. For example, consider a situation where we have a function library in the software architecture. As an architect, we want the library to be used as much as possible so we want to express the rule that all other components are allowed to use the library. At the same time, we want to prevent the library component from using other components, i.e. we do not want the library to be dependent on application-oriented components. These rules are difficult to express using the graphical notation because a dependency has to be created from each component to the library. At the same time, the lack of dependency from the library to each other component would implicitly mean that the library is not allowed to use the other components. It is especially difficult to maintain these rules graphically when the planned architecture is not yet stable. For example, each new component that is added to the planned architecture has to have a dependency to the library. In addition, if the library component is not yet used by a component in the actual architecture, a deviation will be reported. In order to model these rules in a more explicit and general fashion, SAVE also provides a rule editor. Using the rule editor, the user can specify the rule that *all other components are allowed to use the library* and that *the library is not allowed to use any other components*. This rule does not have to be changed if a new component is added to the planned architecture, making the set of rules easier to maintain. The rule editor can also be used to define more detailed rules than can the graphical editor. For example, a rule can specify that a component has to call a certain function in another component, e.g. in a library, and that it is not allowed to call certain functions in the same library. There are actually two types of rules: *relation conformance rules* and *component access rules.* A relation conformance rule defines an allowed or a forbidden dependency between two components, while a component access rule defines simple ports of a component, which other components are allowed to use. These two types of rules can be combined arbitrarily.

The Mapping and the Evaluation Components

Most applications of SAVE involve the comparison of the actual architecture, which was extracted from the source code, to the planned architecture and/or the rules. This form of evaluation is based on the ideas of the reflexion model [5]. SAVE has a mapping editor that allows the user to first map source code entities in the actual architecture to entities in the planned architecture. For example, the user might want to map all classes in the

actual architecture whose name start with "client" to the "client" component in the planned architecture. Once the mapping is completed, SAVE can automatically compare the planned to the actual architecture. The result of the comparison is a new architecture diagram that is based on the planned architecture and where missing and extra dependencies are highlighted. Missing dependencies are those that are modeled in the planned architecture but do not exist in the actual architecture. Extra dependencies are those that are not contained in the planned architecture but exist in the actual architecture. As was mentioned above, type and direction of dependencies are significant.

SAVE allows the user to incrementally specify planned architectures and evaluate them against actual architectures. That way, the user can specify the planned architecture, first on a high level and then add details at a later point. SAVE then evaluates the actual architecture on the respective level of detail. Each evaluation is saved so that the evaluation does not have to be repeated in case the user needs to inspect the evaluation again.

Visualization of compliance checking results is done by overlaying icons on top of dependencies. Convergences (a.k.a. correct) are dependencies that exist both in the planned and the actual, and are decorated with a green check mark (or not at all). Divergences (a.k.a. forbidden, not expected, or extra dependencies – the name depends on the interpretation) are dependencies that do not exist in the planned but in the actual, and are decorated with an exclamation mark. Absences (a.k.a. expected, or missing dependencies) are dependencies that exist in the planned but not in the actual and are decorated with a red cross. Aggregations of multiple result types, i.e. sets of dependencies including expected, extra and missing dependencies, are indicated by a blue question mark. The user can expand such an aggregated dependency in order to determine what the problems might be.

The SAVE Process

In order for any tool to be useful, a process must be defined that describes how the tool is applied to achieve a certain goal. The evaluation goal of applying SAVE is detecting and resolving deviations between the desired software architecture model and the implemented architecture. Figure 1 illustrates how the tool and the process relate to each other. More specifically, it shows what tool components support which step in the application process. The following elaborates on each step in more detail.

Step 1: Modeling the planned architecture

The first step of the process is to capture and model the planned architecture. Because many software architectures are inadequately documented, this step often draws on knowledge from architects and lead developers. SAVE uses information about architecture goals, styles, and components as well as high-level design patterns to model the structure of the planned architecture using a simple built-in editor based on Unified Modeling Language (UML) notation. This step is typically iterative as the architect first models the architecture on a high level and then adds more details to analyze a specific part of the architecture in more detail. This step also helps the architects remember more

about the planned architecture as well as the design rationale behind the architecture decisions that led to this particular solution. From an infusion point of view, this step usually takes some time to get infused because it requires the architects to formulate more explicitly how the components were supposed to be coupled to each other. This information is often not explicitly stated or a long time has passed since the original architecture was formulated making it necessary to explicitly recover such information.

Step 2: Extracting the actual architecture

The next step is to extract the actual architecture from the source code. The SAVE tool parses the source code and produces an architecture view in which files are represented as components and folders as subsystems. Lines between these components indicate dependencies that have been identified in the source code. The SAVE model is hierarchical and the user can either inspect the high-level structure of the system, or zoom into components in order to reveal the structure within components. Dependencies can even be traced back to the source code. From an infusion point of view, this step is typically easier to infuse than step 1, creating the plan, because source code almost always already exists, but planned architectures do not. In addition, if the source code is in fact a Java or C/C++ Eclipse project then this step does not take much effort at all to complete. Even using external parsers for other languages such as ADA and Fortran is very straightforward.

Step 3: Mapping the actual to the planned

The third step is to map the components in the actual architecture to those in the planned architecture using SAVE's simple mapping editor. The mapping is manual and also draws on knowledge from architects and lead developers, but typically does not take much time to complete. For example, all source code class files named "client" can be mapped to all the files that make up the client subsystem, using a regular expression. This mapping is also used to compose more abstract components that are composed of files that are not necessarily stored in the same folders. From an infusion point of view, the mapping step is a relatively new concept to most architects, but once the planned and the actual architectures are in place, the mapping is relatively straight-forward and conducted in short time.

Step 4: Evaluating the actual architecture's adherence to the planned

The fourth step is to automatically compare the actual architecture with the planned one, based on the mapping. SAVE flags each item in the actual architecture that does not match an item in the planned architecture. For example, a typical violation of the layered architecture style is the existence of couplings from lower to higher layers. Another frequent violation is the dependence of libraries and other common assets on application-oriented components. A third common violation is the insufficient insulation of COTS components, which makes it difficult to replace such components with others. A fourth

common mistake is to access a library class directly rather than using its defined interface, which tightly couples the applications to the library implementation. From an infusion point of view, the evaluation step is probably the easiest one to infuse because it is automated and does not require any effort to conduct.

Step 5: Analyzing each deviation

The fifth step is to analyze each deviation to determine whether or not it is critical. This step is manual but relatively efficient given the tool's ability to drill down to the source code. The common deviations described above are generally regarded as critical, whereas import/include statements that are never used are considered harmless. From an infusion point of view, this step requires some training and experience in order to draw the right conclusions based on the evaluation results. For example, even if SAVE has identified an unexpected or missing dependency, it does not necessarily mean that there is a major issue. In order to determine the severity of each violation, the user needs to understand the reason behind the violation. In some cases the planned architecture is incorrect, in some cases there is a violation, but it is minor or intentional. Thus the user typically needs to conduct a couple of SAVE analyses in order to learn how to correctly make conclusions about violations identified by SAVE.

Step 6: Defining a plan for removal of critical violations

The last step is to define a plan for removing the violations deemed critical. In most of the systems we have analyzed, the number of violations was small enough that they could be removed relatively easily. However, that is not always the case. For example, we identified more than 5,000 violations in one large system (not one of the systems analyzed in this paper), and in another one we found a large set of duplicated source-code files infected with bugs. Removing 5,000 violations or removing duplicated buggy code translates into a large reengineering effort. In such a case, it's necessary to first develop a new target architecture and then refactor the existing code so that the actual matches the target. Thus, from an infusion point of view, this step can be the most difficult one to infuse because software developers are almost always under project pressure and there is little time to reorganize the architecture. Another important aspect is that in the aerospace domain, where ground software is just part of a larger mission data system, changes to existing code can trigger extensive system-level regression testing. For this reason, the change control boards (CCB) become increasingly resistant to modifications as mission development progresses. A solution to this problem is to make the improvements over time instead of trying to improve the entire architecture in one huge step. Components of the architecture that are modified for other reasons can often also be refactored to remove architecture violations. For software where test cases do exist, such refactorings are less of a problem because it can be demonstrated by applying the test cases, that the functionality of the system has not been altered by removing the violations.

Infusion Project Details

Above, we summarized the various infusion projects and described the SAVE tool and process. We will now elaborate and describe how SAVE was infused into the various projects and organizations.

Infusion 1: SAVE Applied to Common Ground

As mentioned above, the initial infusion project was established between SIG (JHU/APL), who develops Mission Operations Center (MOC) system software, and FC-MD, who develops SAVE together with Fraunhofer IESE. At the time of the research infusion, SIG was using a shared architecture called the Common Ground System (CGS) to build MOC system software, and CGS was used for all JHU/APL-supported NASA missions. (CGS was later replaced, as described below). CGS uses commercial MOC core software, with extensions for mission specific functions and extensive adaptations made by JHU/APL beginning in the 1990's for the Near Earth Asteroid Rendezvous (NEAR) mission, the Thermosphere Ionosphere Mesosphere Energetics and Dynamics (TIMED) mission, and the Comet Nucleus Tour (CONTOUR) mission. The software architecture had continued to be maintained and enhanced, and was at the time supporting integration and test (I&T) and operations for three deep space missions: the MErcury Surface, Space ENvironment, GEochemistry, and Ranging (MESSENGER) mission, the Solar Terrestrial Relations Observatory (STEREO) mission, and the New Horizons mission to Pluto and the Kuiper Belt. The Common Ground software adaptation process yields a mission-specific ground software system that follows the CGS architecture but which is unique for the mission being supported.

The CGS software architecture has a major impact on the maintainability, usability, and reliability of the ground system. Flight software, science data processing software, and ground equipment software interface with CGS and depend on its services. For JHU/APL's NASA missions to be successful, the I&T and operations software must be of high quality. Towards this end, the software architecture must be planned and managed, and the actual implementation of the architecture must be consistent with the planned architecture.

However, CGS had emerged from the work of at least three teams operating independently over the course of 10 years throughout which little explicit architecture documentation had been produced. For example, JHU/APL originally established architectural goals and design rationale during development of the CGS through periodic team meetings. These goals and rationales were communicated verbally among the developers in the process of designing and building the system but were never documented or checked. And, when the infusion project started, most of the original designers were no longer working on the project. Since systems built on CGS must be maintained for at least 15 more years, a higher level specification was needed so that the architecture could be considered as a whole, but the SIG found it cumbersome and tedious to develop such a specification without tool support. In addition, due to its age, the CGS architecture was dependent on several technologies that had become obsolete.

Thus, the goal of the research infusion task was to increase the maintainability of CGS in order to accommodate for future modifications and to replace obsolete technologies.

The research infusion project followed the SAVE process as it was described above, also see [10] for more details. In this paper, we describe some of the analyses that were conducted during the infusion project:

- How SAVE was used to explore the actual architecture of CGS
- How SAVE was used to check whether a certain design rule was met
- How SAVE was used to determine the feasibility of introducing a layered structure to the architecture

CGS is a large pipe-and-filter architecture involving 83 applications that are launched separately but communicate during run-time. Figure 2 shows the data flow for a subset of CGS. In order to focus the infusion project on a manageable but representative part of CGS, we examined two Computer Software Components (CSCs) from the Assessment Computer Software Configuration Item (CSCI):

- **Archive Server** – a Telemetry data server that serves selected telemetry packets from the archive.
- **Engineering Dump** – a client to Archive Server that extracts selected telemetry points from the archived packets, converts the raw telemetry points into engineering units using calibration data, and formats and stores the converted points in a file.

In Figure 2, the Archive Server (archive_server) and the Engineering Dump (eng_dump) are highlighted and their relationship to the CGS Telemetry pipe-and-filter architecture is illustrated. We will now explore the structure of the Archive Server.

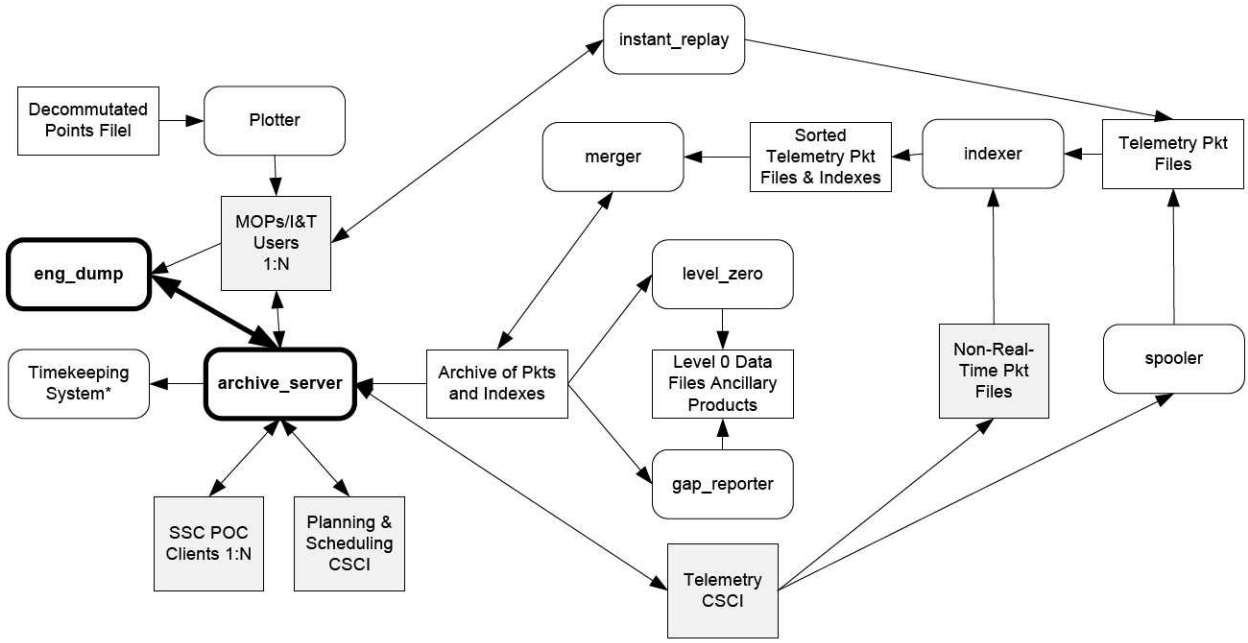

Figure 2: The Pipe and Filter architecture of the CGS

Exploring the actual architecture of CGS

Once SAVE had parsed the source code and had created the actual architecture, the model was navigated in order to create an understanding of the general structure of CGS in terms of views. Figure 3 shows such a view: the structure of the Archive Server. Figure 3 shows that the Archive Server is divided into two main parts: the *app_specific* and the *common*. In this particular diagram, the user has selected the *app_specific* sub-system, which is collapsed and therefore does not reveal any details. Colors indicate the direction of the dependencies. Red indicates outgoing dependencies while blue indicates incoming dependencies. The diagram illustrates that there are only outgoing dependencies (rendered in red) from *app_specific* to the *common* sub-system, which is a library component that is reused throughout the CGS. The dependencies to common show that *app_specific* makes heavy use of *common*, which is desired and intended by the system architects. The lack of incoming dependencies (blue) is also desired because the system architects do not want the library to be dependent on the application.

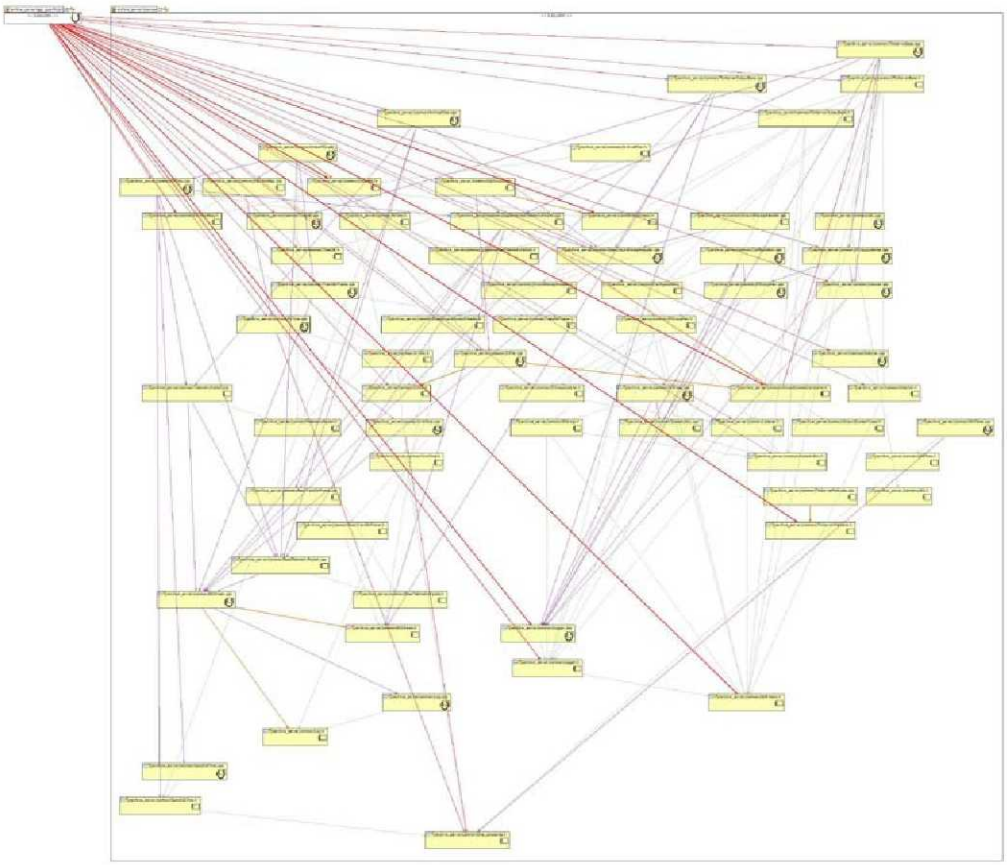

Figure 3: Archive Server's usage of Common

The next step focuses on the internal dependencies of *common*. Even though the diagram is rather busy and some components have many dependencies to other components, most components share dependencies with only a few other components. Few dependencies (i.e. low coupling) are generally desirable. The diagram makes it possible to identify the few components that have many dependencies to other components. In the infusion project, such "suspicious" dependencies were investigated using SAVE's ability to reveal more details allowing potential issues to be identified. A frequently used feature was the one that allows the user to select a certain dependency to obtain more details such as in which file the dependency is initiated and to what file the dependency is directed.

In the infusion project, many such views were created and many dependencies were investigated. The result of this *exploratory analysis* provided the basis for a high-level specification of the actual architecture and an analysis of the architecture searching for high coupling between components, which might increase resistance to change. That information was eventually used for designing the new target architecture with the goal to remove undesired dependencies and prepare it for future changes.

Checking Design Rules in SAVE

The previous analysis provided a start for understanding and qualitatively assessing the architecture and laid the groundwork for evaluating design rules related to dependencies, i.e. specifications regarding how components can depend on each other. The evaluation feature of SAVE, which compares the actual architecture with a planned design and highlights extra and missing dependencies, was used for this purposes. The next example illustrates one of the evaluations that was carried out in the infusion project. The example investigates whether or not a selected design rule is consistent with the needs, goals and design rationale for the modernized target CGS architecture. The design rule reads: "Isolate the external interfaces by requiring that certain types of calls are done through certain modules or class. One instance of this design rule occurs in the *Eng_Dump*. That is, the client socket connection from Eng_Dump to *Archive_Server* should first go through the *TlmClient* class, which implements *Archive_Server*'s client interface protocol. *TlmClient* should then use the *SocketCC* library, which encapsulates the UNIX socket interface (see http://www.ctie.monash.edu.au/SocketCC for details). The planned *Eng_Dump* architecture that encodes the design rule is shown in Figure 4.

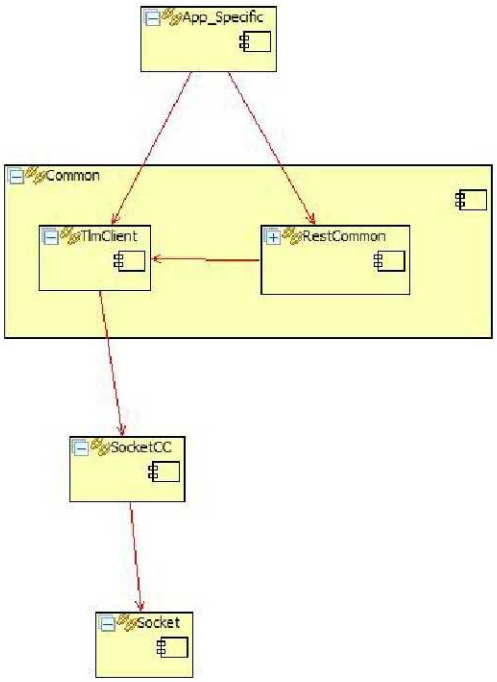

Figure 4: Applications should access external interfaces through defined classes

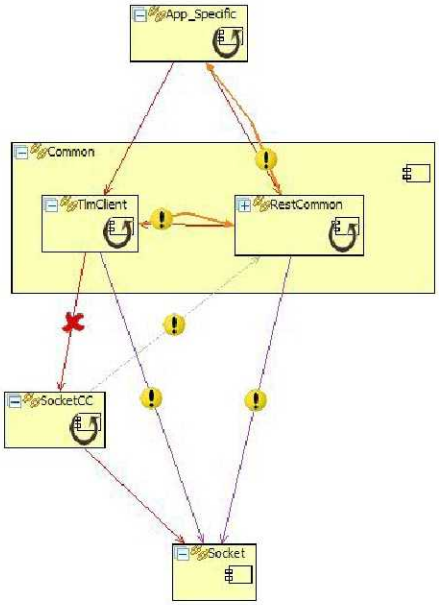

Figure 5: The actual architecture of Engineering Dump bypasses SocketCC

Using the SAVE tool to compare the planned architecture to the actual architecture, it becomes clear that the current implementation of Eng_Dump does not fully follow the design rule. Figure 5 presents the comparison between the planned Eng_Dump architecture and the actual *Eng_Dump* extracted from the source code. We can see that *Eng_Dump* uses *TlmClient*, but *TlmClient* does not use *SocketCC* (i.e. the missing connection between *TlmClient* and *SocketCC* is marked with a red cross).

The reason for these deviations between the planned and the actual architectures is that *SocketCC* became the approved socket interface library long after *TlmClient* was designed, so it is not surprising that *TlmClient* makes direct socket calls. To bring *TlmClient* into compliance with the design rule stated at the beginning of this section, the direct socket calls should be replaced with *SocketCC* class methods. It should be noted that there are also other deviations between the planned and the actual that are annotated with exclamation marks in Figure 5. For example, there is an undesired back link from common to App_Specific. The structure of Engineering Dump (Figure 4) and Archive Server (Figure 3) is similar in terms of their layers. However, it was noted in the analysis that Archive Server does not have issues with back links to App_Specific from common, like Engineering Dump does (Figure 5).

Design New Target Architecture

Based on the analyses conducted as part of the infusion project, the system architects designed a new target architecture for CGS that would improve the structure and therefore the maintainability of CGS while keeping the changes to the actual architecture to a minimum. This was done by re-factoring the actual architecture using the SAVE diagram editor by moving components to different subsystems (existing or new ones created by the user) and analyzing the impact on the architecture. This analysis suggested that the Common part of CGS could be greatly improved by dividing it into layers. Three layers were selected to organize the shared modules of the common component, as depicted in Figure 6:

1. An application layer of shared, application specific functions,
2. A middle layer of domain specific functions, and
3. A utility layer of application and domain independent functions

A basic layered design rule was selected for investigation: a lower layer cannot access a higher layer. To facilitate reasoning about the target architecture, all redundant and deprecated classes were first identified and eliminated. The coupling between the layers was then analyzed, starting with the couplings between the application layer and the middle layer. In Figure 6, the highlighted links decorated with exclamation marks are violations of the basic layered design rule. The upper violations exist because PacketExtractor in the middle layer is accessing Consultative Committee for Space Data Systems (CCSDS) File Delivery Protocol (CFDP) modules in the application layer. This is a known design flaw that was introduced when support was added to CGS for MESSENGER to extract Protocol Data Units (PDUs) directly from CCSDS telemetry transfer frames. PDUs are not telemetry packets and should be processed in a separate module. The lower highlighted links, also decorated with an exclamation mark, from the Ground Support Equipment Telemetry Client (GseTlmClient) in the utility layer back to the middle layer also violate the basic layered design rule. The reason for the violation is that GseTlmClient is domain specific and belongs in the middle layer, and the "back links" actually vanish when the system is properly modeled. Analyzing the coupling from the middle layer to the utility layer reveals extensive coupling, as one would expect for a collection of widely used support functions. However, the need for careful design and testing of utility layer libraries is revealed by this analysis.

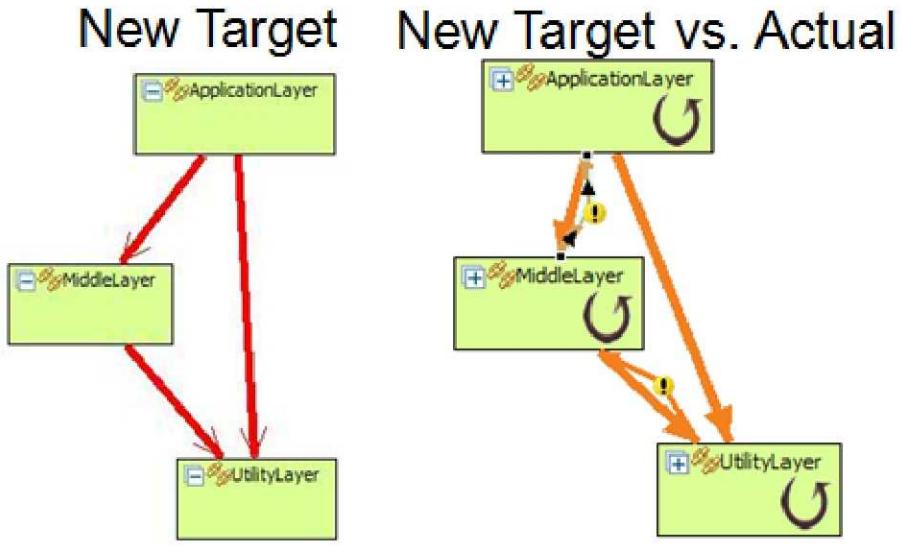

Figure 6: The Common part of the new target is divided into three layers (left), and by comparing the actual architecture to the new target, SAVE identifies a two sets of back links from lower layers to higher layers (right).

Infusion Sustained: Thread-Safe Dependency Analysis

After the infusion project was completed, the JHU/APL members of the team continued using SAVE to analyze CGS for thread-safety. "A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads. In particular, it must satisfy the need for multiple threads to access the same shared data, and the need for a shared piece of data to be accessed by only one thread at any given time" [http://en.wikipedia.org/wiki/Thread-safe]. Thus, if a method is not thread-safe, calls to that method might result in concurrency issues, such as race conditions. In such cases, mechanisms must be put into place to prevent these issues.

The objective of the thread-safe analysis was to identify dependencies from components, for which thread-safety is required, to methods that are known not to be thread-safe. In particular, the dependencies of a class that was recently added had to be analyzed.

To conduct this analysis, the software architecture was extracted from the source code using SAVE and visualized. Figure 7 shows the relevant part of the architecture. The new class *SleRcfIf* was added to the *dsn_tlm_if* telemetry process to support the CCSDS Space Link Extension (SLE) protocol. *SleRcfIf* functions run in multiple threads, so the functions that it calls need to be thread-safe. The diagram shows that functions in two legacy modules known not to be thread-safe are called from *SleRcfIf* (indicated by the dependencies). These dependencies raised the concerns that these functions could, for example, interleave messages from multiple threads and that they could have two threads trying to connect at the same time and that they therefore should wait for access and

check if they are already connected. In order to address the thread-safety concerns, it was decided to control access to the non-thread-safe method with a Mutex.

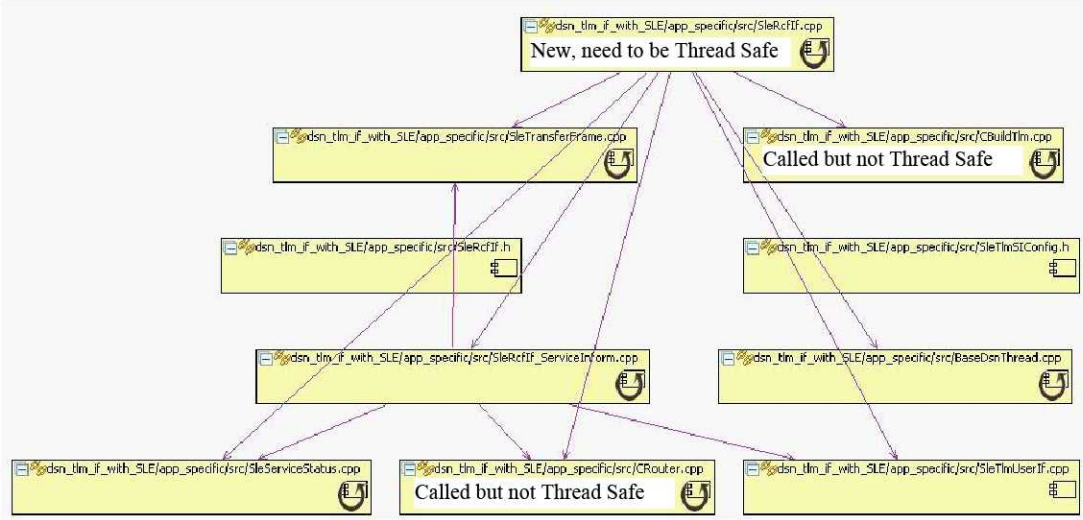

Figure 7: Thread Safe Analysis

Continuation of using SAVE by the Ground Systems Group at JHU/APL

The SIG is currently working on developing the ground system for the Radiation Belt Storm Probes (RBSP) mission. RBSP is a NASA mission to study near-Earth space radiation, which is hazardous to astronauts, orbiting satellites, and aircraft flying high polar routes. The mission will study how accumulations of space radiation form and change during space storms. The RBSPs are two spacecraft in nearly identical low-inclination highly elliptical orbits with identical instrumentation that are required to operate for at least two years.

The SAVE analysis of CGS convinced JHU/APL that there was a need to rework the architecture in order to form a modern replacement for CGS implemented in Java. As part of reworking the architecture, a new core Missions Operation Center software product from L-3 Telemetry-West called InControl was integrated. SIG continues to use SAVE to document, communicate, and control the RBSP ground system software following a process that was defined based on the experiences with infusing SAVE into the regular software development process, as described below.

The JHU/APL Space Department's existing software development process provides a framework in which the SAVE tool and process is used to assess progress towards the modernized CGS architecture. In particular, the software design and implementation processes provide opportunities for refining the specifications of the planned architecture and aligning the actual architecture with the plans.

The process states that an architectural design rule checklist, which elaborates on rules such as those articulated in the infusion project, is developed and maintained for use in each software design review. For any new software development or significant modifications requiring reviews, the following steps are added to the software *design* process:

- In the design phase, define and document the planned software architecture.
- At the design review, review the planned architecture against the architectural design rule checklist.
- After the design review, update the planned architecture and document the updates in the review action item response memo.

The following steps, which are based on the SAVE process described above, were added to JHU/APL's software *implementation* process to keep the planned and actual architectures aligned:

- In the coding phase, extract the actual architecture and compare it to the plan.
- Upon completion of the implementation, review the differences between the planned and actual architecture and specify change requests to reconcile (in the case of software identified as critical, this is in addition to the required code review).
- After the review of differences between the planned and actual architecture, align them and document the alignment steps taken in the change control system.

The modified process is currently being applied. We will document the experiences and lessons learned in detail after the development has been completed.

Infusion 3: Flight Software[2]

Shortly after the infusion project had been completed, JHU/APL's Embedded Application Group (SIE) started experimenting with SAVE because they had heard about the technology from their colleagues. As part of this pilot project, a summer intern applied SAVE to some of their software in order to determine its applicability and value. The goal of the intern's work was to study how JHU/APL's flight software architecture could be engineered to use a framework for flight software provided by NASA Goddard called the core flight executive (cFE) and the core flight system (cFS). The result was presented at JHU/APL's First Flight Software Workshop [6].

Typically, flight software used on JHU/APL space missions consists of one large, mission-specific application. The effect is that any modification of the source code requires a complete reload of the entire flight software image. The risk involved in reloading the entire image is substantial when various pieces of source code are tightly

[2] This part of the paper was prepared by Leah Kelley, Mentor: W. Mark Reid JHU/APL

coupled. An error encoded into one piece of the source code potentially could affect the operation and execution of all tasks.

For the Geospace - Radiation Belt Storm Probes (G-RBSP) mission, SIE wanted to move to a modular publish-and-subscribe system comprising many small applications managed by an executive core. Applications in such a system exhibit little to no dependency with one another, though they are all dependent on the executive core. With no cross-application coupling, changes made to one application would not affect the operation of the other applications. Therefore, moving to such a system reduces the risk of corrupting the entire flight software image when making changes to one application.

A further extension of such modularity is the ability to "plug in" applications to the system after it has been launched. As spacecrafts last longer and longer, there is the possibility that science equipment onboard could be repurposed. In a modular architecture, a new application for such equipment could be written and uploaded to the spacecraft without interfering with other applications, though some adjustment to the message service code may also be necessary.

The SAVE tool was used by the SIE-intern in order to analyze the software architecture of prototype flight applications for the G-RBSP mission. This analysis was conducted independently of FC-MD. The SAVE models were analyzed to determine whether the desired publish-and-subscribe architecture was achieved and recommendations for revisions were made. The G-RBSP team also performed prototyping of heritage application software with NASA's core Flight Executive (cFE) by re-packaging heritage routines into cFE prototype applications. The goal of this prototype was to remove call and access dependencies from application packages, move common utility software routines into a shared library, demonstrate functionality, and evaluate the performance of the flight software in this architecture. SAVE models of this heritage software were created and compared with the G-RBSP models.

As expected, the analysis of the heritage architecture revealed many dependencies between components and subsystems at all levels. In contrast, the prototype applications are almost entirely decoupled at the application level, however many dependencies still exist within individual prototype applications. Some examples of detected unwanted include and data access dependencies are illustrated in Figure 8.

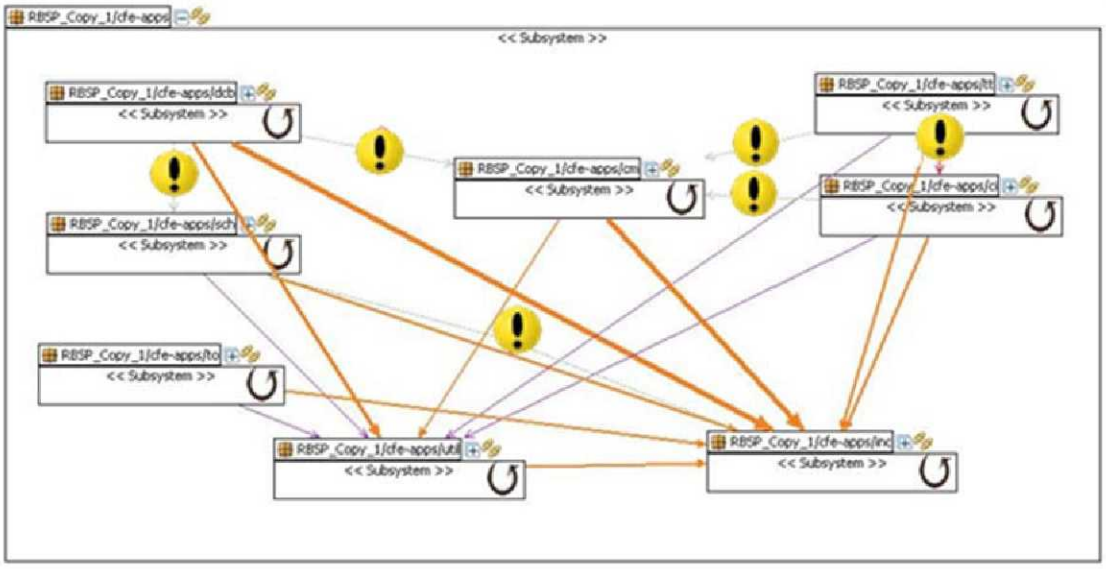

Figure 8: Unwanted includes and access dependencies are decorated with exclamation marks.

For complete decoupling of the prototype applications at the application level, several applications needed to be revised. For example, several applications contained at least one header file that defined data types used by other applications. These shared data types needed to be defined in a separate header file or files in the common-include folder to remove undesired dependencies. Moreover, several applications shared a variable that was defined globally in one application and locally in another application. Changing the name of the local instance of the variable eliminated the access relation between the applications.

Based on this experience, the SIE group concluded that SAVE is useful for creating models of mid- and high-level system architectures in a short period of time, providing developers with a quick way to visualize dependencies and determine whether or not the large architectural picture matches the desired architecture. The SIE also concluded that details of dependencies at the function and data structure level were available but not easily visible in the diagrams. Additionally, since the parser is built for C/C++ code, pure C code that uses C++ keywords as variable names will generate error messages. In this case, the developers were able to avoid that problem because they located and revised portions of the code that will not integrate with C++ code. The SIE also found that SAVE's modeling capability is especially useful when applied to large software development projects involving many programmers. For a smaller team, creating models and revising code only for the sake of the SAVE parser seems inefficient. Though model generation only takes about fifteen minutes, one may have to spend long time searching for all header files and revising portions of the code so that the C/C++ parser extracts the logic correctly. The SIE concluded that a team of five under pressure to get the software working does not have time to spend on modeling under such circumstances. This

feedback was incredibly valuable to the SAVE researchers because they could address these particular problems and improve the SAVE C/C++ parser respectively.

Infusion 4: Applying SAVE to CFS/CFE

Thanks to SIE's SAVE pilot project, the main architect of cFS got interested in applying SAVE to cFS itself. The Core Flight System (cFS) is designed as a common software base for Goddard Space Flight Center (GSFC) in-house missions. The cFS is a system of software components written primarily in C and is comprised of approximately 200 KSLOC when adapted for a specific mission, not including COTS and Open Source software included in the system. The cFS contains a library of common flight software application services and a publish-subscribe messaging middleware. All of these functions are designed for run-time plug and play of flight software applications (both cFS and mission-specific). Sound software engineering principles, such as modularity, layering, and separation of concerns are applied to achieve reuse and ease of integration. However, before the infusion project, there was no systematic automated way to verify whether the actual source code follows these important design decisions. The SAVE is being applied to verify the initial cFS implementation complies with the architectural rules and to help preserve the cFS architectural goals during the evolution and maintenance of the cFS.

The overall goal of this SAVE analysis project, which has just started, but already produced useful results, is to help cFS stakeholders (e.g., architects and engineers) to align and keep aligned the actual architecture of cFS with its planned architecture. This overall goal is being achieved by applying the SAVE tool and processes to CFS software and by achieving the following sub-goals:

1) Define a planned architecture including architectural goals and design rationale based on knowledge about the current software system as well as knowledge about needs for a sustainable, modernized system architecture for the future.
2) Generate a comprehensive high-level description of the actual architecture from cFS source code.
3) Identify deviations between the planned and actual architecture.
4) Create a roadmap that will guide ongoing system development and maintenance towards aligning and keeping aligned the system with the planned architecture.

When the initial phase of the SAVE analysis initiative is completed, SAVE will be used by cFS architects/engineers to continuously maintain consistency between the actual architecture and the planned architecture. A roadmap for the architectural code migration with periodic progress checkpoints will be established and a detailed discussion of the outcomes of this project will be published upon its completion. Architecture violations that have been identified so far, have been registered as discrepancies and will be addressed by the cFS team. For example, one of the minor issues that were detected was that the cFS design rule that cFS applications should not directly use arch and os, but only use src was met by most cFS applications, but the memory manager (mm). This violation was easily eliminated by removing an unnecessary include statement.

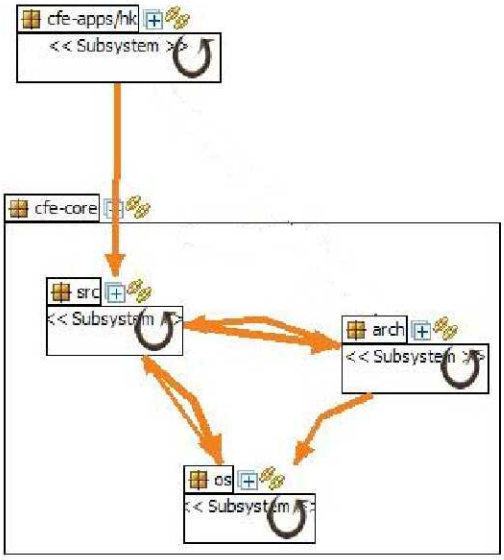

Figure 9: CFS Design Rule: Applications should not directly use arch and os.

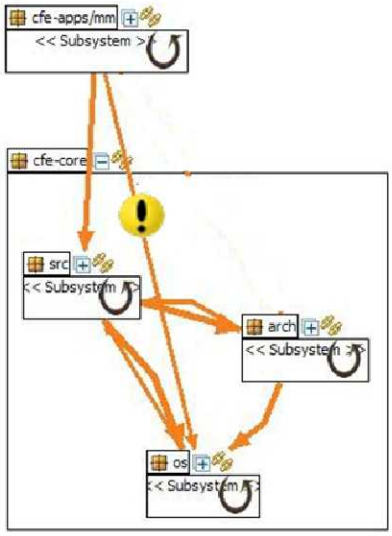

Figure 10: Most applications follow the design rule, but the memory manager.

New Research: Dynamic SAVE

In parallel with the research infusion activities of SAVE, a new initiative has been launched to address needs that have emerged during the collaboration with JHU/APL in analyzing CGS.

As stated above, the GCS consists of a large number of systems that are compiled and launched independently but communicate with each other during runtime. However, with SAVE, single applications are best analyzed and dependencies that go beyond system boundaries are difficult to detect. At the same time, it is the dependencies across systems that lead to problems in integrating the systems. In particular, if one of the systems fails to interact properly, the failure can propagate through the system-of-systems and severely impact its reliability and performance.

Such run-time dependencies are typically not detectable through static analysis but require dynamic analysis, i.e. the analysis of a system during execution. DynSAVE (or Dynamic SAVE) is an extension of SAVE that make use of dynamic analysis to recover information about the runtime interactions between systems. This effort is supported by NASA IV&V's Software Assurance Research Program (SARP) and the technology has already been applied to several of JHU/APL's software systems, and is planned to be ready for research infusion in other projects starting in 2009. We think this development illustrates well how the feedback from research infusion can be used to identify new needs that lead to new research, which again can be infused.

The finding that dependencies between systems cannot be analyzed sparked discussions about what impact such dependencies have on the individual applications. Not only were the inter-application dependencies considered a crucial element of the architecture analysis, but the system-engineers had collected reports on failures that were caused by these dependencies.

These applications are often developed independently by different teams at different locations. While testing procedures are used to ensure that the application in itself functions correctly, there are only limited means available for testing the interaction with other applications. As a result, problems in inter-application interactions occur frequently. However, interaction problems typically do not surface with clear signs as to what the problem is and what caused it. In fact, interaction problems manifest themselves in subtle misbehaviors of the applications. Typically, only a lengthy and effort intensive process is able to reveal that an interaction problem was the cause for the system misbehavior. Additional labor intensive analysis is necessary to determine the exact interaction problem and to locate its source. This is due to a lack of support for both testing the interactions between applications before they were deployed and for analyzing problems that occurred during the operation of the applications.

In order to address these problems, we applied for NASA's SARP program proposing to develop support for extending SAVE to visualize and evaluate the interaction behavior between applications. This support is being implemented as an extension to the already existing SAVE tool to handle dynamic behavior.

DynSAVE, just like SAVE, consists of four main components: (1) extraction, (2) visualization, (3) editor, and (4) evaluation. The extraction component extracts the inter-application interaction behavior from data collected from the running system(s). The interactions are visualized in a diagram. A graphical editor allows the user to specify a planned behavior. Finally, the evaluation component compares the observed interaction behavior with the specification.

Extraction. The interactions between applications are retrieved by observing their communication (i.e. network communication) and extracting the application level information from it. A detailed analysis of the problems that occurred in the CGS had shown that problems related to message sequencing, parameters, and timing caused the interaction problems. Thus, the extraction component retrieves this information for each message.

Visualization. In order to be able to analyze the interaction behavior, it is visualized as a sequence diagram. The sequence diagram shows a sequence of named messages and contains information about the message parameters, and their timing.

Editor. The graphical editor allows the system engineer to model the desired behavior using a sequence diagram.

Evaluation. The observed interaction behavior can be compared to the desired behavior using the evaluation component. The output of this step is a sequence diagram in which deviations between the observed and the desired behavior are highlighted.

Figures 11-13 illustrate how DynSAVE can be applied to a concrete problem. The expected behavior contains assertions to model constraints on timing and parameters. The sequence diagram can also contain loop constructs to model more complex message sequences. The observed interaction behavior, is a list of messages, each with a name, a set of parameters, and a time at which the message occurred (i.e. when it was sent or received). The result of the evaluation is shown in the last figure. The example shows a parameter violation (lightning), a timing violation (clock), and a sequence violation (cross).

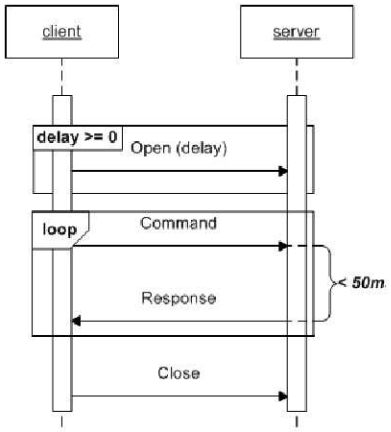

Figure 11: Expected interaction behavior.

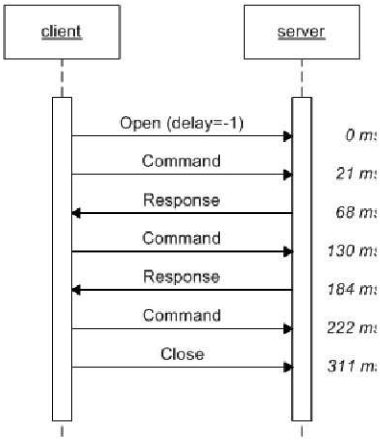

Figure 12: Observed interaction behavior.

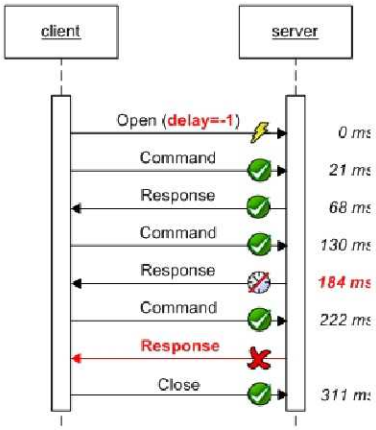

Figure 13: Output of the evaluation.

DynSAVE is a prototype currently in development. Nevertheless, it has been applied to a number of known interaction problems and was able to detect them successfully.

This effort, sparked by the initial work with SAVE, has led to activities that address a gap in the current research and at the same time solve a real problem. In order for such work to be possible and successful, the researcher must recognize gaps in the state of the practice, translate them to a more general problem, and either resolve it using existing research or, as in our case, address research gaps by developing new techniques.

Technology Transfer and Research Infusion: Lessons Learned

Research infusion is a collaborative effort that can only be successful if certain pre-conditions are met and practitioners as well as researchers invest the effort to achieve the goal of improving the state of the practice and maturing the technology. Here, we summarize lessons that we have learned throughout our research infusion work.

Pre-Conditions

The research infusion initiative brings together practitioners who have concrete problems and researchers who want to apply and mature their technology. A collaboration is formed if there is potential for the technology to resolve the concrete problems at hand. For this collaboration to be successful, additional pre-conditions must be met. The following elaborates on conditions that must be satisfied by the practitioners as well as researchers.

Practitioner

Education/Customization/Learning Process/Collaboration/Openness. Research infusion may present a challenge for the industrial organizations because it involves change. First, the practitioners must learn enough about the new technology to understand how it works and how it best could be used in the organization. Second, the practitioners must change the working processes in order to make use of the new technology. Thus, it is essential that the infusion process is gradual and well-supported by the researchers because if the practitioners encounter difficulties, they will quickly revert to the traditional processes. Third, collaborating with outside researchers means that there must be an openness to review by outsiders as well as support for publication and presentation of research infusion results, which requires a desire to establish relationships with the research community.

Need a champion/sponsor to push the idea. In order to bid on a research infusion grant from NASA IV&V, the practitioner must make a compelling business case to his manager for submitting a proposal. Business area executives have limited proposal development funds, so there must be good reasons to pursue this type of work. Technology infusion grants are relatively small in terms of the number of staff-hours they cover, but there are other potential benefits of research infusion to the practitioners:

- Staff capability enhancement
- Product quality improvement
- Organization reputation enhancement
- Opportunity to leverage research efforts in solving real problems in the organization's current programs

In addition to management support, the practitioner's organization must have an established business relationship with NASA, because NASA research infusion requires a civil servant sponsor and an established contract vehicle.

Researcher

Need a mature technology with supporting documentation. When a technology is making its way out of the research laboratory and into the hands of a practitioner, there needs to be support of various kinds because transferring new technology to a third party is very different from using new in-house technology. In-house use of a technology by the inventor of the technology is probably ten times less demanding compared to a situation when a third party, who was not involved in the development of the technology, wants to use it. There are many reasons for this complexity [3]. One is that in-house users

understand the background of the technology, how the technology works in detail and why it works that way, how it is supposed to be used, and what the workarounds to problems are. In addition, in-house users typically *want* the technology to work and therefore are much more forgiving when the technology does not work exactly as expected. In comparison, third party users do not favor a particular technology and *only* wants to solve a particular problem quicker and with less effort compared to the regular way of solving that problem. Thus, third party users easily get frustrated and are quickly willing to abandon the technology if they run into problems. In order to avoid such situations, it is important that the technology to be transferred is relatively mature and that it provides both powerful functions and a user-friendly user interface. In addition, user guides, training material, and personal support are very important. As a matter of fact, the best way to overcome such problems is probably to, at least initially, for the researchers to do the work for the practitioner by using the technology and then transfer the technology in small steps. We will describe such a model below.

A Model for Research Infusion with Concrete Advice

If a sufficient number of pre-conditions are met, a collaboration is established between practitioners and researchers to infuse the technology into the practitioner's organization. It is crucial to always acknowledge the fact that practitioners and researchers have different goals, a different understanding of the technology, and must operate under different constraints. The following defines a process for infusing the technology in a way that keeps both practitioners and researchers engaged, which is necessary to make an infusion project a success.

It is important to remember that research infusion is a sensitive process that is easily interrupted and therefore requires continuous active involvement from all parties. It is an activity that is focused on providing future benefits rather than immediate returns. Since the value of the technology might not be immediately visible, the risk is obvious that the practitioners get distracted by activities of higher priority, such as customer projects, that need their attention. Thus, a successful infusion project requires patience from the practitioners, but in order to ensure a success, the researchers must facilitate and expedite the infusion process and make sure the infusion project results in positive returns for the practitioners as soon as possible.

We addressed this situation by infusing the technology in increments and producing intermediate results. In the beginning of the infusion project, SAVE was applied to extract architecture diagrams that showed the system's high-level structure. While the architecture views were not always complete and detailed, they provided immediate feedback to the practitioners. In incremental steps, more advanced features of SAVE were used to address increasingly challenging analysis tasks that were identified in infusion team discussions. The starting point of each discussion was the intermediate results that were obtained, i.e. the result of applying the improved technology to the system. At the end of each discussion, a list of new analysis goals and needs was documented and served as a task list for the researchers. In this way, the researchers continuously provided the practitioners with small successes throughout the infusion process in order to keep them engaged and stimulate discussions.

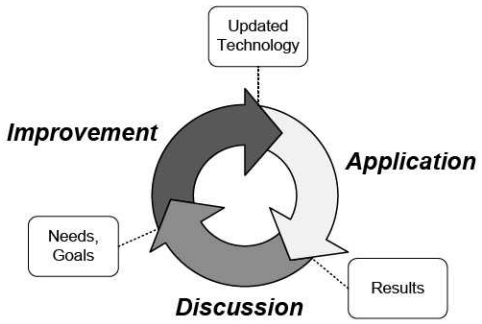

Figure 14: The incremental infusion process

Figure 14 illustrates the model of the incremental infusion process that was successfully used in the original infusion project. It describes a cycle that can be repeated many times. The process begins with the *application* of the technology to the software under study. In every increment, the technology is applied to the system to produce results. In our case, the results were architecture views. The results are then reviewed by researchers and practitioners in a discussion meeting. The goal of such a meeting is to discuss the results and suggest improvements of how the technology can be used to solve additional needs and goals. In our case, we discussed how existing features of the SAVE tool could be used differently in order to achieve the new goals. The researchers then fine-tuned the technology to suit the new goals. The updated technology is again applied to the problem at hand. In some cases, the practitioners must provide additional information or examples to conduct new analyses. This process is applied until all parties are satisfied with the results.

Application. The application of the new technology is in some situations best done in the research lab first. This was also the case for SAVE and the CGS, especially in the beginning of the project when it was still unclear exactly how to address each analysis question. After a number of analyses had been completed, the practitioners had gained more insight into the technology and the researchers were familiar with the domain and SAVE was transferred to the practitioner's organization. This way of working provides the researchers with the opportunity to study the system in more detail, identify possible shortcomings of the new technology that require workarounds, and prepare concrete results that can be discussed in the next meeting. In such a situation, attention must be paid to intellectual property rights and export control regulations. Non-disclosure agreements and necessary protections of export controlled information need to be put in place. This process typically takes some time and it is important that ample time is devoted to put all necessary agreements in place, preferably before the research infusion project starts.

The ***results*** of an increment depend on the technology that is being infused. The most important aspect is to define tangible products as deliverables from each iteration. Concrete deliverables help the researchers to focus on concrete goals and helps

practitioners to understand the value of the technology as well as to communicate the progress of the research infusion process to others.

Discussions. Having a continuous dialog between researchers and practitioners is a crucial component of the entire technology infusion process. Regular meetings serve to re-align the interests of both sides and also prevent the project from losing momentum. Such a continuous dialog allows information and knowledge to flow in both directions and maximizes the benefits for practitioners and researchers. A barrier of communication can sometimes be different terminology used by the two parties. The practitioners might make use of domain specific vocabulary and the researchers might use terms typically only familiar within the research community. Thus, it is important for the researchers to learn and understand the domain and the specific terms used. It is equally important that the practitioners learn and understand the capabilities and limitations of the technology.

The result of a discussion is a list of new ***needs and goals*** that are to be addressed in the next iteration. The goals might specify refinements to the analysis or additional needs that arose when discussing the results of the previous iteration. When infusing SAVE, new goals included analyzing lower layers of the architecture, refining the planned architecture, and establishing a new architecture after analyzing the implementation. It is important that the goals can be achieved in a relatively short period, in order to produce new results quickly. Larger goals can be divided into sub-goals if necessary.

Improvements. After the results have been discussed in a meeting, the researchers improve the technology to meet the requirements that were defined as a result of the discussion. If usability issues are detected during the discussion, the technology might have to be adjusted to be more user-friendly. If no technical limitations have been identified, the technology might not have to be improved and can be re-applied immediately.

The ***adjusted technology*** is customized with the goal to meet additional requirements that have been identified in discussions. With each iteration, the technology becomes more mature to the benefit of the researcher and more tailored to the concrete problem to the benefit of the practitioners.

With a more mature technology, additional goals can be addressed and new results can be produced. This cycle can be executed until the results of the analysis are satisfactory. Through the incremental application, practitioners gain confidence into the technology and learn about its capabilities (and limitations). It is, therefore, easier for them to integrate the technology in the existing process. Practitioners benefit from the incremental analysis as they gain insight into the practitioner's domain, identify limitations that arise in the context of real-world systems, and might discover needs that are to be addressed with additional research (as it was the case in our infusion project).

Recommendations to researchers and practitioners

The goal of the research infusion initiative is to introduce a new technology into an environment with real needs and constraints. In order to make this effort a success for both the technology provider and the practitioners we believe that it needs to employ the following points:

Advice to technology developers, i.e. the researchers who want to infuse their technology

- Pay attention to product usability and support in terms of documentation and training. Product-quality training, documentation, and support are necessary for practitioner buy-in and the technology transfer is going to be smoother if these items are in place.
- If possible, express features and benefits of the new technologies in terms of the project domain so that it is easy for the practitioner to understand what value the technology brings. The reason is that practitioners need to see the technology applied successfully to a problem from their domain to understand and appreciate it. From that, it is easy for them to find other applications for the technology and to explain the technology to their peers.
- Set the right expectations, i.e. be honest and clear about the technology's capabilities and limitations. The reason is that the technology will be applied to real-world problems that will reveal potential weaknesses and limitations of the technology sooner or later.
- Focus on, and learn about the problem, the situation, and the environment in which the technology will be applied. The goal of an infusion project is to solve a real problem to the best possible extent with the technology to be infused. Thus, a good understanding of the real problem that the practitioners want to address will pay off in terms of technology transfer and making the project successful if the researcher provides solutions to exactly those problems.
- Make sure there are quick wins and deliver solutions incrementally. Quick wins will increase the practitioner's interested in continuing to learn about and use the technology because the practitioner will see the value of the technology. In order to keep the practitioner interested, it is useful to continuously deliver results in increments and as intermediate results rather than waiting to the end of the project.
- Be in close contact with the users of the new technology and work with them and help them solve their problem as well as train them in using the new technology. Otherwise the risk is that the practitioners will become stuck and soon abandon the technology because they do not know how to get around the problems they encounter.

Advice to technology adopters, i.e. the practitioners who want to try a new technology

- Make sure there is a match between the technology and the problem to be solved to increase the chances for success. Do this by trying to understand, as quickly as possible, the technologies' capabilities and limitations, even before the infusion project starts, so that the match is as good as possible.
- Make sure that the technology is mature enough to be put in practice and has enough supporting documentation. Also make sure that the researcher spends

enough time on explaining how the technology works and what types of problems it can address. The reason this is important is that the limiting factors are the usefulness of the technology being infused and the practitioner's understanding of the technology. The sooner the technology is understood, the sooner the practitioner can help steer the project in the right direction.

- Make sure the researchers understand the problem, the situation and the environment the new technology will be applied to. This may take some time, but will pay off in the long run because without this understanding, the researcher might spend effort solving less important problems while leaving the important problems unsolved.
- Do not try to solve all problems at once with the new technology, but start small on one or more selected sub problems. The reason is that the project will be most successful and will keep everybody involved interested if there are some quick wins solving problems in the practitioner's domain. Thus, the practitioner needs to carefully pick those sub problems and suggest them to the researcher. Quick wins are best achieved solving small and sometimes obvious problems. Solving such problems will quickly build confidence in the technology and will serve as training for both practitioners and researchers, and will probably provide hints regarding the technology's usefulness, usability, and possibly limitations on scaling up and how such limitations can be avoided.
- Make sure there is enough support and resources behind the project because a successful research infusion will take both time and effort. If not enough resources are devoted to the infusion project and not enough organizational support protects the project, the risk is that more short-term goals, such as developing and delivering a customer project, will make the infusion project impossible to finalize.

Conclusions

In 2006, a research infusion project involving JHU/APL and Fraunhofer was successfully completed, resulting in some confidence that Fraunhofer's Software Architecture Visualization and Evaluation (SAVE) technology – which facilitates documenting, communicating, and managing software architectures – is applicable to software architectures in the aerospace domain. This successful infusion project subsequently led to a series of follow up projects. It also led to the discovery of other needs that could not be addressed by current technologies and therefore spawned the research and development of a new technology that will be ready for infusion in the future. Thus, in this paper, we were able to describe two research infusion paths: 1) The *direct infusion path* along which practitioners are increasingly willing to experiment with a new technology because there are growing indications from earlier infusions that the technology is applicable and provide value in a certain domain; and 2) the *indirect infusion path* along which feedback and results from infusion attempts lead to the identification of needs that are not yet covered by the current technology. These needs can spawn new research projects that lead to new technology that can later be infused into other organizations and projects to address similar needs.

Infusing new technology is difficult, so what makes an infusion successful? Second to the support from NASA IV&V, the most important success factor was probably the fact that the two organizations worked together as one team. The JHU/APL architects formulated questions and tasks to carry out that were important from the perspective of JHU/APL. The FC-MD scientists carried out the analyses and provided answers (sometimes several optional answers) to the questions. The team discussed the implications and carried out additional analyses when needed. Another important success criteria was JHU/APL's quick understanding of the SAVE technology, its strengths and limitations. This understanding was used early in the project to select a manageable piece of the ground control system to be studied for this limited project. The fact that a successful infusion has occurred was demonstrated through the application of SAVE to another of JHU/APL's systems without the involvement of any FC-MD scientists. In addition, other projects at JHU/APL were willing to take the risk to apply the technology. The fact that SAVE revealed some serious issues that otherwise would have been very difficult to detect helped create evidence that the technology works for this particular domain and that it provides some value. Once such evidence existed, other projects showed interest in the technology and were willing to try it out. This is what happened first with the SIE at JHU/APL and later with the cFS/cFE group at GSFC.

So what can we conclude about research infusion based on this experience? First, we would like to state that, from our point of view, successful research infusion occurs when a technology resulting from research is transferred to a real setting and is effectively applied to real world systems. Second, successful research infusion can occur when a new technology is mature enough, is user friendly enough, is packaged and documented enough, and meets a real need. Third, successful research infusion also depends on the complexity of the technology, i.e. while simple technologies require less support and training and therefore are easier to infuse, complex technologies have higher chances to succeed if they are carefully infused over time with the researchers' active support and the appropriate resources are available. This means that both researchers and practitioners have to have enough available time to spend on infusion of the technology and work as one team on infusing the technology. Fourth, successful research infusion can be a risky endeavor, but almost independently of the outcome, it serves both practitioners and researchers because it will always provide some feedback about how to improve the real world system and its processes as well as how to improve the infused technology, the infusion process, as well as the underlying research. Fifth, most importantly are the people involved in the infusion and the support they get from management and other parts of their organizations. Research infusion is risky and requires a lot of curiosity and interest in experimenting with and learning about new technologies. Since immediate results are not always available, the people involved, and their management, also need to be very patient. However, once the technology has been configured to fit the problems of a certain domain, it is likely that the willingness to try it out grows. If that happens, more and more evidence that the technology does work in that domain is going to be collected, and weaknesses in the technology can be used to start new research and development projects resulting in new technologies that can be infused in the future.

It is important to note that the lack of maturity of the technology, discussed above, can play an important role here. The reason is that early in the technology's lifecycle, when technology still has not found its final form, the researchers are still open to constructive feedback related to the technology. This gives the practitioner an opportunity to steer the technology development towards his particular needs. Such flexibility and willingness to change typically does not exist with fully developed tools and technologies.

Our most important conclusion is that it is crucial to provide an environment in which technology resulting from research can have an immediate impact on practice. Such an environment should also let practice have an immediate impact on how the research on these technologies will evolve. The result is mutually beneficial. Practitioners benefit from state-of-the-art technology with limited risk and researchers can align their goals with real needs. NASA's research infusion project is one vehicle that helps establish such an environment.

Acknowledgements

The authors wish to thank Lisa Montgomery and her NASA IV&V SARP program team for supporting this work; Mike Hinchey, Tom Pressburger, and Larry Markosian for helping establishing the SAVE research infusion project; Charles Wildermann, all members of the GSFC CFS team including Kequan Lu for their comments and discussions; Dirk Muthig, Jens Knodel, Lyly Yonkwa, Dominik Rost, Thomas Forster, Roseanne Tesoriero, Patricia Costa, Arnab Ray, Myrna Regardie and all members of the SAVE team as well as students for successful collaboration in the development of the SAVE tool; Leah Kelley for letting us use material from her report.

Bibliography

[1] D. Ganesan, M. Lindvall, M. Bartholomew, D. McComas, and G. Cammarata, "Analyzing the Core Flight Software with SAVE," in *The Second Flight Software Workshop* 2008.

[2] J. Knodel, M. Lindvall, D. Muthig, and M. Naab, "Static Evaluation of Software Architectures," in *Conference on Software Maintenance and Reengineering (CSMR)* 2006, pp. 279-294.

[3] M. Lindvall, I. Rus, F. Shull, M. V. Zelkowitz, P. Donzelli, A. Memon, V. R. Basili, P. Costa, R. T. Tvedt, L. Hochstein, S. Asgari, C. Ackermann, and D. Pech, "An Evolutionary Testbed for Software Technology Evaluation," *Innovations in Systems and Software Engineering a NASA Journal*, vol. 1, no. 1, pp. 3-11, Jan.2005.

[4] D. McComas, "Guidance Navigation and Control/Flight Software Framework," in *The First Flight Software Workshop* 2007.

[5] G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," in *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering* 1995, pp. 18-28.

[6] M. Reid, "Flight Software Architectural Modeling with SAVE," in *First Flight Software Workshop* 2007.

[7] W. E. Riddle, "The Magic Number Eighteen Plus or Minus Three: A Study of Software Technology Maturation," *ACM Software Engineering Notes*, vol. 9, no. 2, pp. 21-37, 1984.

[8] E. M. Rogers, *Diffusion of Innovations* Glencoe: Free Press, 2003.

[9] G. E. Stark, "Measurements for managing software maintenance," in *International Conference on Software Maintenance* 1996, pp. 152-161.

[10] W. C. Stratton, D. E. Sibol, M. Lindvall, and P. Costa, "Technology Infusion of the SAVE Tool into the Common Ground Software Development Process for NASA Missions at JHU/APL," in *IEEE Aerospace Conference* 2007.

[11] W. C. Stratton, D. E. Sibol, M. Lindvall, and P. Costa, "The SAVE Tool and Process Applied to Ground Software Development at JHU/APL:An Experience Report on Technology Infusion," in *IEEE Software Engineering Workshop (SEW)* 2007.