Enhancing the E-business community by software component technology

Daniela Handl, Hans-Jürgen Hoffmann, Ludger Martin

***Abstract*— In E-business persons and/or institutions may permanently co-operate in a closed fashion, they may occasionally meet for a short period of time or a limited transaction; or they may form an *open community* with changing roles of participants, e.g., *peer-to-peer, service provider and client, supplier and customer*. It is a demand to provide architectural guidelines for design with as minimal complexity as possible. We concentrate, due to its importance, on the latter case – open communities in E-business–. Thereby consider software architecture topics in the context of a *realization by software components*, following a flow paradigm and hierarchical composition, and emphasize *functional support offered by components through agent technology*, e.g., for E-mail document analysis and workflow processing. A component development environment, *HOTAgent*, is elaborated to demonstrate our approach in component architecturing. Experience in an ongoing project, *HOTxxx*, is shortly reported motivating our approach.**

We expect, following our ideas, a more integrated, consistent approach of software component design enhancing E-business communities.

Index Terms*— Open E-business communities, component architecture, component composition, functional support offered through agent technology, E-mail analysis, workflow aspects, *HOTxxx

I. Introduction

There are various forms of E-business communities:

- Persons and/or institutions may permanently co-operate in a closed fashion – intranet and private internet –,
- they may occasionally meet for a short period of time or a limited transaction – surfing in public internet –; or,
- they may form an *open community* with changing roles of participants.

In this paper, we concentrate on the latter case – open communities –. Typical occurrences are, to give some examples:

- *peer-to-peer* – e.g., in a *B2B*-correspondents´ community –,
- *service provider and client* – e.g., a commercial business community –,
- *supplier and customer* – typically a *B2C*-trading community –.

Following established software engineering approaches *components* are used to design and implement E-business systems for open communities. Re-usability of components by means of some appropriate system partitioning and, later on, composition techniques is of paramount interest, allowing realization of E-business systems also by non-professional application programmers.

What is a good component architecture to achieve these goals? Szyperski [27] defines: "A *component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component may be deployed independently and is subject to composition by third parties.". Section 2 goes into details. It is important to be open for components hierarchically architectured by enclosing further components as needed.

From a functional point of view we consider components, appropriately partitioned and – may be – composed from enclosed ones, to realize what we call an *agent functionality*. Again, we start with a definition (Hong *et al.* [13]): "S*oftware agents* are entities that function autonomously and perform laborious information processing tasks cooperatively.". To intentionally decrease design complexity (Hoffmann [9]) we have to find a good balance between agents´ intelligence and agents´ simplicity.

In subsections 3.A through 3.D we identify, following some appropriate partitioning, decomposition, and functional division strategies for components, architectural attempts for structuring and mapping components with individually assigned agent functionality. As a running example, we consider a *repair service system* (*RSS*) for electronic equipment in a *B2C* application. E-mail document analysis and workflow processing are main functionalities of the considered *RSS agent (RSSA)*. Customers may buy wearing parts from a catalogue, a rather straightforward transaction. In addition, there is a diagnosis agent which helps in cases of malfunction or breakdown to identify the cause and probably the spare part

D. Handl is with Darmstadt University of Technology, Department of Computer Science, Wilhelminenstr. 7, 64283 Darmstadt, Germany (e-mail: handl@pu.informatik.tu-darmstadt.de).

H.-J. Hoffmann is with Darmstadt University of Technology, Department of Computer Science, Wilhelminenstr. 7, 64283 Darmstadt, Germany (e-mail: hjhoffmann@informatik.tu-darmstadt.de).

L. Martin is with Darmstadt University of Technology, Department of Computer Science, Wilhelminenstr. 7, 64283 Darmstadt, Germany (phone: +49 6151 16 6710 ,e-mail: martin@pu.informatik.tu-darmstadt.de).

to be replaced. This additional service is subject of interest in the example at hand. An initial standard scenario – linear progression – would be as follows:

A customer sends an E-mail document to *RSS*. The E-mail contains a description of the faulty behaviour.

The *RSSA* composed of a linked set of minimal components analyses this description in terms of the defect (the technique of analyzing natural language is beyond the scope of this paper; the analysis is therefore reduced to a keyword matching, see below in subsection 3.A) and thereafter specifies the broken part.

To design and to implement an E-business system following our ideas Martin has researched specific design tools (called *HOTAgent*, [21]). *HOTAgent* is a visual component development environment to construct agents for E-business. Section 4 shortly presents the tools. In section 5 we relate the work presented here to the long-range project *HOTxxx* [10] where it is a part of.

II. Software components for E-business processes

A wide range of transactions is handled over the internet. As more people and companies are participating in E-business, trade procedures become sophisticated and marketable goods multifaceted.

In order to attract customers, several add-on services are popular and often common standard, e.g., providing search facilities, links to possibly useful sites, and detailed product information.

As a result, a customer has a huge amount of possible goods at his disposal and a virtual unmanageable quantity of connected information. Yet this abundance of – at least in theory – accessible goods, is what makes E-business worthwhile.

Software component technology is a possibility to enhance electronic transactions, as will be shown below.

A. Components

Under the point of view of object-oriented software design a component consists of several classes which are encapsulated by a set of well defined interfaces. The interface is the sole public part of the component. All other classes of the component are private. The interfaces ensure the communication between all components. Because of this reason a component can be regarded as a *black box*.

To communicate, every component provides entrances and exits. Lüer and Rosenblum [17, 18] call the entrances *requires ports* and the exits *provides ports*. Communication is based on an event mechanism, which enables all components to communicate with each other independently of their task.

Lüer and Rosenblum demand self-documentation of components. Components need to be well documented to ensure a proper usage. This documentation, including a description of the functionality of the component itself as well as of the respective associated agent(s) which realize their functional behaviour . The documentation should not be deployed separately but should be included in the component(s) themselves.

We speak of *top system component*, *intermediate component(s)*, *minimal component(s)* in order to consider hierarchically composing components; we don´t restrict to tree hierarchy only.

top system component: This is a synonym to the "covering" component of the *main program* as finally realized and conceived to be the entry/activation point for the runnable system.

intermediate components: They (optionally) help to structure the system into coherent parts each one related to an identifyable behaviour. In so far they may be related to *subordinate agents*. The separation should allow to master complexity.

minimal components: These are the leaves in the hierarchy and realize elementary functionality.

B. Software components as a solution

With the depiction of software components in the former subsection 2.A, let's have a look at the accruing advantages, considering some principal tasks of E-business.

One major advantage of E-business are (in theory) the wide ranges of customers, suppliers and goods. So, there is an enormous potential of trading people, dealing with even more trading goods.

Even though in every business process there are universal tasks to be dealt with, the very way in which to fulfil them differs among customers as well as suppliers and of course due to specific properties of goods.

Since components have contractually specified interfaces, capsulating their algorithms for the assigned task, software components may be launched independently. In different businesses there are always similar tasks; so, it is possible to re-use specific components. There might be components able to communicate with the trading correspondents' systems due to a common protocol or other components meeting the demands of individual customers' / suppliers' businesses.

Communication via the components' interfaces is independent of their task, and components should document themselves, so that they may even be broadly re-used. Because of this reason it is also possible to buy components of-the-shelf from third parties. It is not necessary to create all components on it's own.

III. Partitioning

After describing the basics of component technology applied in our work and presenting our running example it is interesting how to divide the *RSS* top system component and its associated agent up into intermediate and minimal software components. We investigate some architectural attempts which are presented in the following text.

A. Linear processing

In our first, initial attempt (see Clausius [4]; no top component put out, please wait for section 3.C / 3.D) all (here) intermediate components are connected in a linear order to

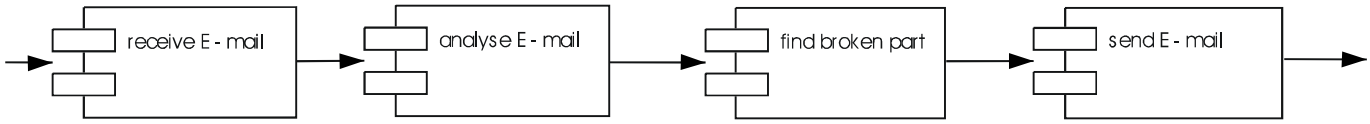

Figure 1a: *RSS* agent with linear processing

construct the RSS. The first component receives the initial data, processes these, and forwards results to the next component. This works according to the fixed pattern until the last component processed the data and provided the results.

Using this attempt the components are very special. The components are created only for this purpose and may not be re-used easily. An other disadvantage is, that all data need to be forwarded from the first to the last component in the chain, e.g., the E-mail addressing data are received by the *receive E-mail* component and must be forwarded to the *send E-mail* component.

The *RSSA* (actually four independent, subordinate components as no top component is provided) may be divided in the following components (see Figure 1a):

receive E-mail: This component accepts an electronic mail with a request for repair service in the guarantee period following purchase of an equipment. After receiving a mail it activates the complete processing of the document. The component forwards the E-mail addressing data and the letter text.

analyse E-mail: The agent needs to analyze the document for a description of the faulty behaviour. To do this, this component is provided. To simplify the agent as mentioned earlier, this components searches only keywords and does not use intelligent algorithms to find symptoms; details of text analysis are beyond the scope of the paper. It forwards the symptoms of the faulty behaviour and the E-mail addressing data.

find broken part: Using found symptoms the broken part can be found in a data base. This component forwards the broken part and also the E-mail addressing data of the customer.

send E-mail: The last task to do is preparing the answer E-mail. The answer must explain the broken part. In the end the E-mail needs to be returned to the customer.

B. Looping and transactions' archiving

Linear processing is not sufficient in considering serial activities in E-business. There has to be an component for starting the process explicitly; in our running *RSS* example we call it the *purchase* component (see Figure 1b; we still don´t put out a top component):

purchase: The customer buys the considered equipment.

There are different sequencing paradigms available for structuring in process models, e.g., control flow, data flow, and, most appropriate in the context of E-business, workflow. Parallel processing (which we do not consider in detail here) encompasses the same paradigms in the threads. In all cases branching to form conditionals and loops may occur. Following this we modify *RSS* (Figure 1a) as shown in Figure 1b.:

guarantee: If the guarantee period expired *RSS* processing is finished.

Now we have a looping structure. In the *RSS* example there has also to be a *wait* component:

wait: The component stops treatment of the process till a new repair request arrives.

And note, in Figure 1a there is no state-conserving archiving involved; any further request by the same customer is handled without re-access to any previously received E-mail. A variant of this architectural attempt includes state-conserving archiving with either

1. An additional "centralized", *semi-hidden/permanent* intermediate component *archive* initialized, e.g., when the customer buys an equipment and alive till end of the *RSS* service guarantee period – see Figure 1b), or
2. specific *archiving information bound to all RSS events* originating from the customer relating to the considered equipment in <u>one</u> mutually exchanged document (see Figure 1c; details of this variant in a more typical context see section 5).

archive: This component realizes a sequential data set archiving all E-mail inqueries and replies for the considered transaction by the resp. customer (belongs to Figure 1b).

... + archive update: These components replace the corresponding components shown in Figures 1a/1b as needed for case (ii) illustrated in Figure 1c.

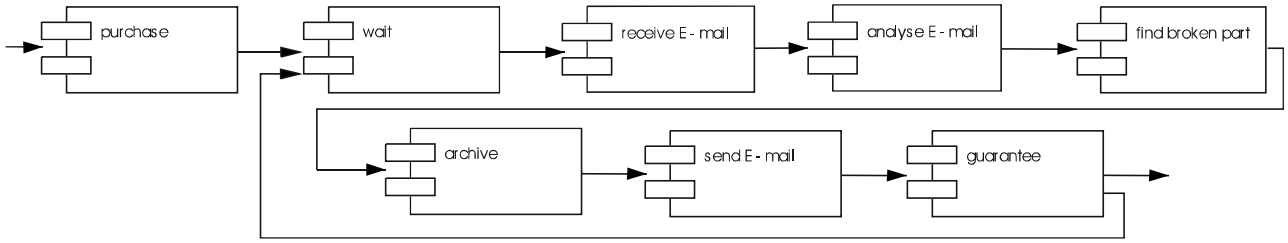

Figure 1b: *RSS* agent with linear processing in a loop involving initialisation at time of purchase, sending request, answering, and eventually finishing (guarantee period ended), central archiving

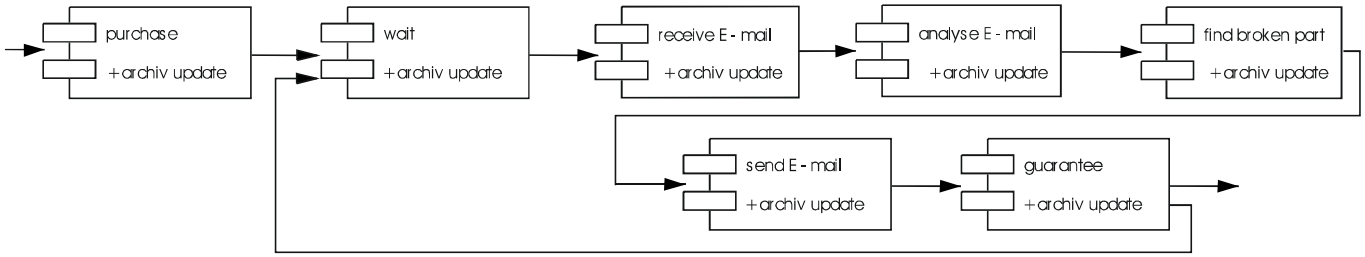

Figure 1c: *RSS* agent with linear processing in a loop involving initialisation, sending request, answering, and eventually finishing (guarantee period ended), archiving all transaction data in exchanged E-mail document

C. Service and control components

Another possibility is to introduce and then to decompose a top *RSS* system component into *service and control components* (see Clausius [4]) to bring the respective functionality and behaviour into sharper focus. Every *RSS* has one *control component* as an intermediate component in the architectural hierarchy. This component, essantially, realizes the main behaviour of the *RSSA* and is used to control the activities of the service components on the next lower level (which may be intermediate or minimal), see especially Figure 2. *Service components* have a good re-usability.

The control component of the *RSSA* is responsible for the following task:

control: This component represents the state of the agent. Its task is to coordinate several work steps. The control component uses several service components and assigns small working steps to them. To do this the control component has requires and provides ports to communicate with service components. The control component is designed specially for the needs of the *RSSA*. Because of this the component needs to be created manually using a textual programming language.

Beside the control component the agent needs service components as described below:

... data base/set: This component can store any data. Stored data can be identified with a key. The data base supports inserting and deleting data. In addition it supports also data base queries with user defined criteria. The data base component may be used on two places in the *RSSA*. In *part data base* the possible symptoms with the parts are stored. The *transaction data set* is used to store data about customers and transactions.

text analysis: The agent needs to analyze electronic documents. To do this in our running example it is only necessary to recognize key words in a text. To simplify the agent, as mentioned earlier, this component searches only key words. If the *RSSA* is used in a real environment it is no problem to replace this component using a more powerful one.

communication: This component is used to handle the communication between the business partner/correspondent and the agent.

D. Composed control components

As mentioned before every control component needs to be constructed specifically for every component. This is done (textually) in a normal programming language. It would be better to architecture such a component also using predefined components. Giesl [6] describes how to compose (top or intermediate) components using (lower level) intermediate or minimal components. We show it in our *RSS* example by composing the control component introduced above (Figure 3). It is architectured using other, minimal components. These minimal components are, e.g., distributor, selector, condition, or repeat components. Here, software design patterns may be introduced into the design.

In the following two sections we present a component development environment, *HOTAgent*, realizing the concepts as discussed above – section 4 –, and its project surrounding where it is part of, the *HOTxxx* project – section 5 –.

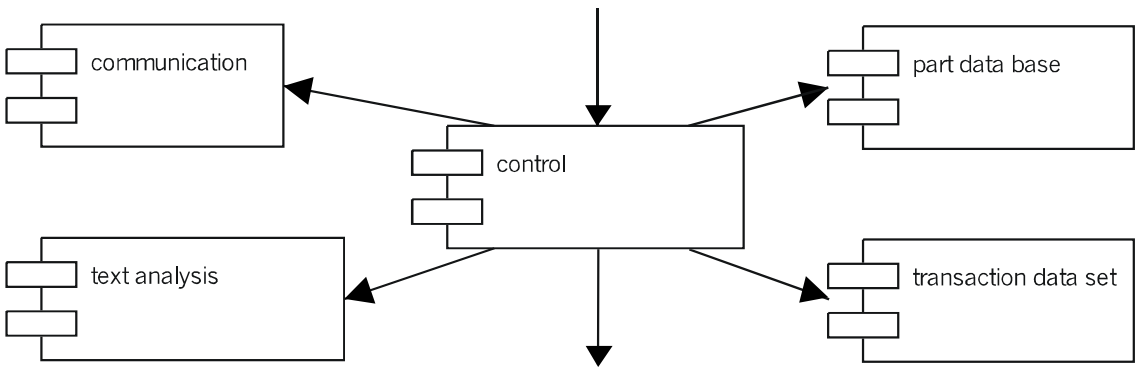

Figure 2: Service and control components

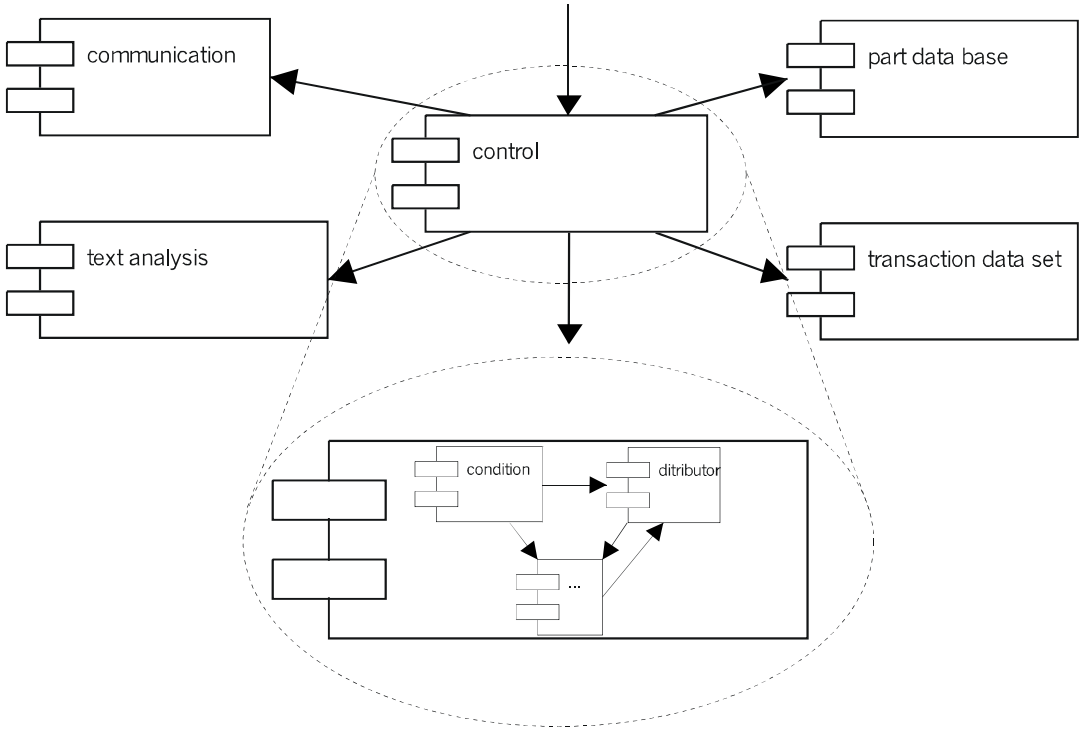

Figure 3: Composed control component

IV. HOTAGENT TOOLS

The *HOTAgent* [21] development environment is presented more elaborated in [20]. It is a component framework to construct agents for electronic commerce. The framework provides a special set of pre-fabricated components to facilitate the construction of agents. The agents could undertake the task of doing routine work, as described above. The *HOTAgent* development environment offers different tools to develop components, to test components, and to compose agents using components. During the development of *HOTAgent* the concept of component technology is consistently used. All developed tools have a similar GUI (Graphic User Interface) to ensure that the developer feels familiar with all tools if he knows one. The different tools are all integrated in each other; so it is, e.g., possible to create a test case while creating a new component.

The simplest way to create new GUI components is to use the *HOTAgent PATTERN COMPONENT* editor (under development). The first step is to place graphical elements into a workspace. The elements have different shapes and colors. The second step is to assign behavioural patterns to the elements, e.g., an element may be moved on a line or plane. The editor creates all necessary classes for the new component on its own. Using this editor, new control mechanisms for a user interface may be created.

The *HOTAgent COMPONENT* editor [22] is an all-purpose editor. It is good for constructing components. It supports the programmer by creating new components using other existing components. They may be placed on a workspace and may be connected with each other and new component requires and provides ports. Using this editor, a programmer may construct components without knowing very much about the underlying component model.

Using the constructed components, it is possible to build complete applications. This is done with the *HOTAgent ASSEMBLY* editor [22, 23]. New applications may be created using existing components similar to the method in the *HOTAgent COMPONENT* editor. *HOTAgent ASSEMBLY* allows easy and clear composition of agents for E-business. It is also possible to maintain composed agents by replacing whole components or changing connections between components.

The *HOTAgent TEST* [24] tool tests components. The tool supports the component developer and component assembler by testing components as black boxes using defined requires and provides ports. *HOTAgent TEST* is suitable for component integration testing. The test tool enables the tester to specify a test case in a visual way. In addition, *HOTAgent REGRESSION* supports regression tests. The tool allows the execution of several test cases at the same time and the inspection of a possibly faulty test case.

The *HOTAgent VISUALIZE* tool [25] provides a three-dimensional visualization of component programs' runtime behaviour based on dynamic analysis. Communication between components is dynamically analyzed and gathered data may be filtered. The visualization is three dimensional and the focus may be chosen: either the whole program execution may be viewed at the same time or the developer may zoom into special regions. *HOTAgent VISUALIZE* is also useful for testing component programs.

V. About the HOTxxx projects

The research described here is part of the ongoing *HOTxxx* project ([10, 12]), started early in 1995. Considering its development it will be possible to motivate the research reported here.

Originally, the work started in research and development of a flexible document management system, *HOTDoc* ([2, 3]). At first, Buchner´s work was intended as a study advancing *OpenDoc*, *OLE*, and *OOE*. The basic idea was to have an *automated electronic document folder* into which a variety of *document parts* may be inserted, even hierarchically nested, viz., editable text, multimedia contributions, a running clock, an operable spreadsheet (*HOTCalc*), a linking mechanism between parts (*HOTScript*) allowing, e.g., to connect spreadsheet data with a business graphic (*HOTDraw*), an interaction facility with buttoned parts, to mention some initial and, in our opinion, interesting features. The system was architectured and realized in form of a Smalltalk framework.

1998 it became obvious that the framework architecture is a limiting factor for further extensions (although, in principle, Smalltalk provides powerful dynamic extension capabilities). Insertion of *HOTScript* instantiations – in its application cases really considered to be individual "components" – and early Internet access extensions exhibited difficulties for integration, especially due to its origin in simultaneous development in students´ thesis work. Further, powerful and complex *HOTxxx* components recently completed (*HOTSimple* – a simulation and planning tool in form of a rather advanced spreadsheet – [15]) and presently under work (*HOTFlow* – an Internet-based workflow control system – [7]; and *HOTAgent* – a routine processing extension, see section 4 –), again, supported this unsatisfactory observation.

During the E-business ad-hoc hype, 1998 – 2001, *HOTDoc*, with these components included, also became a part of *MALL2000* – commercial name *WEST-EAST TRADE* – ([8, 11, 19]) with specific extension for a XML-based communication component for Internet access [16] to support a B2B community of business correspondents.

Component technology in general became a promise to overcome integration difficulties. Under the same basic idea of *HOTDoc*, parts (now components) should be possibly hierarchically nested. Research on hierarchical composition facilities with an adequate concept to master complexity by system partitioning down from the top to minimal components as discussed in this paper shows up to be a solution.

The approach we are presently looking for in further development of *HOTxxx* is already sketched in Figure 1c; archiving information, under the metaphor of a folder, is bound to a document exchanged under workflow control between B2B-correspondents; *one* business activity covered in *one* folder with, for the situation being, a specifically and for a limited period of time established E-business community, organized for activities as, e.g., negotiation of as business opportunity, treatment of an unique business event, etc.

VI. Related work

Now back to the general discussion of software components as a means to enhance E-business communities!

A short notice of Sandholm [26] gives some ideas of components and negotiation techniques in a different application context. He reviews six techniques to make interaction between agents and their components more efficient. He only talks about communication and not how to partition agents into components.

Baster et al. [1] describe the process of building a component end-user computing architecture to bridge the gap between domain expertise and technology skills. They propose to construct a component framework with six layers for task specific components: presentation layer, business process, business components, foundation components, persistent data, and computing infrastructure. One of the main problems is to identify a component candidate. Therefore development experience and sophisticated tools are required. *HOTAgent* supports the user by finding components. Furthermore Buster et al. explain, components need to work consistently which must be guaranteed by the component framework. It is also necessary to leverage the skills of business and technology groups to create good applications.

Hong et al. [13] describe an architecture for software agents using four different types of components: sensing components, communicating components, executing components, and control components. They define a grammar how to combine these types of components. Our approach, *HOTAgent*, is more easy because we have only service components, which include the first three component types of Hong et al., and control components. The structure of control components is similar complex as our control components.

Kinikoglu and Yudav [14] give some more motivating arguments for our work. Their architecture proposal includes involvement of agents dscribing some more general, interesting service components. What they call *Software Components Agents Mediator SWCAM* is related to our control component, however not so clearly distinguished from service components and not the central connection point for them.

"... there will be thousands of open-market components available ..." is a central remark in a paper by Erdur and Dikanelli [5]. We think that our software engineering oriented research will help to master the complexity on partitioning and decomposing components supported by appropriate tools.

VII. Summary

Component technology still lacks some guidance for appropriate structuring, partitioning, and composing of involved components. In the scope of the *HOTxxx* project we discuss some architectural attempts for enhanced solutions useful to support E-business communities. One guiding principle for component-based development may be found in the *workflow process* of an E-business application, i.e., a form of linear processing as discussed in subsections 3.A and 3.B.

Another guiding principle may be differentiating between *service and control components* (subsection 3.C). To master complexity *hierarchical composition* of components may be useful (subsection 3.D).

In the development environment *HOTAgent* (section 4; described in more details elsewhere [20 - 25]) a set of development tools and pre-fabricated components are made available for the construction of E-business *agents*, i.e., software artifacts to achieve business transactions between communicating persons/institution.

In a running example, *Repair Service System RSS*, we demonstrated our architectural approach in an application following our guiding principles for component system design.

REFERENCES

[1] Greeg Baster, Prabhudev Konana, and Judy E. Scott. Business components: a case study of bankers trust Australia limited. Communication of the ACM, May 2001.

[2] Jürgen Buchner. HotDoc, ein flexibles System für den kooperativen Aufbau zusammengesetzter Dokumentstrukturen. Doctoral thesis, Department of Computer Science, Darmstadt University of Technology, March 1998.

[3] Jürgen Buchner. HotDoc, a framework for compound documents. ACM Computing Surveys, vol. 32, number 1, March 2000 (access through ACM Digital Library).

[4] Thorsten Clausius. Komponenten zur Konstruktion von Agenten. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, Juli 2001.

[5] Riza Cenk Erdur and Oguz Dikenelli. A Multi-Agent System Infrastructure for Software Component Market-Place: An Ontological Perspective. ACM SIGMOD Record, Vol. 31, No. 1, pages 55-60, March 2002.

[6] Anke Giesl. Entwicklung von Komponenten-Entwicklungsumgebungen. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, November Januar 2001.

[7] Daniela Handl. HotFlow: E-Commerce processes from a language/action perspective. In: Dilip Patel et al (Eds): OOIS 2000. 6th International Conference on Object Oriented Information Systems (Proceedings), pages 95 – 101. 2001.

[8] Hans-Jürgen Hoffmann. MALL2000+, a vision for a virtual marketplace for businessmen. In J.-Y. Roger et al. (eds.): Advances in Information Technologies, The Business Challenge, IOS Press, pages 247-254, 1998.

[9] Hans-Jürgen Hoffmann. “Less is more” in B2B. In Proc. SSGRR 2001, Intl. Conf. Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, L`Aquila, Italy, August 2001, CD-ROM.

[10] Hans-Jürgen Hoffmann. Unterstützen elektronischer Geschäftsprozesse: Das HOTxxx-Projekt. In P. Horster (ed.) – Elektronische Geschäftsprozesse, it-Verlag, pages 227-241, 2001.

[11] Hans-Jürgen Hoffmann and Daniela Handl. Document exchange as a basis for business-to-business co-operation. In J.-Y. Roger et al. (eds.) – Business and Work in the Information Society: New Technologies and Applications, IOS Press, pages 325 – 331, 1999.

[12] Hans-Jürgen Hoffmann et al. Mall2000 Home page. http://www.informatik.tu-darmstadt.de/PU/projekte/MALL2000, accessed August 24, 2002.

[13] Liu Hong, Zeng Guanghou, and Lin Zongkai. A Construction Approach for Software Agents Using Components. ACM SIGSOFT Software Engineering Notes, 24(3): pages 76 – 79, May 1999.

[14] Pinar Kinikoglu and Surya B. Yadav. An Agent Based Architecture for Componet-Based Software Development, undated: http://hsb.baylor.edu/ramsower/acis/papers/kinikogl.htm, accessed August 13, 2002.

[15] Thomas Kunstmann. Rechnergestützte Simulation und Planung auf der Grundlage von Tabellenkalkulation. Doctoral thesis, Department of Computer Science, Darmstadt University of Technology, February 2002; ISSN 1435-6260.

[16] Stefanie Levasier. Ein XML-Standard für dynamische Arbeitsablaufsteuerungen. Diploma thesis, Department of Computer Science, Chair Programming Languages and Compilers, Darmstadt University of Technology, April 2001.

[17] Chris Lüer and David S. Rosenblum. Wren - An Environment for Component-Based Development. Technical report, Department of Information and Computer Science, University of California, Irvine, September 2000.

[18] Chris Lüer and David S. Rosenblum. Wren - An Environment for Component-Based Development. In Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 207 – 217, September 2001.

[19] Mall2000 project consortium, EU project INCO 977041. Home page. http://www.we-trade.org, accessed August 16, 2002.

[20] Ludger Martin. Visual Development Environment Based on Component Technique. In Proceedings IEEE Symposia on Human-Centric Computing Languages and Environments, pages 346 – 347, September 2001.

[21] Ludger Martin. HOTAgent homepage. http://www.gkec.informatik.tu-darmstadt.de/HotAgent/ accessed November 2002.

[22] Ludger Martin. Visual Composition of Components. The 6th IASTED International Conference Software Engineering and Applications, pages 501 - 508, November 2002.

[23] Ludger Martin. HotAgent Component Assembly Editor. In Schneider, Jean-Guy and Markus Lumpe (editor): Proceedings Workshop on Component Composition Languages, page 25 – 32, September 2001.

[24] Ludger Martin. Visual Component Integration and Regression Test. ICSR7 2002 Workshop on Component-based Software Development Processes, April 2002.

[25] Ludger Martin, Anke Giesl, and Johannes Martin. Dynamic Component Program Visualization Working Conference on Reverse Engineering, pages 289 - 298, October 2002.

[26] Tuomas Sandholm. Agents in Electronic Commerce: Component Technologies for Automated Negotiation and Coalition Formation. In Proceedings of the Third International Conference on Multi Agent Systems, 1998.

[27] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.