

Components – the Past, the Present, and the Future

Jean-Guy Schneider and Jun Han

Swinburne University of Technology
School of Information Technology
P.O. Box 218, Hawthorn, Victoria 3122, Australia
{jschneider, jhan}@it.swin.edu.au

Abstract

Since the early 1990's, component-based software technology has become an increasingly popular approach to facilitate the development of evolving systems as it promised to address some of the problems of object-oriented development technologies. By reconfiguring components, adapting existing components, or introducing new components it was hoped that applications could be adapted to changing requirements more easily than using traditional approaches. But has component-based software technology succeeded? Have we been able to address the problems identified more than a decade ago? Which problems still need further investigations? In this work, we will review some of the goals component-based development was supposed to achieve, investigate whether these goals have been met, and propose a research agenda of topics for further investigation.

1 Introduction

In the early 1990's, it became apparent to both researchers and practitioners that object-oriented technologies were not enough to cope with the rapidly changing requirements of real-world software systems. One of the reasons was that, although object-oriented methodologies encourage one to develop rich models that reflect the objects of the problem domain, this did not necessarily yield software architectures that could be easily adapted to changing requirements. In particular, object-oriented methods did not typically lead to designs that made a clear separation between computational and compositional aspects. In 1994, Udell even claimed that “Object-orientation has failed” [30]. So what could the community do to address this problem and at the same time further advance the state-of-the-art in software development?

At the time, component-based software technology was becoming an increasingly popular approach to facilitate the development of evolving systems [17]. The objective of this technology was to take elements from a collection of reusable software components (i.e. *components-off-the-shelf*) and build applications by simply plugging them together. Hence, it aimed at the production of high-quality software systems with shorter and more cost-effective development cycles. By reconfiguring components, adapting existing components, or introducing new components it was hoped that applications could be adapted to changing requirements of real-world software systems more easily and address the problems of object-oriented development approaches [20].

A decade later, we can observe that considerable efforts have been spent in defining component models such as CORBA [22], COM [26], JavaBeans [28], and more recently Enterprise

JavaBeans [15], the CORBA Component Model [23] and .NET [13]. Furthermore, a number of component-based development environments have been made available to software developers (Delphi, Visual Studio, Visual Age and Eclipse to name a few), components became a popular topic at both industrial and academic conferences, dedicated books about component-based software development were written (e.g., [1, 4, 8, 27, 29]), and many success stories of projects using component technology were published. But has component technology fulfilled the promises it made a decade ago and have the problems identified been addressed? What remains to be done and where should we focus our efforts to further advance component technologies?

The goal of this paper is to reflect on these questions, reiterate on some of the issues stated a decade ago and propose topics for further investigation. Due to space limitations, this should not be considered as a full review of the state-of-the-art, but rather as a starting point to initiate thoughts and discussions about the next decade of component technology.

This paper is organized as follows: in Section 2, we will revisit some of the problems of object-oriented technology identified a decade ago and list some of the questions component technologies were supposed to address. In section 3, we will discuss whether these goals have been met by current state-of-the-art component technologies and highlight some of their deficiencies. We will outline further areas of research in Section 4 and wrap up the paper with some concluding remarks in Section 5.

2 The Past

The origins of component-based software technology go back to McIlroy who proposed viewing components as families of routines which are based on well-defined principles so that these families fit together as building blocks [10]. One of the main motivations behind component technology was *reusability*: components avoid reinventing the wheel. If we had a collection of reusable software components, we could build applications by simply plugging existing components together. However, reusability was not the only motivation behind component technology: independent evolution of application parts, enhanced flexibility, adaptability, and maintainability of software systems, higher-level of application development than “classical” programming paradigms etc. were also on the agenda of researchers and practitioners. But how could these goals be achieved?

In theory, object-oriented programming languages and analysis and design methods would provide a well-suited tool-box for component-based application development, but practice showed that the technology was often applied in a way that contradicted the goals behind component technology, as briefly discussed in the following.

Object-oriented analysis and design methods are domain-driven, which usually leads to designs based on domain objects. Most of these methods make the assumption that an application is being built from scratch, and they incorporate reuse of existing architectures and components too late in the development process (if at all) [24].

In order to successfully plug components together, it is necessary that (i) the interface of each component matches the expectations of the other components and (ii) the “contracts” between the components are well-defined. Therefore, component-based application development depends on adherence to restricted, plug-compatible interfaces and standard interaction protocols. However, the result of an object-oriented analysis and design method generally is a design with rich object interfaces and non-standard interaction protocols.

Object-oriented programming languages have been very successful for implementing and packaging components, but they only offer limited support for flexibly connecting components and explicitly representing architectures in applications. Given the source code of an object-oriented application, one can more easily identify the components, but it can be notoriously difficult to tell how the system is composed. The reason is that object-oriented source code exposes *class hierarchies*, not *object interactions*. In addition, the way objects are interconnected is typically distributed amongst the objects themselves, which hinders a clean separation between computational and compositional aspects needed for component-based development.

Although object-oriented applications can often be adapted to additional requirements with a minimal amount of new code, it can require a great deal of detailed study in order to find out where exactly the extension is needed. Unfortunately, object-oriented frameworks do not make their generic architecture explicit, which results in a steep learning curve before a framework can be successfully used. Since object-oriented frameworks focus on subclassing of framework classes (also known as *white-box reuse*), a detailed understanding of the generic architecture is needed in order to prevent contracts between two classes from being violated. In addition, changing framework classes often implies extensive modifications of application-specific code.

Based on these observations, various research agendas were put forward to address the problems of object technology (e.g., [17, 21]). Briefly summarizing these topics, we can come up with the following list of questions and/or challenges that were formulated in this context:

Functionality: What is a software component and how does it differ from an object? How can we characterize software components? What kinds of abstractions, languages and notations do we need to specify both functional and non-functional properties of components? Are components general-purpose or domain specific?

Interaction: What are suitable ways of expressing compositions of components? What kinds of languages do we need for composition? Do we need implicit or explicit, declarative or imperative composition? Are connectors separate entities or “just” special kinds of components? How does composition help in application evolution? How can we ensure that a composition of components is correct with respect to a given set of requirements, in particular the respective interaction protocols?

Quality: How can we specify (non-functional) quality attributes of components and verify that these attributes are met by a given implementation? Can we reason about the quality attributes of a composition of components given the quality attributes of the components involved (i.e. can we perform *compositional reasoning*)? What kind of formalisms are needed for compositional reasoning?

Management: How can we facilitate the design and implementation of component frameworks? How can we evolve these frameworks? What kind of ontologies do we need to create suitable component repositories? How can we effectively search for components given a set of requirements?

Evolution and Tools: What kind of software tools, development environments etc. do we need to facilitate component-based software development? How can we manage framework development and component repositories to provide the best possible support for both component engineers and application developers?

Methodology: What is the impact of component- and reuse-based development on software engineering methodologies? What kind of development processes and methods do we need to facilitate component-based software engineering? What are the impacts of CBSE from a business perspective?

This list is by no means exhaustive, but characterizes the most prominent questions both researchers and practitioners considered being important. But has the community been able to address these issues?

3 The Present

Over the past decade, considerable efforts have been spent both by academia and industry to advance the state-of-the-art of component technology and address some of the questions we raised in the previous section. In the following, we will briefly summarize some of these efforts.

From an industrial perspective, various component models were defined and the corresponding infrastructure implemented, the most prominent ones being CORBA [22], COM [26], JavaBeans [28], and more recently Enterprise JavaBeans [15], the CORBA Component Model [23], and .NET [13]. Whereas these technologies are still around today, other technologies that were defined did not survive and are now considered being obsolete (e.g., OpenDoc and SOM [5]). Common to all the approaches mentioned above is that they define standards for certain application domains (e.g., distributed client-server systems) and, therefore, address questions in the area of functionality and interoperability. They all define very specific interface standards so that components developed from different vendors can interoperate. Furthermore, considerable efforts have been spent in implementing specialized component frameworks for very specific application domains (e.g., financial services, telecommunication, real-time systems) on top of these component models.

Besides defining the corresponding models, industry has also developed various software tools (e.g., Delphi, Visual Studio, Visual Age and Eclipse), all of which evolve around a particular object-oriented programming language. These languages all offer some reasonable support for component-based programming (e.g., encapsulation of state and behavior, late binding, scalability, visual composition, collection of reusable components, mechanisms for interoperability with other environments), but the underlying semantic models are not powerful enough to provide flexible and typesafe component composition and evolution mechanisms. Moreover, we cannot only identify a lack of support for verifying the correctness of compositions (i.e. correctness is at best ensured based on some form of interface-compatibility), but also a lack of abstractions for building and adapting components in a framework or domain specific way, for defining higher-level *cooperation patterns*, and for making software architectures explicit in source code.

One of the key problems practitioners noticed in the area of a reuse-based development approach was the “not-developed-here” syndrome: how could one trust a third-party software component that was developed by somebody else? Shipping the source code along with the (binary) component was not really a practical solution as this would violate the principal idea of component-based software development and, even more importantly, any changes made to the source code may invalidate the component’s contractual specification. To address this issue, an initiative called *trusted components* was launched by both practitioners and researchers [11]. The main idea of this initiative is to (i) define criteria against which to assess components (i.e. a so-called *Component Quality Model*) leading to the qualification of existing components and (ii)

enable the production of components with fully proved *correctness properties* by creating appropriate frameworks for specifying these properties and tools for automatic verification [11]. Despite promising results, the approach is still relatively new and further efforts are required.

In the early stages, adopters of component technology have thought of software the same way engineers have thought about bridges, buildings, or hardware components: static entities that change little over time. However, experience over the past decade has shown that this view leads to serious problems when it is applied in a component world, not only from a technological, but also from a financial perspective. More and more leading software experts claim that progress in this area can only be made if we abandon this “static” perception and consider software to be a constantly evolving entity [18]. This is even a greater challenge in the area of ubiquitous computing where collaborating software agents have to continuously adapt to changing and evolving environments [2]. Hence, we argue that only few of the questions in relation to methodology have been addressed, but at least we have a fair idea now what does *not* work.

In recent time, *web-services* got a considerable amount of attention as it was thought that this new technology could solve some of the problems the other component standards have not been able to address (c.f. a number of articles published in the Communications of the ACM, October 2003). Most notably, web-services attempt to facilitate interoperability by using (i) XML as an interface description language and (ii) standard Internet protocols for data exchange. This makes web-services an ideal platform for development using Internet browsers and self-describing documents. But are web-services really that new of a concept? In [6], Gokhale et al. make an attempt to compare web-services with CORBA and come to the conclusion that conceptually web-services do not differ from CORBA (and as such from most of the component models introduced at the beginning of this section) as both technologies attempt to solve similar kinds of problems. The most significant differences are that (i) CORBA components have “observable” state whereas web-services generally do not and (ii) CORBA uses a signature-based IDL to specify interfaces whereas web-services interfaces are specified using XML. Hence, it seems that the community tries to reinvent the wheel once again, and before doing so in the future, we should probably reflect more on the problems we have solved and how well these solutions worked.

4 The Future

As discussed in the previous section, component technology has made considerable progress in the last decade, but there are still open questions that need further work. In the following, we will highlight one open question for each of the topic areas introduced in Section 2 we consider being important to be addressed in the near future.

Functionality: Traditionally, an interface specifies the provided and required services of a component in terms of *signatures*. These signatures are described in terms of dedicated language constructs (Java in the case of EJB’s) or using a (language-neutral) interface description language such as the IDL’s of COM and CORBA or WSDL (i.e. XML) for web-services. Despite some approaches of extending the notion of a component interface going beyond a collection of service signatures [7, 14, 25], we are still a long way away of being able to extract enough information from a component’s interface to know (i) how the services of a component have to be used and (ii) how they can be correctly combined with services of other components. In essence, we argue that the interface of a component should describe the services it offers (some form of “IDL”) plus an

abstraction of the semantics of its implementation and usage. It is however not clear yet how this could be achieved with current interface specification techniques.

Interaction: Related to the lack of suitable interface specifications is the notion of *correctness*: when do we consider a composition of components to be correct? The correctness of a composition can be expressed both at a *syntactic* and a *semantic* level: the former is generally associated with a type system whereas the latter is associated with *behavioural* specifications. Over the past years, considerable progress in the area of type systems has been made, but verifying the correctness of compositions of components (and along the same line *substitutability*) is still an area where further efforts are required. Interesting approaches based on concepts such as *regular types* [16], *collaboration types* [12], or *contractual types* [19] have been proposed, but there are suggestions the resulting solutions will need to (i) combine both a static and dynamic approach to checking the correctness of a composition and (ii) component environments have to be taken into consideration.

Quality: Being able to predict and/or reason about specific quality attributes of a composition of components is an area that has attracted considerable attention in the last years and is of particular importance in the areas of performance and security. For example, being able to predict the performance of a system is beneficial for reasons such as bottleneck identification and performance tuning whereas ensuring a required level of security impacts which components can be chosen in security-critical systems [9]. Furthermore, compositional reasoning should also allow us to specify particular quality attributes a component has to fulfill given a predefined composition style and composition environment, respectively. However, state-of-the-art in this area is still relatively immature and further investigations are required.

Management: In an ideal world, there are components available for any task an application has to perform and these components can be simply plugged together. But how do we find a suitable component given a set of requirements? Despite the efforts of the community a *component market* [30] has not really emerged and finding suitable components is still more an ad-hoc procedure than a systematic approach. Admittedly, component registers have been set up where interested parties can register their components (the UDDI for web services being the most recent one). These registries generally work on service signatures specifications and/or some form of textual description given by the developer, and any queries have to be made at this level of abstraction. However, it is generally not possible to search for components given a description in the *problem-domain*. We could think of various approaches to further the state of component repositories (standardization efforts, suitable ontologies, meta-level descriptions etc.), but there is still a lot of undiscovered land to investigate.

Evolution and Tools: As discussed above, verifying the correctness of a composed system at a syntactic level (i.e., at the level of type systems) is not enough and that behavioural specifications need to be considered. However, any kind of behavioural specification that cannot be directly expressed in source code may lead to double maintenance problems (i.e., updating both source code and the corresponding behavioural specification). Hence, there is a definite need for tool support in this area so that specification and verification can be seamlessly integrated into the development process. In fact, we would argue that any kind of formalisms in this area have to be integrated into tools in such a way that they are only present “backstage” and visible to application developers as little as possible. As the state-of-the-art is not very advanced, yet, this seems to be a promising area for further exploration.

Methodology: This is probably the area where the least progress was made. To our knowledge, there still do not exist any development methodologies specifically tailored to component-based development. There are few methodologies that take reuse into consideration at an early stage of development (the OOram Software Engineering Method [24] being the most prominent one), and so-called *agile development methodologies* [3] explicitly avoid reuse-based approaches (at least up to a certain degree). Szyperski even claims that “all facets of software engineering and the entire underlying approach need to be rethought” [29]. Hence, there is still a lot of work to do to come up with a more systematic approach for component-based software engineering.

5 Conclusions

In this work, we have revisited some of the problems of object-oriented technologies that motivated a paradigm shift towards component-based software development. We have also reviewed some of the goals component-based development was supposed to achieve and gave a brief overview of the current state-of-the-art in component technology. Based on this overview, we came up with a list of questions that current component technologies do not fully address (or not address at all) and proposed a list of topics for further discussion and investigation. As we have outlined, there is an emerging need to clarify both technical (i.e. specification of components; component composition; verification of compositions; quality of components) and even more so methodological issues (i.e. when and how to apply the technology) of component software. We also briefly outlined in the context of web-services that we still tend to reinvent the wheel instead of reflecting on what problems have been solved and how well these solutions worked. Hence, a more systematic approach to analyze the state-of-the-art of component technology and where we should spend our efforts in the future might be necessary, if not essential.

Most importantly, however, despite more than a decade of research efforts, we would argue that we still have not gotten the fundamentals of component-based Software Engineering right: as long as we cannot give a well-defined and generally-accepted answer to the questions “*what is a software component?*” and “*how do I correctly compose software components?*” then we have little hope of reaching a level of maturity of the discipline that is acceptable for all stakeholders.

References

- [1] Uwe Assmann. *Invasive Software Composition*. Springer, 2003.
- [2] Guruduth Banavar and Abraham Bernstein. Software Infrastructure and Design Challenges for Ubiquitous Computing Applications. *Communications of the ACM*, 45(12):92–96, December 2002.
- [3] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2001.
- [4] Desmond F. D’Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1999.
- [5] Jesse Feiler and Anthony Meadow. *Essential OpenDoc*. Addison-Wesley, 1996.
- [6] Bharat Gokhale, Aniruddha dn Kumar and Arnaud Sahuguet. Reinventing the Wheel? CORBA vs. Web Services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW 2002)*, Honolulu, Hawaii, May 2002. Alternate Papers Tracks.
- [7] Jun Han. A Comprehensive Interface Definition Framework for Software Components. In *Proceedings of the 5th Asia-Pacific Software Engineering Conference (APSEC ’98)*, pages 110–177, Taipei, Taiwan, December 1998. IEEE Computer Society Press.

- [8] George T. Heineman and William T. Councill, editors. *Component-based Software Engineering*. Addison-Wesley, 2001.
- [9] Khaled Khan and Jun Han. Composing Security-Aware Software. *IEEE Software*, 19(1):34–41, January 2002.
- [10] M.D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*. NATO Science Committee, January 1969.
- [11] Bertrand Meyer. The Grand Challenge of Trusted Components. In *Proceedings ICSE 2003*, pages 660–667, Portland, Oregon, May 2003. IEEE Computer Society Press.
- [12] Mira Mezini and Klaus Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proceedings OOPSLA 2002*, volume 37 of *ACM SIGPLAN Notices*, pages 52–67. ACM Press, November 2002.
- [13] An Introduction to Microsoft .NET. White Paper, Microsoft Corporation, 2001.
- [14] Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. Behavioral Substitutability in Component Frameworks: A Formal Approach. In Mike Barnett, Steve Edwards, Dimitra Giannakopoulou, and Gary T. Leavens, editors, *Proceedings of ESEC '03 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '03)*, pages 22–28, Helsinki, Finland, September 2003.
- [15] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, Second edition, 2000.
- [16] Oscar Nierstrasz. Regular Types for Active Objects. In *Proceedings OOPSLA '93*, volume 28 of *ACM SIGPLAN Notices*, pages 1–15, September 1993.
- [17] Oscar Nierstrasz. Research Topics in Software Composition. In *Proceedings of Langages et Modèles à Objets*, pages 193–204, October 1995.
- [18] Oscar Nierstrasz. Software Evolution as the Key to Productivity. In *Proceedings of Radical Innovations of Software and Systems Engineering in the Future*, Venice, Italy, October 2002.
- [19] Oscar Nierstrasz. Contractual Types. Technical Report IAM-03-004, University of Bern, Institute of Computer Science and Applied Mathematics, August 2003.
- [20] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.
- [21] Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.
- [22] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.
- [23] Object Management Group. *CORBA Components, Version 3.0*, June 2002.
- [24] Trygve Reenskaug. *Working with Objects: the OOram Software Engineering Method*. Manning Publications, 1996.
- [25] Ralf H. Reussner. Formal Foundations of Dynamic Types for Software Components. Technical report, Department of Informatics, University of Karlsruhe, Germany, August 2000.
- [26] Dale Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.
- [27] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.
- [28] Sun Microsystems. *JavaBeans Specification*, July 1997.
- [29] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, Second edition, 2002.
- [30] Jon Udell. Componentware. *Byte*, 19(5):46–56, May 1994.