Component Configuration Management

Magnus Larsson
Development and Research
ABB Automation Products AB
721 59 Västerås, Sweden
+46 21 342666
Magnus.Larsson@mdh.se
www.idt.mdh.se/personal/mlo

Ivica Crnkovic
Department of Computer Engineering
Mälardalen University
Box 883, 721 23 Västerås, Sweden
+46 21 103183
Ivica.Crnkovic@mdh.se
www.idt.mdh.se/personal/icc

Abstract: Component-based programming is nowadays widely recognized approach in software development. Still there are many open problems related to both technical and non-technical aspects of the components. In this paper, we point out the problem of component identification. Since the components are usually binary units deployed in the system at run-time, we do not have the same insight to their characteristics as for software units that we manage at development time. This problem could be solved if the components had built in this information together with the binary code, which can be achieved by defining a standardized identification interface. As such interfaces do not exist in standard component models today, this concept is possible to use only with components built in-house. For external components, extensive tests can, to some extent, compensate the lack of information. To perform an efficient and yet a successful testing we must limit the number of test cases. Which parts of our system can be affected by introduction of a component, or by its updating? We can answer this if we can keep track of changes introduced in the system and their impact on the system. These problems are similar to the problems at development-time solved by Software Configuration Management (SCM) disciplines. In this paper we point out these problems and make proposals for their utilization at run-time using SCM principles.

1 Introduction

When developing a component independently of system development, we meet a number of problems due to the fact that we miss information we usually have during the component development process. One type of problems is related to the components themselves – the component interface, pre- and post-conditions and the component non-functional characteristics such as reliability, requirement on resources, timing requirements, etc. Another type of problems are associated to the relation between the component and the rest of the system. In this paper we address this second type of the problems.

When integrated in a system, the new component has impact on a part of the system. The new component may refer to certain components, and it can also be used by other components. In addition to these explicitly defined dependencies we also have indirect ones, derived from the components that are used by the new component. Finally, we have implicit dependencies, which are related to the system environment (for example timing or other resource constraints). In general, we can expect that some parts of the system are not affected by a change when introducing a new component or a new component version. This situation is shown in Figure 1.

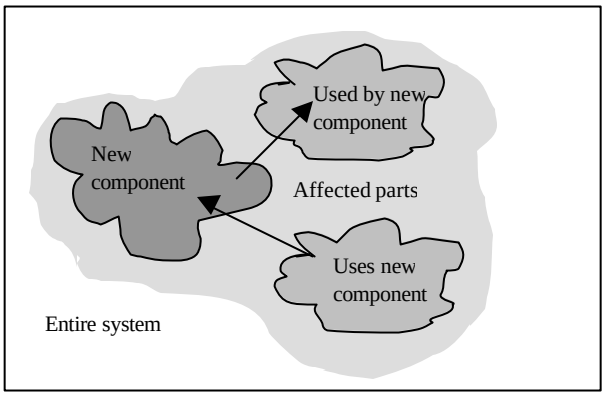

Figure 1. Dependencies between the components

The dependencies are not directly visible in component models available today, such as COM [1][2] or EJB [3].

To limit the uncertainty of the system behavior, we want to identify those parts of the system which might be affected by the introduction of a component.

If we could identify the component versions explicitly, we could specify the entire system as a set of component versions. Two systems, or two versions of a system can be compared and differences on the component level can be identified.

If we could automatically identify the dependencies between the components and their version we could avoid the well-known problems with different versions of shared libraries. The problem is illustrated in Figure 2: We have two programs *Foo.exe* and *Bar.exe*, which share *Common.dll* library, version v_1. At one point of time, we upgrade the new version of Bar.exe, which also includes a new version of *Common.dll*, v_2. The replacement could be successful if version v_2 of *Common.dll* is compatible with version v_1, but if this is not the case the *Foo.exe* can fail. Even if the new version is interface-compatible, *Common.dll* may contain undetected errors, which appears in a combination with *Foo.exe*. *Foo.exe* may then access

some erroneous code and crash even if the library was interface-compatible.

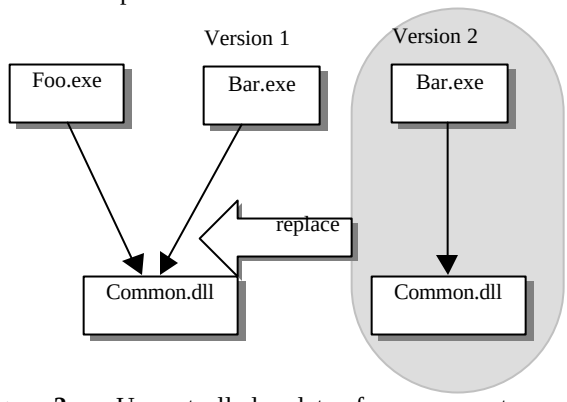

Figure 2. Uncontrolled update of a component

One way to handle multiple versions of libraries is to insert version information into the actual library name as Microsoft does in MFC [1]. For example, names such as *MFC40.dll* and *MFC42.dll* can be used for version 4.0 and 4.2. This prevents name collisions problems but can introduce a vast number of versions which we de not have any control over.

In order to identify the parts of the system that can be affected by the change we want to:

- Identify components including their versions.
- Identify direct and indirect dependencies.
- Get enough information to localize the implicit dependencies.

The identification and dependency management is a typical subject of SCM. The SCM disciplines and their possible implementations for managing components are discussed in section 2. In section 3 we discuss the problem with dependency information which is missing in component models available today. In absence of it, we discuss a possibility of finding dependencies directly form the code. A Dependency Browser, an application that displays dependencies between binary assets, is depicted in section 4. Finally, section 5 outlines further investigations.

2 Component Management and SCM

As a component is a unit of composition, its management is natural related to Software Configuration Management, which main objective is to manage composite entities. However, most of the SCM functions are used at the development-time, and are not enough utilized at the run-time[5]. The major disciplines of SCM are *Version Management*, *Configuration Management* and *Change Management* [7] [8], and we discuss their use for managing components.

Version management takes care of the identification of entities and recognizes different versions of them. We can apply this principle to the components at run-time: Every component in the system should be identified by a name, version number and other version attributes such as creation date, history information, etc. We need the component version identification for two reasons: First, when we update a component with a new version, we want to have possibility to identify that change. Second, in some cases, we want to keep several component versions integrated in the system. Managing different versions of components is important for middle-size or large systems. In the beginning, a component might not have been designed to cover all the requirements from the system that evolves. In general, it is better to release a component containing currently required features and later upgrade it, instead of releasing a full-fledged component to late. Later, when new features are added to the component, it may happen that the new component version is not compatible with the previous one, or that is not fully tested. In that case we want to keep both versions - the new one exploiting new features, and the old one, used by those parts of the system we had not yet changed or tested. When the system must support this type of environment, and when several versions of components are used at the same time, the development time and maintenance increases. However, the experience is that this type of evolution is appropriate for large systems [4].

Configuration and build management methods are used to select and identify specific versions of entities (i.e. to generate a *baseline* or a *configuration*) and integrate them into a new version of the composite entity. It also includes build procedures. The building procedures use information about the dependencies between the entities. These principles can be applied on the run-time system: A system configuration is defined as a set of component versions. By adding a new component or a new version of a component, a new configuration of the system is identified. Similar to *Make* dependencies, which describes the dependencies at build-time, a component should include the specification of the components used (the references to the components used actually exist in the component, but they are hidden in the binary code).

Change management provides information about the changes introduced in the system on an abstract level, what logical changes have been introduced in the process, rather then physical. Change management becomes important when new entity version is created. In a similar way, every component version can include information about what type of change is put in it, regarding to the previous version. This information cannot be automatically generated (which is possible for other type of information, such as version identification and version attributes), but the component developers must explicitly define it. A new component version might be added to introduce new functions in a system, or only to change its behavior, (better performance, better stability), without changing the interface. When replacing a component or a component version we must consider which type of change we permit and what system characteristics we want to preserve, in order to guarantee the system behavior.

To describe this possible impact on the system, we have defined three levels of compatibility:

Input and Output compatibility. A component requires input in a specific format (or maybe have no input at all)

and produces results in a defined format. The internal characteristics of the component are of no interest.

Interface compatibility. The interface remains the same, but the implementation can be different

Behavior compatibility. Internal characteristics of the components, such as performance, resource requirements, must be preserved.

The compatibility criteria can be used to decide if a component can be replaced or not. This decision can be especially important in case of a replacement "on the fly" in a run-time environment.

3 Managing Component Dependencies

Binary components are delivered as shared libraries and executables, which usually have no additional information about dependencies between components. To be able to predict what will happen in a system when a component is installed we need to have information about what part of the system will be affected by the component.

As components can be loosely coupled there is no information connecting different versions of components with each other. In COM for example, a component finds components it refers to through the Windows registry. In the Window registry all installed components store their activation data, such as Interface id, class id, library locations and where to find their stubs and proxies. Connections between components are set up first at run-time. A client uses a unique key to find the server component in the registry and then the COM run-time will load the corresponding component or stub into the client memory [2].

To be able to get full dependency graphs over the system with coherent information about all the components, and the type of change introduced in a component, we need meta-data. With meta-data we mean additional information that is not crucial for the component to run but is valuable for the entire system. Meta-data can be provided as a new interface on the component [5] or stored in a repository where have been placed during the component registration process. Facts about version, name, creation date, compatibility change, interfaces provided and components used, as mentioned in the previous section, are examples of meta-data that will help building a system that has consistent configuration management.

The World Wide Web Consortium has defined a standard to describe components and their dependencies. This language is XML-based and is called Open Software Description (OSD) [6]. However, OSD is mainly designed for web components and do not solve the problems with component dependencies. It is important that meta-data is accessible for third part users. A common standard that makes it possible to describe components in all component models is probably a utopia. We can expect that different types of components will be described in different ways, which is vastly better than not to describe them at all.

As we do not have meta-data incorporated in the standard component models, the only information about the components we can get through binary libraries and executables. The information about which shared libraries are linked to other libraries or programs can be gathered fairly easy. In general this information is linked into the binary code and can be extracted. This information can be used to list the dependencies between different programs and libraries.

The following formal procedure is taken: A component version c is implemented as a library or an executable. A component version has a set of attributes (name, size, creation-date, and others [11] used in different component models), by which it is identified.

The set of all components installed on the system is called S. We define a relation $\rightarrow$ called “depends on”, where $c_i \rightarrow c_j$ if the correct operation of c_i requires the correct operation of c_j. This relation is transitive which means that from the direct dependencies we can derive all indirect dependencies.

The set of all dependencies is defined as

$$D = \{(c_i,c_j) : c_i,c_j \in S \wedge c_i \rightarrow c_j\}.$$

The dependency set D is stored as a baseline before new components are installed. A snapshot of the current configuration is the set of all components and their dependencies:

$$C = (S,D).$$

When new versions of existing components or new components are installed they will affect the configuration

$$C' = (S',D')$$

We identify all component versions that are placed in only one configuration

$$c_{new} \in S_D \; : \; S_D = \overline{S \cap S'},$$

and the dependencies D_Δ of components c_{new}

$$D_D = \{c : c \rightarrow c_{new}\}.$$

All components in S_D and dependencies in D_D can change the behavior of the system and are subjects for futher investigation.

For the dependencies where new components use other components

$$D'_D = \{ c_{new} : c_{new} \rightarrow c\}$$

we test if the input-output domain (i.e. expected outputs from c_{new} for inputs to c) have been preserved or not. If a new range of input to the component c occurs, this dependency should be tested in this new domain range [9].

If a system configuration can contain several component versions, specified ranges in input/output domain can be compared with the current values and used as criteria for selecting a component version to be executed [10][11].

4 Dependency Browser

To show how dependencies can be traced, we have designed an application on Windows NT 4.0, Dependency Browser which parses through the system, finds all shared libraries and generate the dependency three. A snapshot of the current configuration can be shown and saved in a repository. Different versions of snapshots are placed under version control and treated as configuration items. The current configuration, or an old snapshot, can be compared with other configuration snapshots, and the differences between the configurations can be displayed. Typically, before installation of a component, a snapshot can be saved. Then the component is installed, and a new snapshot can be done. The difference graph shows what components have been changed and what are their relations to other parts of the system. The browser can show the entire system, or a specific component and its dependencies, which makes it possible to see a potential consequence of a component update. System integrators can use the dependency browser to view dependencies in the test system, when a new component has been integrated in the system.

All components that depend on the changed component are highlighted and the user can decide and take action upon this information as shown in Figure 3 The dependency browser helps the integrator of the system to verify that nothing unexpected occurs when the system starts. With the tool it is possible to see all the affected files when a component has been updated or installed.

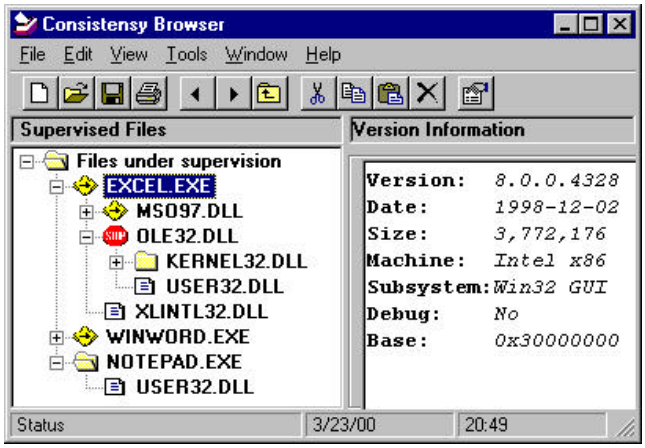

Figure 3. Affected components are highlighted in the browser to alert the user.

The changed or updated components have the stop sign icon while affected components are marked with an arrow icon. Version information of the component is presented in the right pane.

The browser can be used to browse the information and to get an understanding of the effects of the introduction of new and updated components in the system. The tool can browse different configurations and label components as changeable or not changeable. This kind of knowledge is useful if the cause of malfunction in the system is to be traced. An incorrect version of a library may have been installed by mistake and without the dependency information it is difficult to find the real cause of the problem.

5 Conclusion

In this paper we have pointed out the problems with dynamic configuration of systems. Our contribution is a proposal for component configuration management where components can be put under version control. We tie together software configuration management (SCM) and component-oriented programming (COP) with ideas from both disciplines. A simple dependency model is presented and we have shown how to solve the dependency problem for this model when new components are installed. We plan to do more work on a formal description and management of dependencies.

Future work will include the realisation of the Dependency Browser, its implementation for different component models and platforms. In this paper we have treated components as binary entities, i.e. executables or shared libraries. A deeper investigation how dependencies between loosely coupled components can be recorded, will be done. The goal is to be able to predict the behavior of a system before the system update.

6 References

[1] D. Rogerson, Inside COM, Microsoft Press, ISBN 1-57231-349-8

[2] D. Box, Essential COM, Addison-Wesley, ISBN 0-201-63446-5

[3] E. Roman, Mastering EJB, Wiley, ISBN 0-471-33229-1

[4] M. Larsson, I. Crnkovic, Development Experiences of a Component-Based System, 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)

[5] M. Larsson, I. Crnkovic, New Challenges for Configuration Management, System Configuration Management, SCM-9, Springer 1999, ISBN 3-540-66484-X

[6] W3C, Open Software Description Format, http://www.w3.org/TR/NOTE-OSD.html

[7] R. Conradi and B. Westfechtel, Version Models for Software Configuration Management, Software Configuration Management Symposium, SCM-7, 1977, Springer, ISBN 3-540-63014-7, ACM Computing Surveys, Vol. 30, No.2, June 1998

[8] J. Estublier, S. Dami, M. Amiour, Hifgh Level Process Modeling for SCM Systems,

[9] H. Thane, A. Wall, Formal and Probabilistic Arguments for component Reuse in Safty-Critical Real-Time Systems, Technical report CBSE – State of the Art, Mälardalen University, 2000

[10] J. E. Cook, J. A.Dage, Highly Reliable Upgrading of Components, 21st ICSE, 1999, ACM ISBN 1-58113-074-0

[11] Henrik Lykke Nielsen, René Elmström, Proposal for Tools Supporting Component-based programming, Workshop on Component-based Programming, 1999