International Workshop on Large-Scale Software Composition

Rudolf K. Keller
Département IRO
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal, QC, H3C 3J7, Canada
keller@iro.umontreal.ca

Bruno Laguë
Quality Engineering & Research
Bell Canada
1050 Beaver Hall Hill
Montréal, QC, H2Z 1S4, Canada
bruno.lague@bell.ca

Reinhard Schauer
Département IRO
Université de Montréal
C.P. 6128, succ. Centre-ville
Montréal, QC, H3C 3J7, Canada
schauer@iro.umontreal.ca

Content

This report summarizes the International Workshop on Large-Scale Software Composition held at the University of Vienna, Austria, on August 28, 1998 in conjunction with the Database and Expert Systems Applications (DEXA'98) conference. An overall forty people attended the workshop consisting of seven presentations and plenary discussions. In the following, we outline the presentations and subsequent discussions in the four workshop sessions, which included Setting the Stage, Component Modeling, Migration towards Components, and Component-based Modelling of Distributed Systems. The workshop report can be found at <http://www.iro.umontreal.ca/labs/gelo/iw-lssc98>.

Background

Large organizations throughout industry face immense problems when they have to adapt their acquired software systems with millions of lines of code to rapidly changing requirements. Far too often, such systems have evolved from an uncoordinated build-and-fix attitude towards software development and suffer from a lack of methodical support during maintenance. The original design intents of the software systems are obfuscated, or worse, have disappeared altogether. It takes immense effort to implement and test changes as the effects on other software modules and the impact on future reuse are hard to predict.

In Canada, the Consortium for Software Engineering Research (http://www.cser.ca/) has launched a nation-wide series of software engineering projects under the theme "Empirical Evolution from Legacy Code to Modern Architectures". Under the auspices of CSER, companies and universities team up to investigate some of the many issues around this theme. One of these collaborations is the SPOOL project (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems), which is carried out by the software quality assessment team of Bell Canada and the GELO group of the Université de Montréal. This project aims at studying existing telecommunications systems to gain better understanding of the properties and components of object-oriented software in-the-large. For a company such as Bell Canada with expenses in software purchases that go beyond a billion dollars a year, understanding the nature of large software systems and its constituents is a strategic asset. Doing business in the fast growing and changing telecommunications market, Bell Canada depends on software that exposes the software maintenance characteristics of ISO9126, that is, analyzability, changeability, stability, and testability.

Component-based software development (CBSD) proclaims to address these issues of software maintainability. CBSD stands for software construction by assembly of prefabricated, configurable, and independently evolving building blocks. The idea is to assemble software by letting off-the-shelf components communicate with each other. CBSD has gained momentum with the proliferation of development environments based on Microsoft's Component Object Model or Sun's JavaBeans. Yet, reality shows that CBSD has proven mainly effective for systems implementation in well-understood application domains, such as graphic user interfaces, but is still insufficient for the creation of reusable and changeable architectures of large-scale software, such as telephone switches. Nevertheless, the vision of software development by grouping small pieces to wholes, which themselves may be the pieces for a greater whole, is by far the most promising approach towards more efficacious software engineering practices.

As part of the SPOOL project, we address the issues around components and composition of software in the large. To this end, we have constructed an environment, the SPOOL environment, to reverse-engineer, visualize, and query software in-the-large for desirable and undesirable design properties. To obtain a clearer picture on the essence of components and how they should collaborate to meet some complex requirements, we decided to organize the International Workshop on Large-Scale Software Composition and invited researchers from the software component and composition community to talk about their views and latest research results.

Program Committee

All submissions of papers were reviewed by at least three members of the program committee, which included
Israel Z. Ben-Shaul, Technion, Haifa, Israel;
Ritu Chadha, Bellcore, Morristown, NJ, USA;
Gang Chen, Southeast University, Nanjing, P. R. of China;
Prem Devanbu, University of California at Davis, USA;
Patrick J. Finnigan, IBM SWS Toronto Lab, Toronto, Canada;
Volker Gruhn, University of Dortmund, Germany;
Rudolf K. Keller (Chair), University of Montreal, Canada;
Stuart Kent, University of Brighton, UK;
Bruno Laguë, Bell Canada, Montreal, Canada;
Kai-Uwe Mätzel, Ubilab, Zürich, Switzerland;
Peter Mössenböck, University of Linz, Austria;
Mauro Pezzè, Politecnico di Milano, Italy;
David Rosenblum, University of California at Irvine, USA;
Reinhard Schauer, University of Montreal, Canada;
Scott Tilley, SEI, CMU, Pittsburgh, PA, USA; and
Andreas Vogel, Visigenic, San Mateo, CA, USA.

Workshop Goals and Format

The workshop sought participants to cover practical and theoretical issues of software composition addressing the following key questions.

- What are the components of large-scale software systems?
- How can we model components, their internal structure and behavior, their outwards visible services, and their collaborations with other components?
- How can we integrate new component architectures into existing large-scale legacy code?
- How can we achieve assembly of large-scale software systems out of components?

Reflecting these goals, the workshop was organized in four theme sessions, including an initial keynote address. In each session, two or three papers were presented, followed by a short plenary discussion.

Session I: Setting the Stage

Oscar Nierstrasz, the invited speaker, gave an overview of current state-of-practice in software components and composition. He expanded on the importance of components, discussed the need for a conceptual framework for software composition, and reported on the research activities of the Software Composition Group (SCG) at the University of Bern.

Nierstrasz opened his presentation with the argument that component-oriented development be framework-driven. It is the generic architecture of a framework that shapes components and drives their coordination. To devise such an architecture for component assembly, object-oriented programming provides the concepts and constructs, but current development practice hinders the creation of frameworks on which components can collaborate on some overall task. Nierstrasz mentioned four main problems with OO. First, the domain-orientedness of OOA and OOD leads to designs based on domain objects, but neglects available components and standard architectures. Second, the domain objects often have rich interfaces, but component composition depends an adherence to restricted plug-compatible interfaces. Third, OO insufficiently makes object interactions explicit, and the knowledge how objects should be plugged together is distributed amongst them. Fourth, the adaptation of an application to new requirements requires detailed study, even if the actual changes are minimal. To overcome these problems of OO practice, Nierstrasz suggested a conceptual framework for composition that consists of components (black box entities that export and import services), architectural style (formalization of standard component interfaces, connectors, and composition rules), scripts (specifications of concrete compositions), coordination abstractions (implementations of the connections), and glue code (adaptations of components that do not conform to the architectural style).

Nierstrasz further established a list of questions that should help position oneself in the world of components and composition techniques. These question reveal the current ambiguities around the notion of components:

- Is a component a class or an instance?
- Are components bigger or smaller than objects?
- What information should go in a component's interface?
- What kinds of interfaces are good for both usability and adaptability?
- Should components be statically or dynamically configurable? (or both?)
- What is the right way to put together distributed components?
- How can we combine components that adhere to different policies?
- How can we refactor object-based applications into components?
- How can we gracefully upgrade versions of components?

Scripting languages like Python are a good starting point for a composition language that can support various types of components. Python provides for

- Abstractions: objects, classes, higher-order functions
- PITL: packages, exceptions
- Extensibility: objects, dictionaries, keyword arguments
- Reflection: access to object dictionaries, run-time evaluation
- Glue: easy to interface to external components via SWIG

Yet scripting languages leave other important requirements of a composition language virtually unaddressed, such as support for architectural styles, concurrent agents, behavioral reflection, type inference, optional static type check, and formal semantics.

Nierstrasz and colleagues launched several projects addressing some of the above listed questions about components, notably the Piccola and the FAMOOS projects. Piccola is a small composition language, which is currently being developed with the following goals:

- Architectural description
  - Architectural style as component algebra
  - Reasoning via Pi Calculus
- Scripting
  - Static and dynamic component connections via agents
  - Scripts as higher-order abstractions over agent configurations
  - Extensibility and adaptability through "forms" (object/component interfaces, messages, first class contexts for agents)
- Coordination
  - Agents communicating through distributed blackboards
  - Higher-order coordination abstractions
- Glue
  - Gateway agents
  - Adaptors as message interceptors

In the FAMOOS project, Nierstrasz together with industrial partners develops techniques for re-engineering object-oriented legacy systems towards component frameworks. The FAMOOS approach is to apply a re-engineering life-cycle consisting of four phases: reverse-engineering (model capture tools and metrics), problem detection (recognize "legacy patterns"), problem analysis (apply re-engineering patterns), and change propagation (forward engineer).

The second speaker of the introductory session, Nico Lassing,

elaborated on some of the questions raised by Nierstrasz. Lassing reported on interviews they had conducted with tool vendors and users and summarized their observations. The interviewees indicated that their rationale for basing software development on components be improved reusability, increased flexibility, and reduced complexity of distributed deployment. To support these goals, components must have at least the following properties:

- recognizable concept for the developers;
- communication only through interfaces;
- clear definition of what can be expected from the component and what the component expects from the environment; and
- independent distribution.

Both presenters emphasized the lack of crisp definitions for software components and composition, which was clearly addressed by Nierstrasz in form of a question catalogue. We believe that a taxonomy for software components and composition would reduce the apparent confusion around these terms. Without such a taxonomy CBSD will not lose its image of being the latest hype, and companies as well as researchers will be tempted to sell everything that is related under the brand 'component-based'. A taxonomy would also identify niches in which different software engineering communities could gather and have more focused discussions on their research results.

Session II: Component Modeling

The goal of this session was to discuss some of the many issues of component modeling (as summarized by Nierstrasz in the previous session). Particular issues that the organizers hoped would be addressed were

- the differences in the modelling approach between OOD and CBSD; and
- modeling techniques for component coordination.

The first presentation of Jarzabek and Hitz was canceled as both of them were unable to attend. Summarizing their paper, Jarzabek and Hitz suggest a business perspective on software components. They argue that the design of software components be driven by business considerations, and that "software components should correspond to business processes at different levels of an enterprise information architecture".

The second speaker, Stuart Kent, proposed a reference model for CBSD with the core activities of build, find, adapt, and reuse. Elaborating on this reference model, he described components consisting of

- plugs, which comprise the interface and the behavioral specification;
- revealed behavior, which include further specification of the plugs necessary for the adaptation of the component;
- hidden behavior, which consists of implementation details; and
- an executable part.

He further provided a visual formalism to model such components and the composition activities.

Summarizing the discussions, we note that in the absence of standardized terminology for software components and composition, state-of-practice component modeling is much like a potpourri of traditional concepts and techniques that emphasize the creation of independent software building blocks. Most interesting in the workshop were the divergent attitudes towards components by some of the participants and authors. Some argued that modeling of components be driven by the business goals and the subsequent analysis. Others emphasized the danger that models arising from such an approach emphasize the domain and not the components and architectures available in the marketplace (see Nierstrasz talk). Clearly, there is no single solution to component modeling that can consolidate the conflicting needs for a model that, on the one hand, mirrors exactly the business requirements and, on the other hand, should be implemented timely and evolved easily to reflect the changing needs of the business units. The workshop participants could reach consensus that component-oriented development be more oriented towards existing solutions than current modeling practices suggest.

Session III: Migration towards Components

Large-scale software systems that address strategic corporate goals are seldom built from scratch. Rather, the existing system (which works) provides the overall framework for the integration of new components. The challenge is to transform these systems so that they be easier to adapt to new technologies and changed requirements. In this session, we discussed some of the techniques that can help identify and single out components in legacy software systems.

At the beginning of this session, the organizers of the workshop presented their positions on software composition from the reverse-engineering point of view. The first speaker, Rudolf K. Keller outlined the goals and objectives of CSER's research theme "Empirical Evolution from Legacy Code to Modern Architectures", in which the Canadian government and major industries investigate the key constituents of large-scale legacy systems. Keller argued that better understanding of the desirable and undesirable properties of software in-the-large be key for their migration into systems that are more resilient to the ever-changing business needs. Then, Bruno Laguë presented the motivation of Bell Canada to invest in research, techniques, and tools for software assessment and migration. The quality assessment team he heads evaluates software products of potential suppliers for the quality properties as described in the ISO9126 standard, notably maintainability. Laguë reported on creeping loss of architectural consistency in many of the software systems they have assessed over multiple consecutive versions. He emphasized that current software engineering tools for both reverse and forward engineering lack support for the identification and enforcement of a selected architecture. Rounding up the organizers' presentations, Reinhard Schauer presented the SPOOL environment. As part of the SPOOL project, an environment for design recovery and visualization of large C++ systems has been constructed. The C++ source code analysis tool generator GEN++ is used to transfer source code information into a repository with the schema of the UML metamodel. Based on this repository, queries about design information can be started and the results visualized within the UML-style diagrams representing the source code. To reverse-engineer some of the mechanisms that drive large-scale software systems, the SPOOL team developed queries that helped identify

implementations of design patterns. Schauer presented several instances where pattern-based reverse-engineering of design could help draw conclusions about critical parts of the overall system architecture.

The second speaker, Yoshiyuki Shinkawa, proposed a formal model for the recovery of reusable components in legacy software. He suggested that methods aiming at the recovery of reusable components support the following criteria:

- description of both static and dynamic aspects of business processes and information systems;
- easy-to-understand for both business and software people;
- logical and quantitative comparisons between the business and the software domain;
- rigorous expression and analysis of business processes and software; and
- transformations from requirements definitions to software.

The model he presented aims at finding a well-balanced compromise among these criteria. It is based on the combination of Colored Petri Nets and a calculus for communicating systems.

Concluding this session, Vassilios Tzerpos presented a survey on techniques for automated clustering of large software systems. He observed that

- most researchers would base their clustering techniques on structural criteria and naming conventions;
- none of the approaches identified has been assessed in respect to large-scale software;
- many researchers tested their clustering technique only on one real system; and
- most clustering techniques are based on heuristics that try to keep performance complexity below polynomial upper bounds.

Furthermore, Tzerpos gave an overview of cluster analysis as applied to classic disciplines, such as psychology, biology, or statistics, and translated these techniques to software engineering. He also observed that many clustering techniques proposed for software engineering are re-inventions of those already known in classic fields.

Tzerpos suggested that the software community should resort to the well-studied techniques of cluster analysis in classic disciplines. Concluding his presentation, Tzerpos provided a list of research directions in the field of software clustering:

- which kinds of relationships between “software objects” are appropriate from a clustering point of view;
- selection of appropriate algorithms for particular types of software systems;
- analysis of the existing gap between the structures obtained by software clustering techniques and those of the software architecture community;
- test of clustering approaches on large systems;
- incorporation of dynamic aspects into cluster analysis;
- incremental clustering to reflect ongoing changes of the software's structure; and
- integration of the knowledge of the developers into automatic clustering.

Session IV: Component-based Modelling of Distributed Systems.

More and more complex applications that transcend multiple physical sites call for communication, coordination, and distribution mechanisms that go beyond the capabilities of common distribution mechanisms such as CORBA, RMI, and DCOM. In particular, components need to become more resilient to structural and behavioral changes of collaborating components. In this session, we discussed three approaches to such dynamic cooperation of distributed components.

Wolfgang Pree introduced a visionary approach to the self-configuration of small frameworks, which he calls Framelets. The principle idea is to use computational reflection together with naming conventions to let components identify the semantics of the properties of other components with which they should co-operate. In such a setting, components would read the meta-descriptions of other components to identify usable functionality of other components. As Pree argued, this quite simplistic approach leaves many questions open, such as the reliability of dynamic component collaborations. Furthermore, the question about scalability of the approach to large-scale components is left virtually unaddressed.

The second speaker, Israel Ben-Shaul, addressed component composition in the large. He suggested a two-level negotiation model for component composition. The first level negotiates a composition contract among the components and, if successful, the negotiation results will be used to configure the components dynamically. Such a contract involves all activities around the placement of a component in its physical location, such as transfer of the components, their initialization, and their dynamic linking.

Beringer reported on the CHAIMS project, which aims at the development of a language for composition of megamodules, which are autonomous, distributed, and very large-scale components, such as reservation or transportation systems. These megamodules (as Beringer called them) reside at the site of the software provider, and the user and other megamodules interact with them only via a local CHAIMS megaprogram. The CHAIMS system, which is currently being developed, is conceptually based on three orthogonal views:

- the Composition View, which is the only view seen by the megaprogrammer, controls the requests to megamodules;
- the Data View is responsible for services such as data exchange and data interpretation; and
- the Transportation View is only concerned with the transport of messages between megamodule and the megaprograms.

One conclusion of this session is that distributed components at different scales call for different ways of composition. Pree’s Framelets may prove useful for composition of small-scale components as found in user interface frameworks. Negotiation-based component collaboration, as suggested by Ben Shaul, will most likely prove successful for modern architectures that are designed around autonomous components. Beringer’s megamodule approach takes into account the integration of large-scale components into existing legacy environments. We believe that this is indispen-

sable to inject a component-based distributed architecture into large systems, which are hardly developed from scratch.

Conclusion

The workshop attracted many experts in the domain of software composition. It highlighted some major issues of component modeling, migration, distribution, and coordination, yet not unsurprisingly, left many other important questions open, including a concise definition of what is a component. In the following, we revisit the key questions of the workshop and summarize how they were addressed in the presentations and discussions.

What are the components of large-scale software systems?

The workshop participants struggled to answer this question, due to the lack of a clear terminology for components and composition. In the course of this workshop, many views on the notion of components were presented, which include concepts from run-time objects to compile-time classes to large application modules. Nierstrasz summarized these in form of questions, which one should answer first when using the notion of components.

How can we model components, their internal structure and behavior, their outwards visible services, and their collaborations with other components?

Responding to this question, Kent suggested the introduction of a precise, visual modeling language consisting of revealed behavior for specification and adaptation, hidden behavior for design and implementation details, and an executable part. To assemble components, the revealed behavior consists of plugs which have associated class diagrams for their specification. In a similar vein, Keller suggested that component compositions be modeled at the design level. He emphasized the need that component modeling be tightly coupled with framework modeling. Therefore, it would be advantageous to build frameworks as a network of what he called design components, which provide the interfaces for the physical components that hook into the leafs of the framework. For the modeling of component collaborations, Nierstrasz suggested that scripting languages like Python seem to be most appropriate.

How can we integrate new component architectures into existing large-scale legacy code?

This question was addressed by three speakers. Schauer argued that successful migration of the architectures of existing software systems into modern architectures demands prior understanding of the architectural patterns on which these systems were built. He argued that this task of pattern-based software comprehension requires automated support, which is currently being built as part of the SPOOL project. In a similar vein, Nierstrasz reported on the FAMOOS project, which also aims at machine assistance to identify the key structural and behavioral pieces in large-scale software systems. Having a different yet equally important view on component integration into legacy systems, Tzerpos surveyed the many techniques for clustering the physical architecture of a system. Shinkawa, proposed a formal model for the recovery of reusable components in legacy software.

How can we achieve assembly of large-scale software systems out of components?

Some of the many issues of component assembly that were addressed and discussed throughout the workshop are component distribution, automated component coordination, component integration with glue code, precise component specification, pattern-based component assembly, component derivation from business goals, and component clustering.

Most of the workshop participants agreed that the assembly of autonomous components in-the-large exhibits many of the problems of the more traditional, requirements-oriented approach. Most interesting was the opinion of some of the participants that CBSD would shift the focus away from domain analysis, and thus may lead to less suitable software systems. This was countered with the argument that CBSD does require a better integration of existing components into domain analysis and need not compromise the analysis task. Furthermore, the workshop participants agreed that software engineering still lacks sufficient, plug-compatible componentry in-the-large. Catalogues that list in a generalized form the essential parts of an application domain would be very helpful to devise such componentry. This would reduce the many assumptions when building software for reusability, as one would better understand the context in which a reusable part is to be incorporated. In a similar vein, Nierstrasz emphasized that assembly of large-scale software demands a conceptual framework that consists of components, architectural style, scripts, coordination abstractions, and glue code.

Paper Abstracts

DEXA'98-Ninth International Workshop on Database and Expert Systems Applications, Vienna, Austria, August 26-28, 1998, ed. R.R. Wagner. IEEE Computer Society, pp. 765-833.

N. H. Lassing, D. B. B. Rijsenbrij, and J. C. van Vliet. A View on Components, pp. 768-777.

Components are nowadays considered the next step in information system development. Components are assumed to foster reuse and flexibility, and reduce the complexity of distributed deployment. The purpose of this paper is to investigate the properties of components that determine whether the above goals are met. To that end, we explored the literature and had a number of interviews with representatives from tool-vendors, tool-users and software houses. The resulting views are summarized in this paper, and applied to a small example. In our further research, the architecture sketched in this example will be worked out in further detail, and compared with the architecture of similar systems found in industry. Such will deepen our understanding and assessment of architectural choices made.

W. Pree, E. Althammer, and H. Sikora. Self-Configuring Components for Client-/Server-Applications, pp. 780-783.

A mechanistic view of software component assembly implies exact matching and fitting of the particular components. We argue that components for large-scale software construction should have automatic configuration capabilities in order to significantly enhance their reusability and maintainability. The paper sketches a pragmatic approach for implementing self-configuring components, relying on framework technology, meta-information and naming conventions. A case-study drawn from the client-/ server domain illustrates the concepts.

S. Jarzabek and M. Hitz. Business-Oriented Component-Based

Software Development and Evolution, pp. 784-788.

Huge size and high complexity of legacy software are the main sources of today's software evolution problems. While we can ease software evolution with re-engineering tools, in the long term, we should look for a more fundamental and effective solution. Component-based software development (CBSD) technology makes it possible to build software systems as collections of cooperating autonomous application components. This new paradigm has a potential to ease software evolution problems as modification or replacement of components is deemed to be much easier than modification of today's huge monolithic legacy programs. For CBSD to bring promised benefits, we must identify the right components in a given business domain. The claim of this paper is that while CBSD is an important enabling technology, the decomposition of a software system into components must be driven by business considerations. If we let logical models of business processes drive planning and design of software systems, we can avoid creating huge legacy software. Similar approaches may apply to software evolution in other than business domains, too.

S. Kent, J. Howse, and A. Lauder. Modelling Software Components, pp. 789-800.

This paper makes two contributions. (1) it argues that precise visual modelling techniques are important for modelling large-scale software components, as they facilitate the core activities of component-based software development (CBSD): building, finding, adapting, and assembling components. The paper argues for a carefully selected set of techniques based on UML, to provide accessible yet precise component models. (2) it sketches a high level reference model for CBSD to tease out exactly what is meant by the terms 'component', 'component adaptation' and 'component assembly'. The paper illustrates this reference model by giving examples of components and the transformations that can be applied to them, using precise visual models.

Y. Shinkawa and M. J. Matsumoto. On Legacy System Reusability based on CPN and CCS Formalism, pp. 802-810.

Most of legacy systems have been developed in such a monolithic way that they can be used only for solving certain specific problems in the domain where the systems reside. When introducing improved or new business processes into the systems, and/or employing new information technologies such as object technologies or component based software development methods, this monolithic property often becomes an obstacle. Usually one cannot enhance the legacy systems because of this property. In this paper we propose a formal approach which could be soundly applied for modeling such business processes and legacy systems as well. By this approach, we can easily evaluate equivalency and difference between them, and find reusable parts out from the legacy systems. Colored Petri nets (CPN) and a calculus of communicating systems (CCS) are used for comprising of this formal approach. Simple example from a mail-order processing is illustrated to show how this approach works.

V. Tzerpos and R. C. Holt. Software Botryology: Automatic Clustering of Software Systems, pp. 811-818.

It has long been recognized that the decomposition of a large software system into "meaningful" subsystems is essential for both the development and maintenance phases of a software project. We introduce the term Software Botryology for the area of research that attempts to automatically cluster a software system. In this paper, we survey approaches to the clustering problem from researchers in the software engineering community. We also present clustering techniques used in other disciplines, and argue that their utilization in a software context could lead to better solutions to the software clustering problem. Finally, we outline research challenges and open problems of interest.

I. Ben-Shaul, Y. Gidron, and O. Holder. A Negotiation Model for Dynamic Composition of Distributed Applications, pp. 820-825.

Dynamic composition of distributed applications from autonomous components is becoming attractive, and requires new composition models and infrastructures. In this paper we address the problem of deploying components to remote sites as part of dynamic composition process. In particular, we discuss specification and allocation of resources to remotely deployed components. Satisfying the mutual needs of an autonomous component and the environment to which it was deployed suggests a negotiation-based approach. We present a model that allows programmers to specify the negotiations strategy as part of the component, and a 2-phase deployment protocol. The first phase involves negotiation between lightweight objects and results in either a contract or a failure. In the former case the component can tailor itself according to the contract before the actual deployment, and in the latter case unnecessary deployment is avoided. The model has been implemented in Java as part of Hadas, an environment for dynamic composition of distributed applications.

D. Beringer, C. Tornabene, P. Jain, and G. Widerhold. A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules, pp. 826-833.

New levels of software composition become possible through advances in distributed communication services. In this paper we focus on the composition of megamodules, which are large distributed components or computation servers that are autonomously operated and maintained. The composition of megamodules offers various challenges. Megamodules are not necessarily all accessible by the same distribution protocol (such as CORBA, DCOM, RMI and DCE). Their concurrent nature and potentially long duration of service execution necessitates asynchronous invocation and collection of results. Novel needs and opportunities for optimization arise when composing megamodules. In order to meet these challenges, we have defined a purely compositional language called CHAIMS, and are now developing the architecture supporting this language. In this paper we describe CHAIMS and how it meets the challenges of composing megamodules.