A Framework for Evolutionary, Dynamically Updatable, Component-based Systems

Robert Bialek Eric Jul
University of Copenhagen
Department of Computer Science, DIKU
Universitetsparken 1
DK-2100 Copenhagen, Denmark
E-mail:{bialek,eric}@diku.dk

***Abstract*—**
Many distributed applications must provide 24/7/365 availability making the update process of such application a challenging task. Apart from the problem of performing the updates without closing the application there is a problem of dealing with potential version conflicts between application's components. We develop a framework (called DUCS) that facilitates evolution of component-based distributed applications. An example of a simple chat application's dynamic update is used to illustrate what is needed for framework supporting on-line updates and evolution of distributed component-based applications. The developed framework is a three-layer framework that separates update from application logic. DUCS supports dynamic component replacement, state transfer, and interface modifications by utilizing interface adapters. The framework can be built on top of standard virtual machines. A prototype of DUCS, is being developed on top of the Java VM and some implementation details are presented in the paper.

I. Introduction & Motivation

The popularity of component based frameworks, e.g., EJB, and the emergence of internet communication standards, e.g., SOAP, facilitate building large distributed component-based applications. Many distributed applications must provide 24/7/365 availability that is, they must operate continuously. Moreover, as much as 50–70% of the total life cycle cost of an application is spent on evolving the system *after* the initial implementation [4]. Therefore, the update phase is a major part of the whole software life-cycle and optimizing it is important. For long running applications this means that there is a need for systems that allow on-the-fly updates of their software. This is especially true for evolutionary systems.

Updating software is usually done by stopping the application and performing the update whereafter the application is restarted in the new version. Such a stop-and-update method is often too rigid for many applications, e.g., financial services, thus hampering the evolution of such applications.

One of the present solutions to the stop-and-update problem is based on redundant hardware: When updating, an extra server (backup server) is introduced. The backup server overtakes the request processing, meanwhile the main server is being updated. After the update the backup server is removed and requests are again directed to the main server. Although this solution supports dynamic updates, it has the following drawbacks:

- Limited applicability meaning that it works only when the new version is backwards compatible with the old one, i.e., the old server's interface and functionality must be a subset of the new version.
- Unnecessary complexity. Apart from the two servers, we need a gateway solution that will redirect the requests to the correct server. We can risk inconsistency between the versions of the server which may lead to complex synchronization problems.
- Cost of the redundant hardware may be too high in some cases, e.g., satellite systems.

A different approach is to provide a software model supporting dynamic updates. Component Based Modelling (CBM) is a programming model supporting updates: First, components are implementations and architectural abstractions at the same time, which makes the architectural changes easier. Second, extensions are made (almost) independently of other components [3]. CBM is a model supporting architectural changes but it requires additional techniques to perform the updates dynamically.

There is a number of dynamic software update systems:

- For updating programs that are beforehand prepared for the updates [13],
- That support updates of the code but not the interface, or
- That support incremental interface updates only
- That can update only specific parts of an application[18],
- That complicate the update by mixing the update code into the application logic.

All these systems have some drawbacks or unnecessary complications.

The goal of this project is to provide a model for a framework supporting evolution of component based applications and build a prototype for it. The framework shall have the following properties:

- It shall facilitate update process of distributed applications by supporting dynamic object replacement, state transfer, and reflection,
- It shall not complicate the process of building the applications using it,

- It shall be able to be applied to the existing applications with little or no impact on them,
- It shall support automatic propagation of updates,
- It shall be expandable and modifiable.

The goal of the prototype is to identify all (or at least major) requirements for the framework, present how to use the framework, and to evaluate its impact on the applications using it with respect to performance, memory usage, etc.

The paper is structured as follows: In section III, we present an example of an update of a work-flow component-based application and what is needed to perform it correctly. In section V, we present the resulting framework. In section VI, we give some implementation details of our prototype. Finally, we present related work in section VII) and the summary of the report in section IX.

II. Assumptions

Gupta[12] showed that we can not guarantee correctness of an arbitrary update performed to an arbitrary program, meaning that we will not always be able to result in a reachable program state. Therefore, updates must be related to a particular program or program structures[6]. We focus on component oriented programs and on their updates.

We have the following assumptions for the component-based system we are building:

1) The number of components in the application can vary freely and on-the-fly.
2) Application components are only intermittently connected. This assumption is related to the typical client-server applications where clients connect spontaneously and we cannot control the number of concurrently connected nodes.
3) Distributed applications are executed in a heterogeneous environment.
   It is unrealistic toexpect that the implementation language, virtual machine, application framework, hardware or available network bandwidth are the same.
   In the prototype however, we focus on a single implementation.
4) Some parts of the applications are expected to be operational all the time. The continuously running components are assumed to be of high quality meaning that they do not (often) crash. Clearly, often restarting applications would not need dynamic updates facilities.
5) Applications are executed in Virtual Machines. We believe that distributed applications will be written in programming languages that execute in virtual machines (e.g., JVM [2] , CLR [1]), which abstracts the hardware layer making the applications mobile.

III. Illustrating application update

We use a small distributed application to introduce and demonstrate our ideas. We describe the application and what is needed to perform an update of the application in several steps:

1) We start with the application example in section III-A by showing a pseudo code for the first and second version of the application.
2) Then we look at the changes between the versions. We introduce the definition of updates, dynamic updates and correct updates based on the relation between the two versions and the way the updates are perform (section III-B).
3) In section III-C we illustrate how we perform the dynamic updates.

The application is a distributed groupware system: a chat application that allows users to communicate by exchanging text messages through a server.

A. The application

The application includes two versions:

1) The text chat application, which is the original version,
2) The text and graphics application which is the one after the update.

1) Text Chat Application: A simplified version (version 1) of the chat server and a chat client component might look like (written in a Java-like style):

```
component ChatServer {
  provides:
    Result connect(from) {
        addToConnectedList(from);
        return OK;
    }
    Result send(from, to, txt) {
        return to.put(from, txt);
    }
}

component ChatClient {
  requires:
    Result connect(from);
    Result send(from, to, txt);
  provides:
    Result put(from, txt) {
        DisplayFrom(from);
        DisplayText(txt);
        return OK;
     }
}
```

To be registered at the server and later to send the messager new clients call `ChatServer.connect(me);` and `ChatServer.send(me, to, txt);` for sending messages. `me` identifies the client component and `to` describes the destination address, which is acquired from the server. The server merely pushes the text messages to the client using its `put` interface (`txt` comes from the user interface on the client node).

With these components we can have many clients connecting to one server. The clients will be able to send text messages to one other as long as the server component is accessible. To allow continuous operation of the group-ware application, the server is required to be available all the time.

2) Extending the Text Chat Application with Graphics: We wish to upgrade the server (and clients) to support graphics along side with text. The server definition desired in version 2 could be:

```
component ChatServer {
provides:
  Result connect(from) {
      addToConnectedList(from);
      return OK;
  }
  Result send(from, to, txt, grph)
  {
   return to.put(from, txt, grph);
  }
}
```

where `grph` is some graphics. The relevant client code could be:

```
component ChatClient {
  requires:
    Result connect(from);
    Result send(from, to, txt, grph);
  provides:
    Result put(from, txt, grph) {
        DisplayFrom(from);
        DisplayText(txt);
        DisplayText(grph);
        return OK;
     }
}
```

Now users can send both text and graphics to one another using the new server interface `send(from, to, txt, grph)`.

B. Evolutionary changes

The progress from version 1 to version 2 is the evolution of the program. Evolution is a gradual change that takes place over many generations or a process of gradual and uninterrupted changes over a period of time [16]. Changes in software are not random. It was shown that only a certain portion of a system, a portion that remains quantitative within a relatively constant bandwidth changes between releases. This can be observed in our example: the changes happen gradually and address a limited functionality only. The evolution of the components' interfaces can be expressed as a relation between the old and the new version. `send(from, to, txt)` $\rightarrow$ `send(from, to, txt, grph)`. Based on the relation we can build a function that transfers us from the old to the new version. Such functions are called *interface adapters* (for more detail see section V-C).

```
send(from, to, txt) {
  return send(from, to, txt, NULL);
}
```

and back from new to old:

```
send(from, to, txt, grph) {
  return send(from, to, txt);
}
```

We expect that the programmer can always reason about and provide the relation between the two evolutionary changes between the old and the new version of components. If the relation can not be provided then we do not treat such modifications as an evolutionary changes.

To clarify the terminology, we introduce the following definitions: An *update* is the modifications to an application's component where a relation between the old and the new version of the component can be provided. If the components have the same functionality and interfaces, the relation can merely be empty. Consequently, we define a *correct update* as a transition (or a set of transitions) from the old to the new version of the application component allowing as to use the new component instead of the old. For example, the stop-update-restart in new version scenario is a correct update. Because the server, in our example, is the central component in the application and it should stay operational continuously, performing correct updates must be performed without stopping the server. Such update that happens dynamically we call *dynamic update*. And finally we define a *global update* as a set of dynamic or correct updates resulting in replacement of all components involved in the application with their new versions. We see that application code is being changed by the updates. The program code responsible for managing the modifications in the application code will be referred as the *meta-level* or the *update-level* code.

We support correct and dynamic updates. To guarantee that dynamic updates are performed correctly we address several issues described in section III-C. Furthermore, updates may lead to inconsistencies in the system. We will analyze how to handle them in section IV.

C. Perform dynamic updates

Reaching the application state presented in section III-A.2 requires several steps. First, we need to define what we will be updating. To do so, we define an *update request* which consists of `{ComponentName, NewComponentDefinition}`. Update request is used to communicate what we need to update. The update process is initiated by sending the update request to the application's meta-level (which we call the *Update Level*). Before the Update Level starts performing the dynamic update, it needs to check whether the update request is correct. In section III-C.1 we discuss how to check the updates. In the remaining of this section we will analyze when and then finally how the updates should be performed.

1) Checking the update request: As our focus is on applications that have long life-time and must be reliable, we expect that the new version is again a reliable application. Therefore, similarly to Bierman et.al. [**?**] we use a type checking systems to check that the updates do not brake the application's type compatibility. From the NewComponentDefinition, type information is extracted. The type information is used to perform the type check. The new type should be compatible with the component's provide interface by including the necessary interface adapters if needed. In case of type mismatch and lack of the necessary interface adapters the update process is stopped and an optional exception is raised. Otherwise, the update process is started.

2) Timing of the updates: During the update, the process instance of the old component is replaced by the new instance and this must happen at an ”appropriate” time. Dynamically enforcing conditions for correct updates requires runtime support

[14] or they must be performed on inactive code only [13]. We want to support updates of both the active and inactive code.

When exchanging the inactive code the update functions are called by the reflection mechanism before and/or after the call to the components method. The reflection mechanism calls the update layer to check whether the component is to be updated. If so, then the update process described in section III-D starts, otherwise the operation continues. If the component is to be updated but it is in operation, we wait until it is finished. If some components never terminate or have very long execution times we need to perform a hot-swap of such components. Programmers should be aware that some components behave this way. Since the time of updating such code can not be defined [12], we require the programmer to indicate a place for correct update, which we call an update checkpoint. From that point the process is migrated to the new version using the check point state for the state transfer function.

We refer to Szyperkski's definition of components saying that they are units of composition with contractually specified interface and explicit context dependencies only. Components, in contrast to objects, do not have any externally visible state [19]. Infact, a components' state is only visible from its meta-level and consists of component's local variables and program counter. Therefore, to handle such updates, we must provide a *State Transfer Function* (STF) to do the transformation **??**. The STF is therefore attached to the Update Request resulting in:

`{ComponentName, NewComponentDefinition, STF}`

D. Performing the dynamic update

When updating components, we operate on the meta-level, where we must maintain the state of the modified components during their updates. Because state transfer function, as well as the update process, are meta-level operations, they should not influence the application semantics. To call the update code (on the meta-level) we can either use the synchronous or asynchronous method. It should be noted, however, that update process can not be executed together with the updated code. This requires introducing synchronisation methods like semaphores or monitors, which complicates the update logic. Additionally the potential deadlocks must be avoided. This complexity, however, can be fully integrated on the update layer. The easier solution is to use the synchronous invocation of update functions, thus simplifying the the update-process on the cost of application's performance.

IV. Coordination with Old Versions

Supporting the dynamic updates only facilitates changing the system on run time. However, the problem of obtaining a system version with incompatible components is not solved. We can not force all the components to update to the new version because we deal with heterogeneous networks with spontaneously connecting nodes, updating applications in one atomic step is not possible. Additionally, because of hardware and bandwidth limitations, we may not always be able to perform the necessary updates. Therefore, we need to deal with simultaneous cooperation of components with different versions.

In our example, even if we could update the server, we would introduce some component inconsistencies, namely the mismatch between `send(from, to, txt)` and `send(from, to, text ,grph)`. The programmer must therefore analyze the consequences of an update and decide how to handle the potential incompatibilities. There are three options for managing such situation:

- Ignore.
  Leading to avoiding old components and making them useless in the new environment. Usually forcing the users of the components to update them on their own.
- Force.
  To trigger automatic update before the other start communicating. This option has, however, major drawbacks: The running components must be updated either in a stop and restart way, which leads to application errors, or by blocking the whole application by waiting until they finish and then updating them. Furthermore, the other part of the system is required to support updates.
- Adapt.
  To provide adapters for the old version, making it possible to communicate with both the new and the old version components. It has the advantage of supporting seamless evolution of the system. COTS components as well as component in environments not allowing updates are also supported. To support the adaptation we must provide the interface adapters which results in the extension of the update request: `{ComponentName, NewComponentDefinition, STF, InterfaceAdapters}`
  If an interface does not change between two versions, the interface adapter can merely be empty. Interface contracts are not just the interfaces, they can additionally include pre- and post-conditions, as well as non-functional requirements such as space/time constraints, security guarantees, etc. [19]. Changes to the interface contracts should therefore be handled in the interface adapters.
  This methid can be further expanded with update propagation functionality (see section V-E).

V. Layout of the framework

In this section we provide the full description of our framework. We start with the description of framework's layers. Then we present the update request, interface adapters and the state transfer function.

A. Layers

Similarly to our previous work[5], DUCS separates application logic from the update logic. The separation is made to simplify the complex process of performing the updates by assigning responsibilities to different modules.

Our framework is spit into several layers as showing in figure 1:

- VM layer is the run-time environment in which the application is being executed. In our case the VM layer includes a standard Java VM, on top of which we added a meta-VM that provides the functionality of component

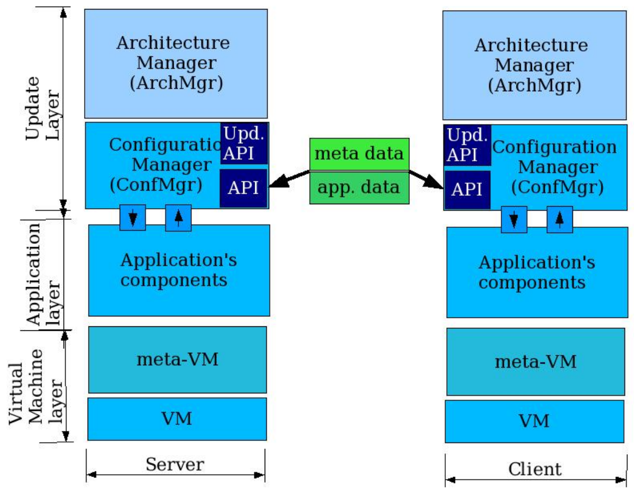

Fig. 1. Three-layer model

Model

loading, unloading, versioning and updating as well as reflection. These functionalities are needed by the Configuration Manger layer, which is a part of the update layer.

- Application layer is the layer, where applications consisting of components are built. The communication between the components goes through their interfaces, which are controlled by their meta-level component, the Configuration Manager.
- Configuration Manager (ConfMgr) layer is the meta-level entity responsible for performing updates on single nodes. The ConfMgr handles object updates and manages interface changes by supporting pluggable adapters. The ConfMgr, provides an update-interface for receiving *Update Requests* (see V-B) that initiate the update process. The ConfMgr can be expanded with additional functionality like security control, proof carrying code, encryption, and more.
- Architecture Manager (ArchMgr) layer is responsible for managing the configuration of components across many nodes. It manages addition, removal, movement and updates of groups of components. ArchMgr is only contacted by the ConfMgr when interface adapters can not handle a version mismatch.

B. *Update request*

Update requests are used to initiate the update process on the Configuration Manager layer. Similarly to Boyapati, et.al., [7] and as presented in the previous example components updates consits of tuples: `{ComponentName, NewComponentDefinition, STF, InterfaceAdapters}`

NewComponentDefinition must include the reified information about the component. It should at least include the type information to allow type checking on update arrival. One update request may include many such component updates together with user information. User information is needed to authorize potential updates by the update layer.

C. *Interface adapters*

Interface adapters are the update layer's functions that solve the mismatches between the new provide side of the interface with the old require sides of other components. Applications should not be influenced by the interface adapters, since they are there only for supporting unfinished global updates.

Interface adapters wrap the provide interface into the adapt functions in the following manner:

```
OldResult adapterName(pars) {
  newPars=convertToNewVersion(pars);
  result=callTheCorrectMethod(newPars);
  oldResult=convertToOldResult(result);
  return oldResult;
}
```

Where `adapterName` represents the name of the method to adapt. Interface adapters can perform other functions then merely the adaptation. For instance, they can perform the pre and post condition checks of the called components.

D. *State transfer function*

State transfer function is to be provided for components which have to be updated while running. In the runtime, we support swapping the application at well-defined places in the application. For such places (e.g., at the beginning of a loop), it is relatively easy to define how to move the program state from the old to the new version. A state transfer can therefore look as:

```
STF(old, new) {
  new.setVar("a", old.getVar("a"));
  ...
  new.setCheckpoint(old.getCheckpoint());
}
```

In the run-time we have to provide the setVar, getVar, to set and get the variable values and set/getCheckpoint to get the current program checkpoint ID. They requested functionalities must be implemented in our extension of the VM in the meta-VM layer .

E. *Propagation of updates*

We can update client components in a fashion similar to the server component's update or we can let it restart in the new version. When the version 1 client contact the version 2 server, the interface adapter function is called and the server calls the meta-level code to perform the adaptation. At that point, the server can attach the update request to the result package which is to be send to the client. In this way, the update request would be propagated in the system. The additional functionality of reacting on version mismatches can be freely decided on the update layer thus supporting flexible update protocols. The only functionality that is required is the ability to contact the other components by attaching the update request, or a small information about availability of an update, to the result message.

The corresponding client could then actively contact the server to receive the update. A dedicated protocol would then be required on the update layer.

VI. Implementation details

Our prototype is a java implementation of the component based systems. Each java class corresponds to one component.

A. Automatic conversion

To automatically convert the java classes to components we use the Javassist [8] framework, which allows run-time creation of wrapper-classes (components).

B. Naming convention

For the prototype we use the dynamic class loading facility in java[15]. Dynamic class loading supports incremental loading of missing code, in this way we can provide the new class implementation. Unfortunately, Java does not support unloading of code. To support dynamic class replacement we need to be able to unload the code first and then to load the new version.

The code unloading functionality is supported by creating a wrapper arund the components. The wrapper component looks as follows:

```
Class ChatServer {
 private Object
  realObject; --> ChastServer_ver1-Object
 send(...) {
    res=realObject.invoke("send", params);
    return res;
  }
}
```

The actual class is renamed to `ClassName_ver1` and the reference to the class object is kept in realObject. When we update the code, the new class is then added by naming it to `ClassName_ver2` and the reference is redirected to the new class instance.

C. Updates

The object replacement functionality is implemented by changing object references (similar to the Gilgul approach [9]). The old class can then be garbage collected.

Additionally we will call the state transfer function when changing the reference.

D. Reflection

The application layer communicates with the ConfMgr layer using reflection on every method call. Again, we implement reflection using the wrapper component. Before every `invoke` method, we check the update level (meta-level). If the meta-level includes an update the method is called.

E. Handling method calls

The component's provide methods correspond to the public methods of the java class. They can therefore be instrumented automatically and wrapped in the call methods. The call to non-public methods happens inside the classes and is not penaltized by the wrapper. However, the count of active objects (object for which we are in the process of calling their methods) should be counted to avoid potential updates of the executing code.

VII. Related Work

Our research has a broad range and covers several areas by combining them into one system. Some of the related research was indicated in the references in the previous chapter. In this chapter, we present only the projects that are very closely related to what we do.

Duggan [11] presents a method for hot swapping running modules. Similarly to our research, he uses programmer-defined version adapters and reflection to add type sharing constraint to the system. He proves that the use of version adapters is computationally correct.

Prose [18] is a system allowing dynamic weaving of aspects into a running code. The solution presented by prose is similar to ours in the sense that it uses a Java Virtual Machine Aspect Interface (JVMAI) to instrument the aspect insertion and removal. We too use a meta-VM to facilitate the component replacement. However, Prose facilitates manipulation only with aspects and does not support modifications of the application code thus being more restrictive than our solution.

SOFA/DUCP project [17] uses components that wrap objects and support dynamic object replacement However, they do not separate the update logic from the application logic. Additionally, SOFA/DUCP project does not use the version adapters to facilitate seamless updates.

VIII. Status & Future work

This paper presents ideas underlying a Ph.D. project on Dynamic Software Updates that runs from 1st of June 2003 to 31st of May 2006. We are in the process of building the basic elements of the framework for future prototyping. Currently, we are programming the meta-VM and incorporating reflection to the ConfMgr into the component model. Next, we will expand the ConfMgr with the state transfer functions and propagation of updates using update requests. By the end of 2003, we expect to have a fully running prototype capable of performing object replacement as well as hot-swapping of object with the state-transfer. Later, we plan to evaluate the performance impact of supporting dynamic updates using the wrappers. Then we will move the update-support (meta-VM) layer to the virtual machine to minimaze the performance impacts and evaluate it against the ”standard” VM.

IX. Conclusion

This paper provides a brief description of an update process to illustrate how a framework (called DUCS) supporting dynamic updates of componet-based applications was built. DUCS unifies research within the area of dynamic updates:

state transfer function [13], dynamic object replacement [9], use of reflection [10], integrated in a layered framework. DUCS supports building evolutionary distributed systems with unknown number of components by utilizing interface adapters.

The contributions of this paper are: 1) a practical example illustrating issues and solutions to them when building evolutionary systems, 2) the clarification of what are updates (changes to the system, where relation between the old and the new version can be defined), 3) the specification of the framework facilitating the process, and 4) implementation detail of the Java-based prototype.

In the prototype, we are extending real world product, a Java virtual machine, with the support for component updates and reflection. The update logic is separated from the application logic so the separation of the application code gives the freedom for optimizing the update functions while keeping the application code (almost) intact. This fulfils all the goals outlined in the introduction. In future, we expect to implement the update logic in a real virtual machine.

References

[1] Clr specification, microsoft corporation. *http://msdn.microsoft.com/net/ecma*, 2003.

[2] Java j2se specification, sun microsystems inc. *http://java.sun.com/docs*, 2003.

[3] F. Bachman. Technical concepts of component-based software engineering, May 2000.

[4] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Analyzing software architectures for modifiability, 2000.

[5] R. Bialek. The architecture of a dynamically updatable component-based system. *COMPSAC*, 2002.

[6] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating, 2003.

[7] C. Boyapati, B. Liskov, and L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report TR-858, MIT Laboratory for Computer Science, July 2002.

[8] S. Chiba. Load-time structural reflection in Java. *http://citeseer.nj.nec.com/chiba00loadtime.html*, 2000.

[9] P. Costanza. Transmigration of object identity: The programming language gilgul. *http://citeseer.nj.nec.com/483031.html*.

[10] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solorzano. Dynamic coordination architectur through the use of reflection. *SAC 2001, Las Vegas, NV*, ACM 1-58113-287-5/01/02:134–140, 2001.

[11] D. Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[12] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[13] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–23, 2001.

[14] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madisson, April 1983.

[15] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. *Sun Microsystems Inc, 901 San Antonio Road, CUPO2-302, Palo Alto, CA 94303*.

[16] R. T. Mittermeir. Software evolution, let's sharpen the terminology before sharpening (out-of-scope) tools. *ACM 2002*, 2002.

[17] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. *Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998*, 1998.

[18] A. Popovici, T. Gross, and G. Alonso. Dynamic homogenous aop with prose, 2001.

[19] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2002.