Distributed Component Technologies and their Software Engineering Implications

Wolfgang Emmerich
Dept. of Computer Science
University College London
Gower St, London WC1E 6BT, UK
w.emmerich@cs.ucl.ac.uk

Abstract

In this state of the art report, we review advances in distributed component technologies, such as the Enterprise Java Beans specification and the CORBA Component Model. We assess the state of industrial practice in the use of distributed components. We show several architectural styles for whose implementation distributed components have been used successfully. We review the use of iterative and incremental development processes and the notion of model driven architecture. We then assess the state of the art in research into novel software engineering methods and tools for the modelling, reasoning and deployment of distributed components. The open problems identified during this review result in the formulation of a research agenda that will contribute to the systematic engineering of distributed systems based on component technologies.

1. INTRODUCTION

The idea of constructing software in the same way as hardware is constructed, i.e. by assembling reusable components is as old as the discipline of software engineering itself. It was, in fact, at the 1968 NATO Workshop, which is commonly considered as the birth of software engineering, that McIllroy introduced the notion of components [30]. During the last 30 years a number of people have refined the notion of components and we shall use Michael Stal's definition that treats a *component* as "a self-contained entity (black-box) that exports functionality to its environment and may also import functionality from its environment using well-defined and open interfaces". In this context an *interface* defines "the syntax and semantics of the functionality it comprises" and "components may support their integration into the surrounding environment by providing mechanisms, such as introspection or configuration functionality" [43].

To date, software engineers have a number of different technologies at their disposal that implement these notions of components and interfaces. A large number of component technologies exist, such as the more established *local* component models, e.g Microsoft's Component Object Model (COM) and Sun's JavaBeans, whose execution is confined to just one machine. These component models have recently been extended to allow for *distributed* execution across multiple machines in, e.g. Microsoft's COM+ and .NET, Sun's Enterprise Java Beans (EJB) and the OMG's CORBA Component Model (CCM). The availability of these technologies, has enabled an approach to software development that is often referred to as *component-based development* (CBD). CBD involves the construction and deployment of software systems that have been assembled from components. CBD includes activities such as the discovery, engineering, procurement of components, as well as the re-engineering of legacy software for component assembly. *Component deployment* denotes the activities that are related to the transfer of components into a run-time environment as well as the configuration and customization of components without changing their implementation.

A large body of literature is available on the different aspects of CBD [50, 15, 52] that we do not attempt to reproduce in this paper. The existing literature has a strong focus on the more mature local component models. We complement that literature in this paper by discussing the implications that the availability of distributed component technologies has for software engineering. We cover two main aspects: We first discuss how practitioners can engineer distributed software systems using distributed components with readily available software engineering methods and techniques. We then review software engineering research results that will, we hope, lead to improvements in the state of the practice.

This paper is structured as follows: Section 2 presents the different developments that influenced, and lead to the notion of distributed components. In Section 3, we show architectural styles that are supported by distributed components in order to give an idea what can be achieved with these technologies. We also discuss the importance of iterative and incremental development processes, such as the Unified Process [21], for distributed development and we show how model driven architecture is achieved using component-specific extensions of the Unified Modeling Language (UML) [44]. Distributed Component-based systems are often also developed in a distributed setting; in Section 4, we review how such development can be supported

To appear in Proceedings of the 24^{th} International Conference on Software Engineering, May 19-25, 2002, ACM Press

using state-of-the art consistency checking and linking techniques. We also discuss in that section novel research into the use of model checking techniques for component-based architectures, as well as component deployment. In Section 5, we indicate gaps in research on the systematic engineering of distributed systems using components before concluding the paper in Section 6.

2. THE ROAD TO DISTRIBUTED COMPONENTS

For a long time it was claimed that object-orientation was the solution to reusability of software. The proponents of this view argued that object-orientation would provide the encapsulation primitives necessary to implement Parnas' ideas of information hiding and postulated that this would naturally lead to reusable software. Object-orientation, indeed enabled the development of reusable class libraries, such as the Standard Template Library [49] or Foundation Classes for Java or C++. These class libraries provide particular type constructors, such as sets, lists, hash tables and so on; however, software reuse in the large has never been achieved by object-oriented development. Reuse of objects is hampered by the large number of fine-grained classes generated during object-oriented modelling that are entangled in a web of association, aggregation and generalization relationships. The large number of dependencies makes it difficult to take classes out of the context in which they were developed and reuse them elsewhere. Reuse is also hindered as classes need to be instantiated or be inherited from and this involves hard core programming.

Components overcome this problem and provide more easily reusable units of code by clustering together related classes into more coarse-grained implementational units that provide one or more well-defined interfaces. More importantly though they provide mechanisms to assemble and configure systems without requiring hard-core programming skills. Examples of such component technologies include Microsoft's Component Object Model (COM) [1], which evolved from the Object Linking and Embedding (OLE) technology and Sun's Java Beans, which drew a large number of ideas from Borland's Delphi. These technologies have in common, that their execution is confined to just one machine and they are therefore often used for constructing user interfaces. They do not provide any mechanisms for distributed deployment (such as load balancing or replication), nor do they provide for distributed communication mechanisms to interact with components that reside on other machines.

Concurrently with the evolution of object models into local component models the industry realized that it was no longer viable to assume that object communication could be confined to just one host. To address that problem, the Object Management Group (OMG) defined the Common Object Request Broker Architecture (CORBA) [39], which supports the distributed communication of possibly heterogeneous objects across machine boundaries. CORBA provides a distributed object model that can be mapped to a large number of object models embedded in programming languages, and an interface definition language that can be used to define the interface of a distributed object in a programming language independent manner. CORBA defines bindings to different programming languages so that client objects can request operation invocations from server objects in one programming language and server objects can be implemented in another programming language. CORBA also provides a number of *services* that support locating distributed objects, managing the state of objects on persistent storage, letting objects participate in distributed transactions and securing access and communication of distributed objects. CORBA was influential in the development of Java's Remote Method Invocation (RMI) specification, which provides for invocation of Java methods across machine boundaries, and for the definition of a distributed invocation capability for COM that is commonly known as DCOM [12]. For a detailed comparison of these technologies, refer to [4, 7].

The difficulties of creating reusable CORBA and RMI objects are similar to those of local object technologies discussed above. In many ways they are even more pronounced for distributed objects as the designer of a distributed server object needs to use particular implementations of persistence, transaction, concurrency control and security services, all of which make it even more difficult to reuse a server object in a different setting. It is these obstacles that led to the development of distributed component technologies. Central to the notion of distributed components is the idea that the designer of a component should only be concerned with the application or business logic and not be burdened with the implementation of location, persistence, transactional capabilities and security. These concerns are provided for by the *containers* that are supplied within application server products and control the creation of components, their activation and deactivation, as well as the execution of transactions.

There are specifications for three main distributed component technologies. Microsoft's COM+ supports the execution of COM components in Microsoft's Transaction Server and thus implements transactional and security capabilities as well as distributed communication. Sun defined the Enterprise Java Beans (EJB) specification as part of their Enterprise Edition of the Java 2 platform [34]. The OMG defined the CORBA Component Model (CCM) [41] in an upwards compatible manner with EJB in order to extent the expressive power of EJB and at the same time provide a distributed component model for languages other than Java.

At the time of writing this paper COM and COM+ are omnipresent in installations of Microsoft's Windows operating systems products and widely used. EJB has been largely successful and it is probably fair to say that it is the most widely used distributed component model at the time of writing this paper. It is implemented in about 30 different application server products. Analysts estimated the size of the application server market in 2001 to be approx. 1.2 billion US$ and the most important vendors are BEA (WebLogic Server), IBM (WebSphere) and Iona (iPortal). There are also a number of open source projects, such as JBoss and OpenEJB. iCMG are the sole provider of a CCM product and a few open source projects that implement the CCM, e.g. OpenCCM [29] and MicoCCM [42].

3. COMPONENT-ORIENTED DEVELOPMENT PROCESSES: STATE OF THE PRACTICE

In this section, we will review the distributed component technologies that we introduced above and show how they are used in practice. We will first provide an idea of the types of software architectures that currently rely on distributed component technologies and discuss two architectural styles in which distributed component technologies play a key role. We will then look at architecture-centric software development processes, most notably the Unified Process and finally we will the means by which the detailed design of distributed components is supported by component-specific extensions of the UML.

3.1 Use of Distributed Components in Architectural Styles

In this section, we discuss architectural styles that have been successfully used in industrial projects by deploying distributed component technologies.

Multi-tiered architectures are layered in such a way that each different layer implements a particular concern and can be executed potentially on different machines. Thus the communication between the different tiers is often achieved by using network protocols or distribution middleware.

We first discuss multi-tiered architectures that are deployed across multiple hosts where components are used in middle tiers. We then discuss how components can be used to wrap legacy systems so that they can be used in a seamless manner in multi-tiered architectures. We then review the relationship between components and web services. We conclude by discussing the use of components in architectures for application services.

Components to Implement Business Logic: Figure 1 shows an example of a multi-tiered architecture that uses a standard web browser as the device to display the user interface. As browsers can be assumed to be installed on any machine, this architecture appeals in situations where the deployment costs need to be independent of the total number of users. E-commerce or large-scale intranet applications are examples of such settings. The user interface to be displayed is delivered in form of HTML pages by a presentation tier that might be implemented using Microsoft's Active Server Pages, Java's Servlets or Java Server Pages and they are transmitted to the display tier using the http protocol. The server pages or servlets, in turn, rely on a business object tier that executes the business logic of the application. As these business objects might be executing on different hosts, they would use Java remote method invocation (RMI) or CORBA's Internet Inter-ORB Protocol (IIOP) to facilitate the required remote interaction. In case of an e-shopping application it would be in this business object layer that the state of the shopping session would be kept by updating a shopping cart object. The business object layer would also drive the transactional behaviour of the application by starting and committing transactions. Finally, the business object tier relies on a persistence tier that is most often built using a relational database in order to implement persistence of the state of business objects and changes to this state within transactions.

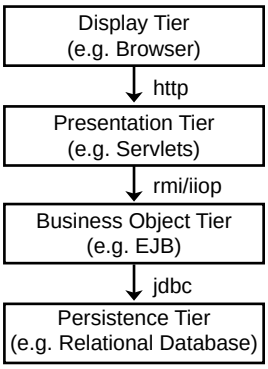

Figure 1: Components for Business Objects

While it would be perfectly feasible to implement the functionality of the business object tier without distributed object technology, the use of EJB, MTS or CCM assists considerably in addressing non-functional requirements, such as scalability and reliability. Distributed component technologies provide very flexible means for changing the deployment of components that implement the business objects. They allow for components to be replicated on clusters of machines in order to bear the potentially significant load of, for example, an e-shopping application with an unknown number of concurrent users. Moreover, they provide the primitives for a number of operations to be executed as transactions and they implement the mapping of the state of business objects onto persistent storage, potentially in a manner transparent to the designer of the application.

Components for Legacy Wrapping: It is probably fair to say that to date most industrial IT projects are not green field developments. Instead new projects need to interoperate with existing legacy IT infrastructures that have proven to be reliable and have received a lot of investment. Examples of such legacy are enterprise resource planning systems, flight reservation systems, financial accounting and settlement systems and so on. The detailed discussion of such an enterprise application integration is beyond the scope of this paper, refer to [8]. Nevertheless, organizations are keen to use new technologies for new developments; for example an airline might want to offer direct access to flight reservations over the Internet. Then a need arises to develop an integration of the multi-tier architecture that we discussed above with the legacy flight reservation system. Again distributed component architectures prove useful as they facilitate the wrapping of these legacy systems in a set of interfaces so that they can then be considered and used in the same way as any other component.

Figure 2 shows how the multi-tiered architecture of Figure 1 has been extended with an adapter tier. The purpose of adapters executing in this adapter tier is to provide a uniform set of interfaces to the Enterprise Information Systems (EIS) tier and to avoid that the complexity of accessing the EIS tier is spread across the remaining business object tier. Adapters would access the EIS tier using whatever proprietary interfaces these provide. Often these are message queues, such as implementations of the Java Messaging Ser-

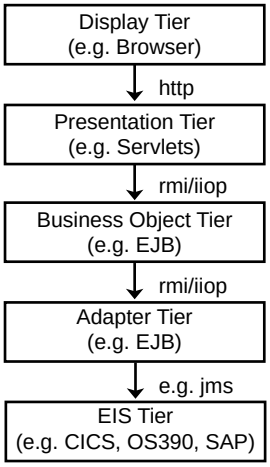

Figure 2: Legacy Wrapper Architecture

vice [14], that define particular messages structures. Connections with legacy systems might also be achieved through databases or files with known schemas or formats or even sockets.

Even though the provision of just one connector interface for a legacy system is a vast improvement over previous practices that involved building dedicated interfaces between different systems, constructing a large number of such wrapper components can be a significant endeavour. To enable EIS system providers to build these adapters so that they can be deployed throughout their customer base, Sun has defined a new Java Connector Architecture [46]. It will allow vendors of EIS systems to implement connectors to their systems in a standardized manner so that any legacy adapter tier is populated by reusable connectors.

3.2 Iterative and Incremental Development

The way distributed components are arranged in a particular software architecture is largely driven by the non-functional requirements that a software system needs to meet. In particular, the choice of programming language, hardware and operating system platform influences which particular distributed component technology can be chosen. Moreover, scalability and reliability requirements may impose the need for replicating components across a number of machines. Security requirements may demand access control, auditing and encrypted communication. Finally new systems often have to be integrated with legacy systems.

There is a considerable risk in any new project that the chosen architecture cannot satisfy the requirements that the stakeholders stated. To date there are no analytic techniques used in practice to identify and mitigate architectural risks. Instead software architects adopt an incremental and iterative development process, which was first suggested by Mills [32], to mitigate such risks. Incremental and iterative development is now part of many process models, most notably the cleanroom software engineering approach [6], the Objectory Process [22], the Unified Process [21], and vendor-specific versions of it, such as the Rational Unified Process (RUP) [26].

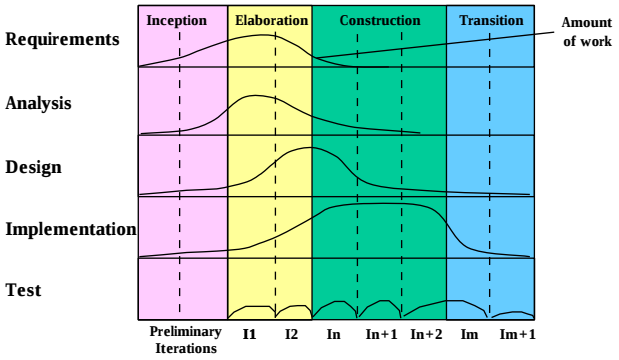

Figure 3: Unified Process: Workflows & Phases

A main reason for the success of the Unified Process and RUP is that it makes risk identification and mitigation central activities. To achieve this, these processes introduce a number of different phases. The Unified Process and RUP identify an Inception, Elaboration, Construction and Transition phase as shown in Figure 3. The development team performs during each phase one or more iterations that consist of requirements, analysis, design, implementation and testing workflows. The result of each iteration is an increment that addresses a particular concern. Increments developed during the Elaboration phase are mostly prototypes that are developed to show feasibility of an architecture and to mitigate development risk, while those during Construction phase are aimed at developing the required functionality. During Transition, the increments are beta-tested and transferred into operation.

The focus during the first Inception phase is on the identification of those requirements that may carry certain risks. For projects that involve distributed component technologies for meeting security, performance, scalability and reliability requirements are certainly going to be among these. The result of the Inception is then a prioritized risk list and a plan of their mitigation during Elaboration phase. Most projects would during the Elaboration phase develop one or more architectural prototypes that elaborate an architecture, i.e. build vertical slices through the different layers shown in Figures 1 and 2. During the test of that iteration these architecture prototypes can then be validated against the previously identified requirements. Once a stable architecture has been developed in this way, iterations in the Construction phase develop new component interfaces and implementations that provide the required functionality.

3.3 Model Driven Architecture

One of the problems with using distributed component technology for building distributed software systems is that there are so many different and incompatible component platforms available. Once a software architect has chosen one platform it becomes very expensive and time consuming to port the developed system to a different platform. For example, components that have been developed for Enterprise Java Beans are substantially different from C# components for .NET and they would virtually need to be written completely anew. To aggravate this problem, the projected lifetime of a distributed software architecture and the components that

encapsulate the business logic is often significantly longer than the life of a component platform itself.

The Object Management Group has developed a set of specifications that are referred to as *Model Driven Architecture* (MDA) to address these problems [48]. The basic idea of MDA is to use the UML for fully specifying both the static interfaces and the dynamic behaviour of components in a platform independent model (PIM). To define platform specific models (PSMs), the MDA specifications define a number of mappings from plain UML to UML profiles, which are platform-specific extensions of the UML. The OMG has adopted a number of these UML profiles, such as a UML profile for CORBA and a UML profile for Enterprise Application Integration and is working on further profiles, such as a UML profile for WebServices and .NET. Further UML profiles have been developed outside the OMG, for example the UML profile for EJB [11] that was developed under the Java Community Process charter by Rational and Sun.

The definition of profiles is enabled by UML's extension mechanisms. In particular, UML profiles make extensive use of UML stereotypes and tagged values. The EJB profile, for example defines a stereotype `<<EJBHomeInterface>>` that indicates that a UML class models an interface between a container and an EJB class. The profiles also list a number of consistency constraints that extend the consistency constraints defined in the UML semantics guides. For example, the UML profile for EJB demands that the type of any method parameter of an EJB home or EJB remote interface can only be atomic (e.g. boolean, int etc), references to remote objects or references to classes that are serializable.

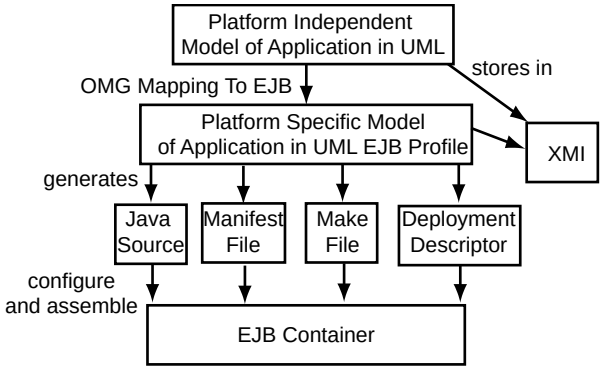

Figure 4: Model Driven Architecture for EJB

Figure 4 shows the artefacts and their dependencies involved in using the model driven architecture approach for developing distributed components for the Enterprise Java Beans platform. The business logic of components is specified in a PIM by a standard UML case tool that stores the model in XMI format [40], an XML encoding [2] for the Meta Object Facility [38]. A mapping tool that implements the OMG standard mapping of UML to EJB then generates a PSM that uses UML extensions defined in the UML profile for EJB. A CASE tool can then translate the PSM into the different components that are necessary for an EJB deployment of the model, such as the Java code, a makefile that is needed compiling the code, a Manifest file that defines the content of the binary archive that is executed by an EJB container and a deployment descriptor that defines on which machines the components need to be deployed.

Even though the Model Driven Architecture specification is relatively young, there are various implementations available. For example, Caboom of CalKey Technologies [23] supports the full specification of components in UML. Caboom provides the usual diagram types of the UML to define the static interfaces of components and uses Activity diagrams for the visual specification of the behaviour of methods. Caboom can then generate EJB, COM and/or .NET components and subsequent deployment information from these UML models. The fact that all component platform specific code is generated by model driven architecture environments, such as Caboom, not only facilitates the portability across platforms but also accelerates the development of the components in the first place as the designers can focus on the graphical specification of components rather than their implementation in a particular component platform. The first projects that have been completed with these tools report productivity increases of about 35-70%.

4. COMPONENT-ORIENTED DEVELOPMENT PROCESSES: STATE OF THE ART

While significant advances in building reliable, scalable and secure systems have been made by using distributed component technologies there is still a long way to go until we can derive a component-based distributed software system for a given a set of requirements in a systematic manner. Most importantly, engineers need to be able to reason about the appropriateness of models of distributed components without having to go through as many development and deployment iterations. Moreover, we need to be able to understand the extent to which component-oriented architectures can be adjusted without having to modify the components, or even worse having to change the platform that executes them.

In this section we provide an overview of some recent software engineering research results that will advance component-oriented development processes. We first assess techniques that support reasoning about static and dynamic properties of models for distributed components.

4.1 Reasoning about Component-based Architectures

When classifying the different techniques that support reasoning about distributed component models, we distinguish static and dynamic techniques. Static reasoning techniques assess the internal static semantic correctness of models as well as their relationship to other artefacts produced during the software development process. Dynamic techniques support reasoning about the behavioural correctness of models.

Static Consistency Definition and Checks: Above we outlined the use of the UML for modelling distributed components in the OMG's MDA. MDA draws on a number of UML profiles. For each of these profiles, the static semantics are defined informally, which is clearly undesirable as these informal descriptions are prone to incompleteness and ambiguity. What is needed is a more rigorous and formal definition of static consistency. Moreover, designers will need

$$\forall_{a \in //Foundation.Core.Association[@xmi.id]} :$$
$$\forall_{x \in a/Foundation.Core.Association.connection/Foundation.AssociationEnd} :$$
$$\forall_{y \in a/Foundation.Core.Association.connection/Foundation.AssociationEnd} :$$
$$x/Foundation.Core.ModelElement.name/text() = y/Foundation.Core.ModelElement.name/text() \Rightarrow x = y$$

Figure 5: xlinkit Rule for UML Core

to be able to assess how conformant their platform specific-models are with a particular profile. This demands the provision of consistency checks of PSMs against UML profiles. To date, hardly any CASE tool used in industry provides such checks. Finally, it is not uncommon that distributed component models are developed in a distributed manner themselves. This means that the consistency checking mechanisms are required to work in a distributed manner, which again most CASE tools do not support.

Motivated by the above requirements, a number of groups have worked on specifying static semantic and inter-document consistency constraints. The literature on software development environments includes many examples of formalisms to specify static semantic constraints and how to support them with development tools. More recently, the OMG has defined an Object Constraint Language [53] that is used to express the static constraints in the UML core meta model.

In a response to a request for information that the OMG has issued to start the OCL revision, Vaziri and Jackson outlined the limitations of using OCL for defining UML and UML profile static semantic constraints [51]. In particular, they noted that OCL is very implementation orientated while a more easily comprehensible constraint definition would need to be of a declarative nature. Jackson also provided a constraint definition for the UML core in Alloy [20] that can then be checked with the Alloy Constraint analyzer.

In our own work, we have developed a formal specification language for static consistency constraints and a standards-compliant distributed execution engine for this language. In [35] we describe xlinkit, a consistency specification language based on first order logic and an associated checking service that executes the language. The operation of xlinkit is quite simple. It is given a set of distributed XML resources and a set of potentially distributed rules that relate the content of those resources. The rules express consistency constraints across the resource types. xlinkit returns a set of XLinks, in the form of a linkbase, that support navigation between elements of the XML resources. We have defined a denotational semantics of our consistency specification language in [35]. Unlike standard first order logic, xlinkit computes hyperlinks between inconsistent elements instead of boolean values.

xlinkit leverages standard Internet technologies. It supports document distribution and can support multiple deployment models. It has a formal foundation and evaluation has shown that it scales, both in terms of the size of documents and in the number of rules. xlinkit has a more flexible architecture than the Alloy model checker. Due to its reliance on XML and Web technologies xlinkit supports checks between distributed artefacts.

xlinkit is a very general technique that is amenable to checking any kind of semi-structured and distributed markup. In [36], we describe an application of xlinkit to UML. We have specified the UML meta model constraints in xlinkit's rule language. We have then tested the performance of xlinkit against industrial UML models that were represented in the XML encoding defined in the XMI standard [40]. The performance measurements revealed satisfactory performance results with most rules executing within a matter of seconds even on files of several megabytes.

Figure 5 shows an example xlinkit rule that formalizes a UML core semantic constraint. For reasons of readability, we show it in the standard mathematical syntax rather than the XML encoding that the xlinkit rule engine understands. The rule specifies that the role names that can be attached to association ends in UML models need to be unique. To achieve that, we allow the use of XPath [5] expressions in our formulae. The vocabulary used in the path expressions shown in Figure 5 is determined by the XMI document type definition defined by the OMG, which in turn was derived from the UML meta model. The first quantifier of the rule uses an XPath expression to select all associations included in a UML model. Then for all pairs of association ends within any of these associations we check that equality of the names implies identity of the association ends.

In [37] we report an application of xlinkit that checks compliance of distributed UML models against the UML profile for EJB. The paper also describes how consistency of distributed UML models can be checked against Java source code and deployment descriptors. Thus, xlinkit can be used to specify static consistency constraints among distributed component models, component implementations and their deployment descriptors.

Qualitative Dynamic Properties: The formal specification of static semantic constraints and the ability to check models against them assists designers of distributed components to check that their UML models are meaningful. This does not necessarily mean that models are correct because static constraints cannot be used to validate any behavioural properties of a model.

Due to their concurrent and parallel nature, the need to check behavioural concerns is more pronounced in distributed component models than for models of sequential systems that execute on just one host. Systems designed using distributed components might deadlock and should satisfy safety and liveness properties.

Two different methods can be used for the automated verification of models against behavioural properties: theorem proving and model checking. Theorem proving techniques can cope with possibly infinite state space models, but their use generally requires manual interaction and guid-

ance. Model checkers do not need such interaction but cannot cope with infinite state spaces. By choosing the right abstractions for models of distributed component systems a finite, albeit large state space can be achieved.

A number of significant advances of model checkers, such as Spin [17] and LTSA [28], were made during the 1990s. Amongst others, the use of Binary Decision Diagrams and compositional reachability analysis [3], together with significantly more powerful hardware have made it possible to check models with very large state spaces. These advances have enabled a strand of applied software engineering research that uses model checking techniques for models of distributed components with the aim to check them against behavioural properties. A common thrust among several of these approaches is the use of the UML and a mapping between the UML meta model and the languages that model checkers use as an input. Cheung [31] and Lilius [27] translate behavioural UML specifications expressed in state diagrams into Promela, the input language of Spin, with an aim to provide a precise semantics for these diagrams. Inverardi and Muccini [18] also derive Promela specifications from UML state charts. They then translate sequence diagrams that specify certain scenarios into linear temporal logic formulae in order to check whether the specified scenarios are behaviourally consistent with the component specifications.

In our own work [24, 25], we focus on model checking safety and liveness properties that are induced by the synchronization primitives and multi-threaded execution capabilities provided by distributed component technologies, such as the CCM or EJB. These component technologies only support a fixed number of such synchronization properties, such as oneway, synchronous, deferred synchronous and asynchronous execution in case of the CCM. Moreover, the object adapters that control component execution in containers provide only a small number of multi-threading policies. These policies control how concurrent invocations are handled by the component and again there is only a few of these policies, such as single threaded execution or use of a thread pool to handle requests. We therefore extend platform specific UML profiles with stereotypes to express the use of synchronization primitives in state diagrams and the use of threading policies in a component class diagram. Moreover, we use object diagrams to define the deployment of components across distributed hosts. From these stereotyped diagrams we then generate a process algebra representation of the synchronization and threading behaviour for which a Labelled Transition System can be automatically computed. We then use reachability analysis to check for absence of deadlocks, safety and liveness properties. If the reachability analysis finds any deadlocks, safety or liveness property violations it produces a trace that leads from the initial state to the state that violates that property. We then abstract that trace into a UML sequence diagram that shows a scenario for that property violation and is more meaningful to the designer of the distributed component system than a trace of labelled transitions.

There are a number of further approaches that use model checking techniques in conjunction with distributed component technologies without using the UML. In [18] Inverardi et al. propose a way of constructing distributed architectures from components in a deadlock-free manner. They use Milner's CCS [33] to prove that the architectures constructed are actually deadlock free. They report in [19] about an application of this technique to DCOM applications.

4.2 Component Deployment

Component technology providers offer a great number of tools that can be used to manage and configure the deployment, but component deployment gained attention among software engineering researchers only relatively recently.

Hall et al. identify the activities that are commonly associated with software deployment in [13]:

- Package all artefacts and configuration descriptions needed to install a system (Release)
- Configure and assemble all artefacts needed to use a released system (Install)
- Bring an installed system into a state that it can be used (Activate)
- Deallocate all resources of an activated system and put it into a state that it cannot be used (Deactivate)
- Modify an installed/activated release by selecting a different configuration (Reconfigure)
- Modify an installed/activated release to adjust it to changes in the operating environment (Adapt)
- Modify an installed/activated release by installing/activating a previously unavailable configuration (Update)
- Remove the installed artefacts of a release from its operating environment (Remove)
- Make a release unavailable (Retire)

Rutherford et al. put these activities into the perspective of deploying distributed EJB-based components in [45]. They then present the Bean Automated Reconfiguration frameworK (BARK), which supports some of the deployment activities. The framework defines a high-level scripting language that can be used for describing deployment scripts that can then be executed on a number of distributed hosts in order to perform installation, activation, deactivation and reconfiguration actions. The tool that executes these scripts provides different kinds of reliability guarantees for the execution of these reconfiguration scripts. The tool supports, in particular an atomic execution of scripts on a number of distributed hosts by means of transactions that are implemented using the two-phase commit protocol.

5. LOOKING AHEAD

In this section, we briefly discuss a number of research questions that may form part of a broader research agenda for software engineering principles, methods and tools in support of the component-based development of distributed software systems.

Quantitative Reasoning: Using the UML together with model checking techniques as discussed above supports reasoning about the presence or absence of certain qualitative properties, such as deadlocks, safety or liveness properties. For distributed component architectures, however, it is also important to be able to reason about quantitative properties

that these models will have when they are deployed in a certain way into a container of an application server. It would be highly desirable to avoid costly risk mitigation iterations during a development process and address the question of whether an architecture scales and performs efficiently and reliably by analytic means. The performance modelling literature includes a large body of work on *stochastic process algebras*, which use distribution functions with which transitions are executed [16, 9, 10]. Due to their compositional nature, process algebras have been successfully used to model distributed component systems as discussed above. It seems natural to extend that research such that performance, scalability and reliability properties of UML models can be expressed and analyzed with stochastic process algebras.

Quality of Service Aware Component Deployment: Once the quantitative characteristics of the interactions between distributed components are modelled it would be appropriate to avoid loosing that information further down the development and deployment process. Instead, it would be desirable to make the containers that execute components aware of them so that they can proactively engage in meeting these characteristics. Thus, the quantitative properties expressed in an (extended) UML model about a set of components become obligations for component execution that is to be met by a container of an application server. These quantitatively enriched models could then be translated into a *service level agreement*, which governs the way how containers execute components. In particular, containers could then monitor their actual performance, reliability and scalability and compare it with the required quality of service expressed defined in the service level agreement. In case of under-performance, containers could then either proactively take steps, such as the replication of particular components in neighbouring containers, to meet the required service level or report service level exceptions to administrators.

Trusted Component Deployment: Important new requirements for component deployment arise when we consider applications that are assembled from distributed components that execute in different administrative domains. This is particularly the case for component-based application services. Firstly, the same components or containers may potentially execute applications on behalf of competing organizations. Consider a storage service provider, which operates a specialized container that is customized for mass storage of stateful components. That container may then be used by potentially competing organizations, who would expect that neither the storage service provider nor any other of its customers can access their data. Secondly, the service level agreements that we discussed above now form part of the contract between the different administrative domains that are involved in providing components for the application. Thus the monitoring of service execution now has to be performed in a trustworthy manner and has to be exchanged, potentially in a summarized form, between the administrative domains as part of any service level monitoring.

Architectural Stability: Architectural stability refers to the property of a software architecture to meet changing requirements. Addition or changes in functional requirements can be addressed in distributed component-based architectures by adding or upgrading the components in a business object layer. Current distributed component containers support hot-deployment of new and hot-swapping of existing components so that these changes can be performed without even stopping applications. Moreover, as discussed above, the deployment lifecycle is fairly well understood. Changes in non-functional requirements, however, can stress an architecture considerably and might lead to architectural break down. Such breakdowns occur if the container or application server that has been selected to execute distributed components does not provide sufficient deployment flexibility to be able to meet the changed requirements and as a result the container or application server has to be changed, which is considerably more expensive than just adjusting the component replication strategy. On the other hand, always using the most advanced container product might be prohibitively expensive for applications that might never exceed the scalability boundaries that could also be met with, for example, open source containers. Therefore in order to achieve architectural stability, it will be necessary to adjust requirements elicitation and management techniques and elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the lifetime of the architecture. These ranges of requirements then need to inform the selection of distributed component technology and subsequently the selection of application server products.

Workflow: The distributed component technologies that we discussed in this paper are all being integrated with messaging technologies that are used to exchange data asynchronously. The Java Messaging Service is now part of the Java 2 Enterprise Edition and the latest release of the EJB specification includes Messaging Beans that are capable of handling incoming and outgoing messages. The CORBA Component model natively supports asynchronous communication by way of messaging and also the component model in Microsoft's .NET integrates with Microsoft's Message Queue (MSMQ). Components that direct the flow of messages through a distributed architecture and in that way control the workflow of an organization are becoming available, too. Examples include IBM's MQSeries Workflow or Microsoft's BizTalk Server. These systems need modelling capabilities for workflows, formal semantics for the workflow modelling languages, analysis methods and tools to reason about workflow models and to monitor the compliance of actual workflows with those prescribed in a workflow model. A great deal of these problems have been addressed in the software process literature. The more mechanical and non-interactive workflows addressed using these distributed component systems seem much more amenable to be solved using the techniques that were originally developed for software processes. It would therefore seem an interesting exercise to apply some of the software process research results to these workflows.

6. SUMMARY AND CONCLUSIONS

In this paper, we have provided an overview of the distributed component technologies that are available to date. We have addressed the state of the practice in using these component technologies. In particular, we have shown several examples of architectural styles that use these technologies to deliver scalable distributed applications and integrate them with legacy enterprise information systems. We have shown why iterative and incremental development processes

are appropriate for the development of distributed components. We have then presented the idea of model driven architecture and sketched how component technology independent development can be achieved. We have reviewed the state of the art in developing distributed components and shown how developers are able to reason about static and dynamic qualitative properties of models of distributed component. We then sketched the current state of the relatively novel field of component deployment. Finally, we have identified quantitative reasoning about component models, QoS aware deployment, trusted execution, architectural stability and workflow across distributed components as some of the items on the software engineering research agenda for distributed component systems.

The market for distributed component technology is worth several billion US$ per annum. The market for professional services that turn these technologies into solutions is probably at least as big. Thus, any software engineering research results that are transferred and successfully applied in this professional services market are likely to have a significant impact. It is encouraging to see that previous trends within the software engineering research community of ignoring current software development practice have to some extent been abandoned. Even though UML and the distributed component technologies available to date are far from perfect and leave a lot to be desired from a formal point of view they are what engineers use in practice. It is probably more important to live with these flaws and package research into a form that shows a clear route to application for practicing software engineers. The research that we have presented in this paper is firmly rooted in currently development practice and stands a fair chance of having an impact.

Acknowledgements

Together with partners in the UK, Italy and Germany, we hope to address some of these questions in an upcoming joint European Research Project on Trusted & quality of service Aware Provision of Application Services (TAPAS) [47]. I would like to thank Santosh Shrivastava, Fabio Panzieri, Paul McKee and Werner Beckmann for the fruitful discussion about distributed component research. Anthony Finkelstein provided valuable comments on an earlier draft of this paper.

7. REFERENCES

[1] D. Box. *Essential COM*. Addison Wesley, 1998.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[3] S.-C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, 1999.

[4] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shin, C.-Y. Wang, and Y.-M. Wang. DCOM and CORBA: Side by Side, Step by Step, and Layer by Layer. *C++ Report*, pages 18–29, January 1998.

[5] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, World Wide Web Consortium, November 1999.

[6] M. Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley, 1992.

[7] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.

[8] W. Emmerich, E. Ellmer, and H. Fieglein. TIGRA – An Architectural Style for Enterprise Application Integration. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering, Toronto, Canada*, pages 567–576. IEEE Computer Society Press, 2001.

[9] S. Gilmore and J. A. Hillston. The PEPA Workbench: A Tool to support a Process Algebra-based Approach to Performance Modelling. In *Proc. of the* 7^{th} *Int. Conference on Modelling Techniques and Tools for Performance Evaluation*, volume 794 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 1994.

[10] N. Götz, U. Herzog, and M. Rettelbach. The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras. In *Proc. of the* 16^{th} *Int. Symposium on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE 93)*, volume 729, pages 121–146. Springer, 1993.

[11] J. Greenfield. *UML Profile For EJB*. Rational Software Corp., May 2001.

[12] R. Grimes. *DCOM Programming*. Wrox, 1997.

[13] R. S. Hall, D. M. Heimbigner, A. v. Hoek, and A. L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proc. of the 1997 Int. Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society Press, 1997.

[14] M. Hapner, R. Burridge, and R. Sharma. Java Message Service Specification. Technical report, Sun Microsystems, http://java.sun.com/products/jms, November 1999.

[15] G. T. Heineman and W. T. Councill, editors. *Component Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

[16] J. A. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Dept. of Computer Science, University of Edinburgh, UK, 1994.

[17] G. J. Holzman. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[18] P. Inverardi, H. Muccini, and P. Pelliccione. Checking Architectural Models Consistency using SPIN. In *Proc. of the* 16^{th} *Automated Software Engineering Conference, Coronado Island, CA*, pages 346–349. IEEE Computer Society Press, 2001.

[19] P. Inverardi and M. Tivoli. Automatic Synthesis of Deadlock free connectors for COM/DCOM Applications. In V. Gruhn, editor, *Joint Proc. of the* 8^{th} *European Software Engineering Conference and the* 9^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, Austria*, pages 121–131. ACM Press, 2001.

[20] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy Constraint Analyzer. In *Proc. of the* 22^{nd} *Int. Conf. on Software Engineering, Limerick Ireland*, pages 730–733. ACM Press, 2000.

[21] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, Reading, MA, USA, 1999.

[22] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.

[23] B. Jaddav. Caboom White Paper. Technical report, CalKey Technologies, Campbell, CA 95008, 2001.

[24] N. Kaveh and W. Emmerich. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Joint Proc. of the* 8^{th} *European Software Engineering Conference and the* 9^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, Austria*, pages 44–51. ACM Press, 2001.

[25] N. Kaveh and W. Emmerich. Safety and Liveness Analysis of Distributed Object Systems. Technical Report RN/02/02, UCL-CS, 2002.

[26] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley Longman, 2000.

[27] J. Lilius and I. Paltor. A Tool for verifying UML models. In *Proc. of the* 14^{th} *Int. Conference on Automated Software Engineering, Cocoa Beach, Florida*, pages 255–258. IEEE Computer Society Press, 1999.

[28] J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999.

[29] R. Marvie and P. Merle. CORBA Component Model: Discussion and Use with OpenCCM. Technical report, Laboratoire d'Informatique Fondamentale de Lille, Villeneuve d'Ascq, France, 2001.

[30] D. McIllroy. Mass produced software components. In P. Naur and B. Randall, editors, *Software Engineering: Report on a Conference by the NATO Science Committee*, pages 138–155, Brussels, 1968. NATO Scientific Affairs Division.

[31] W. E. McUmber and B. H. C. Cheung. A General Framework for Formalizing UML with Formal Languages. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering, Toronto, Canada*, pages 433–442. IEEE Computer Society Press, 2001.

[32] H. D. Mills. Top-Down Programming in Large Systems. In R. Ruskin, editor, *Debugging Techniques in Large Systems*. Prentice Hall, 1971.

[33] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1995.

[34] R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly UK, 1999.

[35] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2002. To appear.

[36] C. Nentwich, W. Emmerich, and A. Finkelstein. Static Consistency Checking for Distributed Specifications. In *Proc. of the* 16^{th} *Automated Software Engineering Conference, Coronado Island, CA*, pages 115–124. IEEE Computer Society, 2001.

[37] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible Consistency Checking. Research Note RN/01/40, UCL Department of Computer Science, 2001.

[38] Object Management Group. *The Meta Object Facility*. 492 Old Connecticut Path, Framingham, MA 01701, USA, 1997.

[39] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*. 492 Old Connecticut Path, Framingham, MA 01701, USA, February 1998.

[40] Object Management Group. *XML Meta Data Interchange (XMI) – Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF)*. 492 Old Connecticut Path, Framingham, MA 01701, USA, October 1998.

[41] Object Management Group. *CORBA Components – Volume I*. 492 Old Connecticut Path, Framingham, MA 01701, USA, December 1999.

[42] F. Pilhofer. Writing and Using CORBA Components. Technical report, FPX, www.fpx.de/MicoCCM/download/mico-ccm.pdf, 2001.

[43] F. Plásil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software – Concepts and Tools*, 19(1):14–28, 1998.

[44] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Reading, MA, USA, 1999.

[45] M. J. Rutherford, K. Anderson, A. Carzaniga, D. M. Heimbigner, and A. L. Wolf. Reconfiguration in the Enterprise Java Beans Component Model. In J. Bishop, editor, *Proc. of the* 1^{st} *IFIP/ACM Working Conference on Component Deployment, Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2002. To appear.

[46] R. Sharma. *Java 2 Enterprise Edition: J2EE Connector Architecture Specification*. Java Community Process, JSR 016, Sun Microsystems, Palo Alto, CA, 2001.

[47] S. Shrivastava, W. Emmerich, F. Panzieri, V. Gruhn, and J. Crowcroft. Trusted and QoS Aware Provision of Application Services (TAPAS). Technical Annex to Project Contract 34069, Commission of the European Union, 2002.

[48] J. Siegel. Developing in OMG's Model Driven Architecture. Technical Report 01-12-01, Object Management Group, Framingham, Mass, November 2001.

[49] A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett Packard, Palo Alto, Cal, October 1995.

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

[51] M. Vaziri and D. Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. Response to Object Management Group's Request for Information on UML 2.0, MIT, December 1999.

[52] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. Addison Wesley, 2001.

[53] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison Wesley, 1999.