A Lightweight Taxonomy to Characterize Component-Based Systems

Holger M. Kienle and Hausi A. Müller
University of Victoria, Canada
hkienle@acm.org and hausi@cs.uvic.ca

Abstract

In this paper we propose a taxonomy to characterize component-based systems. The criteria of our taxonomy have been selected as a result of constructing a number of component-based software engineering tools within the Adoption-Centric Software Engineering project at the University of Victoria. We have applied the taxonomy in our work to characterize the resulting tools and to define the design space of our project's proposed tool-building methodology. Our taxonomy strives to capture the most important properties of component-based systems, resulting in a taxonomy that is both course-grained and lightweight. We believe that it is useful for other researchers in a number of ways, for instance, for component selection and to reason about certain quality attributes of components.

1. Introduction

In this paper we propose a taxonomy to characterize component-based systems. Such taxonomies are useful in a number of ways. For example, they enable a structured comparison of components that can be used as input for component selection. Taxonomies can also identify characteristics that impact the system's requirements and design, as well as the development process. A taxonomy can help to reason about quality attributes of certain types of components and component-based systems. For example, using the criteria defined in our taxonomy one might look at testability of components and conclude that the follow kinds of distribution forms are increasingly difficult to test: white-box, glass-box, and black-box. Similarly, one might conclude that the following types of systems are increasingly difficult to maintain: single-component, homogeneous multiple-component, and heterogeneous multiple-component.

The criteria of our taxonomy have been selected as a result of constructing a number of component-based systems. These systems have been developed within the Adoption-Centric Software Engineering (ACSE) project at the University of Victoria (www.acse.cs.uvic.ca) [40] [41]. ACSE explores tool-building approaches to make software engineering tools more adoption-friendly. Its main assumption is that new tools will be adopted more effectively, if they are compatible with both existing users and existing tools. Research tools are lacking in this respect because they are often built from scratch with limited resources and expertise, resulting in idiosyncratic, stand-alone tools. ACSE proposes to increase adoptability by leveraging off-the-shelf (OTS) components and products that are popular with targeted users.

Following the ACSE methodology, about seven software engineering tools with diverse functionalities have been built over a period of roughly five years (2001–2005). Most of these tools are implemented via programmatic customization of a single OTS product. Specifically, we have implemented (graph) visualizers and editors using Microsoft PowerPoint, Excel, and Visio [67] [32] [30] as well as Scalable Vector Graphics [33]; a metrics visualization tool using Microsoft Visio [68]; reverse engineering environments using Lotus Notes [35] and Adobe GoLive [24]; and a Web site fact extractor using IBM Websphere Application Developer [31].

We have found our taxonomy to be effective in succinctly describing important properties of the components that we have selected for tool-building, and the properties of the resulting tools. Furthermore, the taxonomy can be seen as characterizing the *design space*. The design space for components and component-based systems is large, having many dimensions. The taxonomy has the purpose to focus and structure this design space. The introduced criteria (e.g., origin) are the chosen dimensions in the design space. The criteria's characteristics (e.g., internal vs. external origin) define the (discrete) values within the design space. In our project, the taxonomy has been used to define the subset of the design space that is covered by ACSE's component-based tool-building methodology [29].

Rather than striving for an all-inclusive taxonomy, our taxonomy's goal is to concentrate on the criteria that have a significant impact on tool development. Even though the taxonomy's criteria have been selected based on our own

Attribute	Possible values
user interface	command-line interface, graphical interface
data interface	textual I/O, specific file I/O, database I/O
program interface	textual composition, functional composition, modular composition, object-oriented composition, subsystem composition, object model composition, specific platform composition, open platform composition
component platform	hardware, operating system, programming system, libraries/frameworks, programming language

Table 1. Summary of Sametinger's reusable software components taxonomy [47]

Category	Attribute	Possible values
source	origin	in-house, existing external, externally developed, special version of commercial, independent commercial
	cost and property	acquisition, license, free
customization	required modification	minimal, parameterization, customization, internal revision, extensive rework
	possible modification	none or minimal, parameterization, customization, programming, source code
	interface	none, documentation, API, object-oriented interface, contract with protocol
bundle	packaging	source code, static library, dynamic library, binary component, stand-alone program
	delivered	non delivered, partly, totally
	size	small, medium, large, huge
role	functionality	horizontal, vertical
	architectural level	OS, middleware, support, core, user interface

Table 2. Morisio and Tarchiano's OTS components characterization framework [39]

experiences with implementing component-based system in one particular domain (i.e., construction of software engineering tools), we believe that the taxonomy is applicable in other domains as well.

1.1. Organization of the Paper

We first discuss the work conducted by other researchers to characterize components in Section 2. We then discuss our own taxonomy in Section 3. The taxonomy consists of six top-level criteria. It draws inspiration from existing taxonomies and classifications, but also exhibits unique features (e.g., homogeneous vs. heterogeneous multiple-component systems). When discussing the taxonomy's criteria we also identify opportunities to further refine it. Section 4 closes the paper with a discussion of our taxonomy.

2. Related Work

There are a number of taxonomies and frameworks to characterize and classify components. In the following, we give an overview.

In his book, Sametinger gives a brief survey of component classifications [47]. The discussed classifications reflect the diverse notion of component. Some classifications are targeted at white-box components such as Ada packages (Booch [5]), or are based on classifying programming-language features (Wegner [64]). Other classifications use broader dichotomies such as fine-grained vs. coarse-grained (Kain [28]), specification vs. implementation (Kain), and active vs. passive (Dusink and van Katwijk [16]).

Sametinger has developed his own taxonomy, distinguishing user interface, data interface, program interface, and component platform (cf. Table 1). The user interface of a component can be a command-line or graphical user interface. The data interface characterizes a component's input/output (I/O), which is distinguished as textual I/O, specific file I/O, and database I/O. A component's program interface characterizes how functionality is reused. Lastly, a component's platform characterizes the environment that is necessary for the component to operate.

For OTS components, Morisio and Torchiano provide a comprehensive characterization framework based on evaluation of previous work (e.g., [9] [66] [11]) [39]. Table 2 provides a summary of the characterization framework.

Torchiano et al. present a list of OTS characterization attributes based on the proposal from students made in a fifth year course [55] [56]. Examples of attributes are product

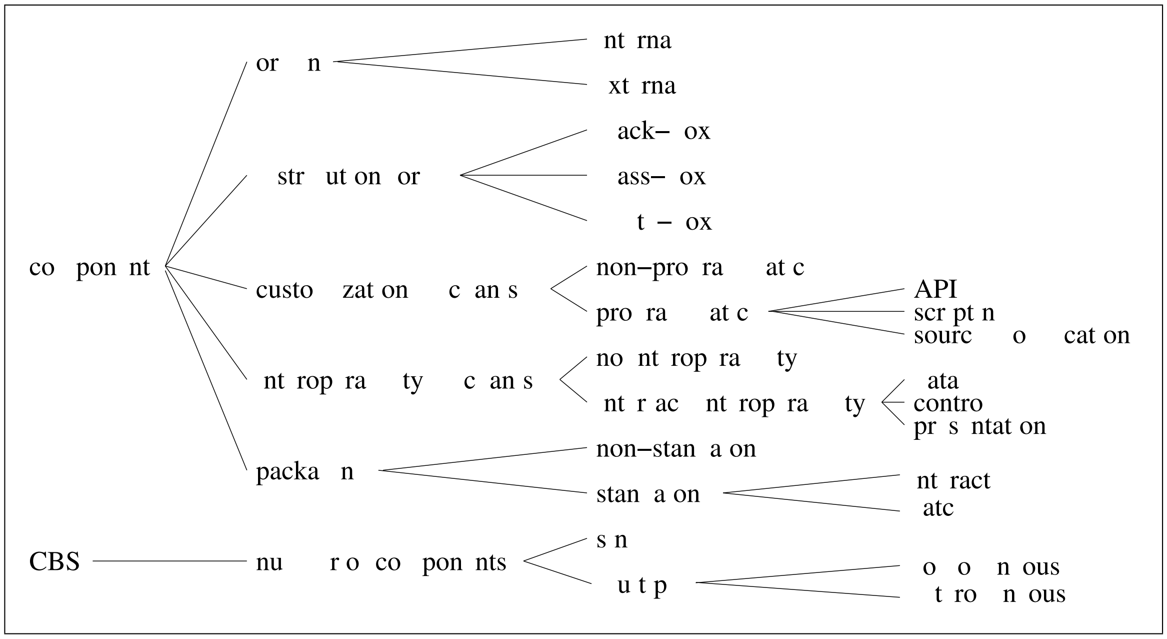

Figure 1. Complete taxonomy

maturity, market share, reliability, hardware requirements, product support, documentation, modifiability, change frequency, license type, and cost of use.

Sassi et al. propose another characterization framework for OTS components in the context of an OTS-based development environment [48]. The framework groups characteristics into: general (e.g., cost, date of first release, and change frequency), structural (e.g., name and number of services), behavioral (e.g., pre/post-conditions and state-transition diagrams), architectural (e.g., component type and architectural style), quality of service (e.g., non-functional properties and possible modification), technical (e.g., conformance to standards), and usage (e.g., similar components and use cases).

3. Taxonomy

This section discusses our proposed taxonomy to characterize components and component-based systems. The taxonomy takes a broad view on what constitutes software components, defining them as "building blocks from which different software systems can be composed" [12]. According to this definition components can be, for instance, OTS products, IDEs, domain-specific tools, application generators, compound documents, frameworks, and libraries.

Figure 1 presents an overview of the complete taxonomy. The taxonomy uses the following criteria: origin, distribution form, customization mechanisms, interoperability mechanisms, packaging, and number of components. As discussed in Section 4, the resulting taxonomy is both small and lightweight. The taxonomy can be divided into two main groups: criteria describing software components ("component"), and criteria describing systems that are built from these components ("CBS").

3.1. Origin

A component's origin or source can be classified as internal vs. external:

internal: The component is developed by the same engineers that build and assemble the final system. In a sense it is a "non-off-the-shelf component" [11] that might be later put into a reuse repository and used for other systems.

external: The component is obtained from an external source (e.g., an in-house reuse repository or a commercial component market), and not developed by the engineers that build and assemble the complete system. Such components are also called nondevelopmental items (NDIs) [11] or off-the-shelf (OTS). NDIs can be further classified by their sources: commercial vendor (COTS),[1] government-owned (GOTS), military-

[1]The commercial character of a component is not easy to define. For example, Carney et al. ask: "Is something 'commercial' if its vendor merely intends to sell it, but has sold none so far? How many sales are needed to qualify something as legitimately commercial? What if the seller does not particularly want to sell it, but is merely willing to sell it?" [10].

owned (MOTS) [60], or developed by research institutions (ROTS) [54] [1].

In contrast to this taxonomy, Carney and Long provide a more fine-grained classification, distinguishing the following types of component sources [11]: independent commercial item, custom version of a commercial item, component produced to order under a specific contract, existing component obtained from external sources (e.g., a reuse repository), and components produced in-house.

3.2. Distribution Form

A component can be made available to clients in various forms. This classification focuses on the availability and modifiability of the component's source code:

black-box: Black-box distribution of code means that the component's source code is not available (i.e., closed source). Instead, the component is made available in some binary form. For example, most shrink-wrapped software is available only as stand-alone executable.

Black-box components have certain disadvantages. For example, since source code is not available, "most forms of software analysis that would help you decide if the software is going to perform safely, securely, and reliably are not available" [61].

white-box: This form of distribution is also called open source [49]. Thus, interested clients can inspect and modify the component's sources. In practice, however, clients treat white-box components as black-box or glass-box for most development efforts [57].

glass-box: Glass-box (or gray-box) distribution means that clients have access to (parts of) the sources but are not allowed to modify them. This distribution form is of interest if the client wants source code for inspection, performance tuning, debugging, and white-box testing, but is not interested in modification [26]. Glass-box distribution is quite common in the embedded systems market [37].[2]

Many discussions revolve around the question whether components of commercial origin (i.e., COTS) mandate black-box distribution or not. However, COTS and black-box distribution are two independent properties as reflected in this taxonomy: Origin vs. Distribution Form. Interestingly, more recent definitions of COTS have widened its meaning to include open source software.[3]

[2]For example, Microsoft has made available the sources of Windows CE under its Shared Source Initiative.

[3]For instance, Torchiano and Morisio's definition: "A COTS product is a commercially available or open source piece of software that other software projects can reuse and integrate into their own products" [57].

3.3. Customization Mechanisms

Whereas components might be reused as-is without any modification, in practice some tailoring or customization (either by the developers or users) takes place. In the context of commercial OTS components, Carney et al. note that "the '-OTS' implies that the software item can be used with little or no modification. But at least some modification, minor or otherwise, is needed for most classes of commercial software"[10].

Support for customization can be divided into non-programmatic and programmatic customization mechanisms:

non-programmatic: Non-programmatic customization is accomplished, for example, by giving command-line switches, by editing parameters in startup and configuration files, or with direct manipulation at the GUI level.

There are many examples of non-programmatic customization. Unix filters and batch-processing tools such as compilers are typically customized via command-line switches. X11 applications can be customized (e.g., fonts, colors, and labels) by changing resource specifications via dedicated tools such as `xrdb` and `editres`. ActiveX controls have properties (typically with default values) that can be modified at design-time or run-time [58, p. 339]. Lastly, GUI-based applications such as Microsoft Office allow user to customize GUI elements such as menus and buttons interactively via check-boxes and drag-and-drop. Note that non-programmatic customization enables, selects, or rearranges predefined component functionality—it does not add or create new functionality.

programmatic: Programmatic customization involves some form of scripting or programming language that allows the modification and extension of the component's behavior. For black-box components, extensions are constrained by the functionality offered by the programmatic customization mechanisms.

Programmatic customization of a component can be accomplished via an application programming interface (API) and/or a scripting language:

API: Most component APIs have to be programmed in C or C++. In this case, the API is described in one or more header files. An example is the FrameMaker Developer's Kit. APIs can also take the form of object-oriented frameworks (e.g., Eclipse) and libraries (e.g., IBM VisualAge). Components that support customization via COM can be programmed in any COMaware language such as Visual Basic, C++, and C#.

The API of a component can be quite large and complex [60]; for instance, Boehm and Abts claim that Windows 95 has roughly 25,000 entry points [3], and Tallis and Balzer report that Microsoft Word has over 1,100 unique commands [52]. Component APIs vary in the extent that they permit clients to affect the component's internal state [14, sec. 3.5]. At the one extreme an API may be read-only and expose very limited information about its internal state; at the other extreme, there may be almost no restriction (e.g., allowing a "quit application" operation). The Together Open API has a high-level read-only interface as well as lower-level interfaces that allows state modifications. The APIs for plug-ins often place well-defined restrictions on the internal state that can be manipulated.

scripting: Components can also offer a scripting language to simplify programmatic customization. Sometimes, scripting is offered as an alternative to a traditional API.

A prominent example of a scriptable component is the Emacs text editor, which can be customized by programming in Emacs Lisp [50] [6]. Similarly, the AutoCAD system can be customized with AutoLISP [20], and UML Studio (`www.pragsoft.com`) has a dedicated LISP dialect called PragScript [27]. Microsoft products typically support Visual Basic scripting. Adobe GoLive can be customized with HTML files containing embedded JavaScript.

source code modification: In case of a white-box component, its source code can be directly modified in order to achieve the required customization. Typically, source code is only modified if a customization cannot be realized via scripting or API programming.

Morisio and Torchiano explicitly distinguish between required modification (i.e., necessary customization mechanisms to build a certain component-based system), and possible modification (i.e., supported customization mechanisms by the component) [39]. This distinction can be of importance. For example, whereas white-box components offer source code modification, component-based systems development rarely makes use of this option (because of future maintenance problems).

Carney and Long provide a classification to which degree a component's code can or must be changed [11]. They distinguish: (1) very little or no modification, (2) simple parameterization, (3) necessary tailoring or customization, (4) internal revisions to accommodate special platform requirements, and (5) extensive functional recoding and reworking. The first two modifications are non-programmatic. The third involves programmatic customization via API or scripting. The last two are modifications not anticipated by the original component vendor and require modification of the source code.

Another classification introduces three types [10]: installation-dependent components (which require the user to take some actions before the software can operate, for instance, in the form of setting environment variables or providing license information), tailoring-dependent products (which require a considerable amount of initialization data such as the definition of schema information or business rules), and modified products (which occurs "when a customer asks a software vendor for (or, conceivably, makes himself) some ad hoc, to-order alteration of a commercial product"). With the last type, the component's modification is a change in functionality or behavior not originally intended by its vendor.

3.4. Interoperability Mechanisms

Wegner states that "interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform" [65]. Interoperability among components can be achieved, for instance, by passing control, sending messages, or sharing data. Note that Customization Mechanisms addresses functionality that is added to the component itself, whereas interoperability refers to the use of the component's functionality by other components (i.e., its clients).

The following classification of component interoperability is based on research in integration of software engineering environments [8] [53] [63]:

no interoperability: In this case, the component does not support any explicit, programmable interoperability mechanism. In this case, the only form of interoperability that the component provides is via its user interface. This can be the case with legacy terminal-based business applications. Even though the user interface is meant to be used interactively by end users, such components can be made interoperable via ad-hoc wrapping techniques. For example, batch applications and textual user interfaces can be wrapped with pseudo ttys (on Unix) or dedicated (screen scraper) tools [47] [59] such as Expect (`http://expect.nist.gov`). There are also approaches that intercept and synthesize GUI events [23] [2].

interface interoperability: Interface interoperability means that the components offers some kind of programmable interface to enable interoperability with other components. The following interface types can be distinguished:

data: Data interfaces provide a simple API to access component-specific data. Sametinger distinguishes between textual, specific file (i.e., binary), and database input/output [47, sec. 9.1]. Examples of data formats that are supported by components are Word documents (textual or binary) and SVG files (XML-based).

Unix's pipe-and-filter architecture uses textual input/output to achieve data interoperability. Another example of data interoperability is provided by early IDEs such as PCTE and Centaur that employed a central data repository. The repository connects the various tools that are part of the IDE by allowing them to store their intermediate results for use by other tools [45] [44].

control: Control interoperability involves an interface that allows a component to pass a message to another component. This functionality is provided by wiring standards such as CORBA and COM (as well as other message-oriented and object-oriented middleware). For instance, the Common Desktop Environment (CDE) offers a message brokering system called ToolTalk [34]. Succi et al. use JavaSpaces tuple spaces to exchange messages between tools [51].

There are many examples of IDEs that are primarily based on control interoperability such as HP's Softbench, Sun's ToolTalk, and Reiss' Field and Desert environments [7] [46]. With these IDEs, a message server allows tools to register their interest in certain events that other tools might send to the server [25] [44]. Tools can report events to the server which are then distributed to the registered tools.

presentation: Presentation interoperability refers to a seamless interoperation at the user-interface level. Components are tightly integrated and have a common look-and-feel. An Example of components that support presentation interoperability are compound documents such as OLE and OpenDoc. In order to achieve such tight integration some kind of wiring standard is necessary. IDEs such as Eclipse, IBM VisualAge, and Microsoft Visual Studio have extensible architectures that makes it possible to add components (i.e., plug-ins) seamlessly.

The three interoperability types introduced above are usually inclusive. Presentation interoperability typically has to support control and data interoperability in order to achieve seamless integration of components. Components with control interoperability also support some kind of data interoperability.

Egyed et al. classify components as reactive or proactive [18]. Reactive components react to user input (and other outside stimuli). Input can be data (e.g., obtained by reading from a file), or control messages (e.g., a mouse click event). A reactive component is passive in the sense it does not initiate interactions with other components; instead, it waits for service requests [17]. In contrast, a proactive component notifies other components of changes in its state. Components often have both reactive and proactive characteristics. However, OTS products and IDEs often have deficiencies in their realization of proactiveness, lacking, for instance, sophisticated event notification; this is the case with Microsoft Office [22] and Rational Rose [17].

3.5. Packaging

Components can be packaged in different ways. Packaging is the form in which the component is used, not the form it is distributed in [39]. For example, a white-box component is distributed with source code, but can be packaged via compilation into a stand-alone executable or a library. The packaging of a component is characterized as follows:

standalone component: Such a component is a standalone program, application, or tool that can be directly executed. The component is standalone in the sense that it can be used without prior integration or customization. Examples of this type of component are OTS products, IDEs, and domain-specific tools. Standalone components can range from huge (e.g., Microsoft Office) to small (e.g., Unix's `echo`).

The taxonomy further distinguishes standalone components whether they are interactive or batch systems [15]:

interactive: Interactive (or conversational) components have a graphical or textual user interface so that a human can participate in the computation. In fact, often the human drives the computation, the system reacting to the user's input. Microsoft Office is an example of an interactive component.

batch: Batch components make a complete program run without human intervention. Input is specified before program start (e.g., in the form of input files or streams). The behavior of the system is typically specified with command-line arguments and configuration files (i.e., non-programmatic customization). Unix filters are examples of batch components.

non-standalone component: A non-standalone component has to be integrated and/or customized before it can be executed. A typical example are linkable components such as object modules or (class) libraries, which must be linked statically or dynamically with other components to obtain an executable [66]. Other examples of this component type are components from application generators (e.g., a scanner generated by `lex`), OTS components (e.g., ActiveX controls), compound documents, (object-oriented) frameworks, classes, and functions.

The packaging of a component is of importance for interoperability and reuse. For example, Unix provides sorting functionality both with the `sort` filter (which is a standalone component), and the `qsort` C library call (which is a non-standalone component). RCS, a tool for version control, is implemented as a set of Unix filters (`rcs`, `co`, `ci`, etc.), but now also offers a C/C++ and Java API (`www.aicas.com/rce_en.html`).

Dusink and van Katwijk distinguish between active and passive components [16] [47, sec. 9.2.3]. Active components run on their own and use operating system services as a means of interoperation (e.g., shared memory or message passing). Passive components are functions, classes, and modules, which are included in a software system by linking or directly including their source code. There seems to be a correspondence between Dusink and van Katwijk's active and passive components, and this taxonomy's standalone and non-standalone components, respectively.

3.6. Number of Components

In contrast to the previous criteria, which describe components, this criterion characterizes a component-based system. A component-based system can be distinguished by the number of components that constitute the system:

single: These systems use a single component on which they heavily depend. Single-component systems often rely on a powerful standalone component (e.g., Microsoft Office) or IDE (e.g., Eclipse).

Carney distinguishes single-component systems into turnkey systems and intermediate systems [9]. Turnkey systems use a single OTS component such as Microsoft Office or Mozilla, on which they heavily depend and which is used out-of-the-box without programmatic customization. Intermediate systems are also built on a single OTS component, but also "have a number of [programmatically] customized elements specific to the given application." In this taxonomy customization is addressed by a separate criterion.

multiple: These systems are (primarily) composed of multiple components, possibly from different vendors and having different characteristics.

Often multiple-component systems have two to four core components which realize a significant amount of the system's functionality. But there are also systems that are composed of more components. For example, Morisio et al. mention a system comprising 13 OTS components [38], and NASA's Hubble Space Telescope command and control system uses even more than 30 COTS/GOTS components [43].

Multi-component systems can be further distinguished as being composed of homogeneous or heterogeneous components:

homogeneous: A system is homogeneous if its components adhere to the same wiring standard. For example, all Microsoft Office products support COM and OLE, and all Eclipse plug-ins use the same extensibility mechanism to integrate into the platform. If a component-based software-engineering tool uses popular standards such as JavaBeans or COM, it can incorporate off-the-shelf components that have been developed independently by third parties.

heterogeneous: A system is heterogeneous if it mixes components based on different wiring standards, interoperability standards, or architectures. Such systems are often composed of OTS products from different independent vendors (e.g., Rational Rose and Matlab/Stateflow [17]), possibly also integrating proprietary, system-specific components [19]. For instance, Garlan et al. describe the building of a software design environment, Aesop, consisting of the following components: (1) a GUI framework (InterViews), (2) an event-based tool-integration mechanism (HP Softbench), (3) an RPC mechanism, and (4) an object-oriented database [21]. Staringer provides another multi-component tool-building experience, integrating (1) a GUI builder (NeXTStep Interface Builder), (2) a spreadsheet application (Lotus Improv), (3) Mathematica, and (4) a relational database (Sybase).

Multi-component systems can have a significant amount of *glue code*, whose purpose is it to bind the (often heterogeneous) components together—in Carney's words, "sometimes cleanly and sometimes crudely" [9]. Glue code realizes functionality such as transferring of control flow to a component; bridging of incompatibilities between components; and handling of errors and exceptions [59]. Whereas components use typed high-level programming languages [42], glue code is often written with a scripting language such as Perl, Tcl, Visual Basic, or Unix shells. For example, ActiveX components are often written in C++, but glued together with Visual Basic [36].

For multiple-component systems an important consideration is which component holds the main thread of control or else implements the control loop [21]. The component that implements the control loop is the system's "driver" and in charge of delegating control to other components. Components are often difficult to make interoperate if more than one components in the system assumes that it is the driver. A standalone component has its own thread of control by definition. Some non-standalone components such as libraries are passive in the sense that they do not have their own control thread—control is passed to them with an explicit invocation from the client. However, other non-standalone components such as GUI frameworks have their own control loop.

Similar to our classification, Wallnau distinguishes OTS-based systems into OTS-intensive systems (which integrate many OTS products) and OTS-solution systems (which consist of one substantial OTS product customized to provide a "turnkey" solution) [62].

Boehm et al. classify component-based system according to the dominating development activity. Based on their experiences, they identify four major project types [4]: (1) assessment intensive projects (which focus on finding a feasible set of components having capabilities that require little or no programmatic customization); (2) tailoring intensive projects (whose main efforts lie in programmatically customizing one or only a few components to realize most of the system's capabilities); (3) glue-code intensive projects (which require a significant amount of glue code design and implementation involving multiple components); and (4) non-OTS intensive projects (in which most of the system's capabilities are provided by traditional custom development).

4. Conclusions

This paper has introduced a taxonomy to characterize component-based systems. The taxonomy has six top-level criteria: origin, distribution form, customization mechanisms, interoperability mechanisms, packaging, and number of components.

While, our taxonomy has been influenced by and reflects previous work on taxonomies, it specifically satisfies two properties:

small and course-grained: We have selected a relatively small number of criteria to characterize a component. Furthermore, each criterion is rather course-grained, having at most three alternatives.

We have identified criteria with potential for a more fine-grained characterization, but these have been deliberately excluded from the taxonomy. Other taxonomies typically have a broader goal such as to aid in a comprehensive evaluation of any kind of component in any application domain.

lightweight: It typically is a straightforward task to assign the taxonomy's criteria to a particular component or component-based system.

In other words, this taxonomy uses *orientation level* criteria, which "paint an overall picture of the component" and whose "values can be often gleaned from developer documentation" [13]. Hence, classifying a component with our taxonomy is a lightweight activity.

To obtain a small and lightweight taxonomy, we have omitted a number of interesting characterizations that do not have a strong direct impact on the construction of software engineering tools itself. Examples of such criteria are market share and execution platform. Such component characteristics can have an impact on the functional or non-functional requirements of component-based systems. For example, an OTS product's market share and execution platform can have an impact on the adoptability of the resulting component-based system.

Whereas this taxonomy has been created primarily for the purpose to characterize components for the component-based building of software engineering tools, we believe that its properties make it also applicable for other classification purposes that go beyond its original context. Also, researchers can use our taxonomy as a starting point to expand and/or contract it so as to better suit their needs.

References

[1] C. Abts. COTS-based systems (CBS) functional density—a heuristic for better CBS design. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 1–9. Springer-Verlag, 2002.

[2] K. Bao and E. Horowitz. A new approach to software tool interoperability. *11th ACM Symposium on Applied Computing (SAC'96)*, pages 500–509, Feb. 1996.

[3] B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1):135–138, Jan. 1999.

[4] B. W. Boehm, D. Port, Y. Yang, , and J. Bhuta. Not all CBS are created equally: COTS-intensive project types. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 2003.

[5] G. Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings Publishing Company, 1987.

[6] N. S. Borenstein and J. Gosling. Unix emacs: A retrospective—lessons for flexible system design. *1st ACM SIGGRAPH Symposium on User Interface Software*, pages 95–101, Oct. 1988.

[7] A. W. Brown. Control integration through message-passing in a software development environment. *Software Engineering Journal*, 8(3):121–131, May 1993.

[8] A. W. Brown and M. H. Penedo. An annotated bibliography on integration in software engineering environments. Special Report CMU/SEI-92-SR-8, Software Engineering Institute, Carnegie Mellon University, May 1992. `http://www.sei.cmu.edu/pub/documents/92.reports/pdf/sr08.92.pdf`.

[9] D. Carney. Assembling large systems from COTS components: Opportunities, cautions, and complexities. In *SEI Monographs on the Use of Commercial Software in Government Systems*. Software Engineering Institute, Carnegie Mellon University, June 1997.

[10] D. Carney, S. A. Hissam, and D. Plakosh. Complex COTS-based software systems: practical steps for their maintenance. *Journal of Software Maintenance: Research and Practice*, 12(6):357–376, Nov./Dec. 2000.

[11] D. Carney and F. Long. What do you mean by COTS? Finally, a useful answer. *IEEE Software*, 17(2):83–86, Mar./Apr. 2000.

[12] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[13] L. Davis, R. F. Gamble, and J. Payton. The impact of component architecture on interoperability. *Journal of Systems and Software*, 61(1):31–45, Mar. 2002.

[14] R. DeLine. A catalog of techniques for resolving packaging mismatch. *5th ACM Symposium on Software Reusability (SSR'99)*, pages 44–53, May 1999.

[15] R. DeLine, G. Zelesnik, and M. Shaw. Lessons on converting batch systems to support interaction. *19th ACM/IEEE International Conference on Software Engineering (ICSE'97)*, pages 195–204, May 1997.

[16] E. M. Dusink and J. van Katwijk. Reflections on reusable software and software components. *ACM Ada-Europe International Conference*, pages 113–126, 1987.

[17] A. Egyed and R. Balzer. Unfriendly COTS integration—instrumentation and interfaces for improved plugability. *16th International Conference of Automated Software Engineering (ASE'01)*, pages 223–231, Nov. 2001.

[18] A. Egyed, S. Johann, and R. Balzer. Data and state synchronicity problems while integrating COTS software into systems. *4th International Workshop on Adoption-Centric Software Engineering (ACSE'04)*, pages 69–74, May 2004.

[19] A. Egyed, H. A. Müller, and D. E. Perry. Integrating COTS into the development process. *IEEE Software*, 22(4):16–18, July/Aug. 2005.

[20] M. Gantt and B. A. Nardi. Gardeners and gurus: Patterns of cooperation among CAD users. *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'92)*, pages 107–117, May 1992.

[21] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. *17th ACM/IEEE International Conference on Software Engineering (ICSE'95)*, pages 179–185, Apr. 1995.

[22] N. M. Goldman and R. M. Balzer. The ISI visual design editor generator. *IEEE Symposium on Visual Languages (VL'99)*, pages 20–27, Sept. 1999.

[23] M. Grechanik, D. Batory, and D. E. Perry. Integrating and reusing GUI-driven application. *7th International Conference on Software Reuse (ICSR-7)*, pages 1–16, Apr. 2002. `http://www.cs.utexas.edu/users/gmark/Publications.html`.

[24] G. Gui, H. M. Kienle, and H. A. Müller. REGoLive: Building a web site comprehension tool by extending GoLive. *7th IEEE International Symposium on Web Site Evolution (WSE'05)*, pages 46–53, Sept. 2005.

[25] W. Harrison, H. Ossher, and P. Tarr. Software engineering tools and environments: A roadmap. *Conference on The Future of Software Engineering*, pages 263–277, June 2000.

[26] S. A. Hissam and D. Carney. Isolating faults in complex COTS-based systems. *Journal of Software Maintenance: Research and Practice*, 11(3):183–199, May/June 1999.

[27] J. H. Jahnke, J. P. Wadsack, and A. Zündorf. A history concept for design recovery tools. *6th IEEE European Conference on Software Maintenance and Reengineering (CSMR'02)*, pages 37–46, Mar. 2002.

[28] J. B. Kain. Enabling an application or system to be the sum of its parts. *Object Magazin*, 6:64–69, Apr. 1996.

[29] H. M. Kienle. *Building Reverse Engineering Tools with Software Components*. PhD thesis, Department of Computer Science, University of Victoria, 2006. To appear.

[30] H. M. Kienle and J. H. Jahnke. A visual language in Visio: First experiences. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 90–93, May 2003.

[31] H. M. Kienle and H. A. Müller. A WSAD-based fact extractor for J2EE web projects. *Technical Report, University of Victoria*, June 2006.

[32] H. M. Kienle, A. Weber, J. Jahnke, and H. A. Müller. Tackling the adoption problem of domain-specific visual languages. *2nd Domain-Specific Modeling Languages Workshop at OOPSLA 2002*, pages 77–88, Oct. 2002.

[33] H. M. Kienle, A. Weber, and H. A. Müller. Leveraging SVG in the Rigi reverse engineering tool. *SVG Open / Carto.net Developers Conference*, July 2002.

[34] G. Kraft. CDE plug-and-play. *Linux Journal*, May 1998. `http://www.linuxjournal.com/article/2362`.

[35] J. Ma, H. M. Kienle, P. Kaminski, A. Weber, and M. Litoiu. Customizing Lotus Notes to build software engineering tools. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'03)*, pages 276–287, Oct. 2003.

[36] P. M. Maurer. Converting command-line applications into binary components. *Software—Practice and Experience*, 35(8):787–797, July 2005.

[37] A. Möller, M. Akerholm, J. Fredriksson, and M. Nolin. Evaluation of component technologies with respect to industrial requirements. *30th IEEE EUROMICRO Conference (EUROMICRO'04)*, pages 56–63, Aug. 2004.

[38] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. COTS-based software development: Processes and open issues. *Journal of Systems and Software*, 61(3):189–199, Apr. 2002.

[39] M. Morisio and M. Torchiano. Definition and classification of COTS: a proposal. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 165–175. Springer-Verlag, 2002.

[40] H. A. Müller, M. Storey, A. Weber, W. Kastelic, H. Kienle, Q. Zhu, J. Ma, F. Yang, D. Zwiers, K. Wong, and J. Pipitone. Adoption-centric software engineering. *CASCON 2002 Technology Exhibition Handout*, 2002. `http://www.acse.cs.uvic.ca/downloads/info/flyer_screen.pdf`.

[41] H. A. Müller, A. Weber, and K. Wong. Leveraging cognitive support and modern platforms for adoption-centric reverse engineering (ACRE). *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 30–35, May 2003.

[42] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–70, Mar. 1998.

[43] T. Pfarr and J. E. Reis. The integration of COTS/GOTS within NASA's HST command and control system. In J. Dean and A. Gravel, editors, *1st International Conference on COTS-Based Software Systems (ICCBSS'02)*, volume 2255 of *Lecture Notes in Computer Science*, pages 209–221. Springer-Verlag, 2002.

[44] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[45] S. P. Reiss. Simplifying data integration: The design of the desert software development environment. *18th ACM/IEEE International Conference on Software Engineering (ICSE'96)*, pages 398–407, May 1996.

[46] S. P. Reiss. Constraining software evolution. *18th IEEE International Conference on Software Maintenance (ICSM'02)*, pages 162–171, Oct. 2002.

[47] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[48] S. B. Sassi, L. L. Jilani, and H. H. B. Ghezala. COTS characterization model in a COTS-based development environment. *10th IEEE Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 352–361, 2003.

[49] D. Spinellis and C. Szyperski. How is open source affecting software development? *IEEE Software*, 21(1):28–33, Jan./Feb. 2004.

[50] R. M. Stallman. Emacs: The extensible, customizable self-documenting display editor. *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, June 1981.

[51] G. Succi, W. Pedrycz, E. Liu, and J. Yip. Package-oriented software engineering: A generic architecture. *IT Pro*, 3(2):29–36, Mar./Apr. 2001.

[52] M. Tallis and R. Balzer. Document integrity through mediated interfaces. *DARPA Information Survivability Conference & Exposition II (DISCEX'01)*, pages 263–270, June 2001.

[53] I. Thomas and B. A. Nejmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, Mar. 1992.

[54] S. Tilley, S. Huang, and T. Payne. On the challenges of adopting ROTS software. *3rd International Workshop on Adoption-Centric Software Engineering (ACSE'03)*, pages 3–6, May 2003.

[55] M. Torchiano and L. Jaccheri. Assessment of reusable COTS attributes. In H. Erdogmus and T. Weng, editors, *2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, volume 2580 of *Lecture Notes in Computer Science*, pages 219–228. Springer-Verlag, 2003.

[56] M. Torchiano, L. Jaccheri, C. Sorensen, and A. I. Wang. COTS products characterization. *14th ACM/IEEE International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 335–338, July 2002.

[57] M. Torchiano and M. Morisio. Overlooked aspects of COTS-based development. *IEEE Software*, 21(2):88–93, Mar./Apr. 2004.

[58] R. van Ommering and J. Bosch. Widening the scope of software product lines—from variation to composition. In G. Chastek, editor, *SPLC2*, volume 2379 of *Lecture Notes in Computer Science*, pages 328–347. Springer-Verlag, 2002.

[59] M. R. Vidger and J. Dean. An architectural approach to building systems from COTS software components. *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, pages 131–143, Nov. 1997.

[60] M. R. Vigder, W. M. Gentleman, and J. Dean. COTS software integration: State of the art. Technical Report NRC 39198, National Research Council Canada, Jan. 1996. `http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-39198_e.html`.

[61] J. M. Voas. The challenges of using COTS software in component-based development. *IEEE Computer*, 31(6):44–45, June 1998.

[62] K. C. Wallnau. A basis for COTS software evaluation: Foundations for the design of COTS-intensive systems. *SEI Interactive, Software Engineering Institute*, 1998. `http://www.sei.cmu.edu/cbs/cbs_slides/ease98/index.htm`.

[63] A. I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments: International Workshop on Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149. Springer-Verlag, 1990.

[64] P. Wegner. Capital-intensive software technology. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume I: Concepts and Models. ACM Press, 1989.

[65] P. Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285–287, Mar. 1996.

[66] D. Yakimovich, J. M. Bieman, and V. R. Basili. Software architecture classification for estimating the cost of COTS integration. *21st ACM/IEEE International Conference on Software Engineering (ICSE'99)*, pages 296–302, May 1999.

[67] F. Yang. Using Excel and PowerPoint to build a reverse engineering tool. Master's thesis, Department of Computer Science, University of Victoria, 2003.

[68] Q. Zhu, Y. Chen, P. Kaminski, A. Weber, H. Kienle, and H. A. Müller. Leveraging Visio for adoption-centric reverse engineering tools. *10th IEEE Working Conference on Reverse Engineering (WCRE'03)*, pages 270–274, Nov. 2003.