TRANSFORMATION OF LEGACY BUSINESS SOFTWARE INTO CLIENT-SERVER ARCHITECTURES

Thomas Rauber
Department of Computer Science, University of Bayreuth, Germany

Gudula Rünger
Department of Computer Science, Chemnitz University of Technology, Germany

Keywords: Legacy software, incremental transformation, transformation toolset, client server.

Abstract: Business software systems in use contain specific knowledge which is essential for the enterprise using the software and the software has often grown over years. However, it is difficult to adapt these software systems to the rapidly changing hardware and software technologies. This so-called legacy problem is extremely cost intensive when a change in the software itself or the hardware platform is required due to a change in the business processes of the enterprise or the hardware technology. Thus, a common problem in business software is the cost effective analysis, documentation, and transformation of business software. In this paper, we concentrate on the transformation issue of software systems and propose an incremental process for transforming monolithic business software into client-server architectures. The internal logical structure of the software system is used to create software components in a flexible way. The transformation process is supported by a transformation toolset which preserves correctness and functionality.

1 INTRODUCTION

Software systems implement business processes of an enterprise and capture essential expert knowledge built up over decades. Due to changes in software and hardware technologies, many enterprises need to consider a change of their business software. Also, the process of changing within an enterprise requires its business software to be adapted or enriched with additional functionality. Thus, enterprises have to face the fact of modernizing their software systems in order to guarantee full functionality. The changes may concern aspects of security, performance, usability, maintainance, as well as interactions with other software systems or cost of the hardware platform used. Often the software also lacks features such as distributed components, adaptivity to heterogeneous platforms or scalability, whose current importance has been unknown when the software systems were built. However, it is extremely difficult to incorporate these feature in today's running business software. The integration of distributed and adaptive behavior may require a complete restructuring of a software system and might effect many aspects of an enterprise and its business process.

In this paper we consider the generation of distributed business software starting from a monolithic legacy software system. The implementation of effective distributed software has been studied in many different ways, including management decision strategies for software replacement, business process modeling or software design principles comprising framework approaches (Lewandowski, 1998), distributed objects (Emmerich, 2000), distributed components (Emmerich, 2002), and middleware integration (Bernstein, 1996). Distributed software design for the legacy problem has been treated with wrapper- and middleware approaches. Both approaches do not meet the requirements of highly complex business software in use. Therefore, distributed and adaptive aspects are lacking in most business software systems although managers and software experts are well aware of these deficiencies.

The contribution of this paper lies in the area of transforming business software systems into distributed software systems with extended functionality. The specific approach comprises the following novel issues in treating legacy software:

Rauber T. and Rünger G. (2007).
TRANSFORMATION OF LEGACY BUSINESS SOFTWARE INTO CLIENT-SERVER ARCHITECTURES.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - DISI*, pages 36-43
DOI: 10.5220/0002348400360043

- The transformation of legacy software into distributed software takes the internal logical software structure into account. This results in an explicit representation of the modular structure which can be exploited as distributed component structure.
- The modular structure can be used to enrich the functionality by including configurative features and explicit workflow integration. This is supported by the creation of configuration files or explicit workflows, depending on the specific needs.
- The transformation of software is performed in an incremental transformation process organized in several steps. The incremental approach enables a separation of different concerns such as logical and modular structure, component building and efficiency. Also the transformation process allows interactive decisions during the transformation process concerning the distributed target architecture and/or the logical configuration level.
- The incremental transformation implements aspects from model driven software development. The model driven architectures (MDA) approach proposes an incremental development of distributed software for different middleware platforms. This paper applies the idea of MDA to legacy software.
- The transformation process is supported by a transformation toolset for transforming monolithic legacy software into client-server architectures. The transformation toolset uses compiler based methods which is possible due to separation of different concerns and the incremental approach. Specification languages are used to specify module and component structures on several levels.

The entire transformation approach brings together recent technology from software engineering, distributed systems, compiler design, and software development of modern business software. The design and implementation of the transformation process proposed in this paper is performed as part of the TransBS[1] project by researcher and practitioners in these areas including software development companies.

The rest of the paper is organized as follows: Section 2 presents background information about distributed systems, workflow technologies, and MDA. Section 3 discusses requirements of effective software transformation. Section 4 outlines the transformation process and the architecture of the transformation toolset. Section 5 concludes.

[1]http://www.transbs.de

2 BACKGROUND

The design and implementation of the transformation process combines several different areas, including distributed systems, component based software design, transformation approaches, MDA, and workflow technology.

2.1 Component-based Software Design

Different approaches have been proposed to support the communication between distributed objects in heterogeneous distributed environments. An example is the Common Object Request Broker Architecture (CORBA) defined by the Object Management Group (OMG). For the development of solutions that are independent from a specific programming language, an Interface Definition Language (IDL) can be used which allows the definition of interfaces of objects and data structures to be exchanged. Based on CORBA and the remote procedure call (RPC) mechanism, the remote method invocation (RMI) of Java has been developed. A distributed execution of the Component Object Model (COM) can be obtained by the DCOM extension. These approaches are useful for the implementation of the distributed framework into which the generated components can be mapped.

Distributed *interacting components* can be realized with different systems:

- COM+ from Microsoft supports the execution of COM components;
- Enterprise Java Beans (EJB) from Sun allows the distributed execution of Java components;
- the CORBA component model (CCM) extends the EJB approach to other programming languages.

These systems provide a general framework for distributed components, but they do not provide support for a specific application area. Moreover, these approaches are not primarily intended for the distributed execution of workflows and there is no direct support for such an execution.

The development of *component-based software systems* is considered by Component-based Software Engineering (CBSE) approaches (Crnkovic and Larsson, 2002). CBSE approaches are mainly based on component approaches like COM+, EJB or CORBA, and provide techniques for developing modular software systems. These approaches often comprise techniques for an incremental development of software systems (Mehta and Heineman, 2002) and for composing existing components according to given requirements and specification (Cao et al., 2005; Zhao et al., 2003). Techniques of Generative Programming (Czarnecki and Eisenecker, 2000) and Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) can also be integrated. The re-use of software components is especially important for long-running projects. CBSE techniques can be used as a general framework to support the distributed execution of workflows.

2.2 Business Software Transformation

The transformation of legacy software has been considered by many research groups. Most approaches concentrate on the transformation of legacy software into modular or object-oriented systems or on the extraction of the business logics. New approaches also consider distributed solutions, e.g. by providing middleware for data integration (Akers et al., 2004; Menkhaus and Frei, 2004). For distributed systems, performance aspects play an important role (Litoiu, 2004), since additional latency and transfer times may be necessary, e.g., when replacing legacy software by web services using protocols like SOAP (Simple Object Access Protocol) (Brunner and Weber, 2002). For the transformation of software systems, approaches like DMS (Baxter et al., 2004) have been developed and have been applied to large software systems (Akers et al., 2004). Graph-based visualization tools are often useful for the transformation (Balmas, 2004). Clustering methods can be applied for identifying modules of the legacy software and will be one choice for providing modules in the TransBS project (Al-Ekram and Kontogiannis, 2004; Lung and Zaman, 2004; Andritsos and Tzerpos, 2003).

Although there are many different approaches, there exists no generally accepted method for the incremental transformation of software systems. An important reason for this lies in the fact that there are many different distributed platforms like CORBA or EJB that cannot be combined in an arbitrary way. Therefore, a distributed realization often requires a new implementation of the business logics.

The Model Driven Architecture (MDA)[2] approach (Siegel, 2005) addresses this problem and uses a model-based approach for the step-wise generation of distributed, component-based software. The step-wise generation starts with an UML specification of the static interfaces and the dynamic behavior of the components in a platform-independent model (PIM). A series of mappings that are given as UML extensions like CORBA profiles or Enterprise Application Integration (EAI) profiles leads to the generation of a platform-specific model (PSM). Based on this, MDA tools are used for generating program code for a specific platform.

[2]http://www.omg.org/mda

Another important aspect of distributed business software is the transparency of the distributed execution and the integration of explicit workflow definitions. The usual feature of transparency of distributed execution seems to be un-appropriate for business software systems, since expensive remote requests may be generated if there is no knowledge of the distributed nature of execution.

The workflow technology for the formal description of the execution of business processes constitutes an important contribution for developing flexible business software. Many workflow management systems have been realized, but few of these consider distributed workflows. The Workflow Management Coalition (WFMC)[3] defines standards for workflow architectures also comprising aspects of a distributed execution. A direct integration of workflows in the technology of distributed components has not been considered often (Emmerich, 2002). The TransBS project described in this paper addresses this deficiency and combines technologies for the transformation of legacy software, distributed platforms and workflow execution.

3 TRANSFORMATION REQUIREMENTS

The transformation of an existing business software system *S* into a distributed software system *DS* can be done in many different ways, depending on the requirements on the resulting software system *DS*. In our approach the requirements are determined by the goal to keep the given functionality but to increase the flexibility of the code so that it can be adapted to different distributed or heterogeneous platforms and enterprise requirements.

We concentrate on business software as it is used in services and commercial enterprises, but our approach is not limited to this line of business. The typical method is to modify and to extend big standard software packages such that the needs of the specific company are met. This is the most cost effective way to create specific business software but still requires a great deal of programming effort since typical software packages in use are of monolithic character. A more flexible standard software package could ease software adaptation for specific enterprises tremendously. This leads to the following main requirements for a transformed software product *DS*.

[3]http://www.wfmc.org

- The resulting software system *DS* should be executable on different distributed or heterogeneous platforms in an efficient way. This includes that remote method or function calls are made explicit in contrast to most distributed software, but that the software can also be adapted easily to a different distributed structure. We call this property *hardware flexibility*.
- The resulting software system *DS* should be flexible such that business software for a specific enterprise with reduced or enriched functionality can be built easily. This kind of *software flexibility* is another goal for the resulting software system.
- The transformation into a software system with hardware and software flexibility should preserve *functionality* and *correctness* of a given system *S*.
- The resulting software system should be *efficient* in the sense that remote method and function calls are implemented with low communication and packing overhead.

To meet these requirements for the resulting software system we propose a transformation process which transforms a given software system *S* into a flexible distributed software system *DS*.

- An intermediate representation given as auxiliary program structure is used to support the transformation process. This representation is language independent and appropriate for the transformation goals. The auxiliary program structure is a hierarchical structure which captures the static software structure. The highest level of the hierarchical structure is the coordination structure. This level calls specific modules which encapsulate the original functionality and which can exhibit a further hierarchical structure. The hierarchical program structure is the basis for a module structure which decomposes the given monolithic program code. This module structure is exploited to create
  a) a flexible program structure which can be used for enterprise specific software (software flexibility) and
  b) for deciding about explicit distributed heterogeneous execution (hardware flexibility).
- Starting with this program representation an incremental transformation process transforms the given software system *S* into a modular system and then into specific distributed software system. The transformation steps are syntax based and guarantee correctness. Interactive decisions guide the transformation process.
- Efficiency in the resulting software system and its specific derivations is achieved by the possibility to design the specific functionality and distributed execution in an explicit way.

To support the transformation process we propose a transformation toolset to interactively transform software. The next section describes this toolset in more detail. The transformation process starts with the input program and the specification of the module structure, which is essential and has to be provided by software experts, e.g. by a clustering method and interactive design. The software experts are also responsible for making the business processes explicit by using a workflow description as described in the next section.

4 TRANSFORMATION PROCESS

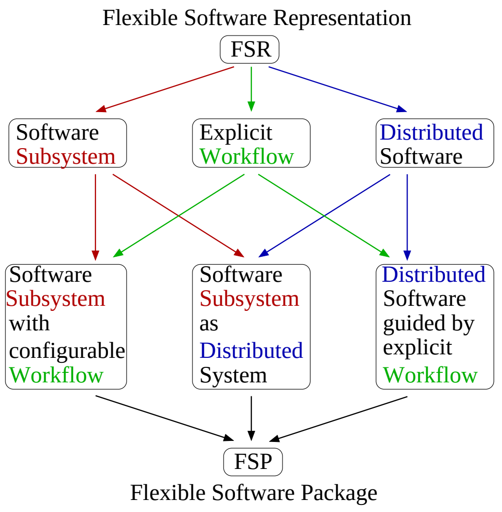
In this section, we describe software modes of the resulting software system, give an overview of the transformation process, and sketch the different components of the framework into which the transformed components are included.

4.1 Software Modes

The transformation process for creating modular flexible distributed software from a given monolithic software package is based on an intermediate software representation which we call *Flexible Software Representation* (FSR). This representation is the starting point for a stepwise transformation of the given software package *S* into a software package with the same execution behavior but different properties concerning the mode of execution.

We distinguish three software modes:

- The *Software Subsystem mode* (SSu) has an explicit modular structure derived from *S* from which subsystems can be built easily. SSu mode systems are executed on the same platform in the same way as *S*. SSu allows the selection of subsystems representing parts of the execution behavior of *S*. The goal is to extract specific functionalities provided by *S* as a separate subsystem.
- The *Explicit Workflow mode* (ExW) describes specific business processes by explicit workflows (in a workflow description language such as XML). ExW captures the business logics of *S* in the form of an explicit workflow. This is the basis for a modification of the workflow to adapt the resulting system to the needs of different enterprises. It also allows the generalization of the workflows to *configurable workflows* that can easily be adapted to different situations. ExW is executed on the same platform in the same way as *S*.



Flexible Software Representation
FSR
Software Subsystem
Explicit Workflow
Distributed Software
Software Subsystem with configurable Workflow
Software Subsystem as Distributed System
Distributed Software guided by explicit Workflow
FSP
Flexible Software Package

Figure 1: Transition diagram for Flexible Software Representation (FSR).

Table 1: Software parts for the three specific software modes SSu, ExW, DS resulting from FSR.

	Flexible Software Representation – FSR		
	Software Subsystem SSu	Explicit Workflow ExW	Distributed Software DS
upper level	coordination program	Workflow description	component cooperation
lower level	Modules/ Functions	Modules	Components
static support	Programming Environment	Workflow Engine	Distributed Runtime System

- The *Distributed Software mode* (DS) has an explicit structure of cooperating components which can be mapped onto a specific distributed system. The structure of DS is based on the modular structure of *S* as expressed by SSu. Additionally, DS allows the mapping of modules to the entities of a distributed system in the form of components. The execution behavior of DS can be expressed implicitly within the components or by explicit workflows.

All three representation modes are created from the FSR. A mixture of these three representation modes results in software representations with enriched properties. The coarse structure of the transformation from the FSR into SSu, ExW, or DS and then into the mixed formats is depicted in Figures 1. For example, the SSu style with explicit workflows leads to a software package which can use *configurable workflows* to easily specify specific software subsystems for a specific enterprise. A combination of ExW and DS, on the other hand, describes an execution mode with explicit workflows that are executed by components in a distributed system. All three features together result in the most flexible software representation called *Flexible Software Package* (FSP), from which a distributed software subsystem can easily be specified by configurable workflows.

The advantage to distinguish the three software modes and to transform the FSR in incremental steps is that not all software products have needs for all features. Due to efficiency reasons only those features which are really needed should be incorporated, since more advanced software features have higher computing demands.

The transformation of the FSR into SSu, ExW, DS, or their mixed forms is done on the intermediate representation level. The intermediate representation consists of an upper level and a lower level. The upper level describes the cooperation and coordination of software parts, thus specifying the possible dynamic behavior of the software. The lower part contains software parts incorporating specific functionalities of the business software. Both together specify the software in an internal format. The transformation process translates both levels separately into a software representation including the newly created features. Upper and lower level constitute the final software (still in an intermediate format) and requires additional static software products to be executable. Upper level, lower level, and static software products for SSu, ExW, and DS are summarized in Table 1.

4.2 Transformation Decisions

The entire transformation is organized in an incremental transformation process including (interactive) transformation decisions. Figure 2 illustrates the coarse structure of the transformation process and the decision tree. One path along the transformation direction corresponds to one specific transformation process producing one business software system using a specific software mode. These software systems can be implemented based on different techniques like EJB or Corba. The transformation process is performed in the flavor of the MDA approach.

The transformation process starts with a *specification level* that exposes the logical structure of the given software system and partitions it into code fragments that are intended to be used as modules in later transformation steps. Currently, the specification level cannot be performed fully automatically. It can

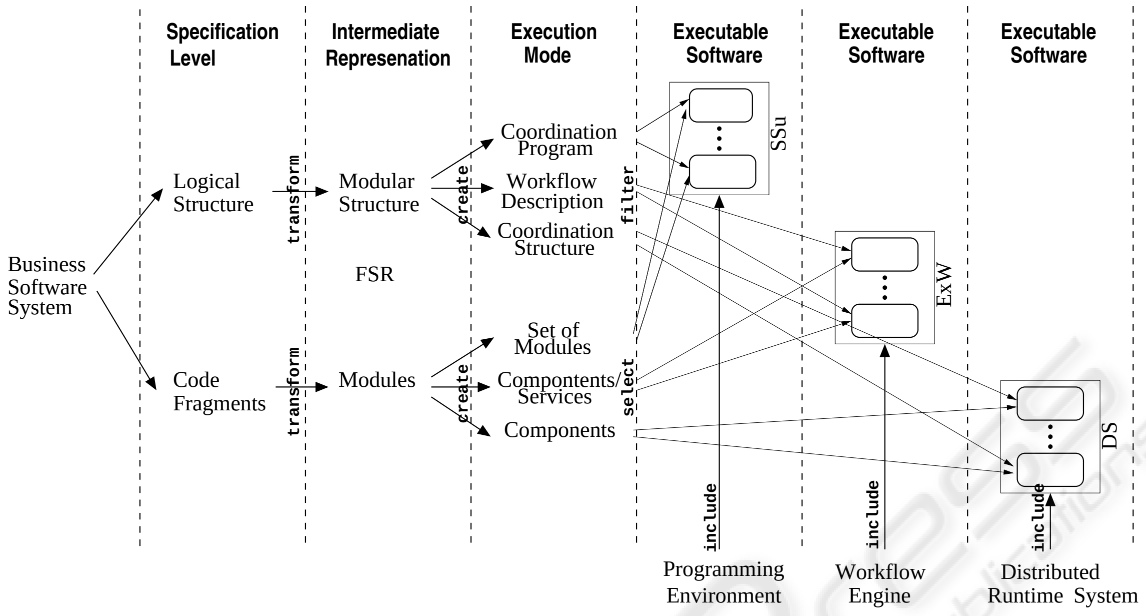


Figure 2: Transformation decision and transformation process.

be supported by clustering techniques and data dependence analysis tools, but still requires manual interaction. In the *intermediate representation*, the logical structure is represented explicitly and the code fragments are represented as modules. Based on the modular structure, the *execution mode* creates a suitable description depending on the software mode selected. This can be a coordination program for the SSu mode, a workflow description for the ExW mode, or a coordination structure for the DS mode. This representation is the basis for the filter transformation which determines which of the components should be prepared for a remote execution. Correspondingly, the modules are transformed into executable modules for the SSu mode or components for the ExW or DS modes. Depending on the subsystem selected, a selection of the modules or components is built and included into the framework for the specific mode.

4.3 Resulting Software Systems

Figure 3 shows the software system generated in the SSu mode. The subsystem has been selected by the transformation process and has been included in the form of modules. The control is provided by a coordination program that contains a part of the functionality of the software package S. The modules can issue data accesses that can be directed to the local database or can be directed to non-local databases in the form of remote accesses. Remote accesses are supported by a data consistency service that is responsible for keeping the database entries in different databases consistent. Further support for remote accesses is provided by a security service and a communication service. All three services are provided as a static part of the framework into which the modules are embedded.

Figure 4 depicts the software system generated for the ExW mode. The difference to the SSu mode consists in the use of a workflow engine which is able to execute workflows comprising the business logics of the software system S. The modules of S are transformed into components that are activated by the workflow engine according to the workflow. Each component can issue data accesses which are performed as local or remote accesses similar to the SSu mode. The workflow engine is able to activate remote components via a remote execution service that is provided as a static part of the software system. Remote execution is managed via the communication service.

Figure 5 shows the distributed software system DS generated by the transformation system for the case that the execution is controlled by a coordination component which realizes a fixed workflow in the simplest case. This component orchestrates the execution of the components. All data accesses of the components are performed via the coordination layer which transfers the accesses to the corresponding services provided by the DS framework. Remote execution of components can be performed via the remote executor service. All remote accesses are done via the communication service that can be performed in different ways.

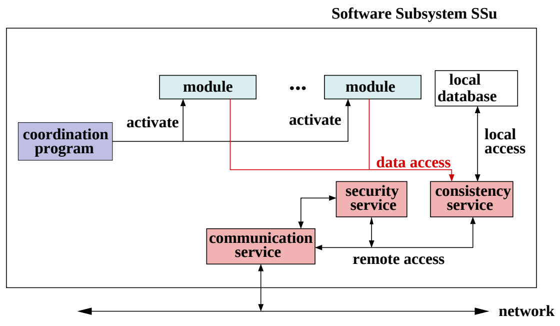


Figure 3: Software Subsystem SSu generated by the transformation process.

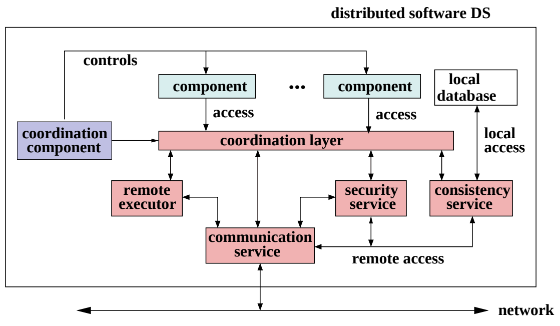


Figure 5: Distributed software system DS generated by the transformation process using a coordination component.

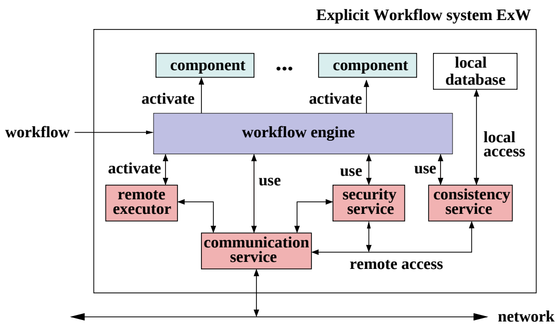


Figure 4: Explicit Workflow system ExW generated by the transformation process working with a workflow engine.

4.4 Transformation Toolset

The incremental transformation process is performed by an interactive toolset providing compiler-based transformation tools and interactive decision tools. The transformation tools work on internal representations of the software product to be transformed. The toolset comprises the following functionalities:

- *Build-intermediate-representation* builds a hierarchical graph structure from a given description of the logical structure in a specification language.
- *Create-coordination-program* creates a complete description of a coordination program calling and coordinating the modules of the software.
- *Create-workflow* creates a complete description of an explicit workflow.
- *Create-coordination-structure* creates a complete description of the most general distributed coordination of components.
- *filter-for-subsystem* extracts a subsystem based on a description of the required subsystem. This description can be provided interactively based on the representation of the coordination structure.
- *filter-for-workflow* selects a workflow which describes the functionality of the resulting business software. Only instances of this workflow will be executable by the resulting business software.
- *filter-for-distribution* creates a representation of the upper level of the distributed software which will be incarnated in the final software. This is a restriction of the most general distributed execution. The restriction is specified by the software expert guiding the transformation process.

The results of the incremental transformation phase is a complete description of the business software to be created, its modes and its functionality in the FSP format. This description is created in an internal format on which transformations can be executed. The final executable business software is created by the following compiler tools:

- *Compile-into-subsystem*,
- *Compile-workflow-subsystem*, and
- *Compile-distributed-system*.

Also the toolset comprises the static programming environments for SSu, ExW, and DS.

4.5 Status of the Implementation

The transformation environment is developed in the TransBS project comprising tool developers and developers of business software products. Both, the design of the transformation process and environment as well as the resulting business software modes, is done incrementally with strong cooperation between the different groups such that the final tool will meet the requirements for creating modern business software. The design of the requirements of the resulting transformed software is finished and the coarse structure of the transformation environment has been developed as described in this paper. The next step is to implement the complete toolset and to conduct a large scale validation using a complex business software system provided by a participating software company. Parts of the transformation process and runtime systems are currently being implemented. This includes prototype versions of a transformation tool with upper and lower level leading to a complete frame program for parallel or distributed (sub)software, a workflow management system for executing distributed workflows specified in XPDL (for the ExW mode), and a distributed runtime system for modular applications in distributed environments. Another important issue is the integration of those prototype versions which is done concurrently with testing the resulting business software systems created by the transformation process. Feedback loops in the implementation process will adapt the prototype versions and its integration. Also, the structure of the transformation, select, filter, create, and include functions has to be refined to yield a flexible and interactive transformation environment.

5 CONCLUSIONS

We have proposed an interactive toolset to incrementally transform existing monolithic business software into a modular system that can be distributed and workflow-based. This leads to a flexible software system that can be adapted to the characteristics of a specific environment and meets the needs of software developers and specific end user enterprises.

ACKNOWLEDGEMENTS

The research presented is funded by the German ministry BMBF 01 ISF 10 A/B/C.

REFERENCES

Akers, R., Baxter, I., and Mehlich, M. (2004). Re-Engineering C++ Components Via Automatic Program Transformation. In *Proc. of ACM Symposium on Partial Evaluation and Program Manipulation*, pages 51–55. ACM Press.

Al-Ekram, R. and Kontogiannis, K. (2004). Source Code Modularization Using Lattice of Concept Slices. In *Proc. of Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 195–203.

Andritsos, P. and Tzerpos, V. (2003). Software Clustering based on Information Loss Minimization. In *Proc. of Working Conference on Reverse Engineering (WCRE2003)*, pages 334–344.

Balmas, F. (2004). Displaying dependence graphs: a hierarchical approach. *J. of Software Maintenance and Evolution: Research and Practice*, 16(3):151–185.

Baxter, I., Pidgeon, C., and Mehlich, M. (2004). DMS: Program Transformations for Practical Scalable Software Evolution. In *Proc. of the 26th Int. Conf. on Software Engineering*, pages 625–634. IEEE Press.

Bernstein, P. (1996). Middleware - A Model for Distributed System Services. *Commun. of the ACM*, 39(2).

Brunner, R. and Weber, J. (2002). *Java Web Services*. Prentice Hall.

Cao, F., Bryant, B., Burt, C., Raje, R., Olson, A., and Auguston, M. (2005). A Component Assembly Approach Based On Aspect-Oriented Generative Domain Modeling. *Electronic Notes in Theoretical Computer Science*, 114:119–136.

Crnkovic, I. and Larsson, M. (2002). Challenges of component-based development. *Journal of Systems and Software*, 61(3):201–212.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

Emmerich, W. (2000). *Engineering Distributed Objects*. John Wiley & Sons.

Emmerich, W. (2002). Distributed Component Technologies and their Software Engineering Implications. In *Proc. of Int.Conf.Software Engineering*, pages 537–546. ACM Press.

Kiczales, G., Lamping, J., A.Mendhekar, Maeda, C., Lopes, C., Loingtier, J.-M., and Iwin, J. (1997). Aspect-oriented programming. In *Proc. Conf on Object-Oriented Programming (ECOOP'01), LNCS 1241*, pages 327–353.

Lewandowski, S. (1998). Frameworks for component-based client/server computing. *ACM Comput. Surv.*, 30(1):3–27.

Litoiu, M. (2004). Migrating to Web Services: a performance engineering approach. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:51–70.

Lung, C. and Zaman, M. (2004). Applying Clustering Techniques to Software Architecture Partitioning, Recovery and Restructuring. *Journal of Systems and Software*, 73(2):227–244.

Mehta, A. and Heineman, G. (2002). Evolving Legacy System Features into Fine-Grained Components. In *Proc. of Int. Conf. on Software Engineering*, pages 417–427.

Menkhaus, G. and Frei, U. (2004). Legacy System Integration using a Grammar-based Transformation System. *CIT - Journal of Computing and Information Technology*, 12(2):95 – 102.

Siegel, J. (2005). Why use the Model Driven Architecture to Design and Build Distributed Applications. In *Proc. of Int.Conf.Software Engineering*, page 37. ACM Press.

Zhao, W., Bryant, B., Gray, J., Burt, C. C., Raje, R. R., Auguston, M., and Olson, A. (2003). A Generative and Model Driven Framework for Automated Software Product Generation. In *Proc. 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, pages 103–108.