Proceedings of the Ninth IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 14-16, 2005, Phoenix, AZ, USA

PROGRESSIVE SOFTWARE SYSTEMS: DYNAMIC SOFTWARE FOR A DYNAMIC WORKPLACE

Kendall O. Conrad
Air Force Research Laboratory
Wright-Patterson AFB, OH 45345
Kendall.Conrad@wpafb.af.mil

Vincent A. Schmidt
Air Force Research Laboratory
Wright-Patterson AFB, OH 45345
Vincent.Schmidt@wpafb.af.mil

ABSTRACT[1]
Valuable time and money are spent developing software systems. Once a system is fielded, the operational and work requirements frequently require additional software changes over the life of the system, which increases system cost and can cause unexpected delays. This paper defines and surveys the progressive software system (PSS), an instance of what is commonly referred to as an evolvable software system (ESS). We describe the fundamental characteristics of the PSS and conclude with a discussion of PSS advantages, disadvantages, and usage tradeoffs.

KEY WORDS
Evolvable Systems, Dynamic Software, Dynamic Work, Progressive Software System, User Support Systems

1. Introduction

One of the biggest concerns when dealing with fielded software systems is upgrading and managing the software to add new and enhanced functionality as the usage requirements and the nature of work change. The standard technique has been to revise the source code and change the needed areas from the code level. Software engineering techniques have advanced enough to allow users to modify the software directly, and even give some software the ability to improve itself. A system of software that is able to change on its own is commonly referred to as an evolvable software system. While there are many categories of evolvable software, this paper uses the term "progressive software system" (PSS) to describe particular characteristics of software evolvability, and shows where the PSS perspective fits within the general realm of evolvable software systems. We begin by introducing several of the current technologies available for upgrading software through the use of evolvable techniques. After briefly looking at the current state of evolvable software systems, we consider the major factors of concern when creating an evolvable or progressive software system. Finally, the paper concludes by discussing the usefulness of progressive software systems, and talks about PSS advantages, disadvantages, and tradeoffs. This paper does not go into exact details of how to create a PSS system, but lays out guidelines and a framework for doing so.

2. Background

Complex software systems are specified to meet specific requirements, originally designed to support designated aspects of work. Once coded, tested, and fielded, users frequently discover new or unanticipated requirements that translate to additional or modified system features and capabilities. This also occurs as the work requirements change over the life of the system. Sometimes these needs can be defined by more functionality, different interfaces, or various types of alterations. When this happens the software developers have to go back and make changes to the source code. This causes a new version of the software to be developed and tested, which typically entails a host of time consuming and costly software engineering activities, including testing, configuration control, and documentation changes, in order for a new version to be produced and released back to the end user. This "upgrade" process often requires users to prioritize the enhancements according to time or monetary constraints, and can result in a modified system that still lacks parts of the needed functionality. The development process itself can also be a burden to the end users, since they may have to wait for long periods of time (sometimes a year or more) while the developers finish and test the changes [1]. But what if the software could progress over time without a need for substantial developer intervention? What if the software could improve itself by adding new functionality, such as the ability to process data faster, or possibly even have a different visual appearance? That is the conceptual basis of evolvable software systems.

The costs (in time and money) involved in having custom software professionally developed and modified are traditionally considered to be characteristic within the software development community. Research has continued in areas for speeding up software upgrades and finding where software updates are needed [2], but these techniques are mainly intended to address the initial design of the system and its maintenance by trained professional software developers. The useful life of the software could be increased, and the overall life cycle cost

[1] Document approved for Public Release AFRL-WS-05-1325.

potentially decreased, if some modifications could be made by the end user or automatically by the software. Conceptual models for these modifications have been biologically inspired, seen in areas such as evolutionary algorithms. Evolutionary algorithms were invented in the 1960s, but weren't very practical until about the 1990s when computers had become advanced enough to use them [3]. With such methods, a system automatically changes itself to find the best possible solution. Since the best solution is not known from the start, the system tests various alternatives. These evolvable systems keep track of the various ideas tried, gravitating toward the most viable solution, and recreate the system based on the best results. These systems continue to evolve over time until all possibilities have been tried or trial limits have been reached.

There are various categories (perspectives) of evolvable software systems (ESS), based on evolutionary goals and techniques used to achieve the evolutionary capabilities, and the degree to which the system evolves. In general, a full ESS is typically considered to be a software system capable of evolving autonomously, without direct inputs from a user. It is able to advance and evolve completely on its own. Software systems based on this model will generally be able to create complex and innovative designs, and are able to run independently once started. Clearly, such a system takes much longer to create and validate than a traditional software design. These software systems might require a supercomputer for execution.

Another perspective on ESS is the morphing system [4]. This type is based on operating system and compiler technology designed to support continued improvements by collecting and managing profile information. Here, the goal is re-optimizing a program during execution. This style of system allows for a low execution overhead and is able to integrate optimization changes without requiring alterations to source code. Real-time modifications are made in the background without user intervention. In addition, these adjustments are generally based on user profiles and are rather limited, only useful for simple changes that impact a single user. Although morphing systems evolve, they do not support modifying and updating user interfaces or adding to system functionality.

Self-evolving software systems (SESS) are another perspective. The creators of the SESS concept say it is "capable of automatically detecting when changing external circumstances or internal conditions can be better handled by alternative software modules and able to dynamically swap these modules into place" [5]. This type of system's strength lies in its ability to maintain itself without intervention from the user by using swappable software modules that allow for reasonably complex changes to be made to the system. The modules that support this system need to be predetermined, making overall changes limited. Since the creation of the self-evolving system is complex, there must also be very reliable self-testing procedures to ensure that available modules do not deteriorate system performance.

A reconfigurable software system (RSS), unlike other techniques described here, is more familiar to most contemporary computer users, and is much simpler to implement. The RSS is a software system that allows for users to make cosmetic and basic adjustments to the software. Since the allowed changes are primarily cosmetic and don't impact overall functionality, these systems are generally straightforward for end-users to employ and much easier for software developers to create. On the other hand this system is very limited in what can be modified, and all possible changes must be predetermined. Examples include software that allows users to select default font, colors, and menu options. The RSS is the lowest and most basic form of ESS.

Another perspective to ESS is interactive evolutionary computing (IEC), in which software modifications are identified by the system. Here, the human user provides the fitness function. This system design is most useful when the form of the fitness function is not known ahead of time or cannot be determined by the computer. These systems may identify changes automatically, but rely on direct human interaction to decide if each and every change should be implemented. For example, the system may suggest a change if the user finds it to be visually appealing. Multiple users can concurrently participate in making these evaluations. A significant drawback of IEC techniques is that the number of evaluation functions is limited by user fatigue, since human evaluation is slow and expensive.

End-user development (EUD) is an ESS technique that allows users and power-users to create or modify software using built-in macro functions and capabilities [6]. This permits users to modify applications as desired, encouraging social creativity and innovation [7]. EUD takes advantage of meta-design, which addresses the problems of closed systems. Even though meta-modifications are available, updates and new functionality may be limited, and the training involved in enhancing some of these systems can be quite extensive and costly.

The final perspective this paper looks at is progressive software systems (PSS). The PSS is a software system in which incremental low impact changes are made autonomously by the system, but the user drives the high-impact changes. In this system, capabilities could be provided in the form of a service using local or remote software agents. This architectural approach makes a PSS easier to implement than a EUD system, and does not require explicit training for its users. New additions to system functions could be incrementally added using upgrades from the service agents. Although these agents can provide the capability for the user to make low-risk changes, designing agents to do more complex changes to the system will be much tougher to produce. The term

"progressive software systems" is also used by other groups [8], [9]. Their usage of this terminology seems comparable with the usage in this paper.

The long development cycle of new software, coupled with the dynamic nature of work (changing of requirements once a system is fielded) makes it appealing to extend the useful lifetime of software systems with approaches that minimize formal developer involvement and maximize end-user input to changing needs. Thus, our goal is to determine the architecture for a software system whereby the system can integrate low-risk changes, but the user can specify and initiate changes that add new or modified functionality, all without the need to undertake a major software redevelopment effort. Identification of the ESS perspective that best meets this goal is an initial step to successfully implementing such systems.

From the definitions of various types of ESS, the PSS is probably the best overall fit. This is because most significant software changes will be driven by end-user request, while the system autonomously integrates the less significant changes. Also, PSS allows for adapting to changing circumstances in the work that the user is doing. The software we desire to create follows more of a progressive change than an evolutionary change, though both are very closely related. Both alter the system for the better, advancing the current system into a superior system. The biggest difference is that almost all changes are autonomously initiated and completed by the system in fully evolvable systems, but progressive software systems rely on the user to initiate high-risk changes and only make the lower risk changes autonomously. The remainder of this paper highlights our ideas of how PSS concepts support evolvable system design.

3. Issues of PSS

The issues involved in the design of a progressive software system need to be considered early in the design. PSS software has all of the same components as a regular software system, but also contains additional components that allow it to grow and progress. Some of the key features include:

- Identifying when the user wants to upgrade the system.
- Supplying the best design for the upgrade or supplying choices of good designs.
- Implementing the desired upgrade.
- Having a way to revert to a previous state or undo a progression.
- Checking the stability of the software; making sure the system performance hasn't degraded.

A progressive software system can only advance to a point before a programmer is needed again to make more permanent upgrades or bring the system up to date with newer technologies. Programmers ultimately have the most control over the system allowing them to make much more complex changes to software than a PSS can achieve on its own. Many of these changes can be made incrementally and modularly using system agents. Technical feasibility can also be an issue: creating a PSS is not common or easy to develop. Experienced software programmers are needed to consider the design and implementation issues. Finally, there may be some features that the PSS cannot accomplish without a significant rewrite. Identifying these issues in the early stages of design is a risk-mitigating strategy.

4. PSS Break Down

Examining the progressive software system at a more detailed perspective gives us a breakdown of how it works. The main PSS components we consider make a progressive software system are:

- Self-testing - Checks if a change in the system would be harmful to the system itself.
- Self-maintaining - Manage changes made to the system.
- Evolve-ability - The software system itself has the ability to implement changes rather than needing a programmer.
- Undo Ability - Allows the user to return to a previous system state.

In addition to this breakdown, Figure 1 depicts orthogonal views of PSS component design (capabilities, patterns, and implementation techniques). Although there is not a specific technique or mechanism we currently suggest when pursuing PSS designs, examining a PSS from each of these views helps to ensure both completeness and expandability in system design. The following sections give examples of the contents of these views.

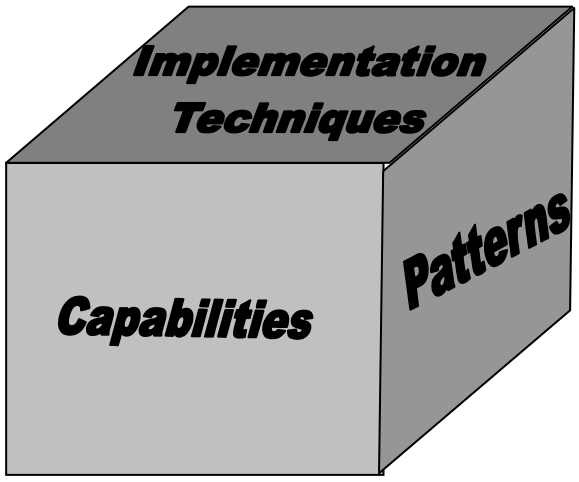

Figure 1: Using PSS Orthogonally

4.1. PSS Capabilities

Listed here are just some of the possible uses that that a PSS could provide for software:

- Optimization of tasks – The system observes patterns of user behavior and offers a pattern-oriented solution. This type of optimization may observe that a user types a particular phrase repeatedly in a certain field and offer a faster and more efficient way of inputting the data with a word completion utility. The system may also observe the user seeking out a particular option in a menu often and offer to place a shortcut icon on the toolbar for them.
- Human Computer Collaboration – The system realizes humans and computers have complementary strengths and weaknesses. Computers do computations very quickly in comparison to humans, whereas when it comes to creativity and aesthetics the human makes decisions more rapidly and effectively. The system can offer to help with the user's tasks if it believes it is a task which it could do more effectively.
- Intelligent GUI – The GUI may require changes over time to accommodate the changing nature of user work. The system observes operator behavior and makes changes accordingly to the interface to allow the user to more effectively do their work.
- User-Defined Functions – Allows for user-created functions to be added to the software. There may be an existing library of functions, but with the ability for users to add their own defined functions to the list.

4.2. PSS Patterns

When considering some of the different PSS capabilities, some patterns or categories seem to emerge. These patterns can be developed into a patterns library, facilitating a more complete and mature software system design. A patterns library is a set of designs that have been proven in prior designs and can be applied to a multiple number of situations as a solution. Example generic patterns include:

- Task Optimization – Speeds work progress and improves reliability.
- User Workload Reduction – Reduces workload by making changes in the system to remove burden from the user.
- Code Stability - Ensures software code does not degrade or lose reliability/stability.
- Graphical User Interface - Allows users to make modifications to look and feel of the system.
- Functional Changes - Allows user to make modifications to specific functionality.

Multiple patterns can be used in solving the same situation: optimizing a task can be at least partly solved using the Tasks Optimization, Graphical User Interface, Functional Changes, or User Workload Reduction pattern. These generic patterns can be used to partially describe system design, without committing too much to implementation details.

4.3. PSS Implementation Techniques

A progressive software system can be implemented using a variety of techniques. This partial list highlights several methods, which are not specific to PSS. However, all of these methods can be used to make allowances for progressive software improvements.

- Patterns Library – Having a set of pre-tested GUI/functionality templates that the software has access to for creating new functions or new displays.
- Profile driven – The system acquires data from the user as they use the software, watching for patterns and analyzing the data to help optimize work.
- DLLs – Dynamic Link Library contains functions and other information, which can be updated for the software to use as needed. May be implemented by end users or administrators depending on design needs.
- Plug-ins – Would allow for adding new functionality to existing software and does not require the software to be recompiled.
- Macros – Scripted steps for accomplishing a task. Macros may be too sophisticated for casual end users to implement on their own, so an administrator or power user may be needed to design them.
- Agents – An agent acts at a near administrative level overlooking systems. Implementing a subscription-based service allows a user to subscribe for desired updates.

4.4. Pros and Cons of the PSS

There are benefits and disadvantages of designing and using a progressive software system. The most distinct PSS advantage is that the useful lifetime of the system is increased due to built-in extensibility. This means that those using PSS software can expect to amortize the overall cost of the software of its extended lifetime, without taking it offline for upgrades. The system's users will have more power and control over their software and its functionality, which ultimately creates a healthier work environment [10].

Since a PSS is able to gain new functionality, this makes it more valuable to the users and keeps them from having to find alternative ways of accomplishing their work (i.e. using a separate spreadsheet or document editor to keep track of data). PSS allows for system enhancements as the nature of work changes over time. For example, if a data feed needs to be updated or is no longer available, the

system can be easily modified to locate and use a new data feed. A PSS will be able to allow many such changes to be made, thus allowing the software to perform properly.

There are also disadvantages to using a PSS. The degree of flexibility built into progressive software systems is expected to cost more in development time and money due to increased system complexity, even though the useful lifetime of the software should bring the overall system cost down. Development efforts should improve as the practices involved in developing PSS advance, reducing this cost.

One of the added PSS design requirements is deciding which users will have the ability to make changes to different aspects of the software. Giving every user the ability to change all aspects could create an unstable software environment, potentially preventing systems from communicating properly. Addressing this issue increases the complexity of the design phase. This problem is essentially a version of the multi-level security situation that must be resolved in the Air Force's Network Centric Warfare environments.

Also, the design must provide a way for the system to ensure that changes being made do not reduce the integrity of the overall system. Whether through tests or system constraints, the design must ensure the integrity of the system through a range of continuous changes. Designing these tests will add time and complexity to the development process and will be vital for PSS success. This is an instance where a patterns library would be very helpful allowing designers to use "tried and true" designs rather than spending time creating new ones that need testing.

Finally, there may be several copies of a certain PSS existing in many locations. When this occurs, it will be important to maintain a strict configuration management mechanism to track and link related changes in sibling PSS instances.

5. Making Sense of Using PSS

The progressive software system strives to automate the low-risk modifications, while allowing the human user to initiate and decide the high-risk changes. Figure 2 indicates a notional view of the tradeoffs associated with a range of progression types. The graph displays the complexity of the modification along the x-axis, with simple modifications occurring at the left end, and nontrivial changes on the right. The y-axis shows the expected likelihood of a modification. The diagonal line indicates our rough expectation that the casual end-user will be interested in simple modifications in a much greater number than nontrivial changes. The strict linear relation shown is for visual guidance only. We've mapped the anticipated complexity of some of the systems discussed early in the paper onto the x-axis for reference.

The graph also presumes that the simpler modifications are expected to be low-risk changes with minimal impacts on functionality and testing. Such changes might include screen format changes or the added display of an existing database element onto the screen. Similarly, we would not expect very many high-risk changes to occur throughout

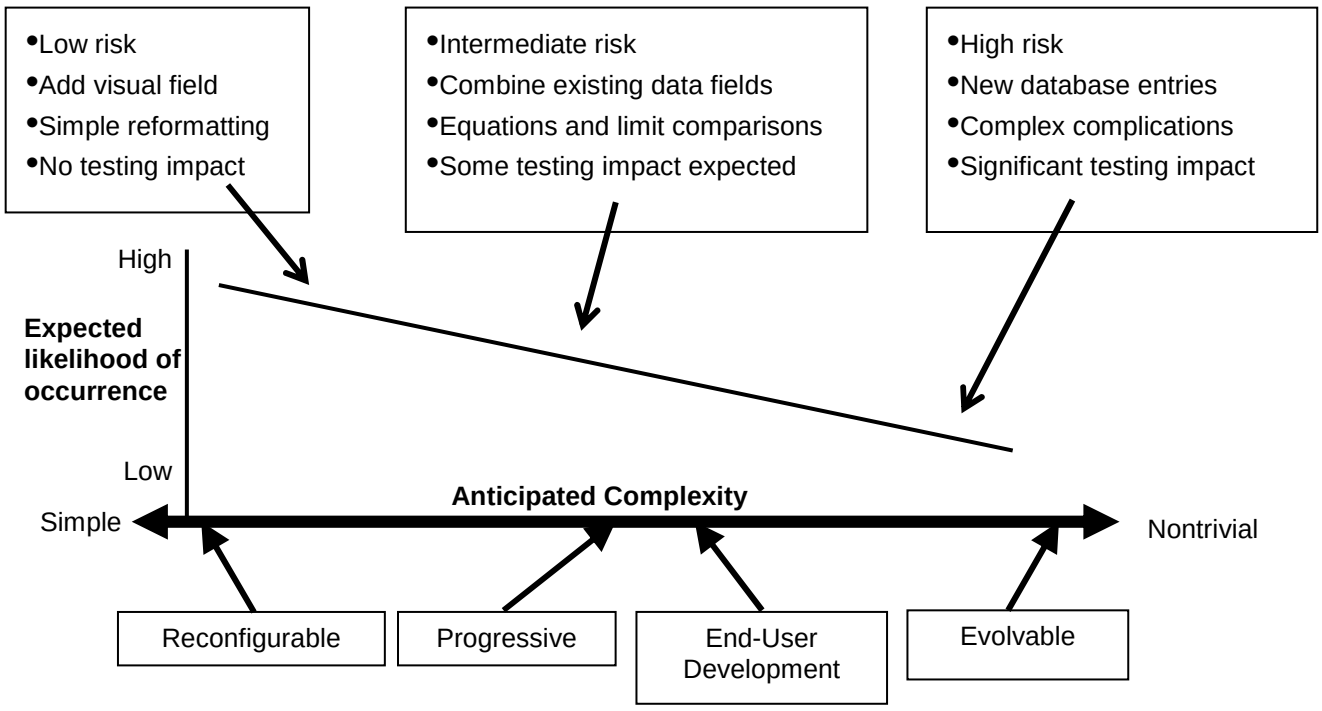

Figure 2: Usability of PSS

the system's lifetime. Such changes would be complex and have an extreme impact on functionality and testing. High-risk changes might include the addition of new database fields or a new visualization of data. A lifetime of intermediate-level changes is to be expected, with some impact on functionality and testing. These changes might include bounded calculations or the combination of existing database entries.

There have not been enough data taken from software systems to create a more detailed graph than the one in Figure 2. The graph gives the impression that these trade-offs are exactly correlated to one another. Though this is not likely true, it is believed that there is a strong correlation between tradeoffs. As a system resembles an ESS more and more, the design complexity will increase, and so will the associated risks. (When referring to risks, this means the different risks that arise during design such as costs, security, time, manpower, etc.) The graph provides a general understanding of the trade-offs and shows how they interact.

At the left end of the PSS spectrum graph, a system is more of a reconfigurable software system (RSS) where the techniques used are very basic. At the right end of the spectrum are fully evolvable software systems (ESS), representing systems such as NASA's evolvable system that creates antennas independently [3]. The graph suggests a designer or user is more likely to consider modifications that are closer to the RSS side rather than the ESS side of the spectrum. Despite this tendency, the PSS will support the full range of operations.

6. Conclusion

The progression of this paper has shown the reasons for the emergence of software systems that have the ability to be improved without the explicit aid of software designers. We have surveyed evolvable software systems, and distinguished the differences between various types of these systems. We have also examined some of the possible implementation techniques that a designer could use in order to achieve a PSS. The paper broke down PSS into its constituent parts and looked at each in detail, finding out what makes up a PSS.

Within the breakdown of the PSS several capabilities were mentioned that can be accomplished using a PSS, such as the user defined functions, intelligent GUI, etc. Design patterns identified were task optimization, code stability, graphical user interface, functional changes, and reduce user workload. We also reviewed several PSS implementation techniques. These included patterns library, profile driven, DLLs, plug-ins, macros, post-compilation, and agents. These orthogonal views constitute different ways of looking at PSS. The key idea behind this orthogonal layout is that there is no set way of creating a PSS; it is open to creativity and does not constrain the designer. The PSS concept is not intended to be a new design methodology of architecture.

Progressive software systems have existed for some time now, and as more software is developed with the PSS ideal in mind, patterns will become more clear and detailed. Additionally, as software technology grows new implementation techniques will arise that will allow PSS to be more easily created and be given more power.

Research into applying PSS concepts to existing software is one of the next goals. This will enable current non-PSS systems to be enriched with the new ideas created by PSS. PSS seems to be the current trend in software development. This trend may not be called PSS by everyone, but the ideas are still be the same. The world needs dynamic software for the dynamic work place.

References

[1] Roth, E., Scott, R Deutsch, S., Kuper, S., Schmidt, V., Stilson, M., Wampler, J. (2005). Evolvable Work-Centered Support System for Command and Control: Creating Systems Users Can Adapt to Meet Changing Demands. *To appear in 2006 Special Issue of Ergonomics on Command and Control*.

[2] IVA (2003). Instability Visualization and Analysis Proposal. *UCSC Engineering*. www.soe.ucsc.edu/research/labs/grase/iva/

[3] Evolvable Systems Group (2004). *Evolvable Systems*. NASA AMES Research Center, www.arc.nasa.gov/exploringtheuniverse-evolvablesystems.cfm

[4] Zhang, X., Wang, Z., Gloy, N., Chen, J., Smith, M. (1997). System Support for Automatic Profiling and Optimization. *Proc. of the 16th Symp. on OS Principles*.

[5] Dellarocas, C., Klein, M., Shrobe, H. (1998) An Architecture for Constructing Self-Evolving Software Systems. *Proc. of the 3rd International Workshop on Software Architecture*.

[6] Fischer, G., Giaccardi, E. (2004) Meta-Design: A Framework for the Future of End-User Development. *Kluwer Academic Publishers*, Dordrecht, Netherlands.

[7] Fischer, G., Scharff, E. (2000). Meta-Design: Design for Designers. *Symposium on Designing Interactive Systems.* ACM Press, New York, NY.

[8] Horizon Business Services (2004). www.caterease.com

[9] Xpient Solutions (2004). http://www.xpient.com

[10] Israel, B. A., House, J. S., Schurman, S. J., Heaney, C., & Mero, R. P., The relation of personal resources, participation, influence, interpersonal relationships and coping strategies to occupational stress, job strains and health: A multivariate analysis. *Work & Stress, 3*, 1989, 163-194.