COMPONENT FRAMEWORKS FOR MODELLING AND SIMULATION

T.Mätäsniemi
Research Scientist
Technical Research Centre of Finland
P.O.Box 1301, FIN-02044 VTT
Finland

P.Laakso
Research Scientist
Technical Research Centre of Finland
P.O.Box 1301, FIN-02044 VTT
Finland

T.Karhela
Research Scientist
Technical Research Centre of Finland
P.O.Box 1301, FIN-02044 VTT
Finland

M.Paljakka
Research Scientist
Technical Research Centre of Finland
P.O.Box 1301, FIN-02044 VTT
Finland

ABSTRACT

In the domain of modelling and simulation, improving the interoperability of software has an enormous potential. Universities and research institutes constantly produce new and improved simulation models, which are aimed at solving specific problems, and when connected to other models and decent user interfaces, can make usable applications and even commercially exploitable products.

The traditional way to connect simulation models is to gather their source codes, define a communication common area, write an executive that defines the order of solution and the interface towards the user, and compile everything together. The resulting application is most likely to become very large and clumsy with high hardware requirements, impossible to extend, and a nightmare for anyone to maintain. This connection method also sets limits to the variety of tools, e.g. implementation languages.

Component technologies (COM, CORBA, EJB) play a major role in the modern information technology. They provide standards and tools for creating distributed applications consisting of heterogeneous pieces of software published in binary format. That way, the implementation of each component is completely hidden, and the maintenance of the system is considerably easier to organise compared to traditional software. The message of the component technologies is that the era of mammoths is over, referring to large monolithic applications.

By appropriately using component technologies in the domain of modelling and simulation, large and widely applicable, yet light-weight and maintainable systems can be easily created by bridging several specialised simulation software components. The prerequisites are an efficient and reliable domain-independent component infrastructure and a solid domain-specific framework that accommodates the components to make the application. The framework's task is to ensure the interoperability of the components by defining the software architecture and the interface standards that the components must conform to.

In this presentation, the pros and cons of distribution and centralisation are discussed from the modelling and simulation point of view, and component frameworks for modelling and simulation are presented with experiences on their use.

INTRODUCTION

Traditionally, process simulators have been monolithic single thread programs. Because of the constraints of the monolithic program i.e. size of the program, calculation speed and maintenance problems, the need has arisen to decompose large programs into components that can be distributed on multiple processors and even across the network, and whose maintenance can be distributed in a similar manner. The recent developments of the software component technologies: operating systems, component model standards and programming tools have made the distribution significantly easier than it used to be.

The operations of any process simulation tool can be divided into modelling, i.e. model configuration, and simulation, i.e. running the simulation by using the solvers integrated in the tool. In this paper, an overview is taken to the underlying technologies and strategies are presented to distribute the software in each of these operations.

COMPONENT TECHNOLOGIES

Object-oriented software development has long been marketed as a solution to all sorts of software engineering problems. It has gained substantial following and improved software development in some areas. However, there are shortcomings that object-oriented development does not answer to. Objects are not units of deployment. Object-oriented development addresses the problems of creating complex software but not the problems of binary deployment. The reasons behind this are multiple. There is no common agreed upon way for objects to communicate over network boundaries and over process address spaces. Component technology addresses the problems of reuse and deployment on binary level.

There is no general consensus of what exactly is a software component. There are definitions [Cha97] such as:

1. Component is a piece of software that is published in binary format and that has a well-defined interface of methods, properties and events, which encapsulate the implementation of the component; not even the implementation language can be recognized from the outside. The methods of the component define the services provided by it, the properties contain its state, and the events determine what callbacks a client can expect from the component.
2. Component has one or more contexts, in which it is applicable, and in which its methods are use cases. The contexts may be expressed as frameworks, which domain-specifically document the appropriate uses of the component. Actually, component development hardly makes any sense without a framework to accommodate the end-result.
3. From the software engineering point of view, a component is a unit of re-use and a unit of configuration management: testing, versioning, maintenance etc.
4. From a non-technical point of view, different components may be owned by different vendors, and they may be clearly separate products with different price tags.

Ideally components should be operational on any software and hardware platform and work with components that are written in any programming language and framework. Currently this is not realistic. Component technologies have varying degrees of limitations with regards to implementation languages, operating systems, component infrastructures and hardware.

At the technical level, components can generally be divided into four categories according to their support for distribution and process type [Cha97]. The categories are presented below.

In-process client components run inside of a container of some kind and not on their own. Containers are often Integrated Development Environments such as Visual Basic and JBuilder and Web browsers. Components of this type are typically Graphical User Interface components. A developer can build an application by dragging these components onto a form and then writing code needed to get them work together. Currently, the leading technologies for in-process components are Microsoft's ActiveX Controls [com] and Sun's JavaBeans [java].

Standalone client components are applications that expose their services in a standardized way to other software. Programs can call standalone component's methods the same way they can call in-process component's methods. The difference is in that standalone components can operate on their own. Many Windows desktop applications such

as Excel and Word offer some of their functionality via COM Automation interfaces. COM and especially its automation is the most common technology for standalone client components.

Standalone server components are processes running on a server machine that exposes their services in a standardised way. This kind of component is accessed via remote procedure calls of some sort. The most visible technologies supporting standalone server components are Microsoft's Distributed COM (DCOM) [com] and Object Management Group's Common Object Request Broker Architecture (CORBA) [corba].

In-process server components are server side components that operate on a suitable container. A common choice for the server side container is a transaction server. Microsoft Transaction Server is an example of such a container, and it requires developers to create transaction programs as COM components. CORBA contains also transaction management services. Enterprise JavaBeans (EJB) is a model for in-process components targeted to run on various kinds of server-side components.

THE OPS FRAMEWORK FOR PROCESS SIMULATION

While in monolithic single-thread programs, simulation time synchronisation and data transfer between components are easy to arrange, both cause problems in case the simulation is distributed to several components. In a single-thread program, simulation time synchronisation is not needed because simulation tasks are executed consecutively, and data transfer in a monolithic program is easy because all parts of the program see the same data space. Clearly, a component system needs a framework to define the control of simulation in a distributed system and the data exchange standard. The OPS (OLE for Process Simulation) framework has been designed to meet this need: to make the configuration and control of a simulation sequence distributed to several components easier.

Architecture

The framework consists of three kinds of programs: one coordinator program, several simulation servers and clients of the coordinator program. In addition, the simulation servers may have clients of their own, and they may have a common historian database. The coordinator program takes care of the data transfer and the simulation time synchronisation. The simulation servers execute the simulation tasks and clients are used to display the states of the coordinator and the simulator servers, and to start simulation sequences. An example of the architecture is in Figure 1.

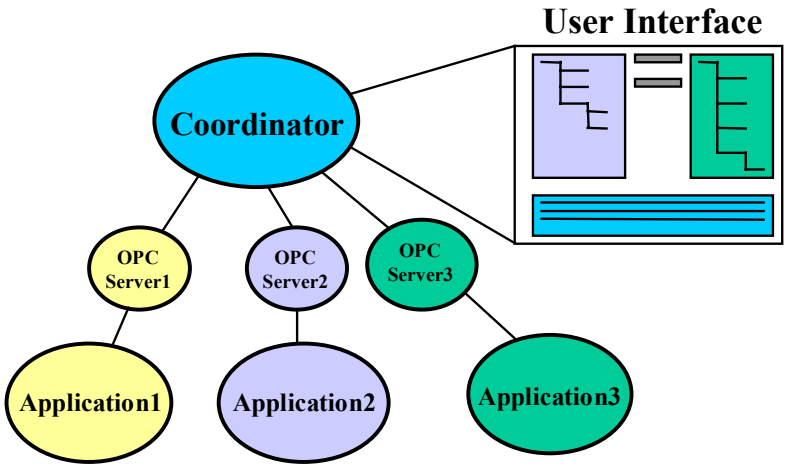

Figure 1 An example of an architecture of framework for process simulation

For data and event exchange, the framework includes interface specifications, which all the simulation servers must conform to and which are used by the coordinator when executing simulation sequences i.e. framework states requirements for the simulation servers. OPC (OLE for Process Control) [opc] interfaces for Data Access with

simulation extensions, Alarms and Events, and Historical Data Access have been selected as the interface standard. To ease the making of server interfaces a kit for making standard interfaces has been developed.

Simulation cases
The framework has been developed with regards to the following simulation cases: consecutive simulation, parallel simulation without and with synchronization.

Consecutive simulation can be either batch like simulation or dynamic style simulation where one batch is repeated perpetually. In batch like simulation, one program calculates its task to the end, sends message that it has completed its task, and transfers data through the coordinator, which transfers data to the next simulator. The simulation sequence is configured in the coordinator, and the programs involved do not need to know about their predecessor or successor programs.

The example case depicted in Figure 2 involves the coordinator, a distributed control system (DCS) with a historian server, a calibration model, a simulator and a user interface client. After the simulation sequence has been started, time series data is read from the historian server of the DCS. The time series are written to the calibration model. After the calibration model has calculated new parameters, an event is sent to the coordinator, which reads the calibrated values and writes them into the simulator. The coordinator starts the simulation at the simulator and when the simulation run is done, the simulator sends an event to the coordinator. The data produced by the simulator can be read on line to the user interface or it can be read at the end from the OPC historian server of the simulator.

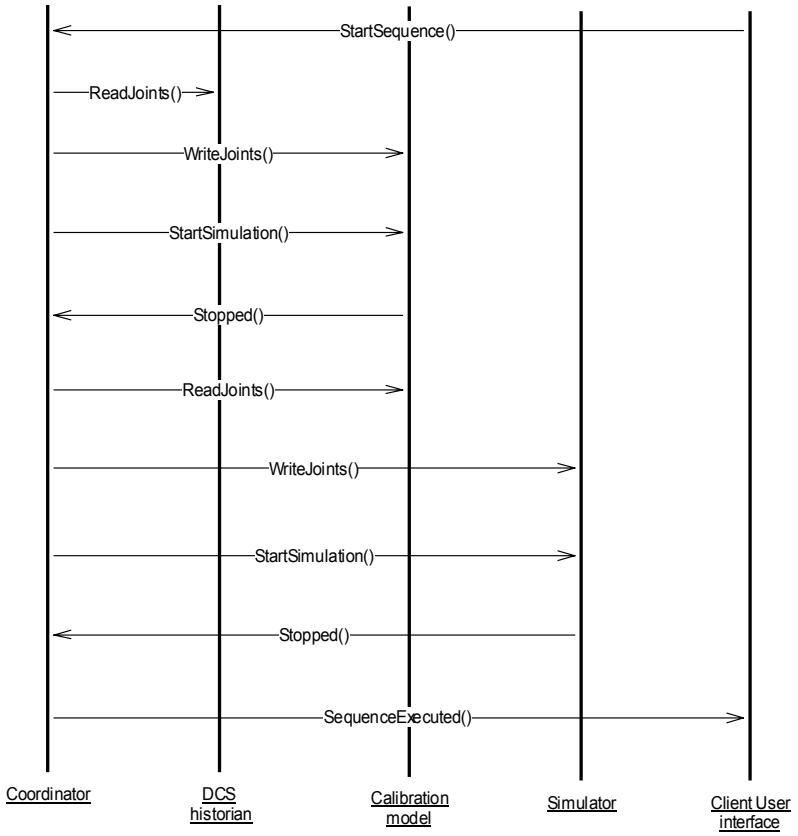

Figure 2 Batch like simulation sequence

In parallel simulation without synchronization, several simulation servers run at the same time, and send data to each other without synchronization by the coordinator. This kind of a simulation suits well for e.g. connecting automation system applications to a simulated process, in which case all components are set to stay in real time.

In parallel simulation with synchronization, the coordinator takes care of the synchronization by receiving events from the simulation components and running the simulation a step at a time. This kind of simulation suits for example to do parallel simulation on several computer hosts.

Experiences
So far only the features of the coordinator program for the parallel simulation without synchronisation have been tested. The coordinator component developed for this purpose is called X-Connector. It uses OPC Data Access interfaces for data transfer. In this example time synchronisation is not needed.

Using X-Connector, one larger demo system has been realised. In the demo, DCS was communicating with dynamic process simulator through X-Connector [Kar99]: measurement values were read from the simulator to the DCS, and controls generated by the DCS were written in the other direction. This kind of a system can be used to plan and test the automation system before it is transferred to a real plant. Furthermore operators can be trained to operate the plant with the real automation system implementation and an operator stations connected to simulator which responds correspond the real plants behaviour.

THE OPD FRAMEWORK FOR PROCESS DESIGN

The different phases of process and automation design involve severe problems due to the variety of tools used by design engineers to do the design. The poor co-operability of the applications causes information breaks between work phases, people and organisations. The resulting overlaps cost a lot every year. The OPD (OLE for Process Design) framework is indented to solve co-operation problems between software applications and to provide a working environment for engineering teams.

Components
The central component of the framework is the Unified Data Model (UDM) exposed to clients by AdviseOPDServer. In Figure 3, UDM is depicted by a white disk, which surrounded by a gray disk depicting AdviseOPDServer. The UDM is a centralized data model, which maintains both transient and persistent information about user groups, users, their workspaces, process models and applications. An application can be a graphical user interface, simulator or any other piece of software connected to the environment.

Communication paradigm
The framework involves two different kinds of communication. Firstly, the client applications retrieve information from UDM and edit its contents by using interface operations of the AdviseOPDServer. Secondly, the client applications are notified by events about changes in the UDM. The AdviseOPDServer has been implemented by using Microsoft DCOM technology, the server interfaces are described by Interface Definition Language (IDL) and the event mechanism is based on connection point technology. To embed a new type of application in the framework, it either has to be encapsulated in a DCOM component that uses the AdviseOPDServer interface, or a front component has to be built for the application that catches events from the UDM, retrieves more information about each occurrence and uses DCOM or any other means of communication to deliver the information to the application. The framework incudes a kit for constructing such components.

Unified Data Model
The Unified Data Model manages process models, user information, client applications and an event subscriptions. The idea is represented as a diagram in Figure 3.

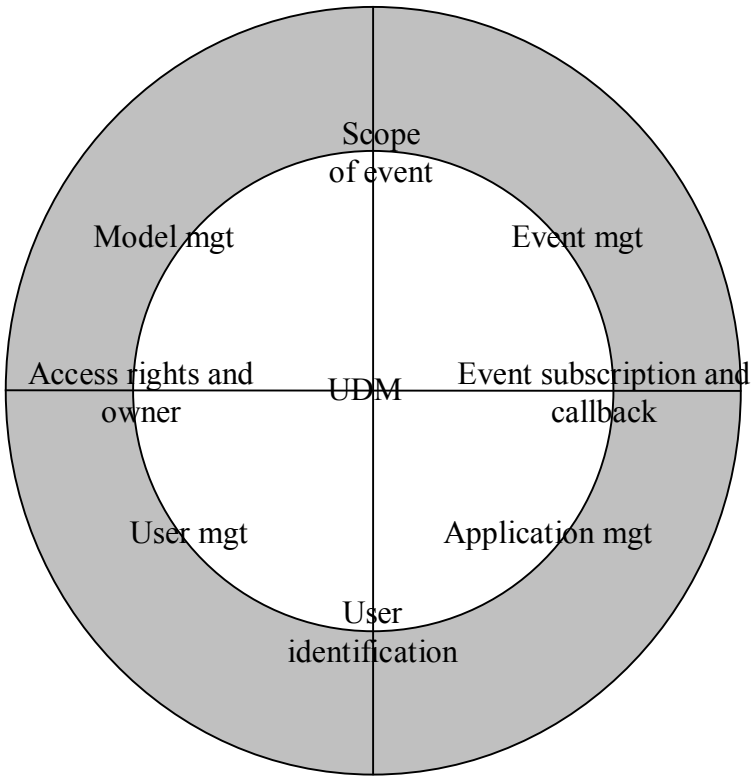

Figure 3 AdviseOPDServer

Model management
The OPD framework has a straight-forward model management scheme. The system consists of workspaces, each of which contains a toolset for building models of a particular domain and running simulation, plus models built using the tools. The toolset consists of model building block types contained in archives, and a configuration of applications that are able to reflect and simulate process models constructed using the building blocks.

Each model contains a hierarchy of components which conform to component descriptions in the archives of the workspace and which may again contain sub-components. The components may represent e.g. process components: pumps, valves and pipes, or collections of them. The component has properties and terminals by the aid of which both the state and the structure of the model can be transferred to each application. Some of the properties may be application-specific, e.g. id's. The terminals are the component's connection interfaces, and they are typed to facilitate connection rules. Connections are modeled as components referring to terminals on other components.

User management
The system has facilities to manage users and user groups that have different access levels to the data model contents, similarly to what many common operation systems have. Each entity in the data model has an owner creator, in addition to which, access rights for other users and groups can be defined. This way, parts of the data model can be protected so that the end-user cannot change or even see all details in the model structure and state.

Application management
Each application in the framework is represented by an application object in UDM. The object maintains persistent information about the application, e.g. its location and how to launch it, and transient information, e.g. whether the application is running. In UDM, applications are present in two roles, which also reflect the different roles of software components in the OPD framework. Firstly, applications are used for representing clients, e.g. user interfaces by the aid of which the user logs in the system, opens workspaces etc. Secondly, server applications represent members in workspaces, and they are launched when the workspace is opened and the modeling and simulation environment is set up i.e. AdviseOPDServer requests server applications to log in the system and

establish connections. After the workspace has been set up, the applications of both roles are handled similarly, and the access rights each application has depend on how its login procedure is configured.

Event mechanism
The purpose of the OPD event mechanism is to notify applications about modifications on AdviseOPDServer and to provide a mechanism for applications to broadcast notifications to other applications. This mechanism is based on the Microsoft DCOM connection point technique. To get notified, the application has to log in the AdviseOPDServer, create a channel for events, and a subscription that includes definitions for filtering events according to their types. The subscription also refers to process areas of interest and the channel to be used for transmitting the notifications.

Distribution and centralization in OPD
UDM is a centralised data model contained in the AdviseOPDServer and reflected to all applications concerned. The solution is entirely different from e.g. HLA [hla], in which every data item is owned by exactly one federate (application). Centralisation always brings along benefits but it also causes some problems.

When the data model is centralised, the model state and structure can be obtained at any time from a single source, and an application does not even have to have a database of its own. A potential problem is that the replication of the model state to different applications that process the state differently may lead to inconsistency. The problem can be avoided by properly co-ordinating the behaviour of the workspace at simulation time and by carefully desiging the workspaces so that all changes really are reflected to all applications concerned.

Compared to the complete distribution, the bridging of the applications to a centralised data model is easier as a lot of the process model management and user management functionalities that would need to be implemented in the front component to a fully distributed system, is included in the centralised data model. The application configuration should probably be centralised even in an architecture with an otherwise distributed data model because somehow the configuration must be saved, loaded, shut down and re-initialised, and somehow the right targets for the model changes must be found. In a system with a distributed data model, the client would probably get the information from a component that manages applications. In OPD, the client does not have to know what other applications there are in the workspace.

It can be clearly seen that the event-based communication is not sufficient for heavy data transfer between simulation time steps or iterations. The framework is open for simulation time data transfer that passes the UDM, e.g. using the OPS framework for process simulation, which has been designed for that purpose.

Especially in case two applications connected to the framework already have a dedicated bridge, there may be the temptation to use UDM for passing messages specific for these two applications only, or to even pass them by other means without notifying UDM. However, that results in a dependency between the components, which is against the idea of OPD, and leads to problems in case the system is to be configured so that one of the two directly interdependent applications is left out.

CONCLUSIONS

In this presentation, the use of component technologies in modelling and simulation has been discussed, and frameworks for creating component-based applications in the domain have been presented. The frameworks define the architecture of the system, and the software components' interfaces and contexts of use. A domain-independent component infrastructure is needed for handling the communication between the components, even distributed across a network. The critical requirements of the infrastructure concern performance, reliability and portability, i.e. independence from vendors of application development tools.

In the design of a domain-specific component framework, one of the critical points for both performance and robustness is the question of distribution and centralisation.

In the OPS framework for process simulation as well as in the OPD framework for process design, the proprietary implementations are encapsulated behind well-defined standard interfaces in software components. In the middle of each framework's architecture, there is a centralised piece of software for managing the application consisting of the

components. The connector software exposes a system-level view to which it is easy to connect with graphical user interfaces to configure the system, and provides services for saving and loading configurations. The customisation work needed for joining an existing piece of software in the environment is minimised by separating the components from each other as completely as possible, which enables also the respective distribution of the software maintenance.

In the OPS framework, every solver may have its own data structures optimised for the algorithm, however linking the solution data structures to the data items on the communication interface may lead to performance problems. On the other hand, sharing data structures on a shared memory would break the encapsulation of the components. Another performance problem arises from transmitting the data from a component to another by the connector component. Here the alternative would be to simplify the architecture by omitting the connector software and adding configuration operations on the component interfaces, which would dramatically increase the work needed for joining an existing piece of software in the environment.

In the OPD framework, every application may have a database structure and a set of concepts of its own because the interface exposed by the connector software may be easily customised by the aid of a front component so that no changes are needed to the application itself. The drawback is again a more complicated architecture than tailor-made systems have, which may lead to problems in performance and robustness. The achievement is the rapid development of new systems consisting of existing pieces of software even with rather low level interfaces.

REFERENCES

[opc] OPC Foundation home page http://www.opcfoundation.org/

[Cha97] Chappel, D., The Next Wave: Component Software Enters the Mainstream. Chappel & Associates April 1997 (http://www.rational.com/sitewide/support/whitepapers/dynamic.jtmpl?doc_key=354)

[com] COM, DCOM and ActiveX page of Microsoft. http://www.microsoft.com/com/

[hla] HLA home page http://hla.dmso.mil/

[Kar99] Karhela T., Paljakka M., Laakso P., Mätäsniemi T., Ylijoki J., Kurki J., Connecting Dynamic Process Simulator with Distributed Control System Using Opc Standard, 1999 TAPPI Proceedings. Process Control, Electrical, and Information Conference. Georgia World Congress Center, Georgia, 1-5 March 1999

[java] Java page of Sun. http://www.sun.com/java/

[corba] Object Management Group homepage. http://www.omg.org/