Configuration Management for Component-based Systems

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

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

ABSTRACT

One of the basic problems when developing component-based systems is that it is difficult to keep track of components and their interrelationships. This problem emerges already in the requirement phase, in which we want to identify and select the most appropriate components. Later, during the assembly and deployment process, or when upgrading components, the problem of components identification and dependency management becomes even more important. One way to maintain control over upgrades is to use component identification and dependency analysis. These are well known techniques for managing system configurations during development, but are rarely applied in managing run-time dependencies. Knowledge of the possible impacts of an update is important, since it can be used to limit the scope of testing and be a basis for evaluating the potential damage of the update. In this paper we analyse different types of dependencies and discuss how to identify and specify them, in analogy with Makefiles. The dependencies can be showed in a form of a dependency graph. The dependency graphs can also be used to facilitate maintenance by identifying differences between configurations, e.g., making it possible to recognise any deviations from a functioning reference configuration.

Keywords
Component-based development, Configuration Management, CBSE, Component Specification.

1 INTRODUCTION

Historically, software configuration management is about managing evolution [4] of systems, but implicitly assumed evolution of systems specified at development phase, i.e. the source code and all other parts involved in the development process. The basic SCM support includes version management, configuration and selection management, build support and then support for the development process. The version and configuration management (selection and baselining) occur *before* the building process. After selecting and extracting particular versions, the items are used in the build procedure. They are then typically managed by *Make* or its different variants. Make can also include a special install rule which is in most cases limited to copying the items created in the build procedure to a specific destination. At that stage the core functionality of SCM finishes, although there may exist many additional steps such as packaging, delivery, installation, etc.

The paradigm of software development is changing. The source code management and build (compile) procedures are becoming less significant part of the development in favor of *assembly* and *deployment* processes. The important functions of these processes are selection and dependency management and they are not localized as one-shot events, but can be spread out to the system execution time.

The process of component identification starts already at design phase, where the system architecture is being defined. As components identified may already exist they can be precisely specified. The architecture will evolve during the development phase and then the configuration mechanisms to identify the changes introduced in the system[5,6]. The components are described by identity, connection and interface type, where a version specification is a part of the component identity.

There are many different definitions of components but in this paper we use Szyperski's definition [13] of a component. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by a third party.

Even at the system execution time we can update the system. Cook has done research on how components can be selected in run-time with respect to the problem domain [1]. In this case, each component must have an input domain explicitly described and a version identifier. The execution environment then determines which version to use at run-time. The disadvantage of this model is that the arbiter must be integrated in the environment and that all components must be simple enough to have a specified input domain.

More related work is presented by Kramer and Magee who outlines how dynamic change to a distributed architecture should be performed to preserve the behaviour of the configuration [7].

SCM deals with complexity of software. As the complexity, as architecture, is visible at development time and disappears at execution time, SCM has been focused on development phase. With introduction of components and possibility of dynamic deployment of components, a management of architecture, structuring and complexity become important at system execution time.

The question is if it is possible to apply the SCM principles and methods developed for managing development phase to the execution phase. In this paper we are concentrated on configuration management, i.e. management of system structures and component dependencies.

2 DEPENDENCIES

Managing dependencies at development phase

Managing structures and dependencies are the key points of the configuration management. When components are to be assembled and managed at execution time it is important knowing the dependencies between the components. We start with an analysis of the dependency management at the development phase.

In SCM the dependencies are managed by Make. The purpose of Make is double: To specify the *dependencies* between items used in the build procedure, and then the *build procedure* itself (Some of advanced versions of Make support selection, which improves the change/build cycle). The dependency specification can be provided manually or by using certain tools, such as *mkmf*, which parse the items and find the dependencies. Those dependencies explicitly refer to other items that are involved in the build process, usually managing only direct relations. If an include file includes other files, it is up to the developer to explicitly define those files, there is no mechanism that support the recursive process for parsing dependencies. Makefiles can also include dependencies referring to the tools used in the build process, which trigger the re-building process if some of the tools have been changed. Other definitions, such as macros, define building options and more advanced variants of Make define these options as dependencies. Note that dependencies do not manage versioning. It is assumed that version selection is already done, either by checkout or a kind of “view to selected versions” mechanism. Makefiles do not cover all dependencies since there are many implicit relations and dependencies that are assumed and which in many cases lead to problems when building or executing the modules. Still we can live with it, and in most cases we can successfully configure and build systems.

Dependencies at run-time

The question is, if the same, or similar, approach can be mapped on run-time configurations? Before analyzing dependencies at run-time, we must consider why do we need that information. The dependencies are close related to the component deployment. Dynamical deployment of components is one of the most important features of components. With information about dependencies we can easily see what can be the impacts of possible component replacement: Which other components use that component? Which components are used by that one and only by that one?

In a system built of components, similarly as for Make, the dependencies can be selected explicitly and implicitly. The explicit dependencies are realized in form of addresses, i.e. pointers to interfaces of the components. Parsing binary components it is possible to find out these dependencies, even if they are not defined explicitly in a configuration repository which would correspond to a Makefile. Even more, it is possible to parse the items recursively. In this way it is possible to build a graph of the explicit dependencies [8] between the components. The process can be somewhat complicated as the references show interfaces and methods, but what is interesting is to find the components to which interfaces they belong.

However, the explicit dependencies do not cover all the dependencies, but also other types must be considered. An application can contain a set of components which do not necessary communicate with each another but are still related. The connection point can be common data, where one component prepares data for another component.

Another type of connection can be “logical”, where only users perceive the logic, not by the systems. For example, several components together can form a GUI towards a user who “sees” a common picture of the components involved. Such a kind of dependencies is not possible to find by parsing components, but they must be specified somewhere else.

Component specification interface and component configurations

The explicit and a subset of implicit dependencies can be encapsulated in components themselves where a *specification interface* can be used as proposed in [9]. This interface will be used to gather dependencies and version information from the component when it is deployed. If no such interface is available it is possible to use the *embedding* pattern to wrap components and make them provide version information.

In the case the components provide that interface, they can provide specifications and in this way it is possible to build dependency graph showing relation between the components. When information about component versions are available, it is possible to check if the system is consistent, or a situation occur, where different components require different versions of other components.

However, a component specification interface does not cover all dependencies, as components can be combined in many, unpredictable, ways. The components can be assembled in packages, or be expressed in frameworks [3], and finally on the operating system level. On each level we should have a possibility to specify the configurations. Some operating systems, such as Windows NT or Windows 2000, have stored that information although not in a consistent way regarding component configurations.

An example is how Windows 2000 manage services, each service keep track on what other services it depends on and also its dependants. This is a good example on how the dependencies between service components are used; it is vital importance that no service shut down other services unintended. However, there is no version information available, in the dependency information, for services that can be used for more accurate component management.

One of the research topics is to find proper model for component configuration, which crucial part is dependency specification.

3 CONFIGURATION MODEL

Configuration graphs

We want to have configurations of components under control and to do that, we represent the dependencies with a directed graph. Components are the nodes in the graph and the dependencies the edges.

When the dependencies have been calculated, it is possible to create a system structure, as defined in [2], with different levels of components. On the lowest level of components are components without dependencies to other component. This system structure is used as a model to calculate quality properties such as complexity and localization factors. The complexity is proportional to the number of dependencies between the components. The localization factor denotes the number of levels between components.

Configurations can be stored under version control for later retrieval. In the configurations it is not the actual component that is stored, it is meta-data which describes the components and uniquely identifies it with a key. New installed components can be compared with a configuration to permit recognition of the affected components in the system. When new components are installed, new nodes in the dependency graph are added. In the same way, nodes are removed if components are removed.

Broken dependencies are detected when an old configuration is compared with a new. New versions of an existing component are identified by the version part of the unique key which identifies all components. New versions simply replace the older in the graph. When comparing graphs, new versions are detected since the keys will be different. Proper dependency analysis requires that a component and its version can be identified.

Nowadays we meet requirements of integration of heterogeneous systems, real-time systems with non real-time systems, or safety-critical systems with general-purpose systems. In these cases it is important to separate those parts of a system that are critical for system execution. By using the dependency graph we have the possibility of marking selected components as critical to indicate that they must not be affected by a component update. A critical component must not be upgraded or affected by an upgrade of another component, whether critical or not. Critical paths are therefore introduced. No component in a critical path may be upgraded without making an active decision to accept the change. Hence more than only the critical components are placed under supervision. On the other hand, components which depend on critical components need no special treatment and can be upgraded without notice.

Dependency browser

Even when the components are missing the specification interface, it is possible to find out (some of the) dependencies by simply parsing them.

We have developed a tool, designated the dependency browser for the evaluation of the presented configuration model. The main requirement for the prototype was to be able to parse a Windows 2000 system for its components and dependencies. An iterative development model was used to be able to show results more quickly with a working prototype. The prototype is able to browse the dependencies in a system and to store them under version control. It is also used to gather information about changes made between two configurations. Certain measurements such as complexity analysis are also provided.

There are different levels of dependency between components in a system; in a Windows system there are dependencies between shared libraries, as well as between static and dynamic COM components. Applications such as Word, Excel or Explorer, are treated as executables with their dependencies obtainable from the executable file itself. Since all Windows executable files comply with the portable executable format it is fairly easy to track the shared libraries but not so easy in the case of COM components. Scanning all shared libraries and executables in a system creates a basic dependency graph. Various features of the tool then extend this graph. The windows registry has been used to gather information about each component, which is then added to the dependency graph. Step by step, a configuration graph is built up for use in configuration management. Processes can be supervised and when new components are dynamically loaded into the memory, the graph is extended with dynamic dependencies. However, the creation of a complete dependency graph at the Windows platform is a tedious task as there are too many dynamic dependencies difficult to detect because they have not been activated during periods of time when the system is supervised. Preliminary results show that it is difficult to identify all the components and their dependencies on the Windows 2000 platform. The configuration theory can be applied when the dependencies are discovered. More effort and research is needed in the area of gathering the dynamic dependencies.

4 APPLICATION ON EXISTING MODELS

Interesting work is to see how the theoretical model presented earlier can be applied to existing component models, such as, COM [10], .NET [14] and CCM [12]. Traditionally COM handles multiple versions of components in a sense that it does not accept new versions of interfaces and states that all new versions must be treated as new components with unique identifiers. This prevents new versions from interrupting the configuration in an unpredictable way. Microsoft .NET[11], is the new runtime environment which supports, among others, C#, handles coexisting components by introducing unique identifiers for each component. Private and shared components are two means of classifying components in .NET. If a component is private, it is not possible for other components to use it and this avoids the versioning problem since no other component can use it and the number of dependencies is one. If a component is declared as shared, many different applications can use it but with the risk of breaking dependencies when updating the system.

The use of components will even increase when new web services are available like any other component. Each web service will provide one or more interfaces that can be used by any client around the world. With the .NET architecture from Microsoft it is possible to write truly distributed component based applications. The clients receive information about what interfaces are available and then they can use proxies which communicate through the Simple Object Access Protocol (SOAP). SOAP is an open standard based on XML and http. Any access to a remote web service in .NET will look exactly as any other component access. All components are treated equal and with the open SOAP protocol it is possible to communicate over different component models. With the object orientation we had to stay with one programming language, and with the first component technologies the language could be chosen but the model was one. Now it is not longer necessary to develop applications for just one component model, for example CORBA and COM can now interoperate freely.

With the .NET architecture it is possible to custom metadata for the components. We think it possible to use these capabilities to create metadata that contains version information. This information can then be used to gather dependencies and valuable conclusions can be drawn.

5 CONCLUSION

In this paper we claim that the component identification and specification of dependencies between them will play an important role, not only during the development (design) phase but also at the run-time. All information must be under version control to make it possible to evolve the systems in an efficient and controlled way.

The challenges for the future work are to find the appropriate information about the components needed in different phases of a software lifecycle: The components should provide information to be easily identified and selected as proper candidates during the requirements and design phases. If the components do not provide information then it must be investigated how version information can be added to the components. The components should be identified as parts of the system that will be built. The system should describe its architecture by referring to the components.

At the run time, the system should be aware of the components integrated in the system. As information required in different phases are slightly different, but at the same time have much in common, it is interesting to find out the most appropriate format of information and possibility of exchanging information.

Another challenge is to define an efficient way of identifying configurations of component-based systems and by using this information make it possible to safely and efficient change the configurations.

6 REFERENCES

[1] Cook J. E. and Dage J. A., "Highly Reliable Upgrading of Components", In *Proceedings of 21st International Conference on Software Engineering*, ACM Press, 1999.

[2] Crnkovic I. "Large Scale Software System Management" Doctoral Thesis Department of Electrical Engineering, University of Zagreb 1991

[3] Crnkovic I., Küster J. K., Larsson M., and Lau K.-K., "Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues", In *Proceedings of 4th IEEE International Baltic Workshop in DB and IS*, 2000.

[4] Estublier J., "Software Configuration Management: A Roadmap", In *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, ACM Press, 2000.

[5] Hoek A. v. d., "Capturing Product Line Architectures", In *Proceedings of 4th International Software Architecture Workshop*, ACM Press, 2000.

[6] Hoek, A. v. d., Heimbigner, D., and Wolf, A. L., Capturing Architectural Configurability: Variants, Options, and Evolution, report Technical Report CU-CS-895-99, 1999.

[7] Kramer J. and Magee J., Analysing dynamic change in distributed software architectures, *IEE Software*, volume 145, issue 5, 1998.

[8] Larsson M. "Applying Configuration Management Techniques to Component-Based Systems" Licentiate Thesis Dissertation 2000-007, Deparment of Information Technology Uppsala University. 2000

[9] Larsson M. and Crnkovic I., "New Challenges for Configuration Management", In *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1675, Springer Verlag, 1999.

[10] Microsoft, The Component Object Model Specification, report v0.99, Microsoft Standards, Microsoft, 1996.

[11] Microsoft, Microsoft, .NET, http://www.microsoft.com/net/.

[12] OMG, CORBA Components, report orbos/99-02-01, OMG, 1998.

[13] Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[14] Wille C., *Presenting C#,* SAMS Publishing, 2000.