EDNA: A Safe, Evolvable, Multi-version and On-demand Deployment System for GNU/EDMA Applications

David Martínez Oliveira[1] and Fernando Martín Rodríguez[1]

University of Vigo. GPI-RV. ETSIT Ciudad Universitaria Vigo Spain,
dmartin@uvigo.es, fmartin@tsc.uvigo.es,
WWW home page: http://wgpi.tsc.uvigo.es/~dmartin/
http://wgpi.tsc.uvigo.es/~fmartin/

Abstract. This paper presents the EDNA deployment system for GNU/EDMA applications. This system provides a safe multi-version deployment system with on-demand installation of applications in network environments. Different strategies are supported including zero-install and multiple component repository, allowing immediate roll-back to previous versions. The proposed system supports run-time update of applications and provides basic self-configuring, self-tuning and self-healing features. Additionally, the special features of the GNU/EDMA system allows the propagation of changes easily between design, implementation and maintenance stages, working at the component interface level. The proposed procedures tightly integrates the testing stage in the process of software evolution and later deployment of it, making deliverable software safer. They also integrate in common software development practices, including common version control tools.

1 Introduction

Software installation is a very important milestone within the software development cycle, that marks the transition from the development stage to the production stage. From this point on, the maintenance stage begins and, as software evolves several installation process will be carried out with each new version of the system.

Therefore, the process of installation and versioning of software must be a confident and well-established procedure within the software development procedure. Additionally, installation process will depend on the final system, arising different problems on different kinds of systems. For instance, it is not the same to install and update an embedded system than a complex computational intensive distributed system. Both systems will evolve in time but the way this evolution will be carried out and the problems derived of this process will be very different in each case.

The first installation process is the simpler one. However, it is not trivial, being very recommendable to provide developers and system administrators with

tools to ensure this first step success. Next installations or updates (as they are commonly named) are even more complex since several compatibility issues must be faced in these processes.

In this paper, the installation and versioning system of the GNU/EDMA[7] environment will be described, and the key ideas behind it will be introduced. This ideas can be applied in a general way to any other platform.

The GNU/EDMA system is a middleware system specially designed to provide object oriented programming and component based features to generic applications, despite of the underlying programming language. The system is physically a library plus a set of language bindings that provides a development environment and a set of functions for Object Oriented Programming (OOP)[2] and Component Based Systems (CBS)[1] development.

So, the philosophy behind GNU/EDMA is to uncouple these facilities (OOP and CBS) from the programming languages. Programming languages simply become the tool used by programmers to provide method bodies to the final application.

This approach allows to set a clear separation between the design and implementation stages of software development and so, introduce an extra degree of freedom for evolving applications.

This is the first contribution of the GNU/EDMA system, which will allow to address design evolution in a simple way, as will be shown in this paper.

Moreover, the GNU/EDMA system also provides a basic Component Based System (CBS) set of features which also uncouples component management from the implementation architecture. This uncoupling allows management of the logical architecture of the applications in an independent and abstract way.

The approach presented in this paper tries to face some of the main problems involved on software installation and further update (evolution) in a simple and general way, showing how uncoupling of features traditionally associated to programming languages makes our concerns (installation and evolution) a more simple tasks.

This paper is organized as follows. Section 2 introduces the GNU/EDMA features related to software versioning and installing. Section 3 provides the detailed desing of the EDNA system built on top of facilities introducced in previous section. Finally a comparison to other systems and some final conclusions are provided.

2 GNU/EDMA Built-in Features

Before describing our approach to software installation and evolution, the main features of GNU/EDMA will be introduced in this section. These features are the ones allowing to build our solution and so, they will state the set of features for translating our results to any other platform.

2.1 Overview

The name GNU/EDMA stands from the Spanish acronym *Entorno de Desarrollo Modular y Abierto*, which can be translated to *Modular and Open Development Environment*.

The author of this paper is the main developer of the system since its birth in 1996. In 1997 the system was included in the GNU project[8]. These last years, GNU/EDMA was extended including most of the research lines in programming technologies, unveiling GNU/EDMA as a flexible platform for research in these fields. Technologies like delegation, object inheritance, aspect oriented programming (static and dynamic), software evolution, coordination approaches, etc. have been added to the system since its born.

GNU/EDMA was, from its inception, an open system aimed to embrace most of available technologies under a common interface, allowing reuse across different systems. For this reason, the system provides a set of extension facilities to deal with interfaces and implementations, and so, it allows simple integration of available technologies within an unique system.

GNU/EDMA tries to provide an uncoupled Object Oriented Programming (OOP) and Component Based System (CBS) environment for software development. GNU/EDMA provides these facilities as a set of services independent of the used programming language, allowing their use even on non-OO programming language or non-CB systems.

The main idea behind this approach is treating component systems and object oriented programming as orthogonal concerns respect software development.

Object Oriented modelling is a general modelling approach. In general, independent of software development. Several different systems can be modelled using this technique, and the key point here is design.

Uncoupling object orientation from software development is translated on uncoupling software design of software implementation. This way, programming languages become simple ways to complete a given design. Design and implementation stages become then clearly separated and related one each other.

This is the current approach seek these last years with the introduction of modern software modelling tools like the ones based on UML diagramming. The problem in these tools is that software design got modified when implementation stage is started. For instance, if the implementation language used for a given system does not support multiple inheritance and the design was created on that assumption, design becomes changed to match the final implementation.

Uncoupling an OO constructor like inheritance from the programming language will keep the design unmodified (in this very simple example) and then keep software development simpler.

This approach is being introduced in new development platforms like .Net[4] and its Common Language Runtime. However, .Net still keeps a tight coupling with the underlying programming language which introduces the artifacts described above, i.e. a languages not supporting multiple inheritance will break a design using that.

The GNU/EDMA system provides an environment where this will not happen, since features like inheritance, delegation or aspect programming are provided by the system not by the used programming language.

After this brief introduction to the philosophy behind the GNU/EDMA system, the set of features of the system related to component versioning and installing will be discussed, since this is the main subject of this work.

2.2 GNU/EDMA Components

The first important concept to understand our approach is the way GNU/EDMA manages components. Within the GNU/EDMA system, there is no difference between classes and components, both are composed of an *Interface Definition File* (IDF hereinafter), and of an implementation file which contains method bodies for the associated class.

This is a common approach on OO-based systems but the main difference with GNU/EDMA is that its IDFs are processed at run-time, instead of using an external interface compiler. Each time a new component is introduced in the system, it takes the associated IDF and compiles it to create internal associated structures. Then the implementation file is also loaded and linked to these internal structures.

Figure 1 shows graphically the representation of a GNU/EDMA component.

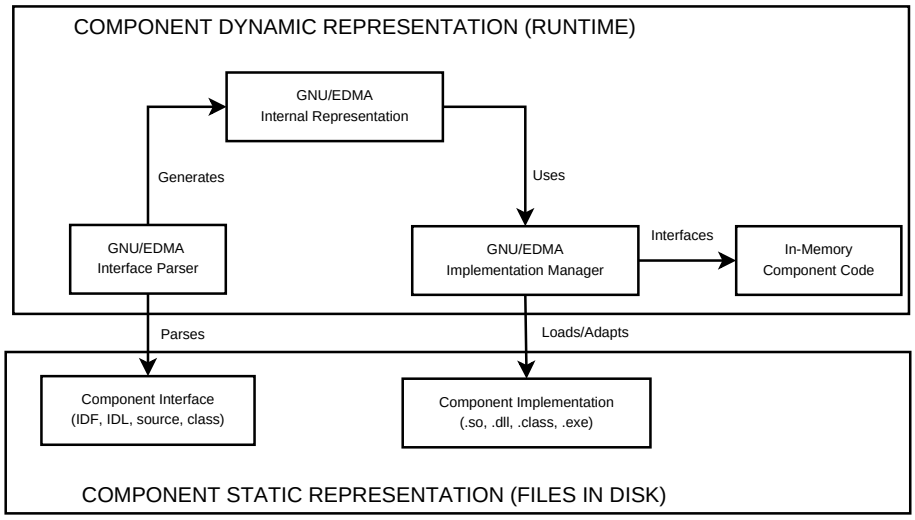

Fig. 1. GNU/EDMA component parts

Figure 1 shows the parts of a generic GNU/EDMA component. The static part of the component is composed of two files, one containing implementation and other containing interface. This files can be the same in practice, but in

the general case two files will be available. These two files forms a GNU/EDMA component.

The static component representation is translated to two main internal blocks. Interface file is parsed by a GNU/EDMA interface parser in order to generate an internal representation of the component's interface. This representation is independent of the interface file format.

Implementation manager (also known as SIU proxies) has two main tasks. First one is to load code into memory and link to it. This process will involve starting an interpreter or a virtual machine able to run the provided implementation code. The second task of implementation manager is to wrap access to implementation from/to the GNU/EDMA system. This task will depend on the component implementation, so different implementation managers must be provided for different implementation files.

Interface parsers and implementation managers are themselves GNU/EDMA components deployed using the default interface and implementation formats that GNU/EDMA understands.

This architecture has two main advantages.

In first place, as all involved operations are dynamics, it is trivial to define components at run-time. This feature will make easier installation of components in a running application and the implementation of dynamic constructors like metaclasses. GNU/EDMA provides a full interface to perform the task described above manually and to implement additional interface parsers and implementation managers.

In second place, this approach allows the system to make more easy the integration of different development platforms. For instance, the *Monna* system (currently in development) allows to integrate .Net assemblies within the GNU/EDMA environment. This system provides a special interface parser able to use the .Net run-time library (actually the Mono implementation) to extract interface information from .Net assemblies and to integrate them within the GNU/EDMA system.

The *Monna* system also provides an implementation manager which allows to instantiate and execute methods included in the assembly within the GNU/EDMA system, at the same time that provides to the .Net run-time a set of methods to access GNU/EDMA.

So, in this case, a component is completely defined with an unique file (a .Net assembly), but developers can provide the component interface as a separated file and only use the assembly as a code provider. This separation is important from the point of view of software evolution and will be discussed later in this paper.

2.3 Class Repositories

The GNU/EDMA system provides a flexible component repository system to manage its components. Each repository contains a list of components and some extra information like the kind of interface parser and implementation manager

to use, its implementation and interface files, component version and some other information which are of no interest for our current discussion.

By default, the system maintains a global repository, accessible to every application in the system. Any change to this repository will be reflected on every GNU/EDMA application in the system, and initially it is intended to hold common components required by most of the applications. Global repository is kept in shared memory and any change on it is immediately accessible to all applications.

Additionally, the system provides tools to define local repositories. Local repositories are identical to the global repository in structure and access way, but these repositories are kept in the private address-space of the process using them.

Developers can set as many repositories as they want, which allow them to follow different strategies to manage their application components. This approach allows, for instance, to have a private repository for each application of a software suite and a common one shared by all the tools in that suite, in addition to the global one holding core facilities like interfacing to the operating system.

The use of private component repositories makes trivial the deployment of applications using a zero-install approach. However, the solution provided by GNU/EDMA has advantages when compared to common zero-install solutions.

In normal zero-install solutions the application and the whole set of components are copied to a fixed place and executed from there. Each application installation produces a new and independent version of the application, and implies distributing the complete set of components the application requires.

In the general case, this is not a big issue, but, as will be described in section 3.5, where an on-demand network based deployment system is presented, a whole update of the system will introduce a performance penalty.

The other main difference of the multi-repository approach presented is that components can be versioned individually. In a normal zero-install approach the whole set of components should be tagged with the application release version, even when several of them did not changed from one version to another. This will produce several versions of the same component which are identical. It will be possible to manage individual component version separately but this does not make too much sense when the application and its components are managed as a whole.

Summing up, the GNU/EDMA multi-repository approach allows to distribute applications using a zero-install strategy but holding the incremental solution for component deployment provided by global repositories approaches. Different versions of a component can be deployed in the same file system place and all of them are available from this same point at the same time that a per-component version policy can be kept.

2.4 Class Version

Each GNU/EDMA component has a version number associated. The version number is composed of two parts; a major version number and a minor version

number. As usual, major version number is used to mark changes on the component interface, and minor version number is increased when internal changes to the component are done but the interface is not changed.

Component version information is kept by system repositories and is managed by the GNU/EDMA component deployment tools. Each time a component is created a deployment descriptor should be built (unless manual installation of the component is preferred). This deployment descriptor allows automatic installation of the component. Figure 2 shows one of these files.

```
ClassName=HELLO_WORLD
NameSpace=examples/hello
Machine=i386
OperatingSystem=LINUX
Implementation=libHELLO_WORLD.so
IDFParser=EDMAIDF
SIUProxy=
MajorVer=2
MinorVer=0
UpdateScript=HELLO_WORLD1_0_2_0.tcc
```

Fig. 2. GNU/EDMA Component Deployment Descriptor

The deployment descriptor contains the version of the component to be installed. In the example the descriptor represents a new version (2.0) of the `HELLO_WORLD` component.

When the component is installed using this descriptor a new version of the component is available, and the old version is kept. This new version becomes the default version, but applications are allowed to force the use of specific versions of a given component using the `edma_class_set_actual_version` GNU/EDMA function.

Therefore, the default GNU/EDMA version management system allows to install different versions of a given component and allows both to co-exist in the system.

Note that, if not stated otherwise, any application using the `HELLO_WORLD` component in our example will begin using the new version automatically. Applications must explicitly state the use of specific versions of components.

Finally, note that the `UpdateScript` field, if provided, will instruct the system to carry out a hotswap update of existing instances of the component. This process will migrate any existing instance to the new version, using the provided *update script* to translate state from one version to another. Next section describes the process.

2.5 Object Hotswap

The different features introduced in previous sections provides tools to evolve applications in easy ways, that is, each time a new update of a given component is available, applications using that component are stopped, the component is installed and the applications restarted. Actually, the presented architecture allows to install any component at run-time if component does not change its interface.

Hotswap allows to carry out this process at run-time, without stopping the applications which is a requirement for certain kind of applications. The GNU/EDMA hotswap system is in development and can be used for simple updates with some minor human intervention.

When the deployment descriptor of the component provides the `UpdateScript` field, the system will start a hotswap update of the component.

In this case, the component update is hold in a list of pending updates and a list of target instances is built. The system will then try to update each component to the new version in a safe execution point using the update script provided in the deployment INES descriptor.

The whole process is summarised in Figure 3

```
if UpdateScript is provided
   Add update to pending_updates_list
   Build list of component instances to update
   for each instance_i requiring update
     If instance_i is in some thread execution stack
        wait until instance_i is released
     else
        create a new instance using the new component
        run UpdateScript to transfer state from old instance to the new one
        swap instance_i reference to fix external instance references
```

Fig. 3. Component Hot swap

This algorithm is executed each time a GNU/EDMA primitive is invoked from the main application, until all instances get updated and all pending updates applied. So, it is carried automatically without special considerations in the application being update.

This approach works on simple scenarios but will fail when the update implies complex dependencies between several components at once. As said above the system is in development and new approaches to solve this problems are being prepared.

Next section shows a simple example on how the hotswap system described so long works.

2.6 From Hello World 1.0 to Hello World 2.0

To illustrate the whole process a complete example is show in this section. In this example, a `HELLO_WORLD` component will be updated to a new version which includes a simple instance state transformation.

The `HELLO_WORLD` component version 1.0 has the developed for a first release of a given application,

Due to some customer request, the component must be updated to provide a per-country greeting. Changing the interface as shown in Figure 4.

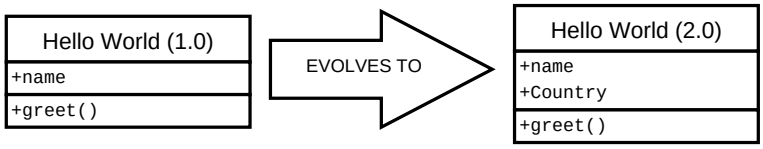

Fig. 4. Hello World Component Evolution

The original `greet` method will produce a "Hello Name" message and the new version of the component must produce a "Hello Name from Country" message. That is, old instance state must be copied to new one.

To carry out this simple update the following INES descriptor must be provided.

```
ClassName=HELLO_WORLD
NameSpace=examples/hello
Machine=i386
OperatingSystem=LINUX
Implementation=libHELLO_WORLD.so
IDFParser=EDMAIDF
MajorVer=2
MinorVer=0
UpdateScript=HELLO_WORLD1_0_2_0.tcc
```

Major version number is increased since the interface of the component was changed and an update script must be provided to keep instance state between versions.

The update script is implemented using the TCC (Tiny C Compiler) and is shown in Figure 5.

The simple function in Figure 5 receives as parameters two component instances. The old one which is being updated and the new one created by the system when update was possible. Then, the script transfers common state from an instance to another and initialises the new field not available in the original component.

After finishing this execution, the system swaps components `newid` and `oldid` so any previous reference to the original instance (`oldid`) get updated to the reference the new one (`newid`).

```
ESint32 HELLO_WORLD_update (OBJID newid, OBJID oldid)
{
  EChar    buffer[80];

  /* Transfer state */
  edma_set_prop_strz (newid, "Name",
                      edma_get_prop_strz (oldid, "Name", buffer));
  /* Initialise new state  */
  edma_set_prop_strz (newid, "Country", "NO COUNTRY");

  return 0;
}
```

Fig. 5. Update script for updating HELLO_WORLD 1.0 to HELLO_WORLD 2.0

Update Script Generation Our current work on hotswaping has unveil the requirement of linking the update process with the design stage, making the update process flow from the beginning of the application life-cycle. That is, once the software update is defined, that update is re-introduced in the system at the analysis stage, reworking the whole cycle towards a new implementation version.

Working this way, will allow design or refactoring tools to take into account the changes involved by the update and so automatically generate most of the actions for components state changes at run-time, making much more probable the success of the run-time update.

2.7 The Exception Management Interface

The GNU/EDMA system provides a general system-level exception management subsystem named EMI (Exception Management Interface). This system allows system administrators to establish how to proceed when a system exceptions is raised.

The `CLASS_NOT_FOUND` exception is the one of interest, for our current discussion.

The EMI subsystem, allows to install a so-called EMI Handler. An EMI handler is a GNU/EDMA component which will be invoked when a given system exception is raised.

This solution was used to setup a class server and allow automatic installation of components from it when a given application is executed. This system is described in section 2.8.

2.8 Previous Experiences

Finally, before introducing the new EDNA architecture for GNU/EDMA application deployment, a previous experience on automatic on-demand network installation will be commented.

This system was introduced in early 1999 on the 0.3r1 version of GNU/EDMA[5]. Several of the concepts introduced in this system will be included in the new EDNA architecture and for this reason, they worth a brief description in this section.

The system was designed to work on LAN environments with GNU/EDMA applications. GNU/EDMA applications run on different machines and each one keeps its own global repository (local component repositories were introduced in later versions of the system).

A general component server is configured in the LAN allowing clients to ask for components using a simple TCP protocol. Each machine running GNU/EDMA applications has a basic GNU/EDMA installation including a EMI handler able to connect to the component server and to ask for a specific component.

So, each time an application is started in the environment, `CLASS_NOT_FOUND` exceptions are raised for each component it tries to instantiate. The associated EMI handler connects to the component server providing some basic information about its running operating system and architecture.

The server responds with a compatible component for each configuration or with an error, if no component is available. Suitable components for a given configurations are:

- A native component for the given operating system and machine architecture
- An interpreted version of the requested component.

That version supported component implementation using Java and Guile (the GNU scripting language). So, if no native version of a given component is available but an interpreted one was found, this last one is transferred to the client machine. This approach allows rapid prototyping using multiplatform languages and later per-component optimization using native versions.

The EMI handler in the client machine gets the implementation and interface files of the requested component and dynamically registers it in its client repository, allowing the main applications to continue execution. The system works similar to virtual memory swap system in operating systems.

This early system allows to improve availability of applications in heterogeneous systems supporting multiplatform implementation of components (Java implementations) and also allows full update of applications wiping out client repositories and simply re-running applications.

2.9 Lesson Learnt

From all this previous work some important lessons were learnt and they will be summarised in this section.

In first place, applications normally do not need complex deployment systems. Most of them can be effectively managed using a zero-install strategy an a simple version control system for the components their use.

This is a consequence of general poor code reuse in applications.

Human factor is critical and code reuse normally only happens when people reusing code knows it very well. If not, people usually tries to write its own code, unless a small suite of tools related is being built.

Think for instance on the set of COM-like system available nowadays. Microsoft developed the first COM system several years ago. From that point on, Macromedia Director included its Xtra extension system based on the same architecture. Mozilla its XPCOM solution, OpenOffice its UNO architecture, etc... There are subtle differences in each system but all of them are based on the same basic idea.

The same can be said for the big set of low level helper libraries available.

The reality is that a lot of effort was put on making code reuse easy, but in practise code reuse is rarely used, except for central core software as for example desktops.

This is a different discussion but what is important for us, is that most of the components of a given application are private to this application or to a small suite of related tools. This makes global component repositories a source of problems (like the well-known *DLL hell* in Microsoft platforms).

So, in general, an application (or small suite of applications coming from the same producer) will be composed by a set of specific components plus a minimum set of shared components which in general deals with the interface to common or inevitable systems like the operating system itself, or core framework foundation classes, desktop interfaces, etc.

In second place, application evolution is a complex task which varies depending on the kind of application being evolved. Two main groups can be defined; application which can be stopped and applications which cannot be stopped.

The main difference between them is the way the change is applied. The way the change is managed and generated is common to both scenarios. In any case, application evolution must be carried out from the conceptual level to the final implementation to reflect the changes in the whole process.

This simple and well-known process is not performed normally. Small patches are applied at the implementation level and a lot of these small patches produces changes not reflected in the higher level life cycle stages (design, analysis). A big effort is required to keep all this information synchronised.

This will soon produce hard to maintain applications. The solution is to produce tools able to keep consistency on all the stages independently of where the changes are made. Such a tool does not exist at this moment and only partial solutions are available.

An unification of the way software is managed at the different stages of its life cycle is required. Such an unification will allow to propagate changes in any stage almost automatically to the others.

Finally, the whole process of application creation and maintenance must be simple and mostly automatic to avoid problems due to error-prone human manipulation.

Taking into account all this facts, the design of the EDNA system to deployment and maintenance of GNU/EDMA applications is described in next section.

3 New EDNA Architecture

In previous sections, the features related to application installation and versioning provided by GNU/EDMA were introduced, as well as the current support provided by the system. However, current support lacks several important features which will be fixed by the EDNA architecture.

The goal of this system is to produce a set of tools and procedures to build GNU/EDMA applications and allow their easy and safe deployment and evolution.

EDNA should provide an easy way to deploy common components in a system which will be heavily used by most of the GNU/EDMA applications, and it also should provide and easy way to deploy applications or small suites sharing a well-defined set of components.

The system should provide tools for real-time and deferred update to cover the needs of normal applications and better-non-stop applications.

Finally, the system should provide an appropriated level of automation to reduce the probability of errors in the processes of evolution and installation of initial and further versions.

3.1 GNU/EDMA Interfaces Extension

The GNU/EDMA system, as pointed early in this text, makes a clear separation between interface and implementation for each component it manages.

GNU/EDMA component interfaces are independent entities which bridges design and implementation stages. Current implementation only captures inheritance and "part of" relationships so direct transformation is not complete. However, the GNU/EDMA interfaces where designed to keep as many information as desired even when the system will only get the one it needs.

So it is trivial to add further information within a component interface and then write tools taking into account this information. This way, the whole design and implementation information for a given application can be kept in an unique file, making changes in any of theses stages available to the other.

So the first point to address by the EDNA system is to introduce additional dependencies within the component interface. The logical targets are: usage, dependency and association relationships. This additional dependencies are directly related to implementation, that is, they are abstract entities at the design level that becomes specific implementation statements . So these relations should be extracted from the source code, or manually managed by developers.

This process is in general not trivial, but using an approach like GNU/EDMA where component interaction always goes through the GNU/EDMA middleware layer it will be a lot more easy than using a classical programming language. Current AOP solution could be helpful in other cases.

The good approach to keep up to date this information is to instrument unit test suites. At the same time that components get validated by a set of unit tests, the dependencies of the component are generated, feed-backing the information to the design stage. This process will also log the exact version of each component involved in the test so a version map of a tested application can be generated during testing process.

In the particular case of GNU/EDMA the system must be instrumented in order to log this information. Instrumentation will be easy since GNU/EDMA application only use a small set of primitives (method invocation, fields accessor, object creation and some dynamic inheritance primitives).

Summing up, the first feature of the EDNA system is to extend the GNU/EDMA interface files to keep additional interface information. This additional information will be generated at the test/verification stage and so, automatically (or mostly automatically) reintroduced in the design stage.

Design tools will work on components interface, managing implementation as opaque data types represented as a set of relationships among a given component and the set of the other components it interacts with.

Any change on a component at the implementation level, will be automatically reflected in the design level after testing, what will determine any new relationship among components due to implementation changes.

Any change at design level will be propagated to implementation level as a new set of component interfaces which will indicate the required changes in design. Design to implementation level will suppose, in general, bigger changes than implementation to design level. This process, design to implementation, will be, in general, supported by additional documentation to instruct developers about the implementation of these design changes.

Note otherwise, that design changes should be less frequent that implementation fixes and in general should imply addition of new features and slight extension of existing ones.

Figure 6 summarises all this process.

In the left side of the figure the different stages of the software life cycle are show. The right side of the figure shows the set of products generated in each stage and stored in the version control system.

The component interfaces are first generated at the design stage, and from that point on, they can be modified at any other stage of the development process, below the design. Implementation, testing or even deployment could produce changes in the components interface not envisioned at the early design stage, and then, interface changes may be induced.

In the same way, the code produced in the implementation stage and the deployment information (and interface relationships) stated at the testing stage are shared from any other stage below the one generated them.

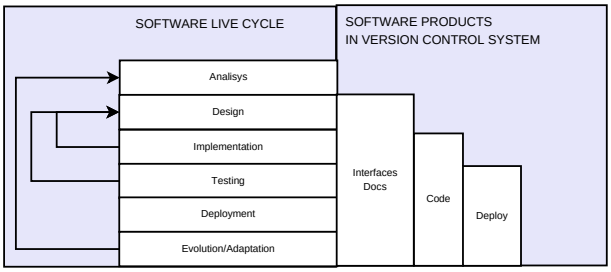

Fig. 6. Software Products vs Software Life Cycle

As explained above in the text, modification of any software product must be supported by additional procedures to communicate the different members of the development team. These procedures would be organization dependant and are beyond the scope of this paper. The key point in this discussion, is that all the software life cycle stages works on the same items, which are clearly bounded.

3.2 Implementation

During implementation stage, several sub-version of the software components, normally named releases are generated and kept in a version control system.

The version control system will also keep a set of tags of these releases which eventually will become a deliverable version. Normally the process of tagging or freezing a version consists on taking an snapshot of the current development version which has been appropriated checked. However, in order to keep evolvable versions of an application additional steps should be carried out.

Therefore, the tagging process should be extended with the following steps:

- Generation of a release version
  The system should check immediate previous tagged version and verify how interfaces of the components (which are also kept in the version control system) had changed.
  If no change in the interface was detected, only the minor version number of the component will be updated. Actually, minor version number should only be increased if there is any change in component implementation. If interface changes are detected, major version number is updated and further steps required.
- Generation of update scripts
  If run-time update is required, at this point a set of update scripts should be generated. The information hold by the component interfaces will be enough to automatically generate any syntactical related transformation, but human intervention is required to face semantic related and arbitrary changes (for example, name changes).

Most of the information required to generate update scripts could be generated automatically if a refactoring tool is used to carry out these interface changes. The discussion of such a tool features is out of the scope of these paper. To know more about current research on these field using the GNU/EDMA system readers can check [6]

- Final Tagging
  Once the real version of the component got determined, the tagging process (using the modified interface files) can proceed. The process will involve the "check in" of all the code, as usual, as well as all the side information generated about version to version transition, including the information generated by the unit test process.
  Note that if no run-time update is required, the second step of the process can be removed.

So, a version release of a EDNA evolvable application will include the following elements:

- A new set of components with updated version numbers
- A complete list of component version list used by the application
- An optional set of update scripts for run-time update
- An optional main application if not implemented as a component.

Following this process the version control system will store a new version tag of a given software system with all the information required for its deployment.

3.3 The NIL Application Approach

Before continuing, the concept of NIL application will be introduced. The concept of implementing the main application as a class is central to modern programming languages like Java or C# which, actually make this mandatory.

However, given this fact, and in order to make simpler deployment of new versions of a given application, a small modification of this concept will be introduced and named *NIL Application*.

Following this approach an unique executable program will exists in the system. This program just sets up appropriated properties to allow the application locate its components. Then instantiates the main application class and starts it.

Note that this is the normal way a Java application is started, using the static method `Main` in the class being executed. The NIL application approach introduce a subtle difference. For the Java case, simply implies a more sophisticated loader, similar to the one provided by the Java Web Start Technology.

For the case of the GNU/EDMA system, such a program will be as simple as the shown in Figure 7

The main difference between the code shown in Figure 7 and, for instance, a normal Java application, is that when the application is executed (method `run`), the whole system, including its EMI handlers (see section 2.7) are up and ready

```
#include <edma.h>

int
main (int argc, char *argv[])
{
  OBJID   app;

  if (argc != 2)
    {
      edma_printf_err ("Invalid Number of Parameters. Aborting\n");
      return -1;
    }

  EDMAInit();

  if ((app = edma_new_obj (argv[1])) < 0)
    edma_printf_err ("Cannot execute application '%s'\n", argv[1]);
  else
    edma_met3 (app, "run", argc, argv);

  EDMAEnd();
}
```

Fig. 7. Basic GNU/EDMA Nil Application

to be used, including the required logic to look for missing classes and manage component versioning.

Note, that the concept is quiet simple, it is just a minor extension of the currently available solutions for application loading. However, building applications this way, within the GNU/EDMA environment, will show several advantages on application deployment and versioning.

3.4 Deployment

At this point in our discussion, deployment of applications using the procedure described up to now is basically trivial for applications that do not need run-time updates.

Now an application is a bunch of components which will be installed to different repositories in the system. The main application becomes another component which will also be installed the same way.

This approach allows several different configurations. In this paper only one is presented as an example on how the system will work.

In our example four main component repositories will be available:

- A global repository holding system wide shared components
- A suite repository targeted to hold components used for a small suite of tools, normally from the same provider

- An application specific repository holding components specific for a given application
- An application component repository holding the main component application

So, an application being deployed will be composed of the following elements:

- The main application component
- A component version file holding the tested components the application is know to work on.
- A package containing components to be installed in the application private repository
- A package containing components to be installed in an optional application suite repository
- A package containing components to be installed to the global repository. This package will in general be distributed apart and referenced by the application throughout the component version file
- A resource package containing data required by the application which will be installed as a directory in the local repository

Installation process simply implies checking availability of required shared components (the ones in the global repository) and then put each component in its appropriated repository. Optionally, the creation of the application private repository will be required for the first time installation.

Note that, since GNU/EDMA allows the installation of different versions of the same component, the following holds:

- Previous version of the application is not modified neither removed. On any problem with new version previous version is still available.
- By the same reason, any modification to the application suite repository will not break any member of the suite, since the old versions of every component are still available.

Finally, note that the list of every component used by an application along with its exact version is a implicit part of the package release. This kind of information is mandatory for certain software development standards and software life cycle recommendations. Normally, this information is provides in a document named "Software Version Description" deliverable along with the software package. The approach presented here will include most of the content of this deliverable as part of the main software package.

3.5 EDNA Class Server. Transparent Installation

The architecture presented so far allows to setup a transparent on-demand installation system using the network. This kind of systems have the additional advantage of automatically install only the components of applications that are actually used.

Suppose that an image viewer application is built. That application will use different components to read and write different image format.

In a normal case, all the components will be installed or user will select manually them during the installation process. Using a network on-demand installation strategy, components will be installed when needed and never used components will never be installed. So, application got self-configured according to the use user does of it, saving machine resources.

To achieve this, the system described in section 2.8 will be rebuild taken into account the new structure of applications described in section 3.4. Figure 8 sketches the general operation of the system.

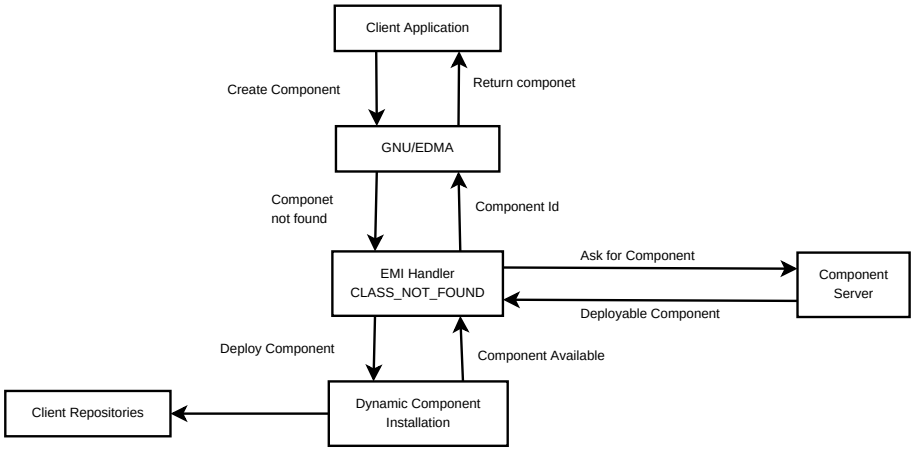

Fig. 8. Normal operation of EDNA Transparent Install

The details of this system is beyond the scope of this paper, but some comments are of interest for our current discussion.

In first place, just a small set of classes is required on each system to let it use the class server. An EMI handler able to communicate to the class server and some basic communication classes. Actually, this communication classes could be included within the EMI handler which is the unique class required to make the system work. This minimal set of classes will be included in the base GNU/EDMA package which will be once installed on each client machine.

In second place a redundant distributed system can be easily build. Since the class server is also an executable component, any machine in the system can launch one of these servers and begin serving files. Initially, the system will not fully replicate the whole database contained by the class server, but this functionality is being considered for a small set of nodes of the network.

The only information replicated along the class servers will be the list of available clients and, therefore, potential backup servers. So, if a client cannot connect to the class server it will try to connect to other servers in its local

backup list. If no server is available the client can launch a new NIL application using the class server component and share, almost its current repositories. The discovery procedure for new class server is still to be determined.

This behavior will introduce basic self-healing capabilities to the system, increasing its availability in the case of a server fault.

This part of the system is in its early development stages and no data is available at the moment of writing this text. The critical point to measure is latency on component installation.

In first installation a non negligible time will be used on transfer and install the basic set of components to begin running the application. However this time will be less than the time of download the whole application form the net. This is a strategy used by some applications nowadays.

The reason is that only the components being used are downloaded so it is expected to have a slight delay each time an user does a new action on the application. Empirical probes will be carried out soon.

3.6 Application Evolution. Upgrading

The system described so far, will install a given application on-demand but, once the application got installed it will not evolve directly without human interaction.

To avoid that, applications should check the last version of themselves at start-up initiating an update process if a new version is available (or almost asking the user if the process must be started). Note that automatic update is safe since the old version will always be available and so, application can be restarted in its old version if the new version is break due to any reason.

The right place to put this is in application initialisation. A helper class will be developed to perform this check in a transparent way. This class will override the `run` method of the application to perform the check before the application starts. If a new version exists, the helper class will download the new version and create a new instance of the application giving control to it.

Proceeding this way developers are allowed to determine the update strategy they want.

- Developers can include the helper class directly in its main application component and so choose when to perform the update.
- An special application loader can be provided to let the user choose when to update an application
- No update strategy is implemented and is the system administrator, who forces an update of system repositories.

4 Related Works

Application installation, versioning and deployment has not been a very popular research line in the academic neither in the industry communities. Obviously it is not the bigger problem within the software development cycle, however it is

an intrinsic part of this cycle, and, as has been shown, it affects previous and later stages of the life cycle.

Traditionally software versioning has been carried out using internal development team policies, normally related to the version control system used during development. Installing and deployment for small application has been reduced a simple "copy this file to that directory" or to ad-hoc solutions for large scale systems.

In the field of small application deployment, the GNU/Debian package management system has been shown this last years as an effective and simple way to install software. The GNU/Debian solution can manage dependencies at the level of packages and provides a comprehensive and robust network installation system. Even solutions for on-demand installation of packages exists like the auto-apt package.

When compared to our proposed system two main differences can be stated. In first place our system is specially target to smaller components, meanwhile GNU/Debian deal with bigger packages which could be decomposed in smaller one if necesary.

The second difference is that our system is being designed to integrate installation and deployment within the software life cycle.

The Microsoft .Net platform is maybe the one facing more directly versioning and installing at the component level. This was a requirement in Windows-based technologies for a long time after unveiling the well-known *DLL Hell* of windows shared libraries and COM services.

.Net framework provides a similar version system to the one presented here and makes a distinction between global and local components, providing zero-install deployment of applications. However it only can distinguish between this two component "repositories". The system presented in this paper supports a more flexible management of repositories which allows flexibler deployment strategies.

The .Net framework also supports the installation of different versions of the same component and allows forcing the use of a specific version of a given component, being similar to our system in this way. The *NIL Application* concept introduced in this paper is not supported directly, being, at first glance easy to implement in this platform.

Finally, official documentation on .Net framework does not include information about building *class loaders* like the ones provided by Java. However last version of the framework seems to provide some of these functionality but its use is not straightforward.

The other big platform to compare to is Java. Java zero-install approach is trivial since the introduction of the .jar files. However, Java does not provide direct support for class versioning and so implementing a version of our solution will require additional coding of helper classes. On the other hand, Java *class loaders* provide a flexible way to set up network on-demand installation systems like the one described in this paper, being an example of this the Java Web Start system.

From a general point of view, component (and application) versioning is a manual task for most commonly used development environment, and its linking with the software life cycle is arbitrary and discretionary according to the development team internal policies.

The approach introduced in this paper is a approximation to integration of the deployment stage (including versioning, installing and evolution) within the real life cycle of software development as a live part of the whole process.

5 Conclusions

In this paper a basic procedure for software development was introduced. This procedure takes into account additional information extracted from the development process for making easier software evolution and deployment of new versions of a given application.

The process also makes the spftware testing stage an integral part of the sequence, forcing to carry out this stage in order to obtain a safer version of the software. Testing will produce important deployment information (list of component versions used) and so, produce an snapshot of a verified working environment.

All this process has been specifically targeted to GNU/EDMA applications, however, the main ideas in it can be easily extended to other component based systems.

Once a deployable version of the application is available, several deployment strategies can be followed. A GNU/EDMA specific strategy, currently in development, has been described.

The concept of NIL application (as a subtle version of the classical application as a component concept) is introduced to generalise application management (versioning and installing) in a given system.

This strategy allows to install different versions of applications and components side by side, never breaking previous version since they are kept until a system administrator completely removes them. So reverting to previous versions is always possible.

The strategy can be extended, with little effort, to work in an small and secure network environment, making application installation and upgrading a trivial task which could even be performed by users.

The proposed solutions will provide basic self-configuration of application and basic self-healing of the whole installation process using a very simple strategy. The system can also expose self-tuning features if different versions of the components, according to different architectures are also managed.

For the moment no empirical result is available since the system is in development stage, but previous similar works carried out some years along do not make us envision any technical problem in its implementation.

References

1. Clemens Szyperski: Component Software: Beyond Object-Oriented Programming. Addison-Wesley
2. Booch, Grady: Object-oriented analysis and design with applications (2nd ed). Addison-Wesley Publishing Company (1994)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J: Aspect-Oriented Programming. In Proc. of ECOOP, Springer-Verlag (1997).
4. Microsoft Corp: The .Net Framework. `http://msdn2.microsoft.com/en-us/netframework/default.aspx`
5. Martínez Oliveira, D., Fernández Hermida, X.: Proceso de Instalacin de Aplicaciones en Red con EDMA. Simposio Espaol de Informtica Distribuda SEID99, Santiago de Compostela (1999)
6. Martínez Oliveira, D., Fernández Hermida, X.: Run-Time Component Extension and Update. Technical Report. University of Vigo. (2002)
7. Martínez D.: GNU/EDMA Web Page. `http://www.gnu.org/software/edma`
8. Free Software Foundation: GNU Project. `http://www.gnu.org`