New Challenges for Configuration Management

Magnus Larsson[1], Ivica Crnkovic[2]

[1] ABB Industrial Products AB, LAB,
S-721 67 Västerås, Sweden
Magnus.Larsson@mdh.se
[2] Mälardalen University, Department of Computer Engineering,
S-721 23 Västerås, Sweden
Ivica.Crnkovic@mdh.se

Abstract. More and more systems are developed using components. There is a move from monolithic to open and flexible systems. In such systems, components are upgraded and introduced at run-time, which affects the configuration of the complete system. Keeping up-to-date information about which components are installed is a problem. Updating a component also affects the compatibility of the system. It is therefore important to keep track of changes introduced in the system. In the product life cycle, CM is traditionally focused on the development phase, in particular on managing source code. Now when changes are introduced in systems at run-time and systems are component-based, a new discipline, *component configuration management* is required. This paper analyses component management and highlights the problems related to component configuration. Requirements on component configuration management are outlined, and some directions to possible solutions of the problems are given.

1 Introduction

In recent years we have recognized a new paradigm in the development process: From a complete in-house development, to a development process which has focused on the use of standard and de-facto standard components[1], outsourcing, COTS (commercials off the shelf). The final products are not closed, monolithic systems, but are instead component-based products which can be integrated with other products available on the market [3]. Developers are not only designers and programmers, they have become integrators and marketing investigators. The new paradigm increases the efficiency of development and the flexibility of delivered products, but at the same time increases the risk of losing product configuration consistency. The higher risk reduces the product reliability, which is a critical factor for certain types of systems, such as real-time and safety-critical systems. Configuration Management (CM) is a discipline, which controls the consistency between the parts of the entire system, and can increase the reliability of component-based products.

[1] Definitions of components are presented in chapter 3.

Software systems based on standard components are the results of a combination of pure development and integration of components. The requirements for conventional use of CM remains, but new requirements related to component management appear in all phases: in the design, integration and run-time. We can expect that the source code management will become less critical, because there is less internal development and because of the fact that source code management in CM is very well established in both theory and implementation. The integration part, i.e. configuration, and version management of the components becomes more important. New aspects of CM arise in the run-time phase, as components are usually loosely coupled, and their update is allowed in the run-time environment.

The importance of CM, and challenges in research and implementation of CM support, are emphasized in the 1998 CBSE (Component-based Software Engineering) workshop [1], as quoted: "In particular, high composeability in a product line setting amounts to mass customization and this introduces tremendous configuration management challenges and support challenges."

Although CM provides good support in the development phase, especially in the coding phase, there is a lack of CM disciplines managing components already developed. This paper points to certain new aspects of CM in managing components. Chapter 2 shows different phases in component development processes and run-time environments and their relations to CM activities. The different compatibility levels of the components are discussed in chapter 3. Chapter 4 gives an overview of the component characteristics related to CM issues. The problems, which appear due to the lack of proper CM support, are presented. Chapter 5 outlines certain models for improving the support and improving the reliability of products.

2 Using CM in Component-based Product Life Cycles

Configuration management is applied in different phases of a component-based product life cycle. Fig. 1 shows an example of a development and run-time process. In the development phase we build libraries from the source code. A component is built by assembling libraries and collecting other types of items such as documentation and executable files. Finally, a typical component-based product consists of a set of components.

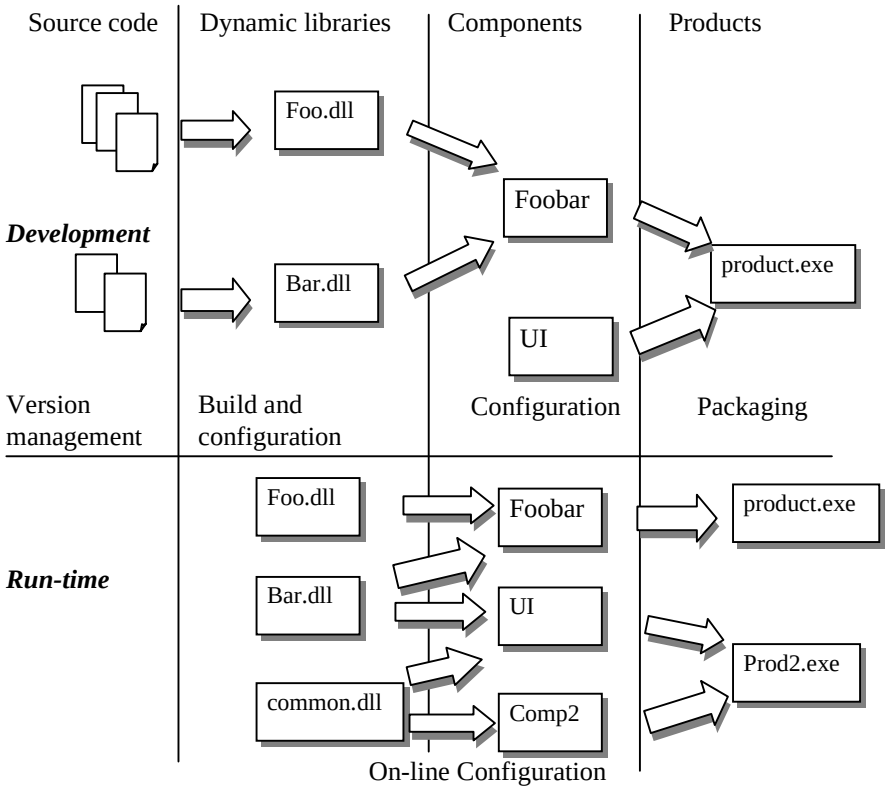

Fig. 1. CM activities in different phases of a component-based product life cycle

In the first development phase, source code management is used to track the introduction of different versions of source code, to enable parallel development, etc. Many CM tools supporting this are available today. The building phase is also supported by CM tools such as different variants of make and configuration tools, the results of the building procedures being connected to the source code. One step closer to CM for components is to use description logic, to describe configurations, in combination with make to build a product[10]. However, this does not solve the run-time issues.

Having control over the source code and producing the system entirely from the source code makes it possible to control the target system configuration. When using imported components, we lose this control, because we only partially know their behavior. It is possible however to manage versions and configurations if we place the components under version control and deliver them as a part of the product.

When delivering components or products, which are part of a target system, we face two problems:

- We cannot predict the behavior of the entire environment of the target system. The system may contain another product, which uses the same component as our

product. The relations between components, and the changes we may obtain by introducing a new version of a component, are uncertain.

- A more serious problem is the dynamic behavior of the system configuration in the run-time environment. If we permit component-updating during the run-time, by updating dynamic libraries, we could be facing a situation in which a new component version works for one product, but not for another. There are also different aspects of updating, such as moving or copying an application from one computer to another, or automatically generation of code.

CM can provide solutions to these problems, and those are new challenges for CM. To cope with the problem, the research and practical implementations must focus on the component management. The following chapters describe the mechanisms of component management and point at the problems related to their identification. Finally an outline of possible solutions for improvement of the component version and configuration management are presented.

3 Component Compatibility

There are different definitions of software components [1]: A component is a non-trivial, nearly independent, and replaceable part of a system which fulfills a clear function. A component conforms to and provides the physical realization of a set of interfaces. A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces which can be discovered at run-time. A component can be deployed independently and is subject to composition by third party.

The importance of components becomes significant where technologies for their development and integration are being standardized. The most prominent component technologies today are Java Beans, COM/DCOM and ActiveX, and CORBA. In this paper, we illustrate component-management problems using COM/DCOM technology, but the same principles are valid with other technologies.

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 is permitted, and which type of compatibility is required. We define three levels of compatibility:

- *Input and Output compatibility*. A component requires input in a specific format and produces results in a defined format. The internal characteristics of the component are of no interest. An example of this type of compatibility is provided by different word-processors producing the same document format. This type of compatibility does not ensure that the interfaces or the behavior are preserved.

- *Interface compatibility* (at development time and at run time). The interface remains the same, but the implementation can be different. A typical example is given by different implementations of ActiveX objects, with the same interface.

Interface compatibility is more demanding than input and output compatibility, but it does not need to have the same behavior.

- *Behavior compatibility*. Internal characteristics of the components, such as performance, resource requirements, etc., must be preserved. Such requirements can be appropriate for real-time systems. This is the strongest compatibility requirement and it includes the previous ones.

The compatibility criteria can be used in deciding 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. It is important to maintain the required level of compatibility to avoid the risk of interrupting the whole system.

4 Managing Components

Components typically consist of shared libraries, where the component functions are implemented. The programs using components do not refer to the libraries directly but to the component interfaces. The libraries are implementations of the interfaces. We need to keep track of changes on both logical and physical levels as well as their relations. Both libraries and interfaces must be identified. Component Configuration Management must work on both levels. Versioning of interfaces is a more difficult task, because the interface is an abstraction without information about the physical representation. For this reason, we separate the problem of managing components onto two levels: Managing libraries and managing interfaces.

4.1 Libraries

Historically there were less problems in this area as all libraries were statically linked into the executables. This prevented the executable from being updated when a new version of the library was released. An advantage of this approach is that the executables are protected from uncontrolled use by the new version of the library. A disadvantage is the necessity to re-link the executable only to incorporate a new version of the library, which is unnecessary work when the library is interface-compatible. Another disadvantage is that all executables which shared the same library must be linked with their own copies of the library. The concept of shared libraries was introduced to avoid this. This was a significant improvement since we could now share libraries and make updates without re-linking the executables while functions were interface-compatible. In Microsoft platforms, shared libraries are designated dynamic link libraries or dlls, which can be loaded and unloaded whenever needed. On other platforms, such as different Unix platforms, shared libraries are loaded together with the main executable.

Unfortunately, the concept of shared libraries introduces new problems related to the consistency of the system, as illustrated by Fig. 2.

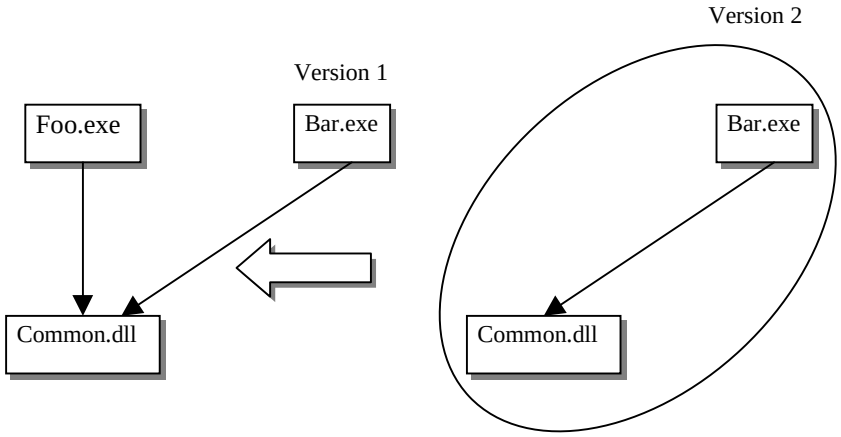

Fig. 2. Foo.exe stops work when the new incompatible version of Common.dll is introduced.

The figure shows how a new version might damage the system. Common.dll version 1 will be overwritten with version 2 when the new version of bar.exe is introduced. The replacement could be successful if version 2 of Common.dll is interface-compatible with version 1, but definitely not if the compatibility level is less. There is a risk that Foo.exe will stop working after the new version of Common.dll is introduced.

The new interface-compatible version of Common.dll may contain undetected errors as it was tested with Bar.exe only and not with Foo.exe. Foo.exe may then access some erroneous code and crash even if the library was interface-compatible.

One way to handle multiple versions of libraries is to insert version information into the actual library name as Microsoft does in MFC [9]. 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 such as developed in Fig. 2. With different names for different version, the situation may be as in Fig. 3.

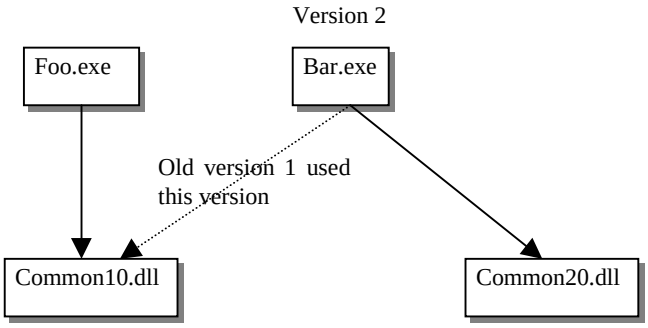

Fig. 3. Common10.dll can now coexist with Common20.dll

This solution is to some extent similar to the static linking of executables, because an executable always uses the same version of the shared library. The solution however

becomes cumbersome when several versions and variants must be installed in the system. There are, for example MFC42d.dll, MFC42u.dll and MFC42ud.dll which are respectively debug, Unicode and debug/Unicode versions of the MFC library. This tight coupling emerges from the design of the C/C++ compilation model, which was not intended to support independent binary components.

Another way to circumvent the problems is to upgrade all executables dependent on a particular library when the new release is ready. This means that both Foo.exe and Bar.exe will be updated instead of Bar.exe only (Fig. 2). This approach can be taken on the assumption that complete control over the whole deployment exists, and from that perspective is very limited.

Suitable support can be achieved with the help of CM functions which keeps track of changes, and by checking which changes are permitted for an executable or a component.

4.2 Interfaces

An interface is a connection between a component and its user. If an interface is changed, the user needs to know that it has been changed and how to use the new version.

Functions exposed to the user are usually designated Application Programmable Interfaces (API). If a change is made in the API, the user must recompile his code. This is the case for compiled languages such as C/C++ but not for interpretative languages such as Smalltalk or Java.

In an object-oriented world, an interface is a set of the public methods defined for an object. Usually the object can be manipulated only through its interface. In C++ the user need only recompile the code when an interface, referred to from the code, is changed.

A disadvantage is that the user of the object must use the same programming language throughout the whole development.

Separation of the interface from the implementation is a means of avoiding this tight coupling. This kind of separation is performed with binary interfaces as in CORBA [3] and COM [6]. Binary interfaces are defined in an interface definition language (IDL) and an IDL compiler, which generates stubs and proxies to make the applications location transparent.

COM solves the interface versioning problem by defining interfaces as unchangeable units. Each time a new version of the interface is created a new interface will be added instead of changing the older version. A basic COM rule is that an interface cannot be changed when it has been released. This makes couplings between COM components very loose and it is easy to upgrade parts of the system indifferent from each other. Fig. 4 shows that it is possible to run new clients together with old server components or vice versa.

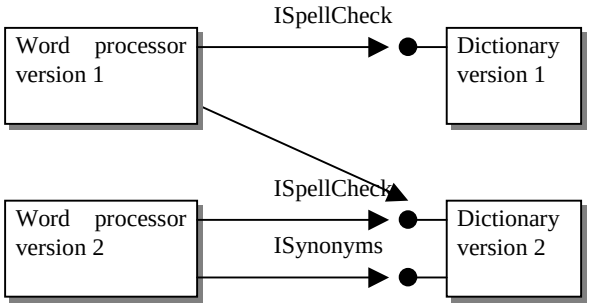

Fig. 4. Possible combinations between old and new clients and their server component.

Even if an interface has not been changed, its implementation can be changed. This increases the flexibility of possible updates, but also introduces the possibility of resultant uncontrolled effects. For this reason, it is of interest to know if the implementation has been changed.

Today there is no support for the handling of components in the configuration management perspective. CM functions should provide information about the changes on the interface level.

5 Proposed CM for Libraries and Components

No or insufficient information is available when a system is assembled from components. There is no standard way to track the dependencies between components. When a system is upgraded with a new program, the programs running already might be affected without notice because the new program may introduce new versions of existing components in the system (see Fig. 2). It is necessary to determine which interfaces (i.e. components) are used by a program or a component.

As a component is placed in a set of shared libraries some control may be obtained by keeping the libraries under control. We propose a component configuration management on two levels, the library level and the component level.

5.1 CM for libraries

Which shared libraries are linked to another library or program can be seen. This can be used to list the dependencies between different programs and libraries. When installing a new program containing libraries the following steps shall be taken:

1. Take a snapshot of the current system configuration.

2. Install the new modules.

3. Take a snapshot of the new system configuration.

The contents of a snapshot are all programs and libraries installed in the system and are treated as nodes in a graph. A number of different attributes are associated with each library. The information for each node in the graph uniquely identifies the module. We propose that at least date, time, size and name shall be stored. Other attributes are which compatibility change is allowed or if a warning is to be given when a particular module is updated.

A snapshot of the system is presented as a dependency graph. Fig. 5 shows an example of how one of the COM libraries depends on other libraries.

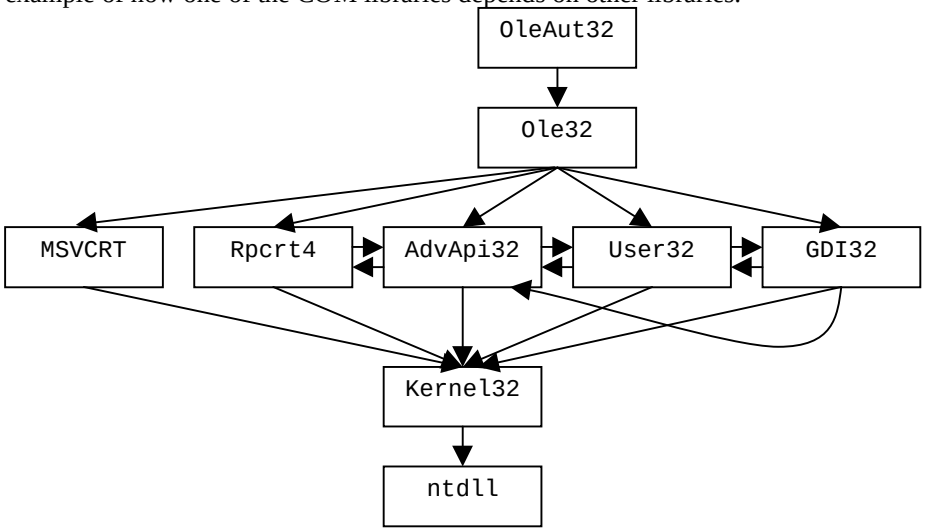

Fig. 5. A dependency graph for OleAut32.dll.

Different versions of snapshots are placed under version control and treated as configuration items. A tool which could browse this information would present the differences graphically to the end user. The user would now gain an understanding of the effects of the introduction of new and updated libraries in the system. An alarm would be activated if a library which should not have been affected is changed. The configuration tool could browse different configurations and could label components as changeable or not changable.

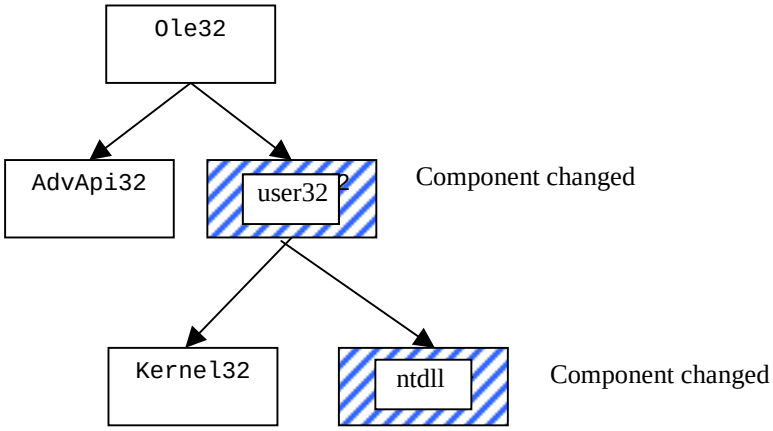

Fig. 6. A dependency graph that shows all changed versions.

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. This kind of identification gives no direct information about which components are changed and which can be affected by the change, but indirect information is available since the physical representation of components are libraries.

5.2 CM for components

In this chapter, we discuss COM as an example. COM treats interfaces in a manner unlike other object models such as CORBA.

COM components expose themselves and communicate through COM interfaces only. Moreover, COM is designed to work with loose references between components. There is no requirement that the clients shall know the class declaration since every class declaration contains implementation details. Components should be able to add or remove interfaces without affecting existing clients.

As components are loosely coupled there is no information connecting different versions of components with each other. A COM component finds its fellow components through the Windows registry in which 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.

Unfortunately, there is no capability in the target system for finding which interfaces are used by a component. This prevents us from getting proper information about all dependencies in the system.

If we do not know which components a program uses in run-time, we must request that knowledge. This can be obtained if the provider of the components implements a specific interface for version management, which we designate IVersion (Fig. 7). The `IVersion` interface can return facts about version, name, creation date, compatibility change, interfaces provided and components used. If the components

had such an interface, it would be possible to write a tool that could browse and record the dependencies between the components.

```
interface IVersion : IUnkown
{
  HRESULT Name([out , retval] BSTR *name);
  HRESULT Version([out , retval] VERSION *version);
  HRESULT CreationDate([out , retval] DATE *date);
  HRESULT TypeOfChange([out , retval] BSTR *name);
  HRESULT History([in] LONG size,
              [out, size_is(size)] HISTORY history[*]);
  HRESULT HasInterfaces([in] LONG numOfElements,
              [out, size_is(numOfElements)] IID
interfaces[*]);
  HRESULT UsesInterfaces([in] LONG numOfElements,
              [out, size_is(numOfElements)] IID
interfaces[*]);
}
```

Fig. 7. IDL specification of IVersion.

- `Name`, `Version` and `CreationDate` identifies the component.
- `TypeOfChange` indicates the compatibility level affected by the change.
- `History` informs about previous versions of the component and which type of change applied between them.
- `HasInterfaces` shows all interfaces provided by the component.
- `UsesInterfaces` lists all interfaces used. This list makes possible the building of the dependency tree of the components.

In the absence of a standard version interface, another method is to parse in some way the dependency data from source code files to provide a list of dependencies with the release of a new product. This has some major disadvantages. Firstly, it cannot be applied to third party components. Secondly, it might work for the first level of dependencies where there is source code, but if other third party components are included, no information can be obtained because of the lack of source code.

A possible partial solution to the problem finding dependencies between components is to track the interfaces from the registry repository. All interfaces are registered in the Windows registry with information about where to find the dynamic link library which implements the stubs and proxies for that particular interface. This mechanism provides us with the information we need to see if an interface has been changed during an update. The snapshot browsing tool has a list of all interfaces apart from the libraries and programs installed. The tool can now warn if the implementation of an interface has been changed. It is possible, using this method, to determine if new interfaces have been registered or if old interfaces have changed implementation.

6 Conclusion

We consider that there is a need for component configuration management, especially during the run-time when components can be changed on the fly. In this paper we have highlighted the different phases in component management in which CM is needed. Support from CM related to component management is rudimentary today and we propose beginning work in a new area, Component Configuration Management.

For want of standardized techniques in component management, we have proposed certain relatively simple methods to identify components and possible changes they can cause in the system. Further work will include a deeper investigation of how to snapshot a system for an insight into the interrelationships between different components. A tool capable of browsing and analyzing an existing system for this should be developed.

7 References

[1] Don Box, Essential COM, Addison-Wesley, ISBN 0-201-63446-5

[2] Alan W. Brown, Kurt C. Wallnau: An Examination of the Current State of CBSE: A Report on the ICSE Workshop on Component-Based Software Engineering, 1998 International Workshop on CBSE, http://www.sei.cmu.edu/cbs/icse98/summary.html

[3] Continuus Software Corporation, http://www.continuus.com/homepage.html, 1999

[4] CORBA, http://www.corba.org

[5] Ivica Crnkovic, Magnus Larsson, Managing Standard Components in Large Software Systems, Position paper on Second International Workshop on Component-Based Software Engineering, Los Angeles, May 1999

[6] Microsoft corporation, http://www.microsoft.com/com

[7] Microsoft Source Safe, http://msdn.microsoft.com/ssafe

[8] Rational http://www.rational.com/products/clearcase/index.jtmpl, 1999

[9] Dale Rogerson, Inside COM, Microsoft Press, ISBN 1-57231-349-8

[10] Andreas Zeller, Versioning System Models Through Description Logic, Proceedings ECOOP'98 SCM-8 Symposium, vol 1439 of Lecture Notes in Computer Science, Springer-Verlag.