VTT PUBLICATIONS 479

A Software Architecture for Configuration and Usage of Process Simulation Models

Software Component Technology and XML-based Approach

Tommi Karhela

VTT Industrial Systems

Dissertation for the degree of Doctor of Technology to be presented with due permission for public examination and debate at Helsinki University of Technology (Espoo, Finland) in Auditorium T2 (Konemiehentie 2, Espoo) on the 11th of December, 2002 at 12 noon.

ISBN 951–38–6011–6 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6012–4 (URL:http://www.inf.vtt.fi/pdf/)
ISSN 1455–0849 (URL:http://www.inf.vtt.fi/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Tuotteet ja tuotanto, Tekniikantie 12, PL 1301, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 6752

VTT Industriella System, Teknikvägen 12, PB 1301, 02044 VTT
tel. växel (09) 4561, fax (09) 456 6752

VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 6752

Technical editing Maini Manninen

Otamedia Oy, Espoo 2002

Karhela, Tommi. A Software Architecture for Configuration and Usage of Process Simulation Models. Software Component Technology and XML-based Approach. Espoo 2002. Technical Research Centre of Finland, VTT Publications 479. 129 p. + app. 19 p.

Keywords process simulation, software architecture, XML, software component technology, model configuration

Abstract

Increased use of process simulation in different phases of the process and automation life cycle makes the information management related to model configuration and usage more important. Information management increases the requirements for more efficient model customisation and reuse, improved configurational co-use between different simulators, more generic extensibility of the simulation tools and more flexible run-time connectivity between the simulators and other applications.

In this thesis, the emphasis is on large-scale dynamic process simulation of continuous processes in the power, pulp and paper industries. The main research problem is how to apply current information technologies, such as software component technology and XML, to facilitate the use of process simulation and to enhance the benefits gained from using it. As a development task this means developing a new software architecture that takes into account the requirements of improved information management in process simulation. As a research objective it means analysing whether it is possible to meet the new requirements in one software architecture using open specifications developed in information and automation technologies.

Process simulation is analysed from the points of view of standardisation, current process simulation systems and simulation research. A new architectural solution is designed and implemented. The degree of meeting the new requirements is experimentally verified by testing the alleged features using examples and industrial cases.

The main result of this thesis is the design, description and implementation of a new integration architecture for the configuration and usage of process simulation models. The original features of the proposed architecture are its openness, general distribution concept and distributed extensibility features.

Preface

My first contact to process simulation was when I was a summer student at the European Laboratory for Particle Physics (CERN) 1995. Later on at the beginning of 1996 I started to do my master's thesis at VTT. At that time I was still more interested in developing simulation algorithms than focusing on the used simulation platform solutions. After graduating I had an opportunity to work in a simulation platform development project P21 (Simulation platform of 21st century) and later on this work was continued in the Gallery (Process model and parameter gallery for process integration) and Advice (Advanced Integrated Component based Simulation Environment) projects. The research and development done in these three projects form the background for the software architectural approach presented in this thesis.

I want to thank my supervisor, Professor Kari Koskinen, and my instructor, Dr. Jari Hämäläinen, for their support and tutoring during the writing process of the thesis. I am also thankful to my group leader, Mr Kaj Juslin, and to the entire system dynamics group at VTT Industrial Systems for the homely and innovative atmosphere they have provided. I want to express my gratitude to Mr Kalle Kondelin, Mr Matti Paljakka, Mr Pasi Laakso, Mr Teemu Mätäsniemi, Mr Jyrki Peltoniemi and Ms. Marja Nappa, especially, for all the discussions that have been essential when developing the ideas of the thesis. I am also grateful to Professors Seppo Kuikka and Kai Koskimies from the Tampere University of Technology for providing expert criticism and suggestions for the thesis. For the financial support, I express my gratitude to VTT, National Technology Agency and the Fortum foundation.

I also want to thank all my friends, especially Mr Sami Majaniemi for all the support, humor and relaxation they have brought to my life. Finally, my warmest thanks to my wife, to my parents and to my sister's family for all the love and support they have given me over the years.

Espoo, October 12th 2002,

Tommi Karhela

Contents

Abbreviations

ACL	Apros Communication Library
AE	Alarms and Events
AIChE	American Institute of Chemical Engineers
AP	Application Protocol
API	Application Programming Interface
CAD	Computer Aided Design
CAPE	Computer Aided Process Engineering
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DA	Data Access
DCOM	Distributed Component Object Model
DCS	Distributed Control System
DDE	Dynamic Data Exchange
DLL	Dynamic Link Library
DMSO	Defense Modeling and Simulation Office
DoD	Department of Defence
DOM	Document Object Model

DTD	Document Type Definition
DX	Data Exchange
EL	Ecosim Language
FOM	Federation Object Model
GML	Gallery Markup Language
GQL	Gallery Query Language
GTP	Gallery Transfer Protocol
GUID	Globally Unique Identifier (see UUID)
HDA	Historical Data Access
HLA	High Level Architecture
HPGL	Hewlett-Packard Graphic Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IIS	Internet Information Server
ISO	International Standard Organisation
I/O	Input/Output
MCTL	Model Client Transport Layer

ModL	Modeling Language of Extend platform
MOM	Message Oriented Middleware
MSM	Model Server Manager
MSTL	Model Server Transport Layer
$NPSH_a$	Net Positive Suction Head Available
OLE	Object Linking and Embedding
OMG	Object Management Group
OMT	Object Model Template
OPC	OLE (Object Linking and Embedding) for Process Control
PC	Personal Computer
PDML	Product Data Markup Language
PDPD	Phenomenon Driven Process Design
P&ID	Process and Instrumentation Diagram
pdXi	Process Data Exchange Institute
RPC	Remote Procedure Call
RTI	Run Time Infrastructure
SCADA	Supervisory Control And Data Acquisition
SGML	Standard Generalized Markup Language
SOM	Simulation Object Model

SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
STEP	Standard for the Exchange of Product Model Data
SVG	Scalable Vector Graphics
TCP/IP	Transmission Control Protocol / Internet Protocol
TEMA	Tubular Exchanger Manufacturers Association
TLS	Transport Layer Security
UI	User Interface
UML	Unified Modeling Language
UUID	Universaly Unique Identifier
VMS	Virtual Memory operating System (currently called OpenVMS)
XML	Extensible Markup Language

Glossary

Architecture – See Software architecture

Component – See Process component

Data-centric approach
A point of view to process modelling and simulation where the emphasis is on the configurational data of a simulation model i.e. on topology and parameter values. The term is used (not rigorously defined) in CAPE-Open specification (CAPE-Open 2000).

First principle model
Model that is based on fundamental physical and chemical laws.

Model-centric approach
A point of view to process modelling and simulation where the emphasis is on the model development and model interoperability. The term is used (not rigorously defined) in CAPE-Open specification (CAPE-Open 2000).

Model configuration
A definition of (or a work process of describing) the interconnections, parameter values and a set of specified initial states of process components that are needed to initialise a process simulation.

Model customisation
A process of setting the proper values for the parameters of a process component inside a model configuration.

Model development
A work process of programming separate unit operation models or other numerical solution algorithms that can be used together with some process simulator(s).

Model resolution
The level of details in a process model.

Model usage
Work process of using an already configured process model. During model usage the user starts and stops the simulation, observes the values and changes the set point values, but he does not make any changes to the model configuration.

Process
Sequence of chemical, physical or biological operations for the conversion, transport or storage of material or energy (ISO 1997b).

Process component
A physical object that is or may be a part of a process.

Process model
A mathematical model of one or more interconnected process components. An experiment can be applied to a process model in order to answer questions about the process.

Process modelling
Constructing a process model. Includes both model configuration and model development i.e. defining process structure and behaviour.

Process simulation
An experiment made with a process model.

Process simulation model – see Process simulation

Process simulation tool – see Process simulator

Process simulator
A program that can be used for process modelling and simulation.

Simulation scheme
The type of the major algorithm inside a process simulator. Can be for example sequential modular, equation-based, sequential and non-sequential modular or modular hierarchical (CAPE-Open 2000).

Software architecture
The fundamental organization of a system embodied in its (software) components, their relationships to each other, and to the environment, and the principles guiding its design and evolution (IEEE 2000).

Software component
A Reusable, executable, self-contained piece of software, which is accessible only through well-defined interfaces (Kuikka 1999).

Topology
Interconnections between process components.

Unit operation
Specified process functions that perform one or more defined transformations on process stream(s).

1. Introduction

1.1 Motivation and background

Dynamic process simulation has traditionally been a tool for a researcher or a specialist to study and troubleshoot process and automation behaviour. Recent development, however, has made the process simulation tools also more suitable for everyday engineering. Tool development has mainly been possible due to the rapid progress in hardware and *software technology*. Therefore, process simulation and information technology used within the process simulation have become an important research subject in the fields of *process and automation technologies*.

Process simulation is a wide concept and can be interpreted in many ways. The scope of the process simulation may vary from the simulation of micro-scale phenomena to the simulation of entire industrial plants. Industrial processes can also be divided into continuous and discrete processes or combinations of these. Furthermore, the chosen process simulation approach can be roughly divided into dynamic simulation and into steady state simulation. The industrial application domain may also differ. Given these criteria for process simulation it can be said that in this thesis the emphasis is on *large-scale dynamic* simulation of *continuous* processes of the *power, pulp and paper industry*.

Process integration means systematic and general methods for designing integrated production systems, ranging from individual processes to whole sites, with special emphasis on the efficient use of energy and reduction of the environmental effects (IEA 1993). This is the original definition of process integration research. The definition as it is stated includes already process simulation methods as part of process integration research. In some cases the definition is also expanded to cover information management methods. In process simulation research, information management methods can be interpreted to cover the development of *model reuse, customisation and configurational co-use* mechanisms. In this work, the process of finding and setting the proper values for the parameters of the unit operation model is called model customisation or simply customisation. Parameters do not have to be only separate values describing physical and chemical properties but they can also be,

e.g., points of a characteristic curve or coefficients of a function describing that curve. In current simulation tools the reuse and customisation mechanisms are not always as efficient as they should be. In practice, process and automation designers often have to model and customise the same sub-processes and equipment for the process simulator over and over again.

Life cycle consideration has become an important topic for the research in the fields of process and automation technology (Lu et al. 1997; Ajo et al. 2001). Process simulation can already be used in the pre-design stage. It can be further developed during the actual design stage. *Simulation-assisted automation testing* is possible for an automation application using a dynamic simulation model and the same model can be used for *operator training*. Furthermore, the model can then be used in the mill after the start-up for offline *operator support*. Offline operator support could include, e.g., testing an unusual operational action with the simulation model before applying it to the real process. The problem with life cycle reasoning in process simulation is that the same tool should support all these different stages. This would require models of different resolution in different stages and easy enough transition from one stage to another.

Concurrent engineering aims to rationalise the process design project by organising the traditionally sequential stages in parallel (Banares-Alcantara 2000). It has been stated for example that process and automation could be and should be designed simultaneously (Bogle 2000). According to Koolen, dynamic simulation is the key to this integration (Koolen 1998). However, the *interoperability* of the dynamic process simulation tools and distributed control systems (DCS) has to be further developed to fully realise the benefits from the integration. In order to verify the actual control schemes and logic running in DCS against a dynamic simulator, *flexible and fast enough run-time connectivity* is needed. Furthermore, if the control application definitions are to be shared between the simulator and DCS, for example in a training simulator project, there should be a way to transform the definition from one system to another.

While software technologies are developing further, software architectures are becoming more open and more extensible. This is also a trend in simulation. CAPE- Open and HLA (High Level Architecture) specifications are examples of such a development (CAPE-Open 2000; HLA 2000). These specifications are, however, very *model-centric*. They focus on independent development of model

blocks that can be attached to the execution of the simulation engine or run-time infrastructure. On the other hand, there are also more *data-centric* standardisation efforts (ISO 1998; Shocklee et al. 1998). The data-centric specifications are often large and concentrate mainly on conceptual design and product data management rather than on architectural issues.

The progress in software technologies offers possibilities for new kind of *integration* of simulation tools. For process simulation, integration means *reuse* of model configurations and parameter values, *co-use* of different simulation tools, easy *extensibility* of simulator functionality and closer *interoperability* between distributed control systems and dynamic process simulators. In order to achieve this kind of integration, a domain specific architectural specification is needed. The specification should take into account both data-centric and model-centric requirements. The main research problem of the thesis is therefore *how to apply current information technologies to facilitate the use of process simulation and to enhance the benefits gained from using it.* As a development task this means *developing a new software architecture* for configuration and usage of process simulation models. The research problem of the thesis is analysed in more detail in Chapter 3.

1.2 The objectives and hypotheses of the study

The main goals of this study are to *define* a software architecture that satisfies the reuse, customisation, co-use, extensibility and run-time connectivity needs of the process simulation domain and to *test* the validity of the hypotheses defined below. The testing is done by *constructing* a prototype software system and using examples and industrial cases to *verify* the expected features of the architecture.

From the data-centric point of view, *reuse* and easy model *customisation* can be most easily achieved if all the users of the simulation environment can share the model configurations. A *centralised repository* in the Internet would satisfy these needs. The configurations themselves are *structural* containing process equipment descriptions and their parameter values. This naturally leads to the use of a structurally oriented description language, such as *XML* (Extensible Markup Language) (W3C 2000b). Furthermore, the *co-use* of different

simulation tools and the fact that the centralised repository is located in the Internet lead to the deployment of some *message oriented middleware* (MOM) for example a protocol such as *SOAP* (Simple Object Access Protocol) (W3C 2000a).

From the model-centric point of view, *extensibility* can be most easily achieved by applying *software component technology*. This way, the extended functionality is well encapsulated and can be described using a component description mechanism. *Flexible and fast enough run-time connectivity* are requirements that can be satisfied by using software component technology and by utilizing a domain specific standard, e.g., *OPC* (OLE for Process Control).

There are two *hypotheses* in this study. Firstly, the same software architecture can be used for *sharing* models between process simulation users, for facilitating the model *customisation* and for *co-using* different process simulation tools. Secondly, a flexible and fast enough run-time connection between a dynamic process simulator and a virtual DCS can be achieved using software component technology and a standardised interface, e.g. OPC. The chosen techniques will be shown to be sufficient for *simulation-assisted automation testing* and for *operator training*. Virtual DCS here refers to a process station that runs in an ordinary PC.

In the second hypothesis it should be noted that several scalable and fast enough connections between dynamic simulators and control systems have been realised over the years. However, these connections have been *tailored* between a certain simulator and control system. The verification will show that configurational flexibility in the data connection and vendor independence can be achieved using a *standardised* approach and still the speed and scalability will remain adequate for small to medium size automation deliveries.

The reuse and model customisation mechanisms have to be developed further in order to make dynamic process simulation tools more suitable for everyday engineering. Until now the models have mainly been reused by copying previously developed models from old projects. This kind of a *template-based* approach will be further extended in this work by storing the models and sub-models into a *centralised repository* in the Internet. Furthermore, the same repository can be used for making the customisation of single process

components easier. Process *equipment manufacturers* can store equipment data in the same repository and this data can be used in the simulation environment for setting the parameter values for an individual process component (5.2.1). In addition to the template-based reuse, a parametricised creation mechanism for structural components will be developed. Instead of copying the content of a structural process component the content can be created using a *user-defined parametricised construction* mechanism (5.2.2).

The co-use of different simulation tools is needed when changing from one project stage to another. As mentioned in Section 1.1, models of different resolution are needed in different stages. Usually one simulator does not support all the needed resolutions or different simulation tools are simply better suited for a specific stage. This implies that a transition from one model to another would be useful. Such a *transformation procedure* will be created and its usefulness analysed in the suggested architecture (5.2.3).

Better run-time connectivity between a dynamic process simulator and a DCS is needed in order to achieve flexible simulation-assisted automation testing and operator training. The performance of the connection is sufficient when the *entire control application* can be tested in one go and the simulation can be run at least in *real time*. Sufficient also means that the *frequency* of the communication is adequate for testing *all* of the desired automation *functionality*. Such a flexible and fast enough connection will be presented in the suggested architecture. The different architectural choices that led to the suggested solution will also be analysed (5.3.3).

1.3 Research approaches and methods

The process simulation domain is studied from the angles of standardisation (CAPE-Open 2000; HLA 2000; ISO 1998), current process simulation systems (Apros 1999; CadsimPlus 2001; Hysys 2000) and simulation research. The *literature study* covers the most relevant topics related to the architectural development of this work. When studying current process simulation systems, three different simulators are selected as a reference set. The study is carried out by answering to the same *set of questions* for each simulator. The questions are structured in such a way that they will measure the *state of the art* with respect

to the above-mentioned connectivity, extensibility, co-use, reuse, and customisation. On the other hand, the questions will also probe the architectural issues related to the studied simulation systems.

The proposed solution to the formulated research problem is *designed* and *implemented*. The hypothesis related to the model configuration is *experimentally verified* by testing the features using *example design cases*. The run-time connectivity features are verified using a set of *industrial cases* where the proposed solution is applied.

1.4 Results and Contributions

The main results of this thesis are the design and description of a new integration architecture for the configuration and usage of process simulation models and the application of current information technologies to implement such an architecture. Furthermore, it will be shown that:

1. More efficient model reuse and customisation can be achieved with the proposed architecture if model configurators and equipment manufacturers share data in a centralised repository (example cases 5.2.1 and 5.2.2)

2. The developed architecture can be used for integrated use of different simulation tools. In practice, the integrated use depends heavily on the simulator tool vendors and their willingness to support such architecture (example case 5.2.3).

3. Flexible and fast enough run-time connectivity between a dynamic process simulator and a virtual DCS is achieved using OPC as a standard data exchange interface (industrial cases 5.3.1 and 5.3.2).

The original features of the proposed architecture are its openness, general distribution and distributed extensibility features. Both configurational and data connections are based on open ‘de-facto’ standards (XML, SVG, SOAP and OPC). The fact that the whole data model of the architecture, including graphics, model topology and parameter values, is represented in the same open format, opens up the possibilities for better reuse, customisation and configurational co-

use. The general distribution features offer possibilities to use simulation and simulation configuration resources regardless of where they are located in the Internet. The security model of the architecture is also designed to meet the requirements originating from the general distribution. Furthermore, the architecture can be extended in a distributed manner. New simulation tools can be linked to the architecture and configurational extension software components can be developed in one place and distributed through a centralised server in the Internet.

The research and development work of this thesis was carried out at *VTT Industrial Systems* (VTT Automation until the end of 2001) within the P21 and Gallery projects, both of which also included other work not reported in this thesis. The author participated in the requirement analyses of the architecture with other members of the project team (5 persons). He is responsible for the design and description of the architecture and for the implementation of the OPC Server and model configuration client tools (Model Explorer, MCKit).

1.5 Structure of the thesis

The thesis consists of six chapters.

Chapter 1. Introduction. Background and motivation, scope, the main research problem, objective and hypotheses, research methods, main results, contribution and original features of the study are represented.

Chapter 2. Related Topics in Simulation, Process and Automation Technologies. Literature study of the state of the art is carried out in terms of simulation standardisation, state of the art in simulator tools and related academic research. Both model-centric and data-centric specifications are studied. HLA and CAPE-Open are studied as examples of model-centric specifications. STEP and PDML are studied as examples of data-centric specifications. The OPC specification is introduced as it has an important role in the prototype implementation of the proposed architecture. The study of the state of the art of the simulators is carried out by answering to an identical set of questions for a set of simulators. First, the different features of the process simulation tools are discussed. Three

simulators are selected for closer inspection. Finally, different modelling languages and other related research is represented.

Chapter 3. The Research and Development Problem. The research and development problem is analysed in more detailed. The requirements of customisation, reuse, co-use, more generic extensibility and run-time connectivity are analysed in separate sections.

Chapter 4. Proposed Architectural Solution. The proposed architecture is described using the practise recommended in IEEE-1471. First, the stakeholders and their concerns are identified using use case analyses. Then the viewpoints for the architectural description are selected. Logical view, data view, security view, component view and process view are described in separate sections.

Chapter 5. Verification. The alleged features of the architecture are verified. This chapter represents the deployment view of the proposed architecture. Different use scenarios, each of which has a certain physical deployment, are represented in their own sections. First, use scenarios for verification of the configurational features of the model of the proposed architecture are represented followed by an introduction of the use scenarios verifying the model usage features.

Chapter 6. Discussion. The reliability, validity, unique features and generality of the results are analysed. The conclusions and future prospects are discussed.

2. Related Topics in Simulation, Process and Automation Technologies

2.1 Introduction

The *information technological approaches* that are used in the modelling and simulation have developed over the years. In this chapter, the state of the art of these approaches is studied. First, the related standardisation and specifications are probed. Then, the state of the art of the current simulation tools is analysed by studying a set of commercial process simulators. Finally, a review of other related research is performed.

A system is a potential source of data (Zeigler 1976). An experiment, on the other hand, is the process of extracting data from a system by exerting it through its inputs (Cellier 1991). In this work, the process model and process simulation are defined as follows:

A process model is a mathematical model of one or more interconnected process components. An experiment can be applied to a process model in order to answer questions about the process.

Process simulation is an experiment made with a process model.

A process simulator or a process simulation tool is a program that can be used for process modelling and simulation. The scale of the process model in the definition is defined as one or more interconnected process components. However, the emphasis in this work is on *large-scale* process simulation. Large scale means typically tens or hundreds of interconnected process components.

The scope of mathematical modelling and simulation is broad in the process industry. On the *micro-scale* level, the process industry may use molecular modelling methods, on *meso-scale* level some process simulation tools and on the *macro-scale* level some process synthesis and strategic planning methods (Klemola & Turunen 2001). The emphasis in this work is on meso-scale to macro-scale modelling and simulation. On these levels, process simulation is used for process and automation design, control system testing, operator

training, plant operation optimisation, process reliability and safety studies, process improvements, and for start-up and shutdown analyses.

Industrial processes can be divided into continuous and discrete processes. The process simulation covered in this work is limited to the simulation of continuous processes. Furthermore, the simulation of continuous processes can be divided into *steady state* and *dynamic* simulation. The main emphasis in this work is on dynamic simulation, but most of the analyses done about reuse, customisation, co-use, extensibility and even run-time connectivity can be also applied to steady state simulation.

The field of process industry is broad covering such sectors as water treatment, food industry, pharmaceutical industry, energy industry, metallurgical industry, pulp and paper industry, oil refining industry and so forth. The main emphasis in this work is on modelling and simulation of the processes of the *energy* and *pulp and paper* sectors. However, many of the results are generic and are applicable also to other fields.

2.2 Related standardisation and specifications

2.2.1 CAPE-Open

The CAPE-Open specification defines the interfaces of software components of a process simulator. The specification was developed in a project sponsored by the European Community running from January 1997 to June 1999. The project goal was to enable native components of a simulator to be replaced by those from another independent source or part of another simulator with a minimum of effort. Several operating companies, simulator vendors and academic institutions participated in the project (CAPE-Open 2000). Currently the CAPE-Open specification is maintained by the CAPE-Open laboratories network (Co-LaN 2002).

CAPE-Open defines a process simulator (known as a flowsheet simulator in the CAPE-Open specification) as a tool that is used in order to create models of the manufacturing facilities, processing and/or transformation of materials. The specification divides process simulators into categories according to the *internal*

architecture or *type* of the simulator. Both terms are easily misinterpreted and thus the internal architecture is referred to as a simulation scheme in this work. The specification names four different simulation schemes: sequential modular, equation-based, sequential and non-sequential modular, and modular hierarchical. The sequential modular scheme is the most common of these schemes. The given properties of the input stream (flow, temperature) and the process unit are used for the derivation of the properties for the output stream. These output properties then act as an input stream specification for the next process unit. This sequence is continued until the last process unit is reached. An example of a process simulation tool using this approach is Aspen Plus (AspenTech 2002). In the equation-based approach every model equation of the process system is solved simultaneously. The number of equations used increases considerably with the size of the process plant. A simulating tool using this approach is e.g. Apros/Apms (Apros 1999). The third category is a hybrid solver often used in simulators meant for both steady state and dynamic simulation. CADSIM Plus (CadsimPlus 2001) is an example of a hybrid solver. The modular hierarchical scheme in the CAPE-Open specification refers most likely to simulators where the solved modules form hierarchical structures. However, the specification does not specify this scheme any further nor does it give any examples of simulators using this scheme.

The specification defines common functionality for all simulation schemes. These ‘conceptual component types’ are simulator executive, unit operations, physical properties and numerical solvers. The *simulator executive* is responsible for installing other components and registering them with the operating system, managing interactions with the users, accessing and storing data and reporting and analysing simulation calculations. The *unit operations* represent physical processing unit operations and the possibility to perform specialised roles such as processing additional calculations to support the process optimisation. An important functionality incorporated in the simulator is to model the *physical properties* and behaviour of the materials used or created by the process. The physical properties include both thermodynamic properties such as specific heat capacity and transport properties such as viscosity. The *numerical solvers* include both mathematical methods for evaluating the equations that describe a unit operation and the methods used to evaluate the overall flowsheet.

The concepts stream and port are defined in the specification. *Stream* is used to describe different internal representations that simulators use to record the different types of flows that exist in physical processes. Streams are divided into material streams, energy streams and information streams. The internal streams in a simulator are not standardised by CAPE-Open. Instead, standard ways are defined for accessing and setting the information in the streams. *Ports* are used to represent a software interface that enables contents of the proprietary simulator streams to be accessed. Ports provide a standard way to fetch data from the simulator executive and to return data to the simulator executive. This is a useful way especially in a sequential solution scheme. A port has a name, direction and type. The specification defines material, energy and information types of ports. Physical property templates are associated with material ports. This ensures that each material port has an available material object. The material object can be a single component material system or a multi-component material system and can contain different physical phases. The similarities and differences of ports and terminals, introduced in this work, are discussed in Section 4.5.

In addition to the interface between unit operations and the simulation executive (ports), also an interface between a unit operation and physical properties component and interface between a unit operation and numerical component are defined. The former is done by providing a library of chemical species, physical property calculation routines and a mechanism for selecting a required set of chemical species and calculation routines. The latter is done by identifying the needed generic numerical objects such as a linear algebra object, non-linear algebra object, differential equation solver object or optimisation object and by specifying the interfaces for them. More technical information on CAPE-Open can be found from (Nougues et al. 2001).

The CAPE-Open specification is *model-centric*. The goal of the specification is to enable replacement of the components of a simulator by the components from another independent source. This is why the specification has to penetrate deep into the solution mechanisms of the individual simulator. The generic blocks of a process simulator have to be identified and the interfaces inside the simulator have to be specified. This is one characteristic difference between this work and the CAPE-Open specification. CAPE-Open specifies *interfaces inside the*

simulator whereas this work concentrates mainly on specifying the interfaces between the simulator and other programs.

The reuse requirement is addressed in the CAPE-Open specification mainly from the model development point of view. The model configuration and customisation requirements are taken into account only by stating that the simulator should support some *neutral file* storage format e.g. pdXi (i.e., STEP application protocol 231, see Section 2.2.3) for storing the model configuration. However, the reuse mechanism for model blocks in the CAPE-Open environment is well defined. It can be further rationalised by a centralised repository approach as will be described in Section 2.4.1. The extensibility needs for unit operations, for physical property calculation and for numerical solvers are well taken into account in the specification. The extensibility needs for model configuration mechanisms are not addressed. The fact that the model blocks are *interchangeable* supports the co-use of different simulator tools and if the neutral file format is supported, the co-use of configurations is probably also possible. CAPE-Open does not specify any run-time data connectivity functionality.

2.2.2 High Level Architecture (HLA)

High Level Architecture (HLA) is a general-purpose architecture for simulation *reuse and interoperability*. HLA was developed under the leadership of the Defence Modelling and Simulation Office (DMSO) in the USA to support reuse and interoperability across the large numbers of different types of simulations developed and maintained by the Department of Defence (DoD). HLA Baseline Definition was completed on August 21, 1996. HLA was adopted as the facility for *distributed simulation systems* by the Object Management Group (OMG) in November, 1998. HLA was approved as an open standard by the Institute of Electrical and Electronic Engineers (IEEE-1516) in September 2000 (HLA 2000).

HLA defines the concept of *federate* as a simulation entity (e.g. simulator), as a supporting utility or as an interface to the live system (e.g. user interface). It calls the set of federates working together a *federation*. Federates do not communicate directly to each other but through a middleman named *run-time*

infrastructure. These concepts are illustrated in Figure 2.1. HLA is used mainly in military training simulators, see e.g. VISTA (Quantum3D 1999), and although it is not particularly a process simulation architecture, the similarities and differences of this architecture and the approach chosen in this work are discussed in Section 4.5. HLA is a model-centric specification that specifies the rules and procedures for designing and implementing functional distributed simulation federations.

The HLA specification consists of three main documents, HLA Rules, Object Model Template, and Federate Interface Specification. The HLA Rules document provides an overview of the High Level Architecture, defines a family of related HLA documents, and defines the principles of HLA in terms of responsibilities that federates and federations must assume. HLA specifies 10 main rules for federations and federates. Rules 1, 3 and 6 are listed here. The rest of the rules and more detailed explanations can be found in (HLA 2000). The relationship of these rules to the approach in this work is discussed in Section 4.5.

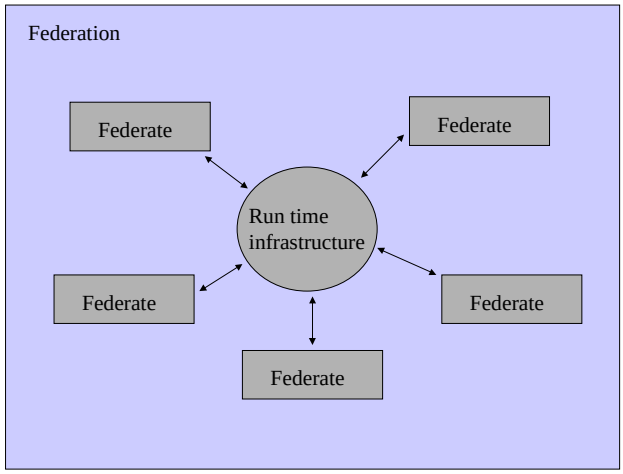

Figure 2.1 The main concepts of the HLA standard.

- Federation shall have an HLA federation object model (FOM), documented in accordance with the HLA object model template (OMT).

- During a federation execution, all exchange of FOM data among federates shall occur via the run time infrastructure (RTI).

- Federates shall have an HLA simulation object model (SOM), documented in accordance with the HLA OMT.

The HLA OMT provides a template for documenting HLA -relevant information about classes of simulation or *federation objects* and their *attributes* and *interactions*. The common template facilitates understanding and comparison of different simulations and federations and provides the format for a contract between members of a federation on the types of objects and interactions that will be supported. The federate interface specification specifies the interface between the federates and the run-time infrastructure. The interface is arranged into six service groups, federation management, declaration management, object management, ownership management, time management and data distribution management. The interface functionality is described in the specification on a general level but also IDL, C++, Ada 95, and Java language mappings are given in (HLA 2000).

2.2.3 Standard for the Exchange of Product Model Data (STEP)

STEP is a family of standards under ISO-10303 for the exchange of product model data (UKCEB 2001). Parts 1, 11, 28, 221, 227 and 231 of the standard are discussed in this section. Part 1 explains the main concepts and gives an overview of the standard. Part 11 defines the EXPRESS description language used in the other parts. Part 28 specifies the way in which XML can be used to encode both EXPRESS schemas and the corresponding data. Part 221 is an Application Protocol (AP) for process plant functional data, part 227 is an AP for plant spatial configuration and part 231 is an AP for the exchange of process engineering data. Parts 1, 11 and 227 have already been published. Parts 28, 221 and 231 are still in a draft phase.

It has been stated that STEP is the principal product data exchange standard in the world (Burkett 1999). The design of STEP attempts to reconcile two objectives:

- Define a set of data elements that are unambiguous.

- Define a set of data elements that are manageable, robust, and flexible.

STEP defines *integrated resources* to meet the second of these objectives. The integrated resources are a collection of schemas written in the EXPRESS language. Integrated resources are designed to be applicable in all usage communities that deal with product data. As a result, the schemas are generic and flexible. The first objective is approached by introducing a technique called *interpretation*. Interpretation explains how a generic integrated resource construction, such as a product, is to be understood within a particular usage domain. Figure 2.2 illustrates the situation (Alemanni et al. 1999).

The formal language, EXPRESS, is specified for data and model formalisation. EXPRESS is independent of the language used for computer implementation. The language specifies generic concepts such as entity, attribute, type, rule and data types. A short example of the syntax of the EXPRESS language is given in Appendix A.

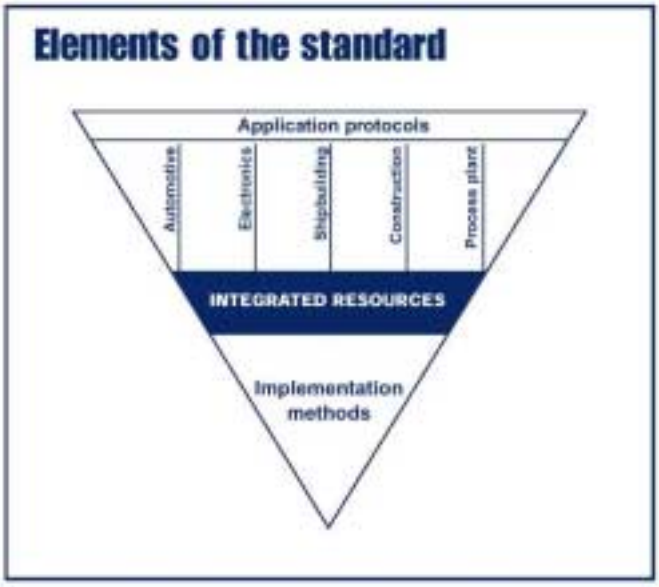

Figure 2.2. Different elements of the STEP standardisation (Alemanni et al. 1999).

Part 28 of the STEP standard belongs to the category of *implementation methods* in the standard family. This draft specification is meant for XML-representation of EXPRESS-driven data. The specification defines two approaches for formulating document-type definitions (DTD) for information described by

using EXPRESS. The two approaches are called *early binding* and *late binding*. Late Bound DTD does not define any constructs that are specific to the schema. It can be used in the same manner for any EXPRESS schema. Early Bound DTD is based on the specific schema and embeds specific aspects, such as names or structures, from the schema in the DTD.

The two approaches can be related using *architectural forms* defined in SGML/HyTime or ISO 10744. This is achieved by identifying the relationship between the two DTDs so that an application can recognise an element defined in one DTD as equivalent to an element in the meta-DTD and process the data according to the meta-DTD. EarlyBound DTD is compliant if it has LateBound DTD as its *base architecture* (ISO 1999). The early and late bindings are of interest also in this work and will be further discussed in Section 4.5. Interesting questions are for example, why late binding is needed and how an XML schema could be used for expressing architectural forms. Appendix A gives an example of both Late and Early Bound DTD and XML documents of the EXPRESS example introduced in the same Appendix.

STEP part 221 is concerned with the functional and physical aspects of plant items. The different aspects of a plant item relate to different activities, but both aspects are described by the same documents, e.g., P&IDs, data sheets and their electronic equivalents. The focus of part 221 (ISO 1997a) is:

- the piping and instrumentation diagram (P&ID), e.g. the arrangement of ink on paper or pixels on a screen;
- the information that can be understood from a P&ID, e.g. the identification, classification and connectivity of plant items;
- and property information about the plant items. This can be accessed using an intelligent P&ID system, but is traditionally presented on an equipment data sheet.

STEP part 227 specifies an application protocol for the exchange of the spatial configuration information of process plants, plant systems and ship systems. This information includes the shape, spatial arrangement and connection characteristics of piping, HVAC (heating, ventilation and air-conditioning) and cableway system components, as well as the shape and spatial arrangement

characteristics of other related plant systems (e.g., instrumentation and controls, and structural systems). (ISO 2001)

STEP part 231 defines an application protocol for the exchange of process engineering data. It specifies the process engineering and conceptual design information of process plants, including process designs, unit operations, process simulations, stream characteristics, and design requirements for major process equipment. The draft specification is rather extensive (more than 2500 pages) including about 40 categories of model items. Some central concepts related to process simulation are: (ISO 1998)

- *Unit_operation* specifies process functions that perform one or more defined transformations on process stream(s) and whose performances are calculated as single logical entities.

- *Stream_data* specifies the process material, thermal or work energy, signals or information flowing past a defined point along a path at a particular time. Streams usually flow into or out of a unit operation or through a port connection associated with process equipment.

- *Process_port* represents a flow of material, energy, or signal through the boundary of a process simulation.

- *Material_data* specifies a physical material at its related stream conditions that is used in or by a chemical process.

- *Substance_model_data* specifies the data associated with mathematical model parameters that can be used to predict thermodynamic, physical, and transport properties of the substance.

Part 221 does not cover the spatial arrangements of plant items within a process plant. Data exchange for activities that involve both functional and spatial data may require the combined use of part 221 with part 227. On the other hand, part 221 does not cover the simulation of process activities either. Data exchanges for activities that involve both functional and process simulation data require the combined use of part 221 and part 231.

The Process Data Exchange Institute (pdXi) is a U.S.-based industrial consortium, organized as an industry technology alliance under the American Institute of Chemical Engineers (AIChE). pdXi was formed in 1991 with the stated purpose of developing open industry approaches for the electronic exchange of process engineering data. pdXi is the sponsor organization for the development of the application protocol 231. AP231 has been under development since 1995 and pdXi has actively participated in the work of the process plant working group developing AP231 and AP221. (Watts 1999)

2.2.4 Product Data Markup Language (PDML)

Product Data Markup Language (PDML) is a set of XML vocabularies and a usage structure for deploying product data on the Internet. PDML is not a single data specification, but rather a structure of related specifications and tools to deploy and use integrated product data. It has been originally developed to be used by the weapon system support personnel of the U.S. Department of Defence. PDML is composed of *application transaction sets*, an *integration schema*, PDML *toolkit* and a *mapping specification* between the application transaction sets and the integration schema. Figure 2.3 illustrates the relationship between these concepts. (Burkett 1999)

The application transaction sets are vocabularies meaningful within a well-defined community. The community is defined as the users of a particular legacy system, such as JEDMICS. JEDMICS is the primary system used by the U.S. Department of Defence for managing technical data. The application transaction sets overlap with respect to the data they include. The relationships between the data defined in these views are established through the mapping to the integration schema. The integration schema is an EXPRESS schema that serves as a neutral integration format for exchange of the transaction set data. The schema is derived from the STEP *integrated resource* parts 41, 42, 43 and 44. The relationship between each application transaction set and the integration schema is specified in the mapping specification. Mapping is more than a conversion between data structures. It encompasses the *interpretation* of data based on contextual values in the same way as interpretations are specified by mapping tables in STEP. (Shocklee et al. 1998)

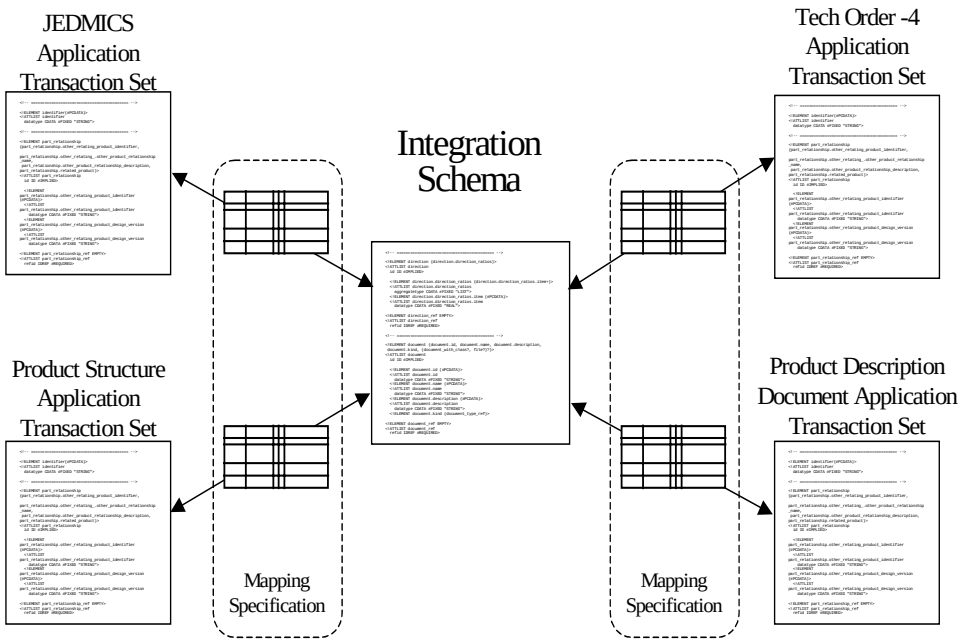

Figure 2.3. Relationship of PDML components (Burkett 1999).

The integration performed in the toolkit is driven by the mapping specifications. Using the mapping specification, the toolkit can convert data encoded according to an application transaction set to the format specified by the integration schema. This integrated data set can then be converted back out to data conforming to another application transaction set (Burkett 1999). The application transaction sets, the integration schema and mapping specification and their relationship to the approaches chosen in this work are discussed in Section 4.5.

2.2.5 OLE for Process Control (OPC)

The OPC Specification is a non-proprietary technical specification that defines a set of standard interfaces based upon the OLE/COM specifications by Microsoft (Microsoft 1995). The application of the OPC standard interface makes the *interoperability* between automation/control applications, field systems/devices and business/office applications possible. (OPC 2001)

OPC consists of several sub-specifications and the OPC foundation has currently many working groups developing new parts to the specification. The most interesting specifications related to this work are Data Access 2.0 (DA), Data Exchange (DX), OPC XML, OPC Commands and OPC Security. So far, only the DA specification has been published. Other specifications are in a draft phase. The Alarms and Events (AE), Batch and Historical Data Access (HDA) specifications are not discussed in this work.

OPC DA servers can be implemented as ‘in process’ or ‘out process’ servers. An *in process* server is a dynamic link library (dll) that runs in the same process as the client application. The components implemented in EXE files are *out process* servers. An out process server can be local or remote according to the location of the server with respect to the client process. The underlying communication is implemented using a *proxy* and *stub* mechanism. The proxy marshals data passed to the interface functions and makes a local or remote procedure call to the server. The stub un-marshalls parameters and calls the correct interface function in the component.

The data model of the OPC DA consists of server, group and item objects. Server and group objects consist of several interfaces. The client can add and delete group objects, handle error codes and access server status data through the interfaces of the *server object*. The client can also browse the server address space, save and load server configurations and handle public groups. However, the latter functionality is only available if the interfaces, marked as optional in the specification, are supported. The server object is a container for group objects. *Group objects* are defined and maintained by the client. Through a group object, the user can add and delete *items*, activate and deactivate groups and items and access data values using synchronous or asynchronous communication. The application and use of DA interfaces in the developed architecture is discussed in sections 4.7 and 5.3.

OPC DA has mainly been used for *vertical data exchange* between control application or field devices and office applications. A connecting client application is needed for *horizontal data exchange* between the control applications (see Figure 2.4). This causes extra overhead in the communication. Because the data goes through the client process, additional copy operations and function calls have to be performed. The OPC Data Exchange initiative has been

put forward to avoid this extra overhead and to avoid the extra implementation of client configuration features into every OPC server. DX specification uses DA for the communication and specifies extra interfaces for the configuration of the DA communication between servers. The connection list object defines the communication between two OPC DA servers. The connection list object consists of connection group objects and connection group object consists of connection objects. Connection group and connection objects correspond to the group and item object in the DA specification.

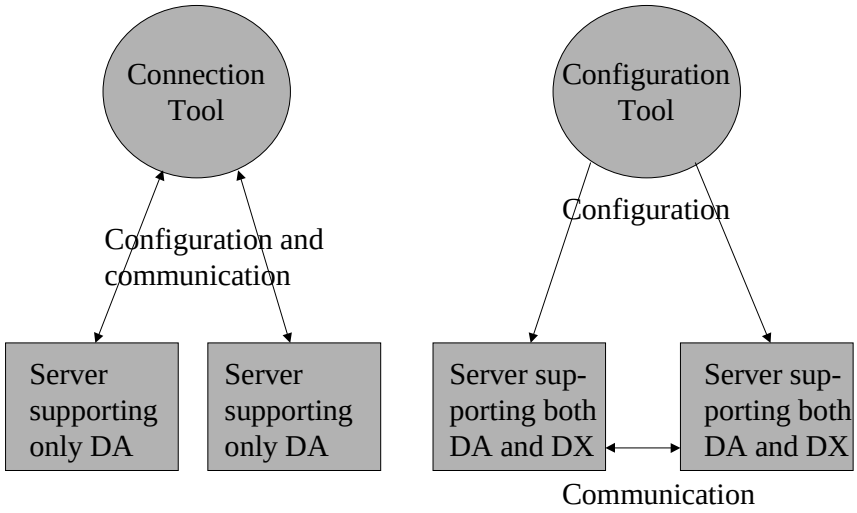

Figure 2.4. Difference between horizontal and vertical data exchange between OPC servers.

The goal for the OPC XML initiative is to develop flexible, consistent rules and formats for exposing plant floor data using XML. The focus is to expose the same data as the existing OPC interfaces expose. Another objective is to leverage the work done on Biztalk, SOAP and other XML frameworks. This will lead to easier web access, for example.

The OPC command specification will be a part of Data Access 3.0. According to the working group, a command is an action that takes a long time to execute and

that changes the state of the server or the data source. The idea is to add two new interfaces to the OPC DA server object. One is meant for retrieving information on the commands that the server supports and the other one is meant for executing those commands. The command information could be for example name and description information. The command execution is *asynchronous* and the command syntax, as well as the command information, is expressed in *XML*.

The OPC security specification specifies three different levels of security for an OPC server. On the disabled security level, launch and access permissions to the OPC server are given to everyone, and access permissions for clients are set for everyone. On the DCOM security level, launch and access permissions to the OPC server are limited to selected clients as are access permissions for the client applications. However, the OPC server does not control access to any vendor-specific security objects. On the OPC security level, the server controls the access to vendor specific security objects. On this level, the server can also support DCOM security.

2.3 Current Process simulation systems

2.3.1 Introduction

The state of the art of the current process simulation systems is analysed in this section. First, the process simulation tools are grouped according to the common features in their simulation approaches. Then, some of the tools are analysed in more detail by posing the same set of questions for each tool. The questions concentrate on the important properties for the requirements represented in this work. A short overview of each simulator is also given.

The number of commercial simulation tools is large. The technology review of the National Technology Agency, TEKES, on process modelling and simulation lists 16 different modelling and simulation tools in Finland only (Klemola & Turunen 2001). In the Netherlands, the Agency for Energy and Environment lists 91 different software tools for process integration, most of them including simulation capabilities (Novem 2000). It would be quite impossible to cover all of these tools. The selection of the studied tools is made according to the topics

represented in Section 2.1 and according to the availability of the tools for testing.

There are many steady state process simulation tools that could benefit from a link to *common software architecture*. Many of these tools focus on a specific application domain, on specific unit operations or on a specific phase in the process design life cycle. Several focused tools have also been developed in Finnish universities and companies (Klemola & Turunen 2001). Typically, the input topology and parameter values are first read from a file, then the steady state for the process is calculated and, at the end, the output is written to a file. Graphical user interfaces are used to produce the input file and to represent the results graphically to the user (Prosim 2000; Balas 1998). Many of the simulators are implemented as monolith applications having a poor extensibility.

Some of the process simulators have a *modelling language* of their own. For example, process simulators built on top of Extend (Extend 2001) platforms, such as Ideas (Ideas 2001) or Flowmac (Flowmac 2001) use the ModL modelling language of the Extend platform. Another example is EL, which is the modelling language used in the EcosimPro platform (Ecosim 2001). In fact, this is one feature that can be used to characterise a simulation tool. Either it uses a modelling language of its own or it uses some general programming language, for example Fortran, C++ or Java.

A process simulation system often has at least two different user groups. The first group develops new models for different unit operations by programming new solution algorithms to the system. The users of the second group are often process engineers configuring the process models using the developed unit operation models. Generalised simulation systems often focus more on developing tools for the first user group. This is natural because the system may be used by a variety of model developers from different application domains, so the tools must be general. However, one can argue whether the result is the best possible for the end user. The *generality* of the simulator can be used as another feature to characterise a simulation tool. At one end we have very generic tools such as Matlab (Matlab 2002) or Modelica (Elmqvist et al. 1999) and at the other end we have very specialised tools for process simulation such as Hysys (Hysys 2000), CADSIM Plus (CadsimPlus 2001) or Apros/Apms (Apros 1999).

Simulators can be also characterised by the *simulation scheme* they use. As described in Section 2.2, for example CAPE-Open divides process simulation schemes into four categories. From the end users' point of view the scheme itself is not important, but rather what can and what cannot be done with the simulator. Thus, the features for the end user are the capabilities of the simulator, e.g. the possibility for the analyses of *process dynamics*, the *resolution* and the *scalability* of the models and the *range of validity* for the models. First principle models often have a larger range of validity than models based only on measurements, for example. The dynamic features are also implemented in a different way in different tools. For example, some of the tools may support only dynamics of flows, tank levels and concentrations without pressure-flow relationships (Wingems 2001).

The application field of the simulation tool is, of course, one feature to characterise a simulator. When selecting the tools for evaluation, the focus with respect to the application field has been on energy and the pulp and paper industry. In other respects, the selected tools support dynamic simulation, are focused on large scale modelling and the model resolution is on an identical level, i.e., pressures, mass flows, enthalpies and concentrations can be calculated in the flow network. The questions addressed to each of the simulation systems are:

Q1. What is the general software architecture of the simulation system?

Q2. How does the simulator support model reuse from both the model configuration and from the model development points of view?

Q3. How can the simulation functionality be extended by the user?

Q4. How does the simulator support configurational co-use with other simulation tools, control systems and process design (CAD) systems?

Q5. What kind of run-time connectivity features does the simulator support, i.e., what are the simulators I/O capabilities?

Q6. What are the multi-user and distribution features in the simulator both during the model configuration and during the usage of the model?

2.3.2 Apros/Apms

Apros (Advanced PROcess Simulator) is a dynamic process simulation tool developed by VTT and Fortum (Apros 1999). The development of Apros software started in 1985. At the beginning, the model libraries were developed for *nuclear power plant* simulation. Later on, support for the simulation of a *conventional power plant* and pulp and paper processes was added. The *pulp and paper* version of Apros is called Apms (Advanced Pulp and Paper Mill Simulator) (Niemenmaa et al. 1998). Apros was originally developed in the VMS and Unix environments, but nowadays the Windows version of the software is most commonly used. Apros has a graphical design user interface in a Windows environment, Grades. The version of the Apros software used in the evaluation is 5.03.

Q1: The General architecture for the Apros system consists of Apros simulation *server* and *client* programs that communicate with each other using a TCP/IP-based communication library known as ACL. Client programs can have data connections and command connections to Apros simulation server. Graphical design user interface Grades is one example of a client program.

Q2: Model configuration with Apros is *modular*. The user selects the process, automation and electrical components from a toolbar and places them on a canvas (or net according to Grades). The user sets the values for the parameters of the component and connects it to other components. Both unit operations and streams are modelled as components (or modules in Apros). Reuse is achieved by introducing a mechanism for the user for transferring the modelled components on a canvas into a higher level component. The user can draw a symbol for this new component and he can export the Apros definitions for the component in a textual format. The graphics is also exported but in a binary format. This set of information can be later imported into the system and reused (template-based approach).

Q3: The user can program his own numerical solution algorithms to Apros using an *external model mechanism*. External models are dynamically linked libraries programmed with C/C++ or Fortran (there are examples and documentation for these languages). The functions programmed by the user have access to the variables in the simulation database of Apros and the routines are called after

each time step during the execution. The mechanism does not limit the access to the variables inside a particular unit operation but the routine can access all the variables in the simulation database. The mechanism is very powerful but also vulnerable to coding errors. There is no specific reuse mechanism for the external models. Of course, the models and their definitions in the database can be copied from one user to another, but they do not obey any standards for the usage in other environments. The automation library of Apros also includes a programmable component type. The user can add simple functionality to the simulation model by using this component type. The component supports an expression language similar to the language used in corresponding components in digital control systems. Using the components of the automation library the user can, of course, build new simple process models. This is, however, a different kind of extensibility. Ready-made solution algorithms are combined to produce new ones, whereas in the external model mechanism entirely new solution algorithms are programmed into the environment.

Q4: There are no specific configurational *co-use* mechanisms between Apros and other process simulation tools, control systems or process design systems. The best way to create tailored connections between other systems is to use Apros *command queues*. These are text files containing Apros database definitions. Apros can dump the content or a sub-content of its database into a queue file. The queue file can then be further processed for use in another system. This kind of tailored co-use can be bidirectional.

Q5: During the simulation execution Apros can write its simulation results into a file or send and receive data through TCP/IP *data connections*. The data is sent and received in data blocks. Among other things this mechanism allows packing of the binary signals. This may be useful in large projects, e.g. in nuclear power plant training simulators. The number of signals handled after every time step (e.g. 100 ms) in real time simulation can be around 30 000. Apros has also an OPC server built on top of the TCP/IP-based data connection. The OPC server is a data client for Apros and exposes the simulation data for other applications through the OPC interfaces.

Q6: Apros is a *single user* system during the model configuration. Even though it may have several command connections linked to it, there is no support for handling different users accessing the same parts of the model. There is also a

limitation that only one simulation run can be executed in one go. However, *several users* could be running the same simulation and observing the same values from different user interfaces. This may be useful, for example in a situation where a trainer is observing the simulation through his own user interface and an operator is using the operator station for operating the simulated process.

The user interface of the client can be located in a different host than the simulator both during the model configuration and during simulation. However, usage over the Internet may be difficult since often the TCP/IP communication ports cannot be opened because of firewalls and other security reasons. The execution of the simulation can also be *distributed*. Apros can be synchronised by using the data connection. In this way, two or more Apros processes can be installed to different hosts and they can calculate parts of the same process in parallel.

2.3.3 CADSIM Plus

CADSIM Plus is a product of Aurel Systems Inc (CadsimPlus 2001). The first version of the CADSIM user interface was developed in 1986 and in the beginning it was used as an interface for a steady state simulator, MASSBAL. CADSIM Plus was commercially released in 1995. It combined the CADSIM drawing interface with a dynamic process simulation engine. CADSIM Plus has been developed on the Windows platform. The version of the simulator used in the evaluation is 2.1. The model libraries of the simulator have mainly been developed for the *pulp and paper* industry.

Q1: CADSIM Plus is a sequential simulator with a hybrid solution method for dynamic solving of flow networks and pressure flow networks. The user interface and the simulation engine of CADSIM Plus form one executable program. The unit operation, stream and control components as well as the import/export functions are implemented as dynamic link libraries. The mechanism enables independent development of the component libraries. The fixed communication protocols between a simulation component and the simulation engine must be honoured. Each component (or module as CADSIM identifies them) must be able to describe to the simulator engine how it is to be

identified on a flowsheet, what its stream variable requirements are and if there are any limits to the number of inlet or outlet stream connections. This is all done with callbacks. There are callbacks also to get the names of unit operation parameters that the user must specify as well as for specifications that can be exported to the streams for specification on those streams. There are callbacks for the names of calculated variables and for the measuring units of all variables. Finally, there are callbacks for the module to set-up, initialise and warm itself up, as well as to perform its calculations, and ultimately matching routines to terminate itself. The components developed by the vendor are collected into a standard library and some optional libraries. The source code of these libraries is open for the users.

Q2: The model is configured through a graphical user interface in the same way as in other process design and simulation tools. The look and feel of the user interface is very similar to *conventional CAD programs* and the resulting flowsheets resemble very much *P&IDs*. *Template based model reuse* is supported. Model topology and parameter value information can be copied from one flowsheet to another. CADSIM Plus includes ready-made drawing parts and sample drawings for this use.

Q3–Q4: The solution algorithms of unit operation and control models are compiled and linked into DLLs. These DLLs can be reused between the users if they are aware of the components others have developed and if they are willing to share the components. The vendor calls the interface of the DLL mechanism Open Simulation System Architecture (OSSA). So far, only third party developers and universities are writing software components to this format so the reuse of the components is not yet possible in other simulation systems. In addition to the DLL extension mechanism, CADSIM Plus provides ready-made automation components that can be used to build new models on a flowsheet. CADSIM Plus supports conversions to AutoCad and a separate utility to export from AutoCad. The simulation results can also be annotated to the existing AutoCad drawings. In addition, graphical conversions to Microstation CAD and HPGL exist.

Q5: CADSIM Plus supports *data exchange* using DDE. By using this link, several CADSIM Plus simulations can be linked together, as can other simulators, control systems or other applications supporting the specified DDE

link. DDE mechanism also includes the possibility for time synchronisation. The capacity of the DDE link is from hundreds to a couple of thousands of variables. The DDE communication can be optimised by grouping the variables according to priority and by setting different communication frequencies for the groups. According to Aurel Systems, they have developed OPC DA client support for the simulator and are currently developing OPC DA Server functionality.

Q6: CADSIM Plus is a *single user* system during the model configuration phase. Either the drawing interface or a custom interface (through DDE) can be connected during run-time. The *multi-user* features during run-time depend on the connected software. Some SCADA systems enable multiple clients to observe values at the same time. The distribution at run time can be done by connecting different CADSIM Plus simulations using a DDE link.

2.3.4 Hysys

Hysys is a simulation and optimisation product family of Hyprotech Ltd. The company is a part of AEA Technology (Hysys 2000). Hyprotech Ltd. was founded in Galgary, Canada, in 1976. The Hysys simulation software was written in Fortran and the first version was released in 1980. It was called Hysim at that time. The software was rewritten in C and C++ later during the 1980s and 1990s and it is designed for the Windows operating system. Hysys is an integrated steady state and dynamic simulator mainly for the *oil refining* processes. The version of the Hysys.plant simulation tool used in the evaluation is 2.4.1.

Q1: The basic software architecture of Hysys is based on COM software components. The Hysys executable is a COM automation out process server. It implements interfaces for accessing model configurational data including topology and parameter values. The model configuration can be changed only when the simulation engine is not running. The COM objects also include interfaces for simulation control. In addition, Hysys can use extension COM components during the simulation. External unit operation models, kinetic reaction models and property packages can be implemented as extension COM objects.

Q2: Hysys takes a model-centric approach for model reuse, i.e. binary extension components can be reused between different simulators. Hysys supports CAPE-Open, so this model-centric approach is very natural. The model configurations can be shared between the Hyprotech product family. For instance, a steady state model can be used by many other Hyprotech programs to optimise the energy use, regress the thermodynamic models, design multi-component separation systems and generate P&ID drawings. The model configuration can be accessed through the COM interfaces and tailored bridges to other programs can be built on top of the interface. The user can also use templates within Hysys. Parts of the model can be exported and imported to other models.

Q3: Extensions can be implemented as COM components to the Hysys environment. The extension type can be a unit operation model, kinetic reaction model or property package. The extension consists of two parts, the extension definition and the COM binary component (cf. server extension definition in Section 4.5.3.) The extensions can be viewed graphically in the Hysys user interface. These views are built using an extension view editor (cf. dialog mechanism in Section 4.5.3). Hysys supports CAPE-Open, so the extensions can also be used in other CAPE-Open compliant simulators.

Other more simulator specific means for extending the functionality are user variables and the internal spreadsheet. If the user needs to add a stream variable that is not currently present in standard Hysys this can be done using user variables. A user variable is a variable with a piece of Visual Basic code attached to it. The user variables can access all the standard variables in the Hysys model. Hysys has also an internal spreadsheet. It can be used, e.g., for calculations where variables from different parts of the model are used. The user can import variables into a cell and perform calculations in the same way as in a stand-alone spreadsheet application. Hysys has also an internal, user-defined unit operation. The user-defined unit operation is a user-defined Visual Basic block with some input and output streams.

Q4,Q6: Hysys does not support any neutral file format so the configurational co-use takes place through COM interfaces. The topology, parameter values and graphics can be accessed through COM objects and exported programatically to other programs. Same way multi-user and distribution features are supported

through COM/DCOM mechanism. The models can be password protected so that only an authorised user can open them.

Q5: Hysys has tailored I/O connections implemented to several control systems. Siemens, Honeywell, Fisher Rosemount, Bailey,Yokogawa and Foxboro are mentioned in the Hyprotech webpage (Hysys 2000). According to Hyprotech they have also implemented OPC functionality to the simulator.

2.3.5 Summary

A variety of software approaches can be found in the analysed process simulators. Legacy features have clearly affected the solutions in some cases. For example, some solutions have used features of certain operating systems more openly than others. All three simulators support features for template-based reuse of ready-made model configurations and external model mechanisms for model development. None of the simulators support accessories for distributed deployment of model configurations and external models. The interfaces for external models are open and well documented in all three simulators. However, all three specifications are different, so the developed external models cannot be used in other two simulators. The CAPE-Open specification, supported by Hysys, seems to be the most promising effort in the direction of open model development.

The configurational co-use between the analysed three simulators, other simulators, control systems and process design systems is quite poor. CADSIM Plus seems to resemble process design CAD programs most closely and it has graphic export and import functionalities to and from CAD applications. This seems to be the best level of configurational co-use in the current simulation tools. None of the tools is extensively using XML or SVG for model configuration. However, inside CAPE-Open, i.e. in Hysys, XML is being used in configuration files for the thermodynamic solver framework (Nougues et al. 2001). XML is also supported in the simulation case definitions in the most recent version of Hysys (3.0). The run-time connectivity features are quite well implemented in all three simulators. OPC seems to have become the de-facto standard also for process simulators. However, a common method for handling simulation control (including synchronisation) and training control lacks from

the current implementations. All the studied simulators are mainly meant for single user local use, especially during the model configuration. During the model usage they support certain distribution and multi-user features.

2.4 Related research and development

Academic research in the field of process simulation software architecture is rather scarce. Schopfer et al. have developed a co-ordination system for process simulators (Cheops) where a *centralised model repository* (Rome) has an important role (Schopfer et al. 2000). The main idea is to develop and deploy unit operation models using Rome and execute the models based on different modelling schemes using Cheops. Schopfer argues that the canonical modelling concepts of Rome enable the specification of models on a level of physical understanding rather than on the basis of implementation-specific constructions. The models can be exported from Rome into textual, e.g. gPROMS or into binary representations e.g. a CAPE-OPEN compliant software component. The use of the repository is justified by the reuse and lifecycle arguments.

Figure 2.5 illustrates the *data model* of Rome. The core of the data model is represented in the model structure and model behaviour packages. Figure 2.5 shows that the concept of the model is associated to a set of mathematical equations and variables describing the behaviour of the model. The model quantities can be published or they can be private. Private quantities cannot be accessed by other models. The mathematical equations of a model can refer to any of its internal quantities or to any published variables of its sub-models (vertical connections). The models can be nested and they can have ports that enable the connections to other models. The models can also publish quantities in their ports in order to make them available for information transfer between different models (horizontal connections). (Schopfer et al. 2000)

Rome is developed using C++ and an object-oriented database, Versant. Rome provides its functionality for other tools via CORBA. This functionality includes manipulation of the neutral model representation or retrieval of information about the model implementation. Rome has also a web user interface known as RomeWWW. The similarities and differences with regard to the centralised

repository approach chosen in this work are discussed in more detail in Section 5.2.2.

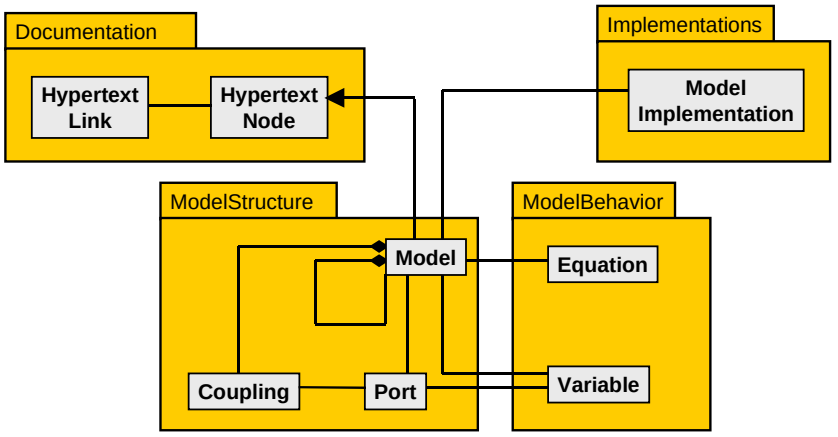

Figure 2.5. Data model of Rome.

It has been stated that the developments in computer-aided process modelling can be classified into four groups; *general modelling languages*, *process modelling languages*, *expert systems* and *interactive modelling environments* (Marquardt 1996). *General modelling languages* can be viewed as equation-oriented simulation languages. MODELICA (Elmqvist et al. 1999), OMOLA (Mattson & Anderson 1994) and the modelling languages of ASCEND (ASCEND 2001) and gPROMS (gPROMS 2001) are examples of general modelling languages. They support hierarchical decomposition and they use concepts of semantic data modelling and object-oriented programming. The definition of these languages is confined to a relatively small number of generic elements.

Process modelling languages are similar to general modelling languages but they are designed to match the specific issues of a particular application domain. MODEL.LA (Stephanopoulos et al. 1990) is an example of a language where the elements are tailored for chemical engineering. Another example of a process modelling language is the computer implementation of the phenomenon-driven process design methodology PDPD (Pasanen 2001; Pohjola & Tanskanen 1998). For example in MODELICA, the basic modelling concepts are generic among all modelling domains (model, parameter, equation, function, algorithm)

whereas for example in MODEL.LA, the basic modelling elements are generic only in a specific modelling domain (port, stream or generic unit).

The aim of the *expert systems* for modelling is to produce a process model from a formal description of the modelling problem initially provided by a user. The implementation is based on some knowledge presentation formalism. *Interactive modelling environment* is not an automatic process model generator like an expert system but rather an interactive construction kit consisting of building blocks. According to Marquardt, there is no system as yet based on this idea. Some characteristics can be found in MODASS. (Marquardt 1996)

The modelling languages mentioned in this section do not only try to model the configuration or the structure of a physical system but they also include equations describing the *behaviour* of the system. This is the fundamental difference between these languages and more data-centric approaches, such as STEP or PDML. It is also the reason why the approach taken in this work is closer to the one chosen in STEP or PDML.

Seven companies including process industry software vendors and end-user companies started to work at the beginning of 2001, to define and to deploy practical XML standards for the process industry. The companies driving this effort are Chemstations Inc., COADE Inc., ePlantData, Inc., HTRI, Kellogg Brown & Root, IFP's Industrial Division, and WaveOne.com, Inc. The PlantData XML standards are initially focused on describing process materials and the design, specification and procurement of shell and tube heat exchanger equipment. PlantData XML standards are envisioned to eventually include the full range of process plant information and process equipment. The aim is to utilise the work done in STEP to produce this new specification. There is no draft specification yet available for this effort. (ePlantData 2001)

3. Research and Development Problem

3.1 Introduction

A new software architectural approach is needed for process modelling and simulation applications in order to satisfy the requirements for reuse, customisation, co-use, extensibility and run time connectivity (see Chapters 1 and 2). Current information technologies offer possibilities for fulfilling these needs. The main research problem of the thesis is thus, *how to apply current information technologies to facilitate the use of process simulation and to enhance the benefits gained from using it.* As a development task this means *developing new software architecture* for configuration and usage of process simulation models. Figure 3.1 describes the above requirements and their relationship to the chosen information technological approach. One should note that the requirements are not isolated, but they overlap as will be shown in the following sections.

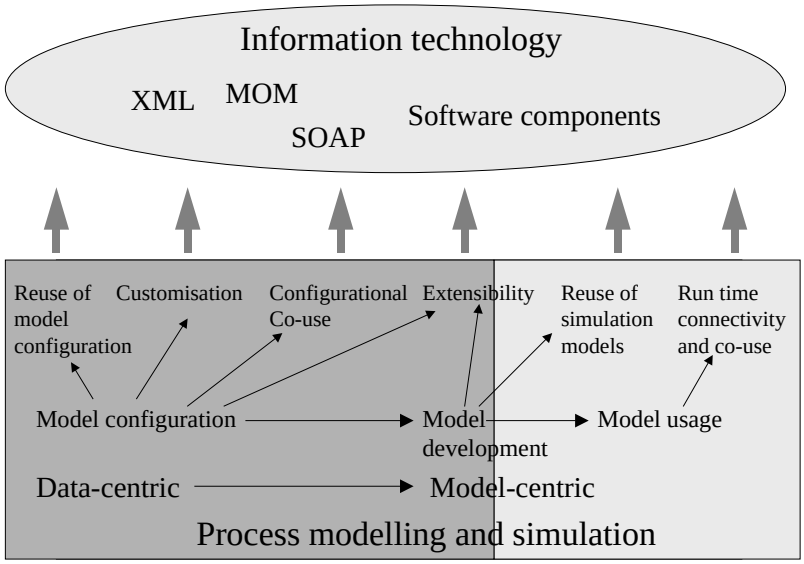

Figure 3.1. Needs of process modelling and simulation.

3.2 Need for better model reuse and easier customisation

Model reuse can be considered from both the data-centric and model-centric points of view. Process simulation tools usually carry comprehensive simulation algorithms and the job of the modeller is to define the model *topology*, i.e., interconnections between the process components and parameter values for the individual equipment. The topology and the parameter values are referred to in this work as *model configuration.*

In spite of the comprehensive algorithms programmed into the process simulation tools, the modeller has to extend the functionality of the simulator in many cases. This process of programming individual unit operation models or other numerical solution algorithms is referred to in this work as *model development.*

Model reuse and easier customisation are becoming more and more important as the models become larger and more detailed. The largest model configurations at the moment may include more than one thousand configuration diagrams and more than 30000 I/O signals between the process and its controls (Puska et al. 2001). Without reuse and easy customisation mechanisms the configuration of such models is time consuming and laborious.

Model configuration especially for a dynamic process simulator is still too laborious. This is a common reason in industrial engineering not to exploit dynamic process simulation. The three main reasons for this *laboriousness* are: (i) *poor reuse* of earlier model configurations, (ii) difficulty in finding the proper parameter values for unit operation models and (iii) *lack of* appropriate *co-use* between the process simulation tools and other process design systems (CAD).

Poor reuse is often a consequence of bad *encapsulation* mechanisms and of a working process that does not support *sharing* of the model configurations. The more detailed a model is the more an engineer has to know about its content in order to use the full power of the model.

It can be also difficult to find right parameter values for some unit operations, e.g. for dynamic simulation for heat exchangers and burners. Sometimes the unit

model is desired to be customised so that it would simulate the equipment of a specific manufacturer. In this case, the correct parameter values have to be somehow obtained from the *equipment manufacturer*.

Reuse of the developed extension algorithms is difficult due to the fact that different simulators usually support different *simulation schemes* and the developed models are very often designed for that particular scheme. As described in sections 2.2.1 and 2.4.1 there is some standardisation and academic research efforts in the area of reuse (CAPE-Open 2000). Schopfer describes different schemes with the following examples (Schopfer et al. 2000):

"During flowsheet design the specification of a simulation experiment for, e.g., Aspen+ is based on the selection and parametrization of modules that are subsequently used in the underlying modular simulation scheme. To study the transient behaviour of the process using dynamic simulation tools such as gPROMS an equation-based representation of the system to be simulated is well suited for the underlying iteration scheme. As a third example, control engineers use Matlab/Simulink based on (often linear) state space representations to design and assess control configurations."

In this work, the approach to this problem domain is different from the standardisation or from the referred research. Section 3.4 explains in more detail how extension components are not only seen as unit operation models or numerical solution algorithms but rather as a *generic* way for extending the functionality of the simulation server. The functionality can contain a search algorithm that goes through the database of the simulator, a dimensioning algorithm for sizing of particular type of equipment or a parametricized constructor for creating the contents of a structural component. These software components have *well defined interfaces* in the architecture and they can use the defined interfaces of the simulation server. The problem of different simulation schemes in different simulators is not addressed in this thesis. Reuse of the developed software components can be achieved through a centralised repository. However, the suitability and correct use of a particular software component designed for a particular simulator depends wholly on the end user and on the documentation generated by the model developer.

3.3 Need for better configurational co-use

Different simulation tools are used at different stages of the process and automation delivery. The balances are calculated in the pre-design stage using spreadsheet calculators or simple steady state simulation tools. The steady state models are developed further during the actual design phase or the dynamic behaviour is already analysed using a dynamic simulator. Data from the equipment manufacturers is also needed at this stage. During the automation design stage, the dynamic simulator can be used for designing the control schemes. Furthermore, when the DCS is configured, the application can be tested with a dynamic process model. Dynamic models can be also used for operator training and support. The *life cycle* consideration is presented in Figure 3.2.

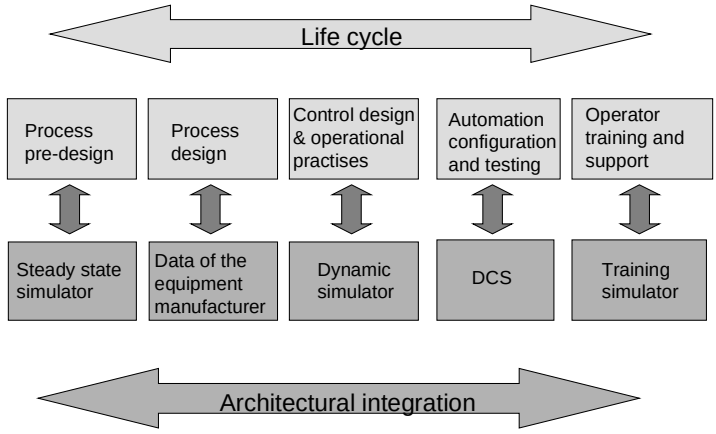

Figure 3.2. Need for architectural integration from the co-use point of view.

The problem with the use of different tools is the *re-configuration* that has to be done when changing from one tool to another. There are two ways for approaching this problem. Either a simulation tool should support all the different stages described above or different simulation tools should utilise the same model configuration. If one simulation tool is made suitable for all of this use, it should support different model *resolutions* at different stages and it should be flexible enough from general usage to detailed troubleshooting. Co-use would still be needed, e.g., if the model configurations were to be *reused* by different organisations using a different tool. The problems with the co-use approach are

that different tools should support the same interfaces. This may be difficult to achieve due to the large number of simulation tool vendors.

In this thesis, however, better co-use is defined as one of the sub-problems. *Openness* and better co-use capabilities are justified features in both approaches. Also according to a recent modelling and simulation technology review by (Klemola & Turunen 2001) co-use should be enhanced. The report lists two interesting conclusions and recommendations related to the co-use, connectivity and reuse objectives in this work.

- Integration of modelling tools should be enhanced. Tools and models that support technology transfer, such as operator training and support systems, should be further developed.

- Knowledge management will be more and more important (cf. Gallery).

The co-use needs do not only appear between different simulation tools but also between simulation tools and *process and automation design systems* (CAD). A concrete example of such a need is a training simulator project. There are several ways to implement a training simulator. At one end, there stands a case where the operator displays and automation of the actual DCS are used and only the process is modelled in a dynamic simulator. At the other end, thereis a case where the displays are *emulated* and the process and automation are both simulated in a dynamic simulator. The choice of way depends mainly on two facts. Firstly, is the automation application available before the operator training is due to happen? Secondly, if the application is available, how much does the extra automation hardware cost in case the first extreme is chosen?

The fact that the DCS manufacturers have nowadays also PC-based versions of their system has lowered the costs considerably. In practice, the emulated approach is often necessary because the implementation of the automation application is not ready before the operator training. This leads to a situation where the automation is implemented first to the simulation system and then again to the DCS. If there were better co-use between the systems or if the project could be scheduled differently this *re-configuration* could be at least partly avoided. The problems related to the configurational co-use between a simulator and a DCS are discussed further in Section 5.3. The problems related

to the co-use of process design systems (CAD) and simulators are not addressed in this thesis.

3.4 Need for more generic extensibility

As mentioned in Section 3.2, sometimes the numerical solution algorithms of the simulation tool are not adequate and the user has to extend the functionality of the simulator. The extensibility of the simulator depends on the interfaces it supports for this purpose. The *tightness* of the *integration* of the user-developed model to the execution of the rest of the simulation model is also different in different simulation tools. There are problems both in tight and loose integration of the model extensions. The more closely the model extension is bound to the *simulation scheme* of the simulator the more difficult it is to use the model anywhere else. On the other hand, using loose integration the user may have fewer means for influencing the *model as a whole*, which is often needed to increase the calculation speed and model resolution.

The user may also have other needs than simulation functionality for extending the simulation tool. Model development is not the only area where extensibility is needed. Extendable functionality is also needed in *model configuration*. This kind of more *generic extensibility* may support for example *reuse*, *customisation* and *co-use* requirements.

The *template based* approach for model reuse is sometimes not efficient enough. Suppose for example that the user has integrated a sub-process as a structural process component. This sub-process contains n sub-components, for instance tanks connected to each other with pipelines. Now, when *reusing* the model the user may have a need for a sub-process with n-1 or n+1 tanks. With the template-based approach all of these different variations of the model structure have to be stored in order to be able to copy them to the new project. This takes a lot of space and effort. Instead, the user may want to parametrisize the creation of the sub-process component so that in the creation he can give the number of tanks as an input parameter for the *constructor* of the sub-process (cf. class constructor in object-oriented languages). In order to achieve this the user must be able to extend the functionality of the creation of the sub-process. This means

that the constructing algorithm is programmed using some programming language.

The simulator user may also want to extend the functionality of the simulator by creating different *dimensioning tools*, depending on the application area in which he is using the simulator. The selection of a pump based on hydraulic properties in a pipeline dimensioning is an example of such a task. If the simulator does not already contain such a tool there might be a need to build one. This is possible without the tool vendor only if the system is extensible enough.

A *transformation* of the model configuration from one tool to another is needed in order to *co-use* the different simulation tools. This transformation can be done through a common model or directly from one tool to another (5.2.3). In both cases, a transformation algorithm is needed. The programming of this algorithm can be also seen as extending the functionality of the simulation tool.

All the sub-problems described above are examples of the need for more generic extending mechanisms. The more generic approach taken in this thesis naturally also leads to a *loose integration* of the model extensions to the simulation scheme of the simulator.

3.5 Need for more flexible run time connectivity

Already in the 70's, dynamic simulation models were connected to a control system in order to *test* the functionality of the controls and to *train* the operators (Juusela & Juslin 1976). However, the connections were *tailored* for a specific simulation tool and for a specific DCS and they also lacked the flexibility for *on line* modification of the connection. A dynamic connection would enable the testing of the control schemes section by section even while they are being configured to the control system. The fact that the solution is independent of any particular DCS or simulator would enable the use of different tools even at the same time.

The requirements for a functional run time connection are *scalable* and *fast* enough data exchange, flexible on line configuration and a possibility to control the execution in both of the connected systems. This control includes *simulation*

and training control features. An interesting simulation control requirement is the need to *synchronise* the execution of the entire system consisting of a DCS and a simulator. The synchronisation can be realised approximately by setting the execution speed of the systems to real time. If more accurate synchronisation is needed, the systems have to be synchronised in every time step. More accurate synchronisation is necessary, e.g., in cases where the system is executed faster or more slowly than real time. Simulation and training control features are discussed in more detail in Section 5.3.

Scalability is needed when dealing with large applications. Connectivitywise, the size of the application is measured by the number of input and output (I/O) variables. In a nuclear power plant application the number of I/O variables can be around 30000 (*large*), in a conventional power plant usually less than 10000 (*middle-sized*), and in a small application around 1000 (*small*). The scope of this work is from small to middle-size applications. The *speed* of the connection is needed in order to test all the control loops of the automation application. Speed and scalability are not separate requirements. Usually the I/O variables are divided into groups depending on the communication frequency they need. The more variables there are at high frequencies the more load there is on the connection. In this work, the scope is limited to frequencies less than 10 Hz in real-time simulation. The speed and scalability are discussed in more detail in Section 5.3.3.

Flexible run-time connectivity is not only needed for a DCS connection. Also run-time *simulator to simulator* connections, flexible connections to *SCADA systems* and connections to *PC-based operator displays* may be needed. The simulator to simulator connection can be used for example in dividing the simulated model to several simulation engines and running them on different machines. The connection to SCADA systems can be used for example for automation testing or for training purposes (Zamarreno et al. 2000).

4. Proposed Architectural Solution

4.1 Introduction

Software architecture can be defined as the fundamental organisation of a system embodied in its (software) components, their relationships to each other, and to the environment, and the principles guiding its design and evolution (IEEE 2000). A system, on the other hand, is a collection of (software) components organised to accomplish a specific function or set of functions. The architectural solution proposed in this thesis is described according to the practice recommended by IEEE (IEEE 2000).

The conceptual model of architectural description is shown in Figure 4.1 as a UML (Fowler 1997) class diagram. According to the conceptual model, every *system* has an *architecture*, it fulfils one or more *missions* and it is influenced by its *environment* (Figure 4.1). The system also has several *stakeholders* with specified *concerns*. The architecture can be recorded by an *architectural description* that is organised into one or more architectural *views*. Each view addresses one or more of the concerns of the system stakeholders. A view consists of architectural *models* and it is used to refer to the expression of a system architecture with respect to a particular *viewpoint* covering one set of concerns. The relationship between a viewpoint and a view is analogous to the singleton relationship between a class and its instance. (IEEE 2000)

An architectural description should include the following elements (IEEE 2000):

a) Architectural description identification, version, and overview

b) Identification of the system stakeholders and their concerns judged to be relevant to the architecture

c) Specification of each selected viewpoint to organise the representation of the architecture and the rationale for those selections

d) One or more architectural views

e) A record of all known inconsistencies among the required constituents of the architectural description

f) A rationale for selection of the architecture

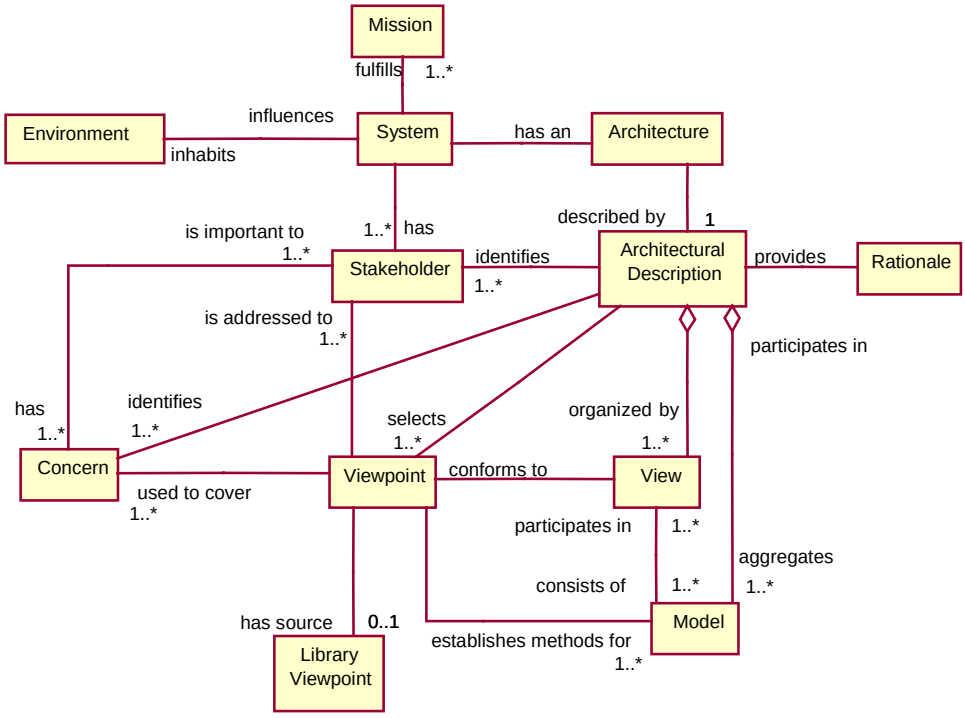

Figure 4.1. Conceptual model of architectural description according to IEEE Std 1471-2000 (From IEEE Std. 1471-2000 Copyright 2000 IEEE. All rights reserved.).

Stakeholders include at least *users*, *acquirers*, *developers* and *maintainers* of the system. However, the documentation given here describes the proposed solution for the research problems listed in the previous chapter and thus does not provide full software documentation. Therefore, some of the required items are discarded, e.g. the version numbers and the full analyses of the concerns of the acquirers, maintainers, and developers. In other words the viewpoints are selected to cover the concerns of one stakeholder group, i.e. the users, and its sub-groups.

The identification of the different sub-user stakeholders and their concerns is described using UML *use case diagrams*. The users of the system are divided into four major categories *kernel developers*, *providers*, *model configurators* and *model users*. These stakeholders and their concerns are analysed in the next section.

4.2 Use case analyses

4.2.1 Kernel developer

The users of the proposed simulation architecture can be divided into four major categories: kernel developers, providers, model configurators, and model users. As will be explained in the security view, these four user categories have different security roles in the architecture. The kernel developers are the most powerful users of the system. Of course the administrators of the model server hosts may have functions not allowed to the kernel developers, such as adding users to the operating system or installing software components. However, the use cases of administrators or maintainers are not analysed here. It should be noted that the concepts of *model client, model server, model server extension* and *model client extension* are used already in this section but explained in Section 4.4.

The four user categories correspond also to four *security roles* in the architecture as will be seen in the security view section. These security roles form subsets so that the kernel developers have all rights for interface functionality allowed to the providers, the providers have all rights for interface functionality allowed to the model configurators and the model configurators have all rights for interface functionality allowed to the model users.

The use cases of the kernel developer are shown in Figure 4.2. The actions specific only for the kernel developers are the development and publication of client and server extension software components. The kernel developers are also responsible for adding new departments to the model servers and adding new users to the departments. Departments are main level elements in the data model of the architecture as will be described in the data view section.

Building a client or server extension is a task of programming a software component that conforms to the interface specification of the corresponding client or server extension sub-type. Server extensions may use the GQL interface of the model server in their implementation whereas client extensions can use configuration or data connections in the same way as model clients use them. Publishing a client or server extension is a task of writing a description for the extension in GML format and uploading both the description and the binary component to the model server. The format for the client and server extension description can be seen in the data view in Figure 4.8.

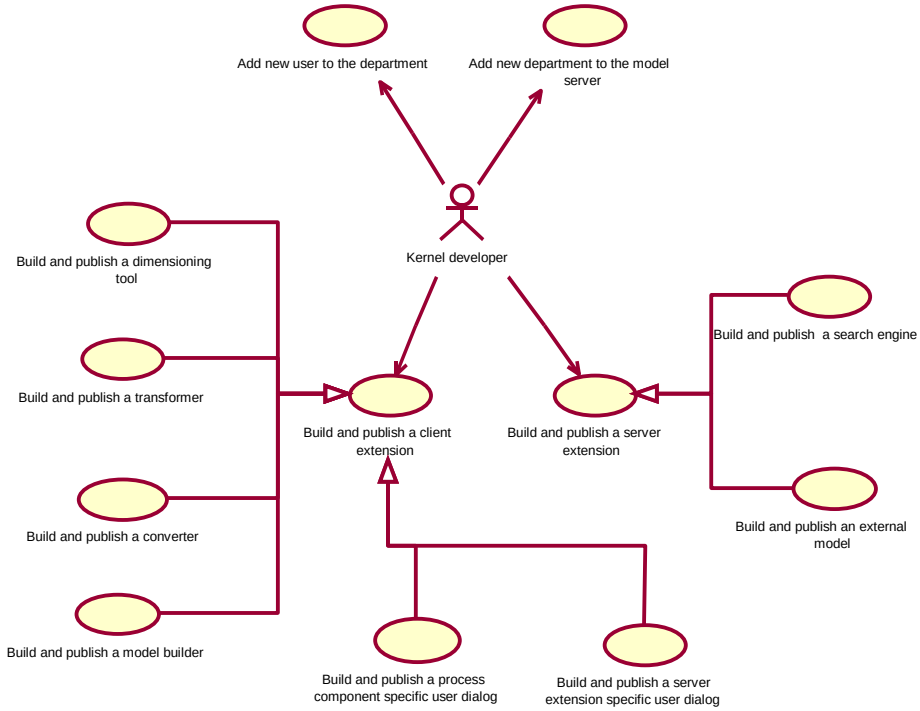

Figure 4.2. Use cases of kernel developer.

4.2.2 Provider

The provider users provide *model templates* and *process component data* for the model configurators. Equipment manufacturers and library developers are providers. However, securitywise these two actors belong to the same provider role.

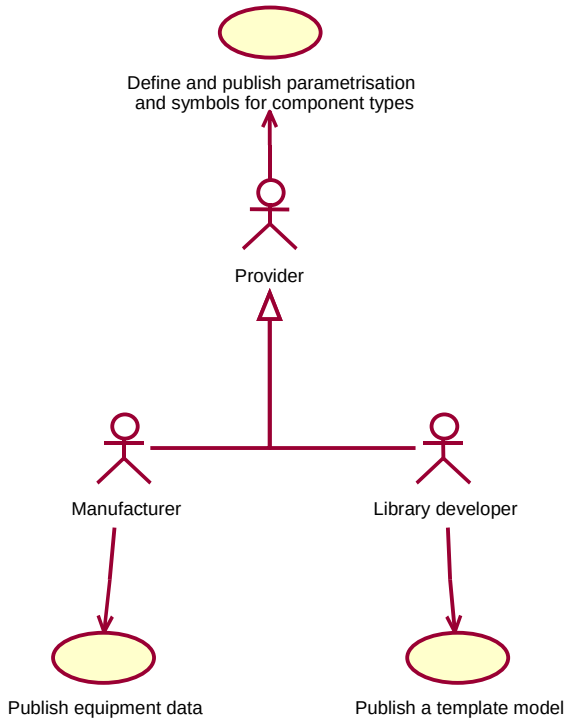

Figure 4.3. Use cases of provider.

The use cases of providers are shown in Figure 4.3. The providers are responsible for the development and maintenance of type and symbol libraries. Type and symbol libraries are elements of the data model of the architecture. They contain process component type descriptions and graphical symbols as will be explained in the data view section. The library developers publish template models for the model configurators. One model server in the architecture has a special role for this. It acts as a model and parameter value repository for all simulation users. This model server is called the *Gallery*. Equipment

manufacturers may also publish process component data of their equipment in the Gallery server. The use of the Gallery server in different scenarios is discussed further in Chapter 5.

4.2.3 Model configurator

The model configurators use the building blocks and tools that the kernel developers and providers have published to configure simulation models. The model configurators are *process and automation designers* working at consulting offices, equipment and automation supplier companies or research institutes.

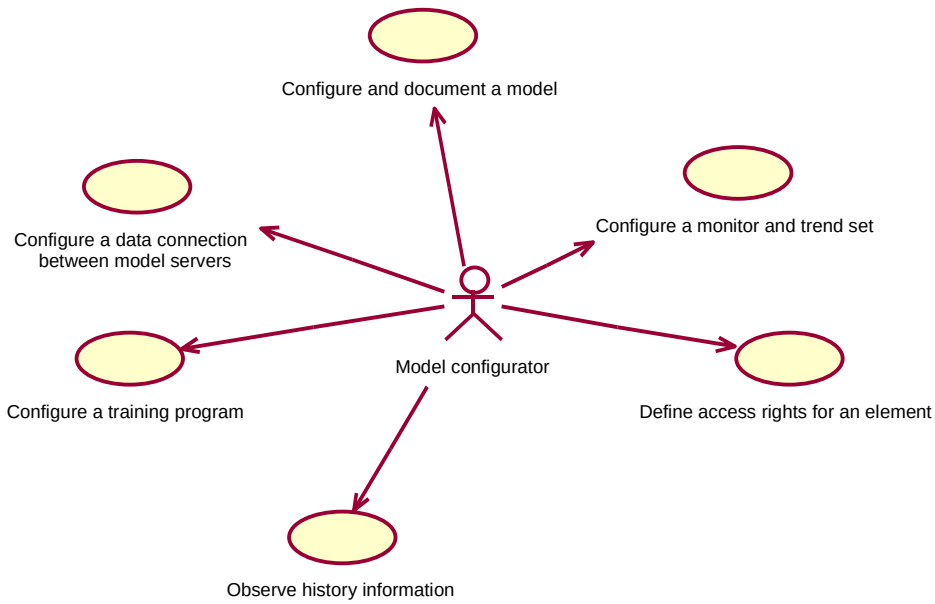

Figure 4.4. Use cases of model configurator.

The use cases of the model configurator are shown in Figure 4.4. In addition to configuring and documenting simulation models using a model configuration user interface, the model configurators define monitor and trend sets using *vertical* OPC DA data connections. There might be several different monitor and trend sets configured in the same model using different model servers as the data source for each set.

The model configurators also configure *horizontal* OPC DA data connections between model servers. The configuration is done using a data exchange user interface. The data connection between the model servers can be synchronous or asynchronous depending on the simulation execution in the model servers.

Training programs are configured for the instructor's interface by the model configurator. Training programs are operational situations in the modelled processes that are of particular interest in *operator training*. For example, a training program may consist of a malfunction, shut down or start situation.

The kernel developers, providers and model configurators all define *access rights* for the elements they create to the model server. Access rights for reading, writing and executing are given for user groups as will be explained in the data and security views. All three user groups may also observe *history information* about the elements created and modified in the model server.

4.2.4 Model user

Simulation models configured by the model configurators are used by the model users. Instructors, engineers and operators are model users. Instructors and operators use simulation models during simulation-assisted training. *Engineers* are model users who use the model for example for testing of the control system. The engineers may also use demonstrational models for example for marketing purposes. The engineers can also browse the database of the Gallery when searching for suitable process components. It has to be noticed that securitywise these three actors have the same role of a model user. The use cases of the model user are shown in Figure 4.5.

Operators use an operator's user interface for observing monitor and trend data and for changing values of the control parameters, e.g., set point values and binary controls. Vertical OPC DA connections can be used for observing data.

The engineers load model and communication configurations, they can start and stop a simulation or communication, and they can also save and load different model states. One model configuration may have several states as will be explained in Section 4.5. The engineers may also browse the content and documentation in the model server.

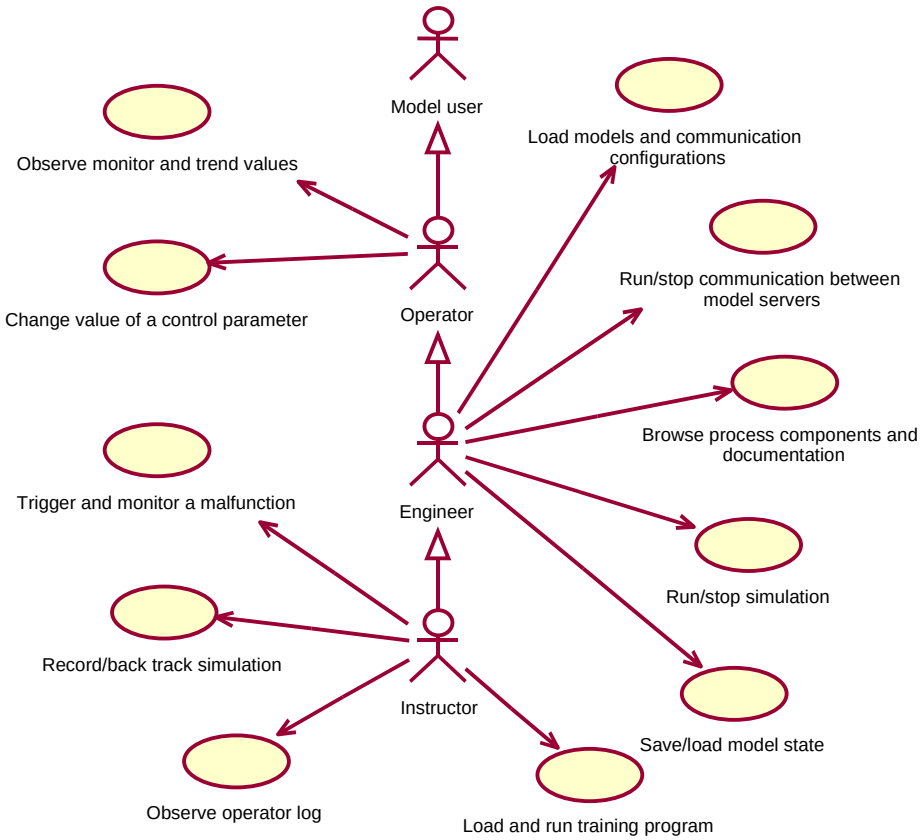

Figure 4.5. Use cases of model user.

Instructors load and run different training programs using an instructor's interface. They can record and backtrack the simulation, trigger and monitor malfunctions and observe the operator log information. Of course, in many cases the instructors do the same things as the engineers and the operators and the

engineers do the same things as the operators. However, securitywise all three are equal model users.

4.3 Viewpoints

The viewpoints selected for this architectural description are the *logical, data, security, component, process and deployment* viewpoints. The selection has similarities with the "4+1" approach (Kruchten 1995). The data and security viewpoints have been added. The physical viewpoint has also been renamed as deployment viewpoint. In addition the scenarios are not only seen as a supporting viewpoint for the other viewpoints but also as a route from other viewpoints to deployment viewpoint. The different viewpoints and the description procedure are shown in Figure 4.6.

In the *logical viewpoint* the system is decomposed into a set of key abstractions. The services of the system to its users are described using these concepts and their relationships. This viewpoint serves all of the four user groups and it is used to cover all of their concerns. Together with the use case analyses, logical viewpoint describes the functionality provided by the architecture. The logical view of the proposed architecture is analysed in Section 4.4 using the UML class diagram format.

The *data viewpoint* describes the data model of the system. This is the data model common to all software components in the architecture. Individual software components may have their own inner data models that are not described in this viewpoint. The data viewpoint addresses all the concerns of the programmers of the system, the kernel developers, the providers and even the model configurators. The data model of the proposed architecture is described in Section 4.5 using a UML class diagram. The name of the data model is GML (Gallery Markup Language).

The *security viewpoint* describes the security model of the system. In a distributed, multi-user environment the security viewpoint as a separate viewpoint is justified even though many of the security issues may be discussed already in the other viewpoints. The security model as a whole is described in Section 4.6. This viewpoint especially addresses the publishing concerns of the providers.

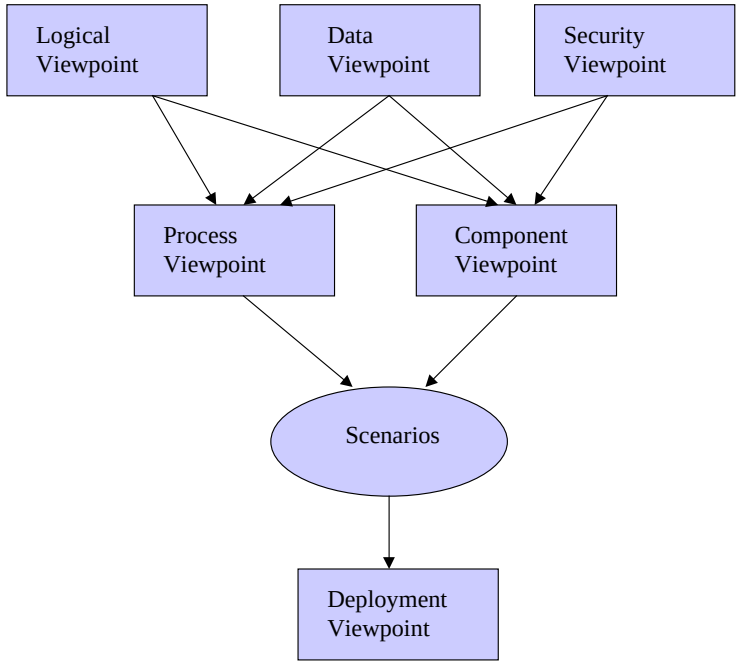

Figure 4.6. Different viewpoints and the description procedure. The level of generality in the architectural description with respect to implementation techniques decreases in the viewpoints further down.

The software components and their relationships are analysed in the *component viewpoint*. The system is broken down into subsystems and program libraries in the way they are developed in the software development environment. The main interfaces of the system are identified and specified. Usually one software component is developed by one developer or a small number of developers. This viewpoint addresses all the concerns of programmers of the system and the kernel developers. The software component organisation of the proposed architecture is described in Section 4.7 using the UML component diagram format.

The *process viewpoint* takes into account non-functional requirements, such as performance and availability. It addresses issues of concurrency and distribution, and how the main abstractions from the logical viewpoint fit within the process architecture. This viewpoint addresses all the concerns of the kernel developers and programmers of the system. The process view of the proposed architecture is analysed in Section 4.8. UML component diagram is used to describe the process view.

The *deployment viewpoint* describes the physical architecture of the system. The proposed architecture may have a different deployment in different cases. These different deployments use certain features of the proposed architecture. Each case can be described as a use scenario. The scenarios are chosen so that they measure the desired customisation, reuse, co-use, extensibility and flexible connectivity features of the proposed architecture. The scenarios are described using the UML collaboration diagram format. The objects in the collaboration diagrams are instances of the classes in the logical view. Each scenario has also a physical deployment. This deployment is described as a UML deployment diagram. The different scenarios and deployments are represented in Chapter 5. This viewpoint addresses all the concerns of all of the users. The viewpoint clarifies the concepts and mechanisms of the architecture by giving examples of the usage of the software system. In particular, this viewpoint provides the cases to verify the two main hypotheses of this thesis.

4.4 Logical view

The logical view to the proposed architecture is described in Figure 4.7 as a UML class diagram. The architecture consists of *model clients* and *model servers*. As will be shown (Chapter 5), the model server can be a *steady state simulator*, *dynamic simulator*, *database* or a *process station* of a control system. The non-exclusive specialisation in the diagrams refers to the possibility that a model server instance can be any combination of the sub-types at the same time. The model client can be a *model configuration user interface*, *data exchange configuration user interface*, *the instructor's user interface* or *the operator's user interface*.

The most important part of the architecture is the interface between the model clients and model servers. In the logical view this is modelled as a *connection*. The connection can be a *configurational connection* or a *data connection*.

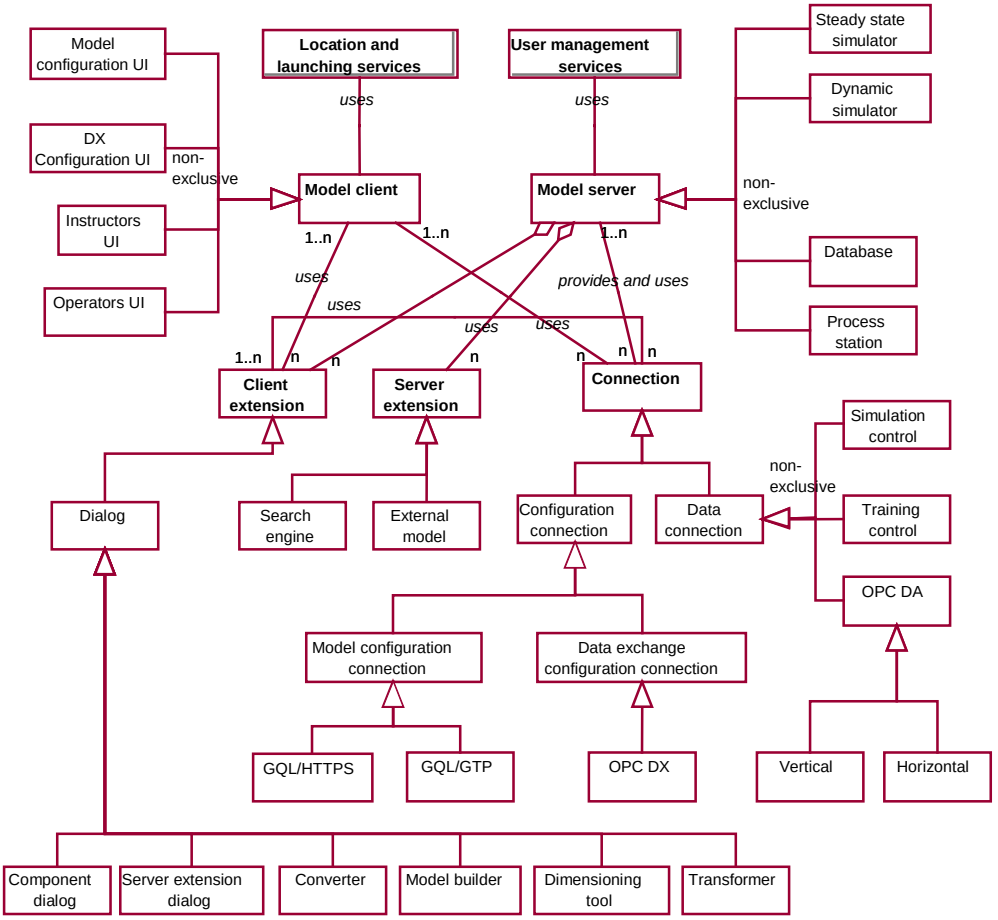

Figure 4.7. Logical view of the architecture. The main concepts are marked in bold type.

The configurational connection can be a *model configuration* or a *data exchange configuration connection*. The interface used for the model configuration is known as a GQL (Gallery Query Language) interface. The interface is described using a protocol that resembles SOAP (Simple Object Access Protocol) (W3C

2000a), and it is further discussed in Section 4.7. The underlying protocol for transporting GQL requests can be either HTTPS (Hyper Text Transfer Protocol Secure) (W3C 1999) or tailored Gallery Transfer Protocol (GTP) based on the TCP/IP (Transmission Control Protocol /Internet Protocol).

The data connection can be an *OPC DA connection*, *simulation control connection* or *training control connection*. Again, these types are non-exclusive. Instead, a data connection instance can be any combination of these three sub-types at the same time. Furthermore, an OPC DA connection can be *vertical* or *horizontal*. A vertical OPC DA connection is created between the model clients and model servers whereas a horizontal OPC DA connection is created between two model servers. Simulation and training control connections are further discussed in Chapter 4.6.

Server and client extension mechanisms have been designed to the architecture in order to achieve generic extensibility. These mechanisms can be used for easier model customisation, reuse and co-use purposes as is shown in Chapter 5.

A server extension is a software component that is invoked and executed as a part of the model server. It has an invocation interface that depends on its type and it can use the GQL interface of the model server. The server extension can be a *search engine* or an *external model*. Search engines are used for searching or selecting process components from the database of the model server. External models are algorithms that are used to extend the simulation functionality of the model server.

A client extension is a software component that is invoked and executed as a part of the model client. It has an invocation interface that depends on its type. The client extension can be a *component dialog*, *server extension dialog, dimensioning tool, transformer, model builder or converter*. Component and server extension dialogs are used as specialised user interfaces for different process components and for the different server extensions. Dimensioning tools are used for finding the right parameter values for process components in the database of the model server. Transformers are used for transforming a process component or a set of process components and their parameters from the type description system of one model server to another. Model builders are used for parametrisized creation of simulation models. A converter is a client extension

for converting the data of the equipment manufacturer to GML format. Client extensions can use both configurational and data connections independently of the model client.

The architecture also provides *location and launching services* for the model clients. Through these services, the clients can list the available model servers in different hosts, launch the desired model servers and create connections. The architecture provides *user management services* for the model servers for authentication of the users and for finding the registered role for the different users. User roles are discussed more in the use case analyses and in the security view sections.

4.4.1 Rationale

The distributed *client-server* pattern, the division of connections into configurational and data connections, and the use of client and server extension mechanisms are the most important architectural choices that can be seen in the logical view.

The lifecycle of numerical solution algorithms in a process simulation is longer than the lifecycle of a configuration user interface. The development of the solution algorithms of for example Apros started already in 1985 and the algorithms are still in use today. Meanwhile the same solver has had several different user interfaces in different operating systems and in different operating system versions (Apros 1999). Thus, it is rational to separate the solution algorithms from the user interface code. As described in Section 2.3, none of the examined simulators support full multi-user features. The possible need for multi-user features and solver distribution is also an argument for separating the solver code and user interface code even into different processes. Furthermore, it is easier to connect legacy simulators to the architecture using this approach. One can always argue whether tighter integration would lead to better usability. However, this fact depends heavily on how good the software interface between the solver and the user interface is.

The model configurators need both *configurational access and data access*. However, the model users do not need configurational access. Thus, it is

beneficial if the architecture is designed and implemented so that it can function separately with both configurational and data connections. These connection types are also very different in context and performance. Furthermore, OPC as a de-facto standard in the field of automation is a justified choice for the data connection whereas XML and XML-based message passing are reasonable choices for expressing and manipulating the structural model topology and process component parameter values. XML is designed for expressing structural data. It is also designed to be well suited for network usage. There is a good choice of ready-made tools and software components supporting its usage, e.g. parsers, editors and databases.

The requirement of *generic extensibility* leads to the decision of the use of the server and client extension mechanisms. The functionality of the model server can be extended at run-time by using software component technology and an application programming interface to the database of the model server. User interfaces for different process components and for different server extensions can be created run-time using the client extension mechanism.

4.5 Data view

The data view explains the common data model for the architecture. This is the data model that the model clients and servers must agree on in order to communicate with each other. The data model is different for a configuration connection and for a data connection. This section mainly discusses the data model for a configuration connection and, at the end, mapping to the data model of a data connection is introduced.

The data model of the suggested architecture is shown in Figure 4.8 as a UML class diagram. The data is processed in the architecture in XML format and thus the class diagram has one to one correspondence to XML. The XML format is GML (Gallery Markup Language). The diagram can be read so that classes are XML elements, class attributes are element attributes and aggregation relations are element containments. The administration element, its sub-elements, and the rights elements are discussed in the security view. The other elements are explained in this section.

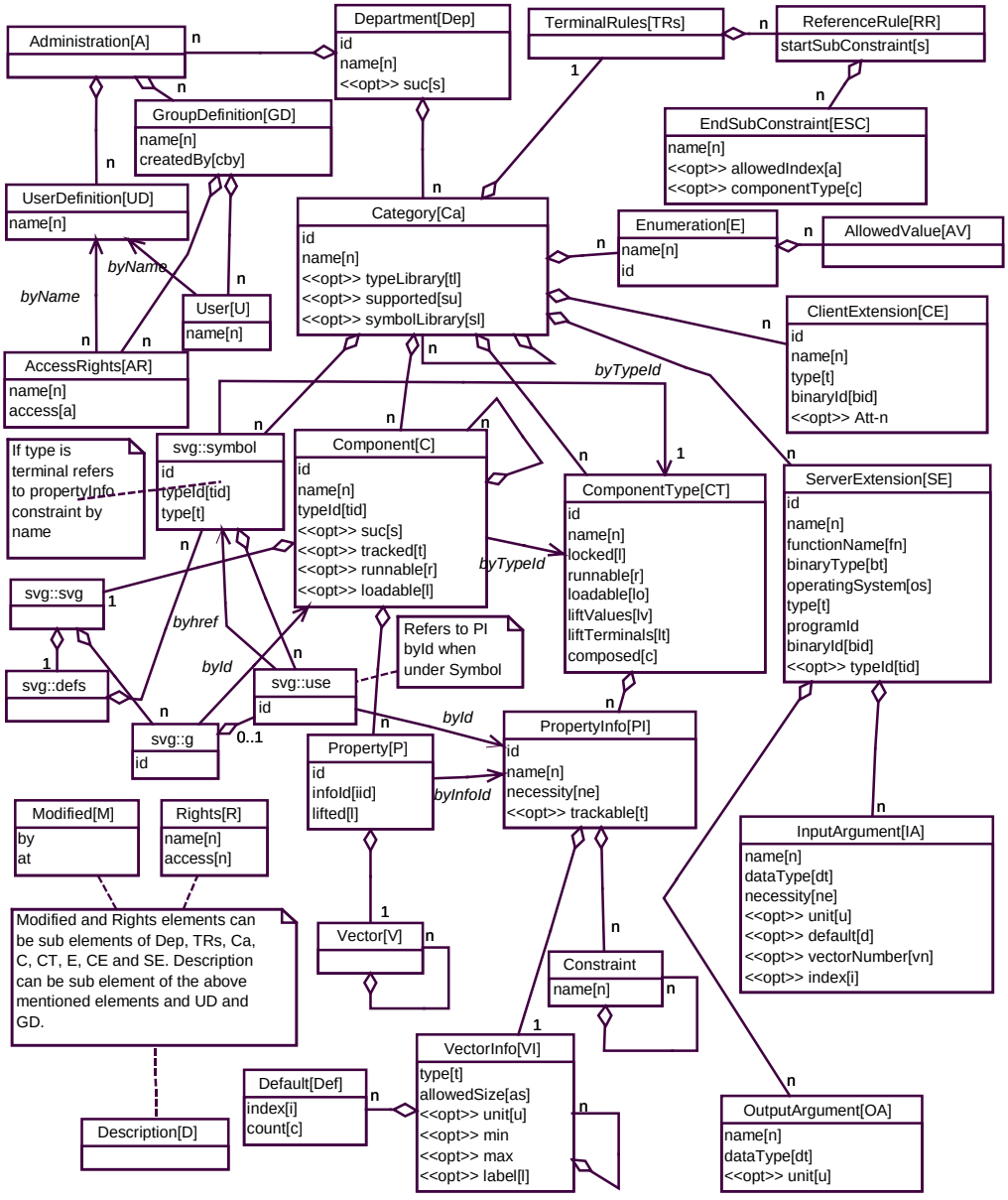

Figure 4.8. Data view of the architecture.

4.5.1 Basic elements

The data in the model server is divided into *departments*. A department is a root-level concept with a name and an id. The users and user groups are defined at the department level. The model configurations under one department conform to the modelling rules defined in that department. Thus, in many cases the department has correspondence to the model server type, e.g. a certain simulator, linked to the architecture.

The departments consist of *categories*. The categories can be normal categories, type libraries or symbol libraries depending on the attribute values. The modelling rules of the department are defined under the *type library*. The type library consists of component type descriptions and one *terminal rule* description. *Component types* are descriptions of the process, automation and electrical components that can be modelled under the department in question. The component types can be placed under normal categories in the type library. Terminal rules describe the reference rules that are used when connecting process components under the department. The graphical *symbols* for different component types and terminals are defined under the *symbol library*. Graphical symbols are defined by using SVG (Scalable Vector Graphics) and they have references to the component type as will be explained later. It is up to the kernel developers and providers to define the component types, symbols and terminal rules.

Many elements in the data model have an id attribute. In the architecture, universally unique identifiers (UUID) (DCE 1998) are used as ids. In addition, a UUID has to be prefixed with a letter so that the id conforms to XML id specification. ‘a’ or ‘b’ is used as a prefix in the data model. For example, aD061ECA0-C4B7-457a-A0E3-ADCB0CAECEAB could be a valid id in the architecture. One can argue that 37 characters take a lot of space for an id. However, as will be seen, the use of UUIDs is justified in a distributed modelling environment.

The model configurations are represented under normal categories as *component* hierarchies. The components refer to component type descriptions. This is one rationale for the usage of UUIDs as identifiers. The reference to the component type description has to be the same regardless of to which model server the

model configuration is copied. The components consist of *properties* that refer to the corresponding *property infos* in the type description. The property values are expressed as *vector* hierarchies that are also described in the type description. The graphics of each of the model configuration levels are expressed as SVG under the *svg element* of the parent component. There are references from the SVG to the components as will be explained later. It is up to the providers and model configurators to define the model configurations. The data model does not contain a concept for a model. The root components that contain other components are referred to as models.

Model configurations may take a lot of space in XML format. When the data is processed in the different interfaces in the architecture, short names for the elements and attributes are used. These short names are expressed in brackets in Figure 4.8.

4.5.2 Component type description mechanism

Component types are used to describe the process components. The models are configured under the department using component types from the type library of the department. It should be noticed that the component hierarchy is not described in the component type description mechanism. The parametrisation of the process component is described using property infos. The necessity of property info can be optional, implied or required. If the necessity is required it means that the property is always created under the component and the user always has to give value for the property during its creation. If the necessity is implied then the property is also always created. However, if the user does not give the value the value is taken from the default element in the property info description. If the necessity is optional the property is created under the component only if the user wishes to create it. The component type is marked locked as long as the provider or kernel developer makes changes to it. When the type is locked no instances of that type can be created. After the component type has been opened other users can start to create instances of that type and the type description cannot be changed any more. This mechanism ensures that a component always conforms to its type description.

The values of a property are described under the property info using *vector info* hierarchies. Vector info has type, allowed size, unit, min, max and label attributes. The type of vector info can be any primitive type (boolean, short, integer, float, double, string, file, reference) or it can be a complex data type (vector, enumeration). A file property is a reference to a file. A reference is a string value that defines the UUID of the referred element. If the type is a vector it means that the instance vector element has vector elements as its sub elements. These sub vector elements are all described in the corresponding vector info elements in the description. The enumeration types are described by the kernel developers and providers under the *enumeration* elements. The description is done into the same type library as the component types that are using the enumerations.

4.5.3 Client and server extension description mechanism

Binary client and server extension software components are described also in the data model as XML elements. There can be many XML entries for one binary software component. The *client and server extension elements* are placed under a type library. It is up to the kernel developer to define the client and server extension entries. In the client extension definition the kernel developer defines the name, type and component type or server extension reference list. The type can be a component dialog, server extension dialog, dimensioning tool, transformer, model builder or converter. The kernel developer also provides the binary software component for the extension. This software component is referenced using the binaryId and programId attributes. In the server extension description the kernel developer gives the name, type, function name, and some other attributes. The function name refers to the name of the function in the interface of the software component. In addition, the kernel developer defines the *input and output arguments* for the server extension. Some of the arguments are predefined for the server extension type in question, but the other arguments are free. The input and output argument definitions and the function name are used by the model server when invoking the server extension. The server extension type can be a search engine or an external model. In external models a reference type of argument (both input and output) would be useful and will be specified to the data model in the future.

4.5.4 Connection mechanism

The user can connect components using a connection mechanism. The connection is done by using *terminal properties*. The terminal property can be a *start terminal* or *end terminal*. Thus, the connections can be directional if needed. The start terminal and end terminal properties are described in the component type description using property infos with start terminal or end terminal constraints. The type of vector info of a terminal is reference and the allowed size is two. The first vector position is used for a reference from the start terminal to the end terminal (connection reference). The second position is used for the mapping mechanism (mapping reference). In addition, the start and end terminals can be typed. The kernel developers and providers can create terminal rules for guiding the connection of the process components by using the typing mechanism. The *type of the terminal* is expressed as a sub-constraint for the start and end terminal constraints. The corresponding rule for the terminal type in question can be found under the terminal rules element.

4.5.5 Mapping mechanism

The mapping mechanism is used for two purposes. The terminals of a component at a lower level can be mapped to a higher level (*terminal mapping*) and the *value properties* at a lower level can be mapped to a higher level (*value mapping*). There are two encapsulation rules in the data model. First, there cannot be a connection reference between the terminals of components that have a different parent component. Second, the mapping reference can refer only to the properties of components at the next sub level. These rules force the connection references between different component hierarchies to go through the upper level and thus the component encapsulates its content. The value mapping property is described using property info with a value mapping constraint. This vector info has the allowed size of one and its type is reference. The value mapping property can refer to a non-terminal property at the next sub level. This reference makes the value property of the lower level also visible at the higher level.

The mapping mechanism for terminals and values, described above, requires that the mapping properties are defined in the component type description. This kind

of mapping is useful when the model configurator uses the top down modelling method. In this method, the upper level component and its interface are described first. After that, the content is configured for it. However, in some cases the model configurator does not know the upper level interface and the bottom up modelling method is used. An empty component is created and its content is configured. After that, terminals and values are mapped to the higher level in order to define the interface for the empty component. This would be impossible using the mechanism described above, because the component type description for the empty component would be already open and could not be changed. This is the reason why there are two different mapping mechanisms in the data model. The above-described mapping mechanism is called *static mapping*, because the descriptions for the mapping properties are predefined and can be found in the component type. The other mapping mechanism is called *dynamic mapping*. In dynamic mapping, the property from the lower level can be lifted onto a higher level even though its property info description cannot be found at the upper level. Dynamically mapped property refers to the property info at the lower level. Whether the value properties and terminals can be dynamically mapped can be controlled by the kernel developer and provider using a lift value and lift terminal attributes in the component type. They can also control whether the component can have child components using a composed attribute.

There can be several departments in one model server and thus several type libraries. In other words, it is possible to configure a model to a model server using the component types of another model server. However, the model cannot be necessarily simulated in the model server. The component types the instances of which can be simulated in the model server are located in a type library marked as *supported* by the server. There can be database model servers where none of the type libraries are supported, i.e. the model server does not contain any solution algorithms for simulation. The component types the instances of which form entities that can be simulated are marked as *runnable* by the server. Usually these are the component types of the root components of the model configuration hierarchy. In the same way, the model server can mark the component types the instances of which form entities that can be loaded to the memory of the model server separately *loadable*. This mechanism is meant for model servers that cannot handle all different model configurations at the same

time. These model servers usually have their own file format for saving the model configurations.

4.5.6 Documentation and history mechanisms

The model server keeps track of the changes made to the data in the server. This is illustrated in the data view using *modified elements*. Modified elements can be sub elements of departments, categories, client extensions, server extensions, terminal rules, enumerations, component types and components. Whether the changes made to the component are tracked or not depends on the tracked attribute value in the component. Furthermore, the changes in the property values change the history information of the component only if the kernel developer or provider has marked the corresponding property info as trackable. All the above mentioned elements that can have modified sub-elements can also have *description sub elements*. The user can write description information to the different elements in the data model and the description elements help generate user documentation from the data.

4.5.7 Graphical descriptions

In modern simulation tools, the simulation models are configured using graphical diagrams. However, in many cases the graphical descriptions are separate from the model parameter and topology information. Often the graphics also has its own format. In the GML data model graphics is also expressed in XML format. The Scalable Vector Graphics (SVG) standard is used for this purpose.

The kernel developer can draw graphical symbols for different component types, terminals and connections. The symbols are published under the symbol library category. SVG symbols for terminals refer to the terminal type using the name of the terminal type (sub-constraint name under the start terminal or end terminal constraint in the property info) that has to be unique under a certain department. The SVG symbol for the component type refers to the component type by type id. In addition, the component type symbol can use terminal symbols. These SVG use elements refer to the corresponding property infos by id.

The graphics of the child components is represented under the SVG element of the parent component. When the graphical information is returned from the server to the client programs, the needed terminal and component type symbols are copied under the SVG element using a SVG defs declaration. Connection symbols behave differently in this respect. They are only used as templates when the connection lines are drawn and after that the dependency of a specific connection line to its symbol template disappears. In other words, the connection symbols are not copied under the defs element. The connection line symbol graphics is also limited to SVG polylines. The style of the polyline can be different for different connection symbols.

Components are expressed under SVG g-elements with SVG use elements. The g-elements refer to the actual component by id and the use elements refer to the symbols using the href attribute. A connection component also behaves here slightly differently. It is described also under the g-element that refers to the corresponding component, but it does not use any symbol information. Instead, the graphics of the connection line is described separately for every instance under the g-element.

An example of the usage of the data model is given in Appendix B. The example includes also some graphical descriptions. The example illustrates the reference mechanisms explained above. It should be noticed that the graphics part of the data model has not yet been verified in the implementation.

4.5.8 Monitor, trend and state definitions

Monitor and trend definitions should also be a part of the model configuration. They have not yet been designed to the data model and thus not represented in Figure 4.8. Monitor definitions are related to the svg-definitions of a component. The monitor definitions should be represented in the same way as the graphics of the sub-components are represented in the svg element of the component. The extra requirement is that the monitor definitions should also include the coordinates for retrieving the value through data connections, i.e. the model server host, model server and property (item) id should be included in the definition. It would also be convenient if the user could group the monitor definitions to monitor sets and activate and de-activate them when needed.

Trend definitions are usually related to the root (model) components. However, it could be convenient if the trend definitions were made under any component. In the same way as in monitors, the trend definition should include the coordinates for the retrieved property values, i.e. the model server host, model server and property (item) id.

The different states of the model should be also part of the model configuration. This feature has not yet been designed to the data model either. In current process simulators the model configuration is saved when the state of the model is saved. When working with big models this usually means that many large files have to be stored that include the same configurational information over and over again. Only the calculated variables may have different values. This could be avoided if the state of the model could be saved and activated inside one model configuration. In the suggested data model this would mean for example adding additional vector hierarchies under the property element in the component instance. Each of the vector hierarchies would correspond to a saved state. The different states would be listed in the root (model) component and the active state could be marked. Of course, if the model configuration was changed, i.e. components were deleted or added, the corresponding states with the same default values would have to be generated for the added components.

4.5.9 Data model of data connection

The data model of the data connection is much simpler than the data model of the configuration connection. As explained in Chapter 2, OPC DA uses three concepts: server, group and item. The mapping of this model to the configuration connection data model is quite intuitive. The items that can be monitored and observed using trends are the component properties. In the simulation control connection the run, stop, load and save functions are directed to the runnable and loadable components. In the training control connection the malfunction concepts are most likely modelled as malfunction components in the model server. Different interfaces are discussed in more detailed in component view.

4.5.10 References to other specifications

Though there are many mechanisms embedded to the data model some commonly used mechanisms are also left out. For example inheritance could be useful in some cases for component types. The kernel developers and providers could arrange the component types in a type library into inheritance hierarchies. However, in the field of process components the *inheritance* hierarchies may not be so evident and would require very careful planning.

In CAPE-Open, ports have a name, type and direction. In the same way terminals in the suggested data model have a name, type (typing sub-constraint) and direction (start or end). However, it has to be understood that although ports in CAPE-Open describe the actual interface between the software component (containing the solution algorithm or behaviour) and the simulation engine, terminals (and references between them) in the suggested data model only describe the topology of the process. A model server and server extensions can use this topological information any way they wish in the inner solution algorithms. The Rome repository described in Section 2.4 is based in many respects on the CAPE-Open specification, so it has also the same differences compared to the suggested data model.

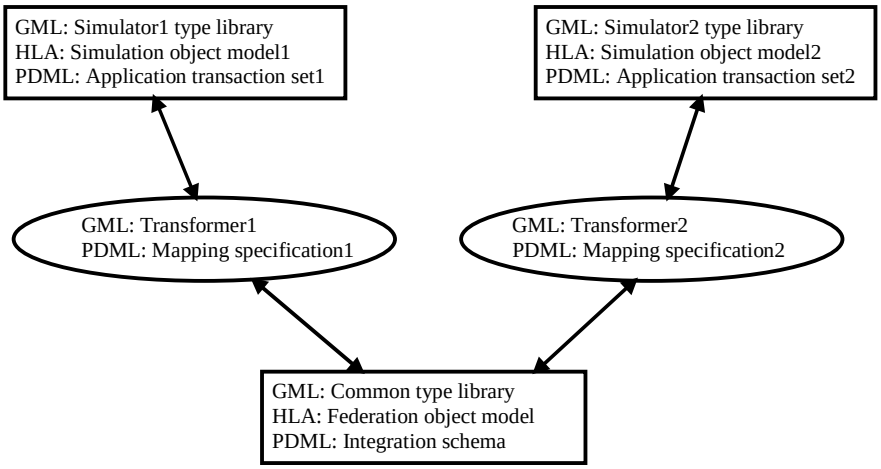

Figure 4.9. Similarities between approaches using different specifications.

Three HLA rules were listed in Section 2.2.2. The last one of the listed rules says that a federate shall have the simulation object model documented and the first one says that a federation shall have the federation object model documented. The department may have a correspondence to the model server type, e.g. a specific simulator, linked to the architecture. Thus, the type library of that kind of a department can be interpreted as a documentation of the simulation object model for that model server. Furthermore, the user can also define a department that does not have a corresponding model server type. The type library of that department can be used to document a federation object model common to several model servers. Transformations can be done between the different departments. If a *common model* exists the transformation can always go through that common model and when new model servers are linked to the architecture they have to define only the transformation to the common model. However, the common model may be difficult to describe. In those cases, the transformations can be done straight from one department corresponding to a certain model server type to another.

Also the PDML approach discussed in Section 2.2.4 has similarities to the suggested data model in the same way as above. Type libraries of certain model server can be interpreted as application transaction sets in PDML. The integration schema, on the other hand, can be modelled as a common type library. Mapping specifications are transformers between model configurations in the different departments. The similarities between approaches using different specifications are shown in Figure 4.9.

A DTD or XML schema can be written for component type description to assist the kernel developers and providers in writing valid component types. This approach is similar to the late binding approach of STEP Part 28, discussed in Section 2.2.3. Another possible approach is to use early binding. This means that the kernel developers and providers write DTD or XML schema descriptions for component types. These descriptions are valid according to the ‘meta-DTD’ or ‘meta-Schema’. In order to implement simple functional model servers the ‘meta-DTD’ or ‘meta-Schema’ has to be described in a very compact manner. This means that the kernel developers have to write similar component type descriptions as in the suggested approach but using XML schemas. It is, however, easier to write language than to write grammar. Thus, the *late binding* approach was considered to be simpler for the end users.

4.6 Security view

The security model of the architecture can be divided into three parts. The lowest level is the *encryption* of the configurational data between the model servers and model clients. Configuration connections that use HTTPS as a transport protocol are always encrypted using SSL/TLS technique. Also those data connections that use HTTPS underneath use the same encryption. The client authentication feature of SSL/TLS is not used. The encryption feature is especially important when using Internet-based model and parameter Galleries or any other model server located in the Internet. Equipment manufacturers or library developers do not want their information to end up to a third party. Without the appropriate encryption it is possible to steal configurations from the line.

Every connection in the architecture is authenticated. The *authentication* is done using the user name and password. It should be noticed that in data connections the third level security of the OPC security specification has to be implemented for the model servers in order to support the required security model also in the data connection. The users of the architecture are categorised into four *user roles* that correspond to the users listed in the use case analyses. The functions of the main interface of the architecture (GQL) are categorised accordingly. This means that a user in a certain user role can call only the functions allowed for that user role. The user roles are nested so that the kernel developer can call all the functions that are allowed for the provider, the provider can call all the functions allowed for the model configurator and the model configurator can call all the functions allowed for the model user.

The third level in the security is the *access mechanism* inside the data model configurated by the user. User definitions are added under the administration element in the department by the kernel developer. Model configurators can create user group definitions, add users into them, and define access rights for the user groups. Model configurators can also define rights for different elements in the data model by specifying the user groups for whom the access is granted. The access can be read or write.

4.7 Component view

The component view describes the software *component structure* and the main *interfaces* of the architecture. The component view as an UML component diagram is shown in figures 4.10 and 4.11. A pure *layered architecture* would mean that the software components at one level can access only software components at a lower level. In this sense, the architecture is not layered. However, the components are grouped into layers because these layers form functional groups. The component view is divided into two figures so that Figure 4.10 represents the view of a model server and Figure 4.11 represents the view of a model client.

The *transport layer* handles the data transmission between the model clients and model servers. The main software components of the transport layer are MCTL (Model Client Transport Layer), MSTL (Model Server Transport Layer) and MSM (Model Server Manager). MCTL and MSTL are program libraries that offer functionality for higher level components. MSM is a demon process that is used for user management and launching services. MSM is always running in a host where the model servers are executed. In GTP-based communication, these are the only components that are used in the transport layer. With HTTPS, commercial web server and stub components are used. The idea on the server side is to catch the HTTPS messages at the stub and redirect them to the model server in the local network using GTP. MSM and GQL services are used through MCTLStub. Data services are used through OPCXMLStub. The message used for the communication between OPCXMLProxy and Stub comply with the draft specification of OPC XML.

The next layer in the model server handles the persistence, data access and extensibility of the model server. All the software components in this layer as well as the transport layer software components can be reused when legacy applications or new applications are linked to the architecture. OPCKit implements the OPC DA and OPC DX interfaces for the model server. In addition, it implements the simulation control and training control interfaces that are extensions to the OPC DA functionality. GQLKit implements GQL functionality for the system. The GQLKit handles security issues and persistence so that all GML elements that are created to the GQLKit are persistent. The idea is that the user of the OPCKit and GQLKit libraries can register call backs to the

libraries and customise the response to different functions in his own code. The GQL specification contains a function for invoking the server extension components. In this way, the client can use the extended functionality in the server. In addition, the server extensions can be invoked from the simulator code. The server extensions themselves can access the GQLKit using an inner GQL interface.

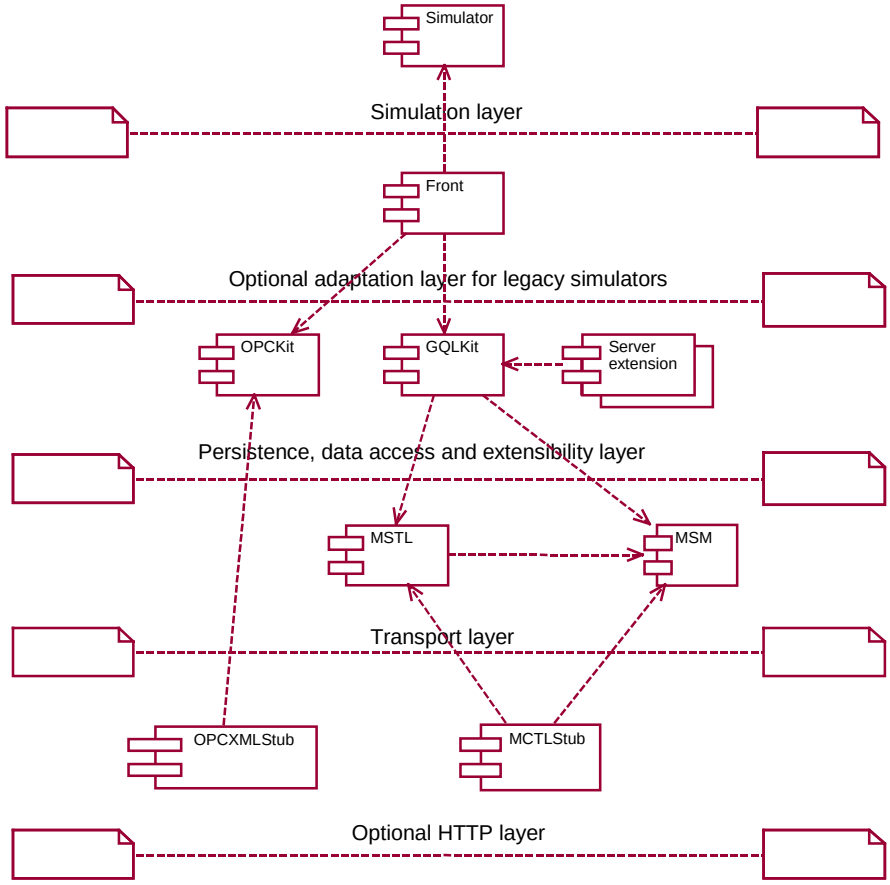

Figure 4.10. Model server software components.

When legacy simulators are linked to the architecture it is often beneficial to implement an adaptation software component between the simulator and the generic kit libraries. In this software component known as Front (often prefixed

with the name of the simulator for example AprosFront) the model access and data access functionality is customised for the simulation tool in question.

On the client side the user interface can use the services of the transport layer directly or it can use an optional *adaptation layer*. Different GQL messages are expressed as function calls in the MCKit interface. In the same way, the OPCXMLProxy implements the OPC DA interfaces for the HTTPS model server. This means that the client can access the interfaces as if they were ordinary OPC DA interfaces in the local network. Client extensions are invoked from the user interface and the client extensions can access the model servers using GQL and transport layer services. The configuration user interface implemented for the architecture is called Model Explorer.

The most important interfaces in the architecture are the GQL, OPC DA, OPC DX, simulation control and training control interfaces. The GQL interface is the function interface that is used to manipulate the GML data model described in the previous section. The GQL interface can be used through different software components in the architecture. MCKit for example exposes the GQL interface as a COM automation interface whereas GQLKit exposes it in C++ format for server extensions and for simulator vendors. The GQL interface between the model servers and model clients is defined using SOAP like XML messages. The following is an example of a GQL request and response. It can be seen that the GQL syntax could be easily changed into SOAP. However, when the implementation of the system was started the SOAP standard was still in a draft phase and SOAP tools were not as advanced as today. This was the reason for using a tailored syntax in the message passing.

```
<Request>
   <SetCategory>
      <Department>aDAF105DD-0A80-4ec4-AB57- 54F5D7E71D16</Department>
      <Parent>a7E83E3E0-5DEC-4e98-AF53-3890E7ADD740</Parent>
      <Name>Enumerations</Name>
      <Id>a03874FE8-DA5E-46b3-9154-46E0BB5D8E97</Id>
   </SetCategory>
</Request>
```

```
<Response status="ok">
   <Return seq="1"/>
</Response>
```

The user can create categories to the server by using the SetCategory -function. The GQL specification defines almost 100 functions for the manipulation of GML data. These functions are not listed in this thesis.

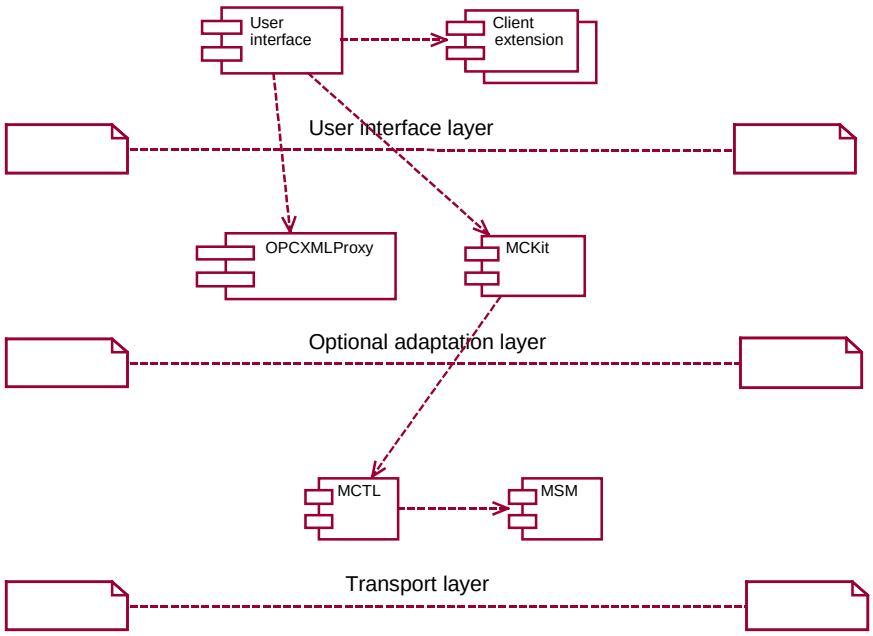

Figure 4.11. Model client components.

The OPC DA interfaces are used for accessing simulation data in the model servers. The simulation results can be monitored and trended through OPC DA. The data exchange between different model servers is also handled using OPC DA. If the model server is OPC DX compliant this kind of horizontal data connection is configured using the DX standard. If the server does not support OPC DX then the data between the model servers is transmitted through the data exchange configuration user interface. The data from the HTTPS model server is transmitted to another model server in the same way using the data exchange

configuration user interface. The user interface uses the OPCXMLProxy/stub mechanism underneath. The data exchange configuration user interface implemented for the architecture is called X-Connector.

The simulation control interfaces are used for loading and saving models, for listing and changing the model states, for running and stopping the simulation and for querying the simulation control variables such as real time ratio or simulation current time. One important feature that is still missing from the simulation control interface is synchronisation. There may be a need to *synchronise* the execution of two or more model servers. For example, when the dynamic simulation in two model servers is running faster or slower than real time, synchronisation is needed. Different synchronisation models are discussed further in Section 5.3.3. The training control interfaces are used for recording functions and for malfunction control. Both simulation and training control interfaces are implemented as extension interfaces for the OPC server object.

One HTTPS model server has a special role in the architecture. This model server is called Gallery. Gallery is located in the Internet so that it is accessible for all users of the architecture. Gallery can be used for *sharing models* between different users and for transmitting process component *parameter values* from equipment manufacturers to the simulation users. Furthermore, client and server extension tools can be distributed through Gallery. When downloading and uploading models and component information it is important to detect whether the model or component already exists in the server. This is another rationale for the usage of GUIDs.

4.8 Process view

The process view discusses the *distribution, concurrency and performance* issues of the architecture. In addition, the mapping from the logical concepts to the physical processes is explained.

The process view of the architecture as an UML component diagram is shown in Figure 4.12. The model server and model client are implemented as processes in the architecture. In addition, launching and user management services are implemented using a separate process, a model server manager.

The architecture is *distributed* so that the model clients and model servers can be located in different hosts. When they are in the same local network, the GTP and RPC mechanism of COM are used. When they lie distributed over the Internet, HTTPS protocol is used. The transport layer chooses the right protocol according to the configuration done during the installation of the system. A model server manager process must be running in every host where the model servers are executed. The administrator of the system configures the different model servers and user roles to the model server manager. The distribution model of the architecture is illustrated in Figure 4.12. It should be noticed that the configuration in the figure is only one possible deployment of the architecture.

The architecture is designed so that *several model clients* can connect to one server simultaneously. This is needed in the data connections for example if several trainees are observing the values of one and the same simulation. If horizontal data connections are configured the model server may need to transfer values also to other model servers. Concurrency is needed in the configurational connections e.g. in the Gallery server, where several process designers access the database at the same time. However, the configurational connections are not event-based, so the model clients have to refresh the view from time to time in order to receive up to date information.

When working with simulators, *concurrent* configuration usage is not practical. First of all, simulators do not support concurrent simulation of separate parts of the model at the same time (Section 2.3). Secondly, concurrent model development in the same simulator is not the way process designers like to work. Large models are broken down into pieces and the pieces are modelled in separate simulators by different model configurators. After the pieces have been tested the larger model is assembled from the pieces. It is probably not worth the effort to develop a process simulator that could handle this concurrent development inside one simulator.

The *performance* of the architecture can be divided into the performance of the configurational connections and the performance of the data connections. The model configuration may be large and when expressed in the GML format it may take a lot of space. For example, a typical medium-size power plant model (Tuuri & Juslin 1995) takes 15 MB when expressed using GML (does not include graphics). Copying model configurations of this size from one server to another takes time depending on the transmission speed of the network connection.

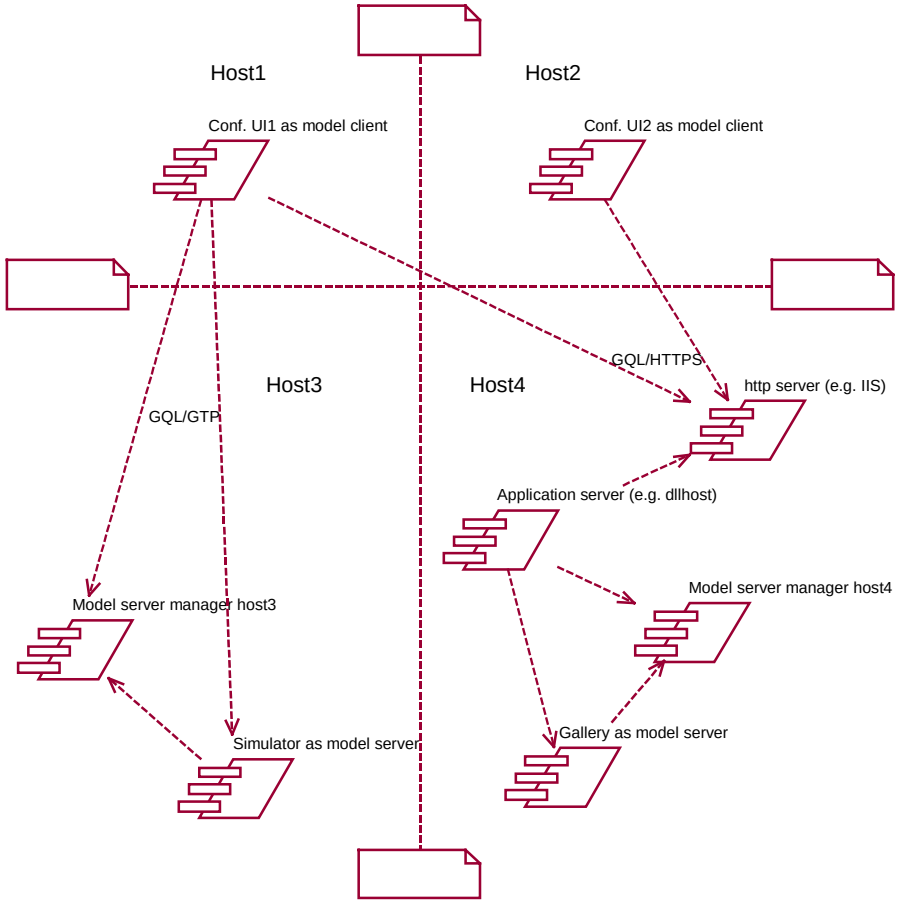

Figure 4.12. Process view of the architecture using an example deployment to illustrate the distribution.

The model configuration has to be parsed in the model server. Different XML parsers have very different performances. For example, if the power plant model is read to the DOM parser of Microsoft (version 2.0) it takes around 2.6 seconds (Pentium III 800 MHz). The same value with the Xerces parser of Apache XML project (version 1.51) is more than 40 seconds. Due to the delays in the handling of large XML documents some of the GQL functions are specified as asynchronous. This means that in normal GQL functions, the response is returned immediately as an answer to the request. In an asynchronous GQL function, the client gets an identifier as a response to the request and later on it may query the status and the result of the original request.

The functions handling large XML documents such as the functions handling model configuration and type and symbol libraries are made asynchronous in the specification. Furthermore, packing is added at the transport layer level if the transmitted XML package is large. Otherwise, the GQL interface is designed so that the XML requests and responses are of reasonable size (< 1MB).

The performance of a data connection is crucial for many tasks. For example in automation testing and in training simulator cases a fast data connection between the control system and the simulator is vital. This is why efficient implementation of OPC DA interfaces is important in the architecture. Different implementation approaches were tested and they are further discussed in Section 5.3.3.

5. Verification

5.1 Introduction

This chapter presents the use scenarios to verify the hypotheses (see Section 1.2) of the thesis. The use scenarios also form the *deployment view* to the proposed architecture. The deployment viewpoint describes the physical architecture of the system. Different deployments use different features of the proposed architecture. The use scenarios are chosen so that they measure the desired customisation, reuse, co-use, extensibility and flexible connectivity features of the proposed architecture. Section 5.2 focuses on the use scenarios related to model configuration and Section 5.3 focuses on the use scenarios related to model usage.

Use scenarios are described using the UML *collaboration diagrams*. The class instances in the diagrams are instances of classes represented in the logical view. It should be noted that the objects are not implementation objects and that the sequence of actions also contains actions initiated by the user. Each physical deployment is also described using the UML *deployment diagram* format.

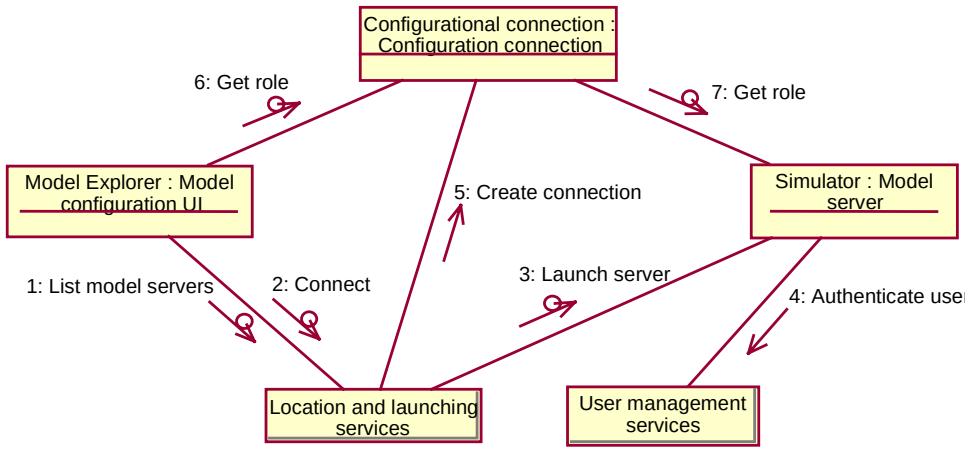

Figure 5.1. Launching scenario.

Many of the use scenarios include the creation of a configuration connection as a common scenario. This *launching scenario* is shown in Figure 5.1. First, a model client asks the location and launching services to list the model servers located at a certain host. After the user has selected a model server, the model client requests a connection to that model server. In this connection request, also user name and password are given. The location and launching services launch the desired server (if not already running) and the server authenticates the user using user management services. If the user is accepted, a new configuration connection is established. Now the model client can send GQL requests using this connection. The first request is very often “getRole” giving information about the role of the user at the model server.

Data connections are created using the COM mechanism and services of the COM library. Each model server instance has a program id that can be used to establish an OPC connection to the server. Connection to a remote server also needs the name of the remote host. When the remote host is located in the Internet, the OPCXMLProxy/Stub mechanism handles the creation of the data connection. It should be noticed that data and configurational connections are independent of each other.

5.2 Verification of the configurational features

5.2.1 Model customisation using manufacturer data and dimensioning tools

In the proposed architecture, a centralised database model server, known as the *Gallery*, can be used for sharing model configurations and component parameter values among the users. In the following example, a pump manufacturer uploads information about *centrifugal pumps* into the Gallery and the simulator users can use the data for setting the parameter values for a pump in the simulator program. The selection of the pump is based on the hydraulic properties.

Figure 5.2 illustrates the use scenario for model customisation using manufacturer data. The main phases in the diagram are:

1. Kernel developer defines a component type description to Gallery (messages 1, 2 in the diagram)

2. Kernel developer programs and uploads extension tools to Gallery (3, 4)

3. Provider downloads converter from Gallery (5, 6)

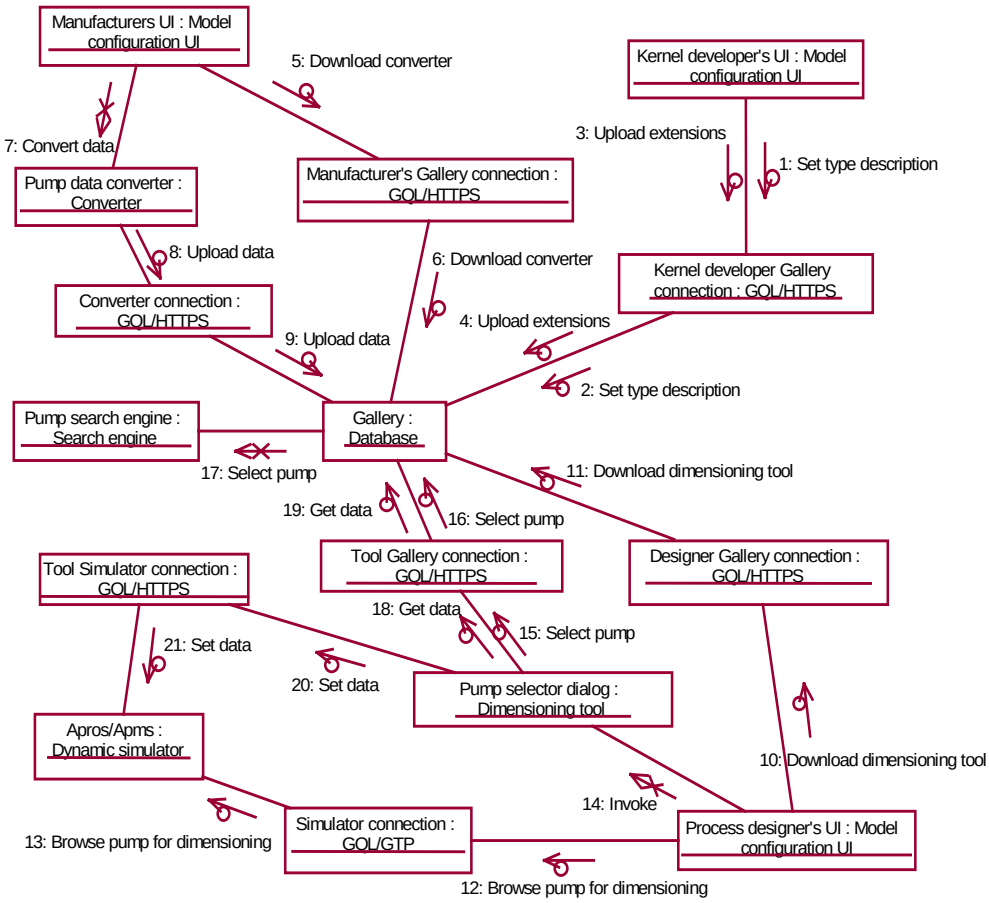

Figure 5.2. Use scenario of pump dimensioning.

4. Provider uploads equipment data to Gallery (7, 8, 9)

5. Model configurator downloads dimensioning tool from Gallery (10, 11)

6. Model configurator selects the equipment from simulator for dimensioning (12, 13)

7. Model configurator dimensions the equipment (14, 15, 16, 17, 18, 19, 20, 21)

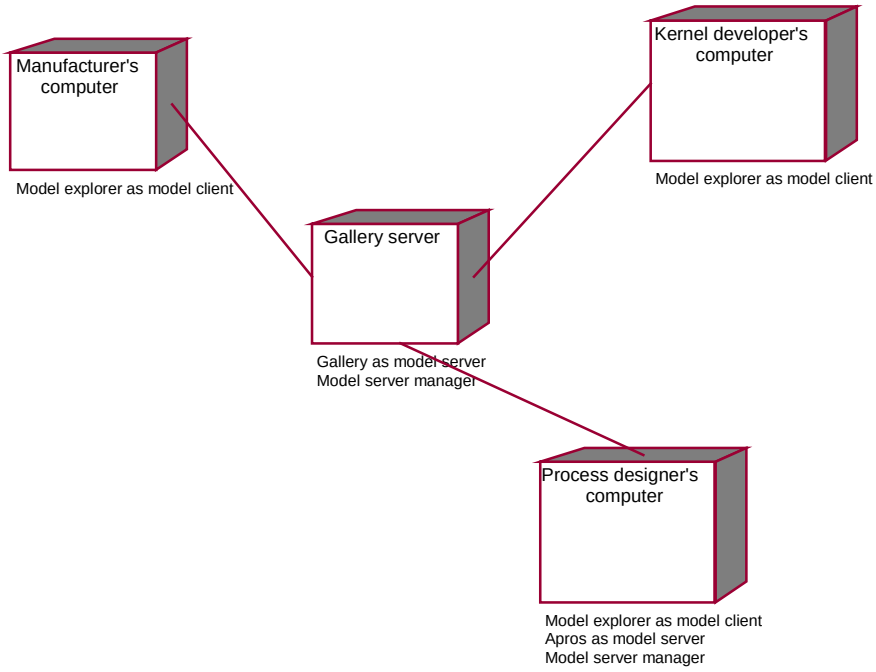

Figure 5.3. Deployment view of pump dimensioning scenario.

First, the kernel developer writes the component type description for the centrifugal pump to the type library in the Gallery department at the Gallery model server. The kernel developer also programs the software components for pump data conversion, for pump selection based on hydraulic properties and for pump dimensioning to the simulator (Apros/Apms). The dialogs are implemented as *ActiveX components* (Chappell 1996) and the pump selector is implemented as in process *COM automation component* (Microsoft 1995). In addition, the kernel developer uploads the client and server extension components and their descriptions to the Gallery model server. Model Explorer is used as a configuration UI for all users in the use scenario.

The pump data converter is described to the Gallery model server as a client extension of the type *converter* and it is associated to the centrifugal pump component type description. The purpose of the software component is to convert the hydraulic pump data files of the manufacturer (Sulzer Pumps Finland in this case) into GML components as described in the component type description of centrifugal pumps.

The pump search engine is described to the Gallery model server as a server extension of the type *search engine*. The purpose of the software component is to go through the pump instances in the database and select suitable pumps according to the hydraulic selection criteria i.e. capacity, head and $NPSH_a$.

The pump dimensioning tool is described to the Apros/Apms model server as a client extension of the type *dimensioning tool* and it is associated to the component type description of an Apros/Apms pump. The purpose of the pump-dimensioning tool is to offer a user interface for the pump search engine server extension and to transform parameter values from the Gallery centrifugal pump instance to the Apros/Apms pump instance.

The manufacturer uses the pump data converter for setting and updating pump data in the Gallery database. In the example case, 10 pump instances are set to the database. The model configurator downloads the dimensioning tool from the Gallery to the Apros/Apms model server and sets the parameter values for the pump in the simulator using the pump search engine in the Gallery and the dimensioning tool dialog in Apros/Apms. The characteristic curve, e.g., for the selected pump is automatically transformed and set to the simulator pump. Transforming is needed because the curve is expressed in a different way in the Gallery type description than in the Apros/Apms type description. More information about the selection algorithm and about the type description for centrifugal pumps can be found in (Luukkanen 2001).

Dialogs use their own connections to the model servers. Separate connections make them more independent for use in other model clients than Model Explorer. The deployment view for the use scenario is shown in Figure 5.3. Four different computer hosts are involved. The processes used in the different hosts are written under the host symbols.

A similar dimensioning case to that of the centrifugal pumps is also implemented for shell and tube *heat exchangers*. The problem with the use of heat exchangers in the Apros/Apms environment has been the customisation of the unit operation model. There are many parameter values (number, material, length, diameter, loss coefficient of tubes, roughness in both tubes and shell, flow area and length at shell side etc.) that have to be given in order to dynamically simulate a heat exchanger (pressure/flow solution).

The use scenario for the heat exchanger deployment is very similar to the use scenario in the pump case (Figure 5.2). The kernel developer implements and describes the dimensioning tool for the Apros/Apms heat exchanger component type. In addition, he implements and describes the heat exchanger selector as a search engine to the Gallery database. The shell and tube heat exchanger component type description is made according to the TEMA (TEMA 1988) specification. So the instances that are used in the selection are not instances of a specific manufacturer, but standardised instances according to the specification. In other respects, the use scenario is very similar to the pump customisation use scenario described in Figure 5.2. More detailed information about the selection and dimensioning algorithms can be found in (Soini 2001).

The two example cases show how the architecture can be used to enhance the efficiency of model customisation by providing

1. centralised repository, Gallery, for process component data and

2. generic distributed extensibility features for building selectors and dimensioning tools for different process components.

5.2.2 Model reuse using centralised repository and parametricized construction

The proposed architecture can be used for reusing the model configurations. Model reuse can be done in three different ways. In *template-based* reuse, a model template is copied to the Gallery database and access rights are set for other users. When someone else takes the template in use, it may have to be modified for the particular usage.

In *parametricized construction*, the logic of building the model configuration is programmed into a model builder client extension component. The model builder software component is stored in the Gallery and access rights are set for other users. When the model builder is used, the user can set the functional parameters through the user interface of the model builder and the software component builds the model configuration according to these parameter values.

The third way for model reuse is to *combine* the template-based and parametricized construction approaches. Some of the basic templates can be stored to the Gallery database and the model builder uses these templates as input and modifies them according to the parameters given by the user.

A machine screen is used here as an example to demonstrate the three reuse mechanisms. The purpose of the machine screen at a paper mill is to remove impurities from the pulp suspension before paper machine head box. The aim of the example case is to use the reuse mechanisms to store the machine screen assortment of a certain manufacturer (Metso Paper) into the Gallery database and use this information to ease the work of the model configurator. The difference from the pump case described in the previous section is that a machine screen is often not a primitive component in the simulation environment. It can be modelled as a hierarchical model in e.g. Apros/Apms. Even though the example model is very small it should be noticed that the same reuse mechanisms could be used for much larger hierarchical model entities.

The feed in a machine screen is typically axial or tangential. The machine screen can be modelled as a *hierarchical* component in the Apros/Apms type library. However, the two different feed types require different sub-structures (Figure 5.4). In both cases the flow is split to accept and reject. In the case of the tangential feed there is a pressure loss in the accept line. Instead the axial feed produces an extra head into the accept line.

A *clean template-based* approach would mean that the manufacturer maintains the hierarchical Apros/Apms model palettes at the Apros/Apms department of the Gallery database. The different feed types and different equipment sizes should be maintained in the Apros/Apms type library format. This could be a reasonable option if the manufacturer is also an Apros/Apms user and if all the other Gallery clients who needed the machine screen data could use it in this

format. It can be seen that a clean template-based approach is best suited for sharing models between users who use the same simulation tool.

A clean parametricized construction mechanism would mean that the manufacturer or kernel developer of the Apros/Apms model server maintains a model builder software component. Using the model builder, the model configurator could build the different machine screen types to the simulation model. If the manufacturer wants to make changes to the assortment of machine screens, the component has to be changed, recompiled and shared through the Gallery database. This is laborious and, thus, a clean parametricized construction mechanism is not so feasible if there are dependencies to a changing data set.

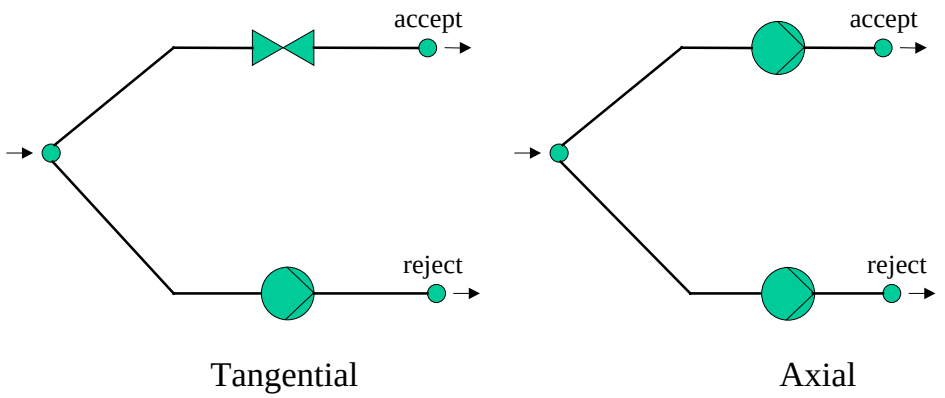

Figure 5.4. Sub-structures for a machine screen in Apros/Apms for axial and tangential feeds.

A *combination* of the two mechanisms is best suited for the machine screen use case. It means that the manufacturer maintains the information on machine screens in the Gallery type library format. This component type description is described in such a way that it contains all the needed information on machine screens for all the different simulation tools linked to the architecture. It should be noticed that the machine screen component instances in the Gallery department are not hierarchical. Now the kernel developer of Apros/Apms implements the model builder for machine screens for the simulator. This model builder can be used for selecting the desired machine screen from the Gallery and for building the appropriate hierarchical model to Apros/Apms. The use

scenario is represented in Figure 5.5. The deployment diagram is the same as for the pump dimensioning case (Figure 5.3).

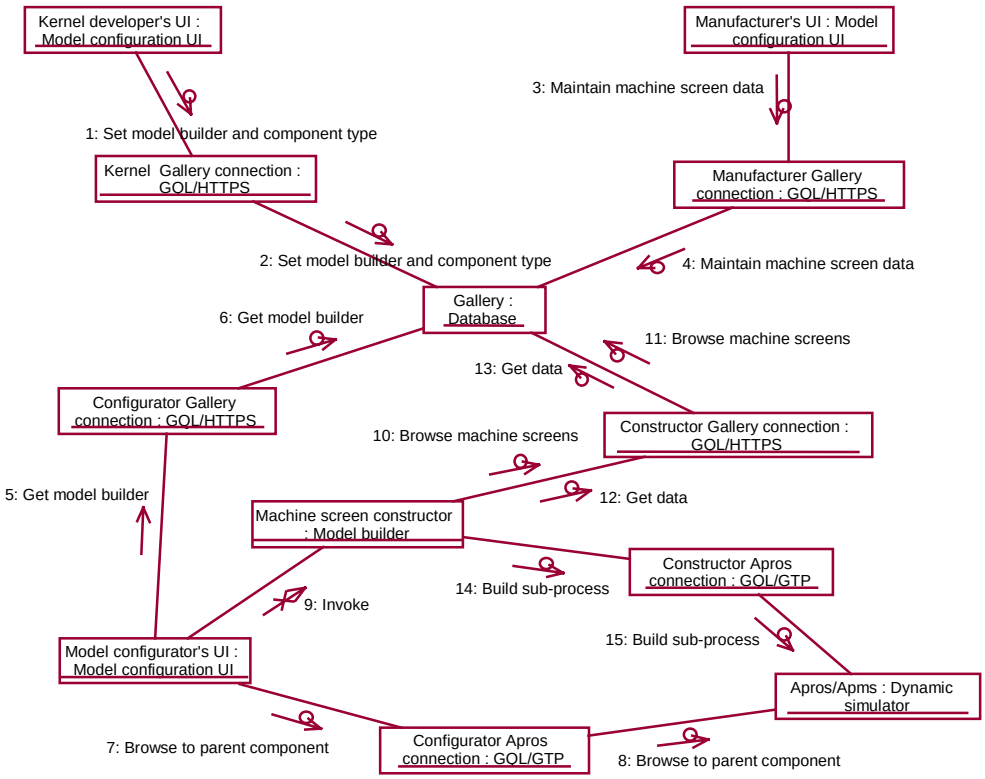

Figure 5.5. Use scenario for combination of template based and parametricized construction of machine screens in Apros/Apms.

The main phases in the use scenario in Figure 5.5 are:

1. Kernel developer defines component type for machine screens, programs model builder and uploads them to Gallery (1, 2)

2. Provider maintains machine screen data according to component type description (3, 4)

3. Model configurator downloads model builder from Gallery to Apros/Apms environment (5, 6, 7, 8)

4. Model configurator uses Gallery data and model builder to build appropriate sub-process into Apros/Apms (9, 10, 11, 12, 13, 14, 15)

The data-centric approach of the architecture can be seen from the examples represented in this and in the previous sections. The extension mechanisms i.e. software components are used for data-centric purposes. This is the main difference between the approaches of Gallery and Rome (see. Section 2.4). Rome as well as CAPE-Open take a *model-centric* approach to the reuse and thus the software component technology is used for extending the simulation algorithms rather than for supporting model configurational issues.

Version control is an important issue when sharing models through an Internet-based database. If a model is updated by several users, for example, it should be possible for them to check out the model or parts of it so that the other users could see that the model is being used by someone else. This kind of version control features, other than the history mechanism (4.5.6), have not yet been implemented to the architecture.

5.2.3 Co-use of a steady state simulator and a dynamic simulator

The proposed architecture can be used for *configurational co-use* between different process simulators. Prosim and Apros simulators are used here for an example of configurational co-use. Prosim is a simulation tool developed by Endat (Prosim 2000). It focuses on steady state simulation and optimisation of power plant processes. A small part of a *power plant* process containing the turbine and feed water tank was taken as an example configuration (Figure 5.6). In this process, steam from the superheater of the boiler is led to the turbine. Most of the steam from the turbine is led through the condenser to the feed water tank whereas some of it is tapped directly to the feed water tank. The water is pumped back to the pre-heaters of the boiler from the feed water tank.

The use scenario of the configurational co-use is shown in Figure 5.8. The deployment view for the use scenario involves only one computer where both simulators and the user interface are located. Before the model configurator can use the architecture as described in the use scenario both of the simulators have to be incorporated into the architecture. The kernel developer has to define the

type libraries for the supported component types of the simulators. The kernel developer also has to implement a *transformer* client extension component that is used in the configurational transformation. The component types for some of the process components used in the example are shown in Appendix C. The component types for both Apros and Prosim are listed. From the type descriptions one can see the difference in the parametrisation of the process components in the two simulators.

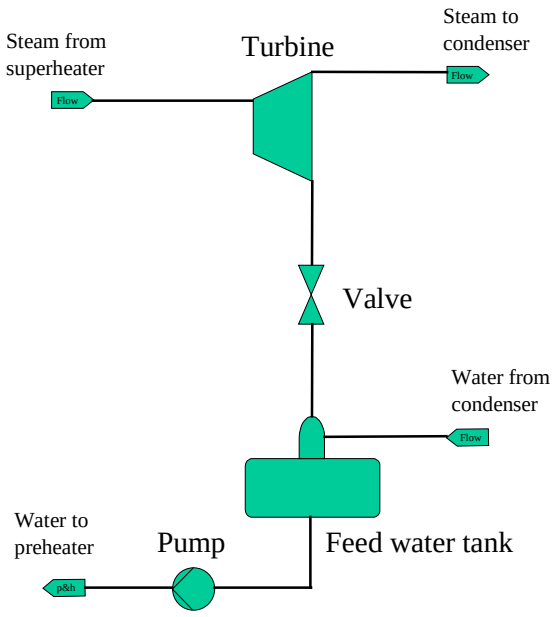

Figure 5.6. Example process.

The mapping from Prosim types to Apros types is done in the transformer component. For example, the flow in boundary in Prosim is mapped to two points and a connecting pipe in Apros. The first point and pipe are excluded from the flow net solution and the other point is a part of the simulation (Figure 5.7).

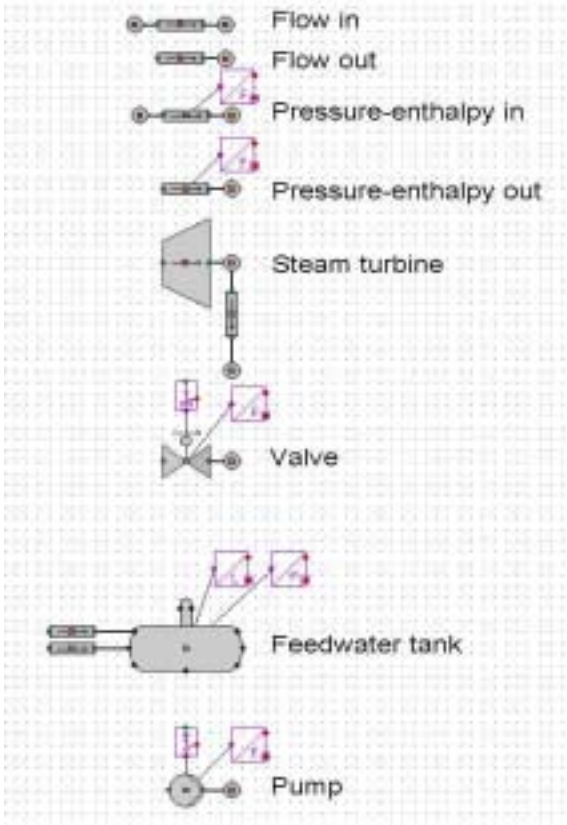

Figure 5.7. Mapping from Prosim components to Apros components in the use scenario. The Prosim component name is shown on the right and the corresponding Apros components are shown on the left.

It can be seen that the Apros representation for all of the components is more complex than the Prosim representation, i.e. *one to many mapping*. This makes the transformation somewhat easier from Prosim to Apros. The transformation to the other direction has not been implemented. In the use case sense it is not as important since the workflow usually goes from steady state to dynamic modelling.

In the case of one to many correspondence, Apros components can also be packed as higher level components that correspond to the Prosim components. The mapping can be seen in Figure 5.7. Measurement and actuator components are added to the mappings in order to lift the flow measurement values and control signals to the interface of the higher component.

After the kernel developer has implemented the transformer tool the model configurator can use it. The model configurator builds the model (Figure 5.8) to Prosim and simulates the steady state for it. Then he uses the transformer for transforming the model to Apros. The generated Apros model has to be considered only as a *template*. In the dynamic model the model configurator also has to configure the controls for the model in order to successfully simulate it. Of course, wizards for configuration of simple control loops could be added also to the transformer, but the model configurator will probably wish to configure the controls himself.

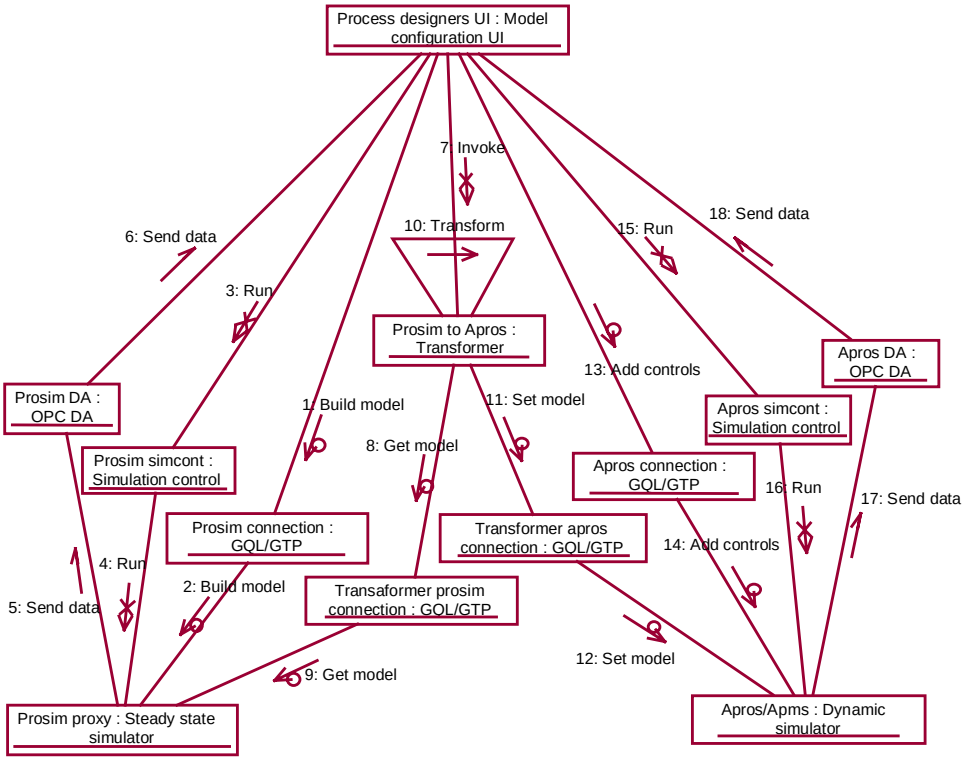

Figure 5.8. Use scenario for configurational co-use.

The problem with this approach is the *updating* of the model. Once the model has been transformed from Prosim to Apros the model starts to evolve in two places. One can always design different intelligent update operations to the

transformer. However, according to the author's experience it is difficult and time consuming to develop and maintain a transformer that is reliable for such usage. The main advantage for using a transformer is therefore that one gets a template of the model to start with so that everything does not have to be configured all over again. After that, however, the models can be let to evolve separately.

One way of implementing configurational co-use would beto use a *common model* (third type library) between the simulators as described in Chapter 4. The kernel developer implements the transformer from the common model to Prosim and the transformer from the common model to Apros. The changes are made only to the common model and then projected to the simulators. However, the common model is most likely a union of the parametrisation in both simulators and thus the Prosim model configurator must also deal with the Apros parameter values.

In addition to configurational co-use the simulators may also have *run time co-use*. This means that the steady state values calculated in the steady state simulator can be transferred to the dynamic simulator to be used as an initial state. This kind of a horizontal data connection between the tools can be configured using X-Connector and OPC DX. The items of the connection can be kept inactive and the connection can be refreshed when the user wants the initial state to be transferred.

5.2.4 Empirical models in the architecture

In the proposed architecture, a centralised repository can be used for *sharing measurement data*. The following example shows how a *disc filter* manufacturer can upload measurement data (called leaftests) of the disc filters to the Gallery. The data can be used in a dynamic process simulator in order to better adapt the disc filter model to a certain process. The represented example demonstrates the *extensibility features* of the architecture. Due to the lack of measurement data the example, however, has not yet been completely implemented.

The use scenario of the disc filter case is shown in Figure 5.9. The main phases in the use scenario are:

1. Kernel developer implements an external model and dialog for it and uploads them together with the component type description to Gallery (1, 2)

2. Provider uploads disc filter leaftest data to Gallery (3, 4)

3. Model developer uses external model and related measurement data in simulation (5–16)

4. Disc filter end user can access the measurement data and external model through the dialog indepent of the simulation usage (17–23)

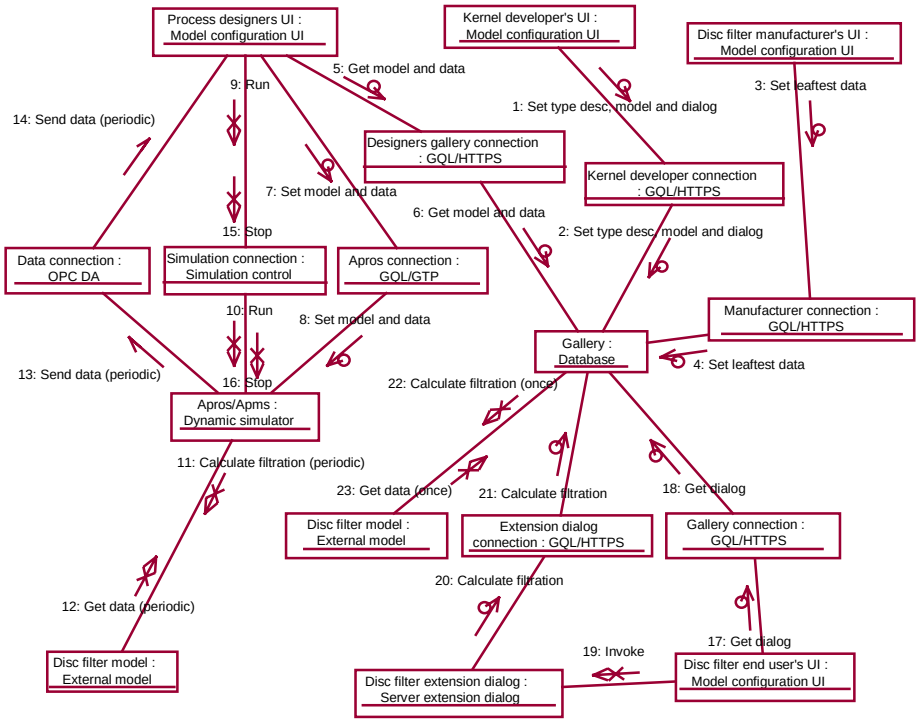

Figure 5.9. Use scenario of disc filter external model.

The filtration model is implemented as an *external model* (dynamic link library). The model takes the rotational speed, inlet consistency, freeness and angle of offset as input parameters. It calculates the drainages and consistencies for different filtrates (cloudy, clear and super clear) (Mäkinen 2001). The model is implemented by the kernel developer. The kernel developer also provides the component type description for the leaftest data of the disc filters. The disc filter manufacturer (GL&V in this case) maintains the leaftest data in the Gallery database. Simulator users can download the model and the related measurement data to the Apros/Apms environment and use the data if they need to adapt the disc filter model to a certain process. In this case, the model is used periodically by the simulation engine when it asks the filtration values from the external model during the execution of calculation time steps.

In addition, the model can be used by the disc filter end users. They can invoke the model using the server extension dialog implemented by the kernel developer. In this case, the external model is called only once using the input arguments given by the user. It should be noticed that the model server provides fast GQL API for server extensions to access data in the database. Although this API has the same functionality as the GQL interface, it is used internally which makes the execution of GQL calls faster than between model clients (or client extensions) and the model server. The deployment view of this use scenario is allmost the same as for the pump dimensioning case (see Figure 5.3). Only difference is that the disc filter end user is accessing the system from own separate host.

5.3 Verification of the run time connectivity features

5.3.1 DCS testing

The configurational features of the architecture discussed in the previous section have a larger role in the design phase of a process and automation delivery. The focus of this section is on the later phases of the delivery life cycle, i.e., *automation testing and operator training*.

The suggested architecture was applied to an automation modernisation project of an Estonian power plant. The project was carried out by Fortum Power and

Heat in 1999 and 2000. The Estonian power plant is an oil shale combusting power plant consisting of 8 blocks each with two boilers and one steam turbine. Each block has an electric output of 200 MW. The control system used in the modernisation was MetsoDNA (NelesDNA at that time) and the process was modelled using the Apros simulator. (Rinta-Valkama et al. 2000)

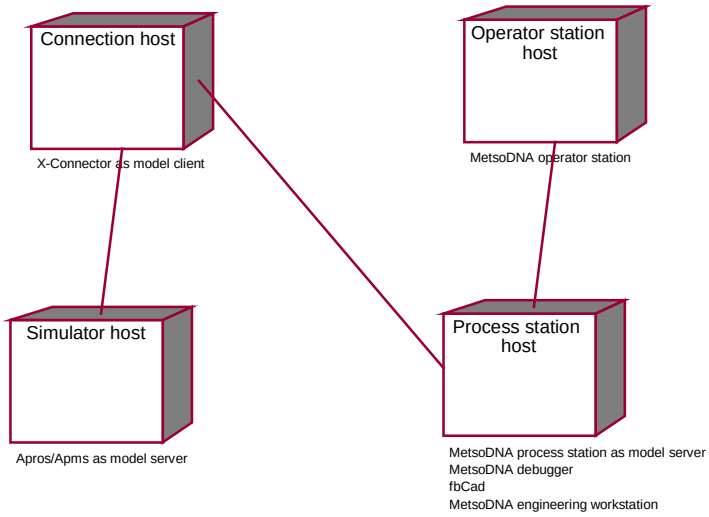

Figure 5.10. Deployment view of the architecture in the DCS testing case.

The physical deployment of the architecture used in this case is represented in Figure 5.10. The process model ran in one of the computers. The control software ran in two computers with the process stations in one and the operating station in the other one. The fourth computer was used to handle the communication between the two systems. *X-Connector* was used for the creation of a vertical data connection to the systems and for transmitting values between the connections. It should be noted that at that time DX specification did not yet exist. There were about 2000 signals conveyed simultaneously through this link in a real time simulation.

The use scenario of the architecture is represented in Figure 5.11. It should be noticed that in the diagram Grades, MetsoDNA debugger, MetsoDNA operator station and fbCad are described using the concepts of the proposed architecture even though they do not satisfy the architectural criteria of a model client. They do not use any connection types of the proposed architecture. In the same way, the MetsoDNA engineering workstation is not a model server. However, the applications are included in the diagram to give a better understanding of the testing procedure.

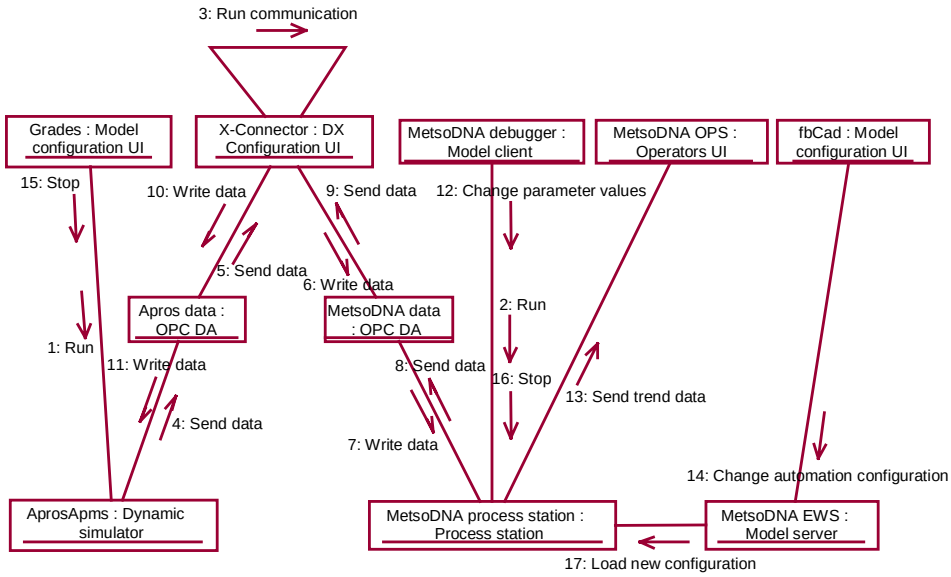

Figure 5.11. Use scenario of the architecture in the DCS testing case.

Rinta-Valkama describes the testing procedure in the following way (Rinta-Valkama et al. 2000):

"The testing procedure was rather similar to the control loop commissioning procedure at site. Each control loop was tuned based on a step response experiment done with the simulator. It was then checked that the loop was stable, followed the set point and had acceptable dynamics. If the loop did not function properly, the fault was located using trend plots of the operator station and the debugger of the automation system. Finally, the problem was fixed and the function block diagram was updated. Most loops were tested in this fashion. These included steam temperature,

steam pressure, drum level, flue gas oxygen, electric power, feedwater tank pressure and mill temperature control loops. The master controls were tested in a more diverse manner. The master control of the power plant includes the control of the electric power, turbine initial pressure and pressure in the two boilers. The configuration of the master control was not yet decided, when the simulation tests were started. Different variations of the master control were tested by simulation, and compared against each other. After the operation in steady state and in load changes was checked, it was verified that the control loops worked properly in certain disturbance situations. These included a forced draught fan trip, a fuel mill trip and quick by-passing of the pre-heaters."

The experiences from the use case were mainly positive. Quite many changes were made to the automation application already at the factory during the simulation-aided testing. Without the testing many of the changes would have had to be made at the site. The communication was *flexible* to configure and easy to modify. The *capacity* was also adequate for most of the tests. However, there were some cases where the *speed* of the data transmission was too slow and the controllers could not be tuned realistically. Also, the scalability of the data connection was not adequate for running the entire plant at the same time. The testing of the entire plant at one go would have included around 7000 signals between the DCS and the simulator. The OPC connection at that time was able to transmit around 2000 signals between the systems. The lack of *synchronisation* between the simulator and the DCS was also a problem with the motor operated valves, because some control pulses were doubled and some lost. It was possible to get around this problem in the simulator, but in these cases the functionality of the valve position control could not be tested.

It will be shown in Section 5.3.3 that with a more efficient OPC implementation the speed and scalability problems of the data connections have been fixed. Also the synchronisation issue is discussed in more detail in Section 5.3.3.

5.3.2 Training simulator support

The suggested architecture was applied in a *training simulator* delivered by Metso Automation to the Suomenoja power plant in Espoo. The automation modernisation was done in the plant in 1993 and the training simulator, including the coal block So1, was delivered in December 2000. The Suomenoja plant delivers electricity and district heating for the cities of Espoo, Kauniainen

and Kirkkonummi. The maximum electrical power of the plant is 120 MW and the capacity for district heating is 350 MW. (Yli-Petäys et al. 2001)

The process was modelled using the Apros simulator and the MetsoDNA distributed control system was connected to the simulator by OPC data access interfaces and the X-Connector tool. The use scenario of the architecture is represented in Figure 5.12. An *instructor's interface* was used to operate the system. First, a snapshot corresponding to a certain starting point of a training program was loaded to both of the systems and then the execution was started. In addition, the communication configuration was loaded and started. Around 1300 signals were transmitted through the OPC link in a real time simulation. The instructor could trigger different malfunctions through the instructor's interface and he could follow the operations of the operator using trend plots. The operator operated the process through a normal operator UI, similar to those in the control room. The deployment view of the system is shown in Figure 5.13.

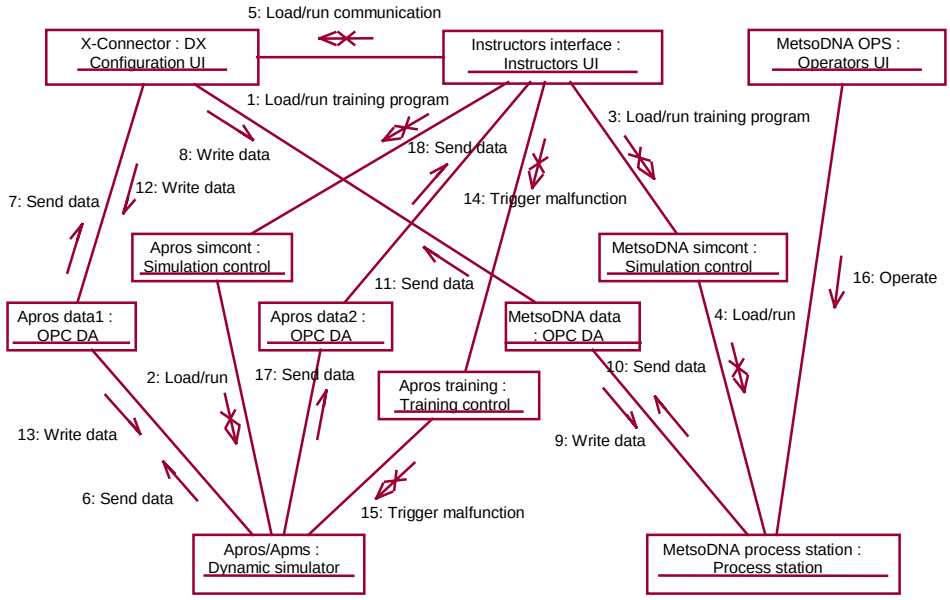

Figure 5.12. Use scenario of the architecture in the training simulator case.

The use of *extension interfaces* is necessary in both control system testing and training simulator cases. OPC DA does not specify the functionality for starting and stopping the simulation or for the training control features such as

malfunction control, recording and backtracking. Also the ability to save and load snapshots is essential. This functionality is defined in the data connections of the proposed architecture. *Simulation control and training control* interfaces are defined as extensions to the OPCServer COM object.

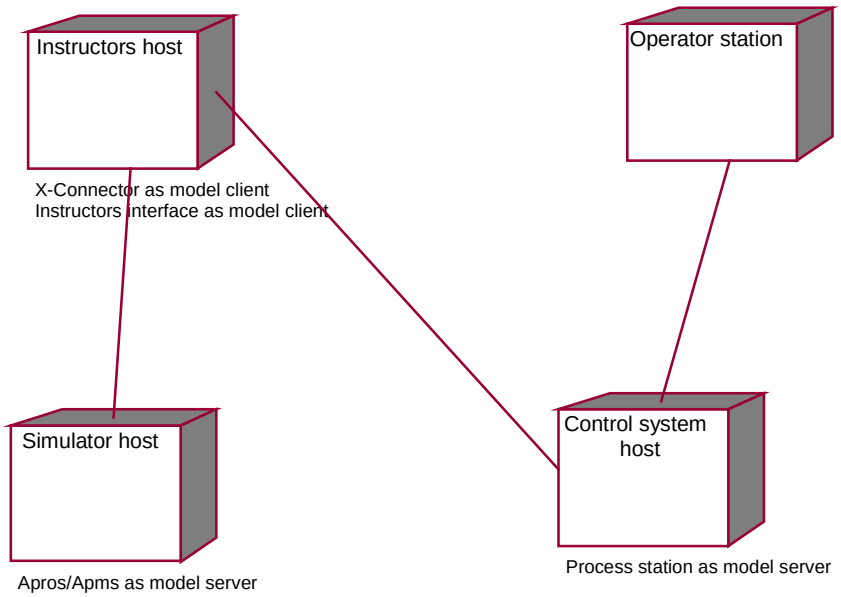

Figure 5.13. Deployment view of the architecture in the training simulator case.

The experiences from the project were positive both among the training simulator developers and the operators. In an inquiry made among the process personnel (22), the average grade for all questions (8), on a scale from 1 (unsatisfied) to 5 (satisfied) was 3.9 (Yli-Petäys et al. 2001). The compound system was easy to operate through the instructor's interface. The *speed and scalability* of the OPC connection was also adequate for a training simulator of this scale.

The architecture was also applied in a training simulator delivered to the Sunila pulp mill. A new bleach plant started production in June, 1998. During the autumn of 1997, a dynamic simulation model in the Apros/Apms environment was developed based on the mill's P&ID's, functional descriptions of the equipment, and input from the experts of the main equipment deliverer,

Ahlstrom Machinery Corporation. The model was connected to the TotalPlant Alcont control system of Honeywell-Measurex and was used for operator training during spring, 1998. The main controls of the automation application were simulated in the simulator and the rest of the automation application was executed in the DCS. (Lappalainen et al. 1999; Tervola et al. 1999)

At that time, the OPC implementation of the simulator and DCS was not as sophisticated as it is today. The Modbus protocol (Swales 1999) was used for the communication from the OPC server to the DCS process station. Despite the 'crude' implementation 700 signals were transmitted between the systems in a real time simulation. The simulation control and training control features were not ready in the compound system at that time. For this reason, the start-up of the training system and the shift from one training situation to another were quite time consuming and difficult.

In all of the three industrial cases, one of the conclusions was that the dynamic simulation model could be used also in other phases of the process or automation delivery. In the case of the Estonian power plant the model was detailed enough also to be used in the operator training. In the Sunila case, many problems in site testing of the automation application were solved as the model was connected to the doubled DCS system simultaneously with the commissioning at the mill. With more careful planning and scheduling the model could have been efficiently used for testing of the automation. In the Suomenoja case the simulator has been used after the training period during plant operation also for trouble shooting in different operational practices in unusual situations, e.g. in a malfunction situation of a process component.

As already discussed in Chapter 3, there are several possible choices to implement a training simulator. *Direct connection* of DCS and the simulator, as the ones described above, is one of these choices. However, a direct connection is not always possible due to the schedule of the automation delivery project. Despite the extra work, sometimes the automation application has to be simulated in the simulator and the operator displays have to be *emulated*. The experience from a medium size power plant training simulator shows that in the emulated case the work is divided between the different work package so that the process modelling takes about 20%, automation modelling takes around 30% and display emulation about 50% of the work. Thus, 80% of the work has to be

re-done when the actual control system is configured. It should be noticed that if the simulator and DCS supported the same kind of configurational connection, the automation configuration part (30%) could be at least partly reused.

5.3.3 Speed and scalability of the data change

The idea of using OPC between the simulator and DCS (Karhela et al. 1999) imposed a couple of problems with scalability and speed at the beginning. As described in the Estonian power plant case the limit in the scalability seemed to be around 2000 signals in a communication cycle of 200 ms. Recently the OPC implementation has been studied more carefully and two major improvements have been made that are also incorporated into the suggested architecture.

Firstly, the industrial cases were carried out with a configuration where the OPC servers of both the simulator and DCS were located in different processes than the actual simulator engine and the process station. The OPC interfaces have now been *incorporated* into the same process with the simulator engine as described in the component view of the proposed architecture. Tests for the data transmission were run using efficient interfacing between the OPCKit component and the Front component (see Figure 4.10). The simulator input and output results are shown in Tables 5.1 and 5.2. In order to illustrate how the enhanced performance of processors will affect the throughput of the OPC server, the performance values achieved with a PIII 500MHz (Windows NT 4.0 operating system) are compared with the values achieved with an AMD Athlon 1.2GHz (Windows 2000 operating system), both with 256 MB-RAM. In the output tests all double items were continuously changing and the requested update period was 200ms. ‘New’ refers to the new OPC implementation and ‘old’ refers to the old implementation of the various processes. (Peltoniemi et al. 2001)

Table 5.1. Performance of synchronous write (Peltoniemi et al. 2001).

	Processor	Items	Time (ms)
new	PIII 500MHz	1	1,3
new	PIII 500MHz	1000	2,8
new	PIII 500MHz	10000	29
new	PIII 500MHz	50000	160
new	AMD 1.2GHz	1	1,5
new	AMD 1.2GHz	1000	2,3
new	AMD 1.2GHz	10000	27
new	AMD 1.2GHz	50000	120

*Table 5.2. Performance of event-based data exchange 2. * denotes that the processor time consumed for the simulation is included in the CPU server column. This is a case where the OPC server is integrated into the simulation engine. The consumed CPU times are divided into privileged, user and total time. proc. denotes processor. (Peltoniemi et al. 2001)*

	Processor	Items	CPU	%	servr	CPU	%	client	CPU	%	proc.	CPU %
			priv.	user	tot.	priv.	user	tot.	priv.	user	tot.	simulator
old	PIII 500MHz	5000	1	41	42	1	4	5	3	49	52	5
old	PIII 500MHz	8000	1	70	71	1	5	6	6	80	86	5
new	PIII 500MHz	5000	1	9	10	1	4	6	2	13	15	*
new	PIII 500MHz	10000	1	19	20	1	5	6	3	24	27	*
new	PIII 500MHz	30000	2	46	48	10	13	23	26	59	85	*
new	AMD 1.2GHz	5000	0,1	5	5	0	0,1	0,1	1	5	6	*
new	AMD 1.2GHz	10000	1	9	10	1	4	5	7	13	20	*
new	AMD 1.2GHz	30000	1	33	34	5	10	15	13	43	56	*
new	AMD 1.2GHz	50000	2	56	58	10	14	24	26	70	96	*

The communication between two model servers through X-Connector was also tested. Table 5.3 shows these results. From the results it can be seen that with the more sophisticated OPC implementation the number of variables could be increased with a factor of ~2–4 (lines 2 and 5 in Table 5.2 and lines 1, 3 and 4 in Table 5.3).

Table 5.3. Performance of two model servers connected with a cross-connector application. The requested frequency was 200 ms with constantly changing item values. (Peltoniemi et al. 2001)

	Processor	Items	CPU % X-Con	CPU % servers	CPU priv.	% user	proc. total
old	PIII 500MHz	2500	6	45	7	50	57
new	PIII 500MHz	2500	6	12	8	17	25
new	PIII 500MHz	5000	19	17	10	30	40
new	PIII 500MHz	9000	38	27	21	59	80
new	AMD 1.2GHz	9000	22	24	9	41	50
new	AMD 1.2GHz	11000	39	37	17	64	81

The second improvement in the speed and scalability of the OPC DA connection between the model servers is to get rid of X-Connector in the communication. The communication configuration can be done flexibly from a common user interface like X-Connector, but the communication itself should go *directly* from one system to another. This is achieved in the proposed architecture by implementing OPC DX features to the OPCKit component (See Figure 4.10). Even though there are no hard numbers yet available from the second improvement it is evident that this will further improve the speed scalability of the OPC DA connections.

Synchronisation is needed in the data connections in order to run the compound system faster or more slowly than real time. As described in the Estonian power plant case synchronisation is sometimes needed also in real time simulation. Different synchronisation models in the architecture have been designed and tested but none of them have been proved yet to be efficient enough. The current synchronisation model is based on the OPC data change event. In this model, the client application, e.g. X-Connector, is used as a synchroniser between the model servers. The client application gives the permission for the model server to continue to the next time step by releasing the data change event. However, this model will not work properly in the DX scenario where servers exchange data directly between each other. Thus it would be beneficial if the DX working group of the OPC Foundation would specify such a synchronisation model.

The industrial cases show that a standardised approach to the data connection of a dynamic simulator and DCS provides a *flexible* and *vendor independent* way

of configurating data connections between the systems. With sophisticated implementation the *speed* and *scalability* of such a standardised data connection can be made adequate for simulation-aided automation testing and operator training purposes. Important features in the connection include *simulation control* and *training control* functionality. The simulation control functionality includes also synchronisation of the compound system as one very important feature.

6. Discussion

Information management is becoming more and more important in the fields of process and automation technologies. In most cases, dynamic process simulation has been applied quite separately alongside with the traditional methods for process and automation design, automation testing, operator training, and for different kinds of troubleshooting during the process and automation delivery project. It has been characteristic of the usage that simulation tools and the use of simulation tools have been loosely bound to other tools (also other simulation tools) used in the design, testing and operation, i.e., manual work has been required in order to transfer the configuration and simulation data from one tool to another. Thus, in the information management sense the situation has been quite poor.

In this work, information management in process simulation is divided into *model development*, *model configuration* and *model usage*. Furthermore, more detailed problems of customisation, reuse, co-use, extensibility, and run time connectivity are identified for these problem sub-domains. Since model development is quite well addressed by the current standardisation efforts and simulation tools, the main emphasis of this work is on model configuration and model usage.

The main results of this thesis are the design and description of a new integration architecture for the configuration and usage of process simulation models and the application of current information technologies in the implementation of such an architecture. Most parts of the proposed architecture are implemented and verified in the prototype implementations. The parts that are not yet implemented, including graphics, monitors and trends, are noted in Chapter 4. Nevertheless, they are included in the architectural description in order to make the description complete.

It is shown by using pumps, heat exchangers and machine screens as examples (5.2.1 and 5.2.2) that more efficient model *reuse* and *customisation* can be achieved with the proposed architecture if the model configurators and equipment manufacturers *share* data in a centralised repository. Even though full-scale industrial cases were not carried out, the benefits can already be shown by the examples. Potential problems concerning the reliability of the results may

arise from the scalability of the XML-based solution. The simulation models stored to the centralised repository may be large and the equipment manufacturers may have a lot of component data in the database.

Using a steady state and a dynamic simulation tool as an example (5.2.3) it is shown that the same architecture can be used for *configurational co-use* between different simulators. The benefits of such a co-use depend heavily on the simulator tool vendors and their will to support the architecture. The support means that the interfaces of a model server, i.e. configurational and data connections are implemented in the simulator. Simulator vendors should also maintain transformer software components from their departments to other departments. The benefits of the co-use also depend on the logic implemented to the transformer software components, in other words how well they handle round-trip engineering and updating problems etc. For simulator tool vendors the tools of the architecture provide user interface functionality, plug and play configurational and data connectivity to other tools and possibility for distributed usage. If these features were offered in a free of charge, common, middleware package, it could be appealing and beneficial at least for small simulation tool vendors such as universities, research centers or simulation development teams inside larger companies in process industry.

The analysed industrial cases (5.3.1 and 5.3.2) show that a *flexible and fast enough run-time connectivity* between a dynamic process simulator and a virtual DCS is achieved using OPC as the standard data exchange interface. The chosen techniques are shown to be sufficient for simulation-aided automation testing and for operator training (5.3.3).

The emphasis of the thesis is on *large-scale dynamic* simulation of *continuous* processes of the *power, pulp and paper industries*. However, the proposed architecture is quite generic and could also be used in other application fields inside the process industry, in steady state process simulation and optimisation, and even in configurational descriptions of batch processes. The purpose of the proposed architecture is to improve information management in process simulation. The information management aspect becomes more important as the sizes of the processes grow. Thus, the benefits of applying the architecture are more significant in a large-scale simulation.

The *original features* of the proposed architecture are its openness, general distribution concept and distributed extensibility features. Both configurational and data connections are based on open 'de-facto' standards (XML, SVG, SOAP and OPC). The fact that the entire data model of the architecture, including graphics, model topology and parameter values, is represented in the same open format opens up the possibilities for better reuse, customisation and configurational co-use. The general distribution features offer possibilities to use simulation and simulation configuration resources regardless of their location on the Internet. The security model of the architecture was also designed to meet the requirements originating from the general distribution concept. Furthermore, the architecture can be extended in a distributed manner. New simulation tools can be linked to the architecture and configurational extension software components can be developed in one place and distributed through a centralised server on the Internet.

The logical, data and security viewpoints to the proposed architecture are the most generic parts of the architectural description. The core of the logical viewpoint is purely conceptual, which reflects the architectural concepts related to the requirements of model configuration and model usage. The data viewpoint does not have to be mapped to XML (except the graphich definitions that are described using SVG) and can be considered as a conceptual model for representing model topology and parameter values. Thus, these parts are very general and not bound to any implementation techniques.

Support for some parts of the data model (graphics, monitors and trends) has not yet been implemented to the model client and model server tools. These parts are important for the prototype implementation of the architecture to achieve its full potential.

The distributed configuration architecture could be used in different phases of the lifecycle of the mill by the different companies involved. The *version control* features of the architecture have to be improved for this kind of heavy distributed design to be possible. Version control features, e.g. check in, check out, get latest version, should be included for different elements in the data model.

Further studying should be also done in order to include *3D graphics* to the data model. 3D models of the mill are often made as a part of the design process. It would be beneficial to map the relations between this information (e.g. STEP AP227) and the configurational information needed in the simulation models.

The interface and data model specification could also be extended to include concepts and functionality for *project management* and *maintenance*, so that the architecture forms a complete technical service frame for the network of companies involved in a mill lifecycle.

The proposed architecture is an architecture for model configuration and usage. The model development part was not considered in detail because the current specifications and current simulation tools already offer bases for model development. A *model-centric* specification combined with a more *data-centric* approach suggested in this thesis would lead to a comprehensive architectural description for process simulation. Extension interfaces for unit operation, physical property packages and reaction kinetics packages would have to be specified as suggested for example in the CAPE-Open specification. Of course, the simulation engine itself would also have to support these interfaces.

Simulation and *training control* features in the data connections are necessary for model usage in simulation-aided automation testing and operator training. Features such as running and stopping the simulation, synchronising the execution of the model servers, recording and back tracking, and malfunction triggering are not included in the current OPC specification. It would be beneficial if the DX specification for example could take a stand to the *synchronisation* issue. The synchronisation features could be added as an option to the specification. Other simulation and training control features could be added to the command part of the DA specification.

References

Ajo, R., Hakonen, S., Harju, H., Järvi, J., Kaskes, K., Lenardic, E., Niukkanen, E., Nurminen, T., Ritala, P., Tolppanen, M. & Tommila, T. 2001. Laatu automaatiossa, Parhaat käytännöt (In Finnish). Finnish Automation Society. ISBN 952-5183-12-2.

Alemanni, M., Aerospazio, A., Fuchs, I. & Gillies, S. 1999. Introducing Step. http://strategis.ic.gc.ca/stepguide, [referenced 26.10.2001]. ISBN 0-662-64382-8.

Apros, Apros web documentation. 1999. http://www.vtt.fi/aut/tau/ala/apros.htm, [referenced 8.10.2001].

ASCEND. 2001. ASCEND Documentation, http://www-2.cs.cmu.edu/~ascend/ [referenced 6.11.2001].

AspenTech. 2002. Web documentation of Aspen Plus, http://www.aspentech.com, [referenced 30.4.2002].

Balas. 1998. Balas documentation, http://www.vtt.fi/ene/balas/, [referenced 8.10.2001].

Banares-Alcantara, R. 2000. Concurrent Process Engineering – State of the Art and Outstanding Research Issues. http://capenet.chemeng.ucl.ac.uk, [referenced 26.9.2001].

Bogle, D. 2000. State of the Art of Research in Flexibility, Operability & Dynamics. http://capenet.chemeng.ucl.ac.uk, [referenced 26.9.2001].

Burkett, W. Product Data Markup Language. 1999. A White Paper. http://www.pdml.org/, [referenced 26.10.2001].

CadsimPlus. 2001. CADSIM Plus home page, http://www.aurelsystems.com, [referenced 2.12.2001].

CAPE-Open. 2000. CAPE-Open specifications: Conceptual Design Document (CDD2) for CAPE-OPEN project. http://www.global-cape-open.org/CAPE-OPEN_standard.html, [referenced 26.9.2001].

Cellier, F. 1991. Continuous System Modelling. New York: Springer-Verlag.

Chappell, D. 1996. Understanding ActiveX and OLE. Redmond, WA: Microsoft Press. ISBN 1-572-31216-5.

Co-LaN,. 2002. CAPE-Open laboratories network, http://www.colan.org/, [referenced 2.5.2002].

DCE. 1998. Distributed Computing Environment, UUID draft specification, http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt, [referenced 23.2.2002].

Ecosim. 2001. Ecosim homepage, http://www.ecosimpro.com/, [referenced 7.11.2001].

Elmqvist, H., Mattsson, S.E. & Otter, M. 1999. Modelica — A Language for Physical System Modelling. Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Hawaii, August 22–27, 1999.

ePlantData. 2001. ePlantData homepage,
http://www.ePlantData.com,[referenced 7.11.2001].

Extend. 2001. Extend User Manual, http://www.imaginethat.com, [referenced 22.11.2001].

Flowmac. 2001. Flowmac Home page,http://www.papermac.se/, [referenced 22.11.2001].

Fowler, M. 1997. UML Distilled, Applying the standard object modelling language. Addison-Wesley. ISBN 0-201-32563-2.

gPROMS. 2001. gPROMS Documentation, http://www.psenterprise.com/gPROMS, [referenced 6.11.2001].

HLA. 2000. HLA Specification. US Department of Defense. http://www.dmso.mil/public/transition/hla/index_html [referenced 26.3.2002].

Hysys. 2000. Hysys documentation. http://www.hyprotech.com/hysys/, [referenced 8.10.2001].

Ideas. 2001. Ideas Home page, http://www.ideas-simulation.com, [referenced 22.11.2001].

IEA. 1993. Process Integration Definition. http://www.tev.ntnu.no/iea/pi/definition.html, [referenced 26.9.2001].

IEEE. 2000. IEEE Std 1471-2000, Recommended Practice for Architectural Description. ISBN 0-7381-2518-0.

ISO. 1997a. ISO 10303-221 - Functional data and their schematic representation for process plant (Draft 6.6.1997). International Standard Organization (ISO).

ISO. 1997b. ISO 10628 – Flow diagrams for process plants. International Standard Organization (ISO).

ISO. 1998. ISO 10303-231 - Process engineering data - Process design and process specifications of major equipment (Draft 24.11.1998). International Standard Organization (ISO).

ISO. 1999. ISO 10303-28 - XML presentation of EXPRESS-driven data (Draft 21.12.1999). International Standard Organization (ISO).

ISO. 2001. ISO 10303-227 - Plant spatial configuration. International Standard Organization (ISO).

Juusela, A. & Juslin, K. 1976. A Nonlinear Simulation Model of a BWR Nuclear Power Plant. In: Technical Research Centre of Finland, Electrical Engineering Lab. Report 20. 83 p.

Karhela, T., Paljakka, M., Laakso, P., Mätäsniemi T., Ylijoki J. & Kurki, J. 1999. Connecting Dynamic Process Simulator with Distributed Control System Using OPC Standard. In: TAPPI. Proceedings of Process Control, Electrical, and Information Conference. Pp. 329–337.

Klemola, K. & Turunen, I. 2001. State of Mathematical Modelling and Simulation in the Finnish Process Industry, Universities and Research Centres. In: National Technology Agency, TEKES. Technology Review 107/2001. ISBN 952-457-033-5.

Koolen, J.L.A. 1998. Simple and Robust Design of Chemical Plants. Comp Chem. Eng., 22. Pp. 255–262.

Kruchten, P. 1995. Architectural Blueprints-The "4+1" View Model of Software Architecture. IEEE Software 12(6). Pp. 42–50.

Kuikka, S. 1999. A batch process management framework. Domain-specific, design pattern and software component based approach. In: Technical Research Centre of Finland. VTT Publications 398. 215 p. ISBN 951-38-5541-4.

Lappalainen, J., Tuuri, S., Karhela, T., Hankimäki, J., Tervola, P., Peltonen, S., Leinonen, T., Karppanen, E., Rinne, J. & Juslin, K. 1999. Direct Connection of Simulator and DCS Enhances Testing and Operator Training. In: TAPPI. Proceedings of of TAPPI 1999 Engineering / Process & Product Quality Conference. Pp. 495–502.

Lu, M.L., Batres, R., Li, H.S & Naka, Y. 1997. A G2 Baser MDOOM Testbed for Concurrent Process Engineering. Comp. Chem. Eng., 21. Pp. 11–16.

Luukkanen, P. 2001. Pump selectors in an integrated simulation environment (In Finnish), Master's Thesis, Lappeenranta University of Technology.

Mäkinen, M. 2001. Disc Filter External Models in an Integrated Simulation Environment, Master’s Thesis, Lappeenranta University of Technology.

Marquardt, W. 1996. Trends in computer-aided process modelling. Computers and chemical engineering, 20, 6/7. Pp. 591–609.

Matlab. 2002. Matlab documentation, http://www.mathworks.com/products/matlab/, [referenced 22.2.2002].

Mattson, S.E. & Anderson, M. 1994. OMOLA - an object oriented modelling language. In: Elsevier Science. Recent advances in computer aided control engineering. Amsterdam, Netherlands: Pp. 291–310.

Microsoft. 1995. The Component Object Model Specification, http://www.microsoft.com/com/comdocs.asp [referenced 28.2.2002].

Niemenmaa, A., Lappalainen, J., Laukkanen, I., Tuuri, S. & Juslin, K. 1998. A multi-purpose tool for dynamic simulation of paper mills. Simulation Practice and Theory 6. Pp. 297–304.

Nougues, J.M., Pinol, D., Rodriguez, J.C., Sama, S. & Svahn, P. 2001. CAPE-OPEN as a mean to achieve interoperability of Simulation Components. In: Scandinavian Simulation Society. Proceedings of SIMS 2001 Conference. Pp. 3–21.

Novem. 2000. Web page of Netherlands Agency for Energy and Environment, http://www.interduct.tudelft.nl/PItools/tools.html, [referenced 7.11.2001].

OPC. 2001. OPC home page. OPC Foundation. http://www.opcfoundation.org, [referenced 24.10.2001].

Pasanen, A. 2001. Phenomenon-Driven Process Design methodology Computer implementation and test usage. In: Technical Research Centre of Finland. VTT Publications 438. 140 p. + app. 26 p. ISBN 951-38-5854-5.

Peltoniemi, J., Karhela, T. & Paljakka, M. 2001. Performance Evaluation of OPC-based I/O of a Dynamic Process Simulator. In: SCS. The Proceedings of the 2001 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS). Pp. 231–236. ISBN 1-56555-240-7.

Pohjola, V.J. & Tanskanen, J. 1998. Phenomenon Driven Process Design Methodology: Formal representation. CD-ROM paper prints of CHISA'98, Prague, Chech Republic.

Prosim. 2000. Prosim web documentation. http://www.endat.fi/, [referenced 8.10.2001].

Puska, E.K., Norrman, S. & Nihlwing, C. 2001. Three-dimensional Core Models in Research Simulators. Proceedings of M&C conference at Salt Lake City.

Quantum3D. 1999. Visual Stealt Application for DIS/HLA Applications, http://www.quantum3d.com/pdf/marconi.pdf, [referenced 1.5.2002].

Rinta-Valkama, J., Välisuo, M., Karhela, T., Laakso, P. & Paljakka, M. 2000. Simulation Aided Process Automation Testing. In: Elsevier Sciense. Proceedings of IFAC's Conference on Computer Aided Control System Design. Pp. 277–280. ISBN 0-08-043660-9.

Schopfer, G., Wedel, L. v. & Marquardt, W. 2000. An Environment Architecture to Support Modelling and Simulation in the Process Design Lifecycle. Proceedings of AIChe Annual Meeting 2000, Los Angeles, 12–17.11.2000.

Shocklee, M., Burkett, B. & Yuhwei, Y. 1998. Product Data Marup Language specification Version 0.6. http://www.pdit.com/pdml/doc/WD-pdml-19981103.doc, [referenced 26.10.2001].

Soini, S. 2001. Heat Exchanger Dimensioning in an Integrated Simulation Environment (In Finnish), Master's Thesis, Helsinki University of Technology.

Stephanopoulos, G., Henning, G. & Leone, H. 1990. MODEL.LA. A modelling language for process engineering. I. The formal framework. In: Elsevier Science. Computer and chemical engineering, 14, 8. Pp. 813–846.

Swales, A. 1999. Open Modbus/TCP Specification 1.0, http://www.modicon.com/openmbus/, [referenced 1.4.2002].

TEMA. 1988. Standards of the Tubular Exchanger Manufacturers Association, 7. Edition. New York: TEMA Inc. 224 p.

Tervola, P., Lappalainen, J., Rinne, J., Leinonen, T., Peltonen, S., Karhela, T. & Juslin, K. 1999. Bleach Plant Training Simulator Featuring Enhanced Linkage between Simulator and DSC. In: TAPPI. Proceedings of TAPPI 1999 Pulping Conference. Pp. 1031–1045.

Tuuri, S. & Juslin, K. 1995. Uuden voimalaitos- ja automaatiokonseptin varmistus esisuunnitteluprojektin aikana (In Finnish). In: Finnish Automation Society. Proceedings of Finnish Automation days 1995. Pp. 453–457.

UKCEB. 2001. UK Council for Electronic Business, STEP Homepage, http://www.ukceb.org/step/ ,[referenced 18.2.2002].

W3C. 1999. HTTP Specification, http://www.w3.org/Protocols/Specs.html , [referenced 19.2.2002] .

W3C. 2000a. SOAP Specification, http://www.w3.org/TR/SOAP/ , [referenced 19.2.2002].

W3C. 2000b. XML Specification 1.0, http://www.w3.org/TR/2000/REC-xml-20001006 , [referenced 22.2.2002].

Watts, S. 1999. Web document. http://www.kbintl.com/e-x/stories/99apr005.html, [referenced 26.10.2001].

Wingems. 2001. Wingems homepage, http://www.pacsim.com/WG/default.shtml, [referenced 7.11.2001].

Yli-Petäys, J., Pyykkö, J. & Ropponen, H. 2001. Voimalaitossimulaattorin toteutus ja käyttö voimalaitoksen henkilökunnan koulutuksessa (In Finnish). In: Finnish Automation Society. Publication Series number 24. Pp. 222–227. ISBN 952-5183-17-3

Zamarreno, J.M., Acebes, F. & Alves, R. 2000. OPC-based real time simulator: architecture and practical example. In: Scandinavian Simulation Society. Proceedings of SIMS Simulation Conference 2000. Pp. 109–118.

Zeigler, B. 1976. Theory of Modelling and Simulation. New York: John Wiley.

Appendix A: An example of EXPRESS language and late and early binding

The following is a short example of an EXPRESS language syntax.

```
SCHEMA pipe_example;

TYPE pipe_material
= ENUMERATION OF (copper, aluminium, stainless_steel);
END_TYPE;

ENTITY pipe
length: REAL;
diameter: REAL;
material: pipe_material;
END_ENTITY;

END_SCHEMA;
```

A late bound DTD can be written for the EXPRESS language. The entire specification for this DTD is given in the ISO-10303-28 draft standard. The valid XML document according to the DTD for the EXPRESS example shown above is the following:

```
<?xml version="1.0"?>
<!DOCTYPE express_driven_data SYSTEM "express-dtd-v6.dtd">

<express_driven_data>
<schema_decl>
<schema_id>pipe_example</schema_id>

<type_decl>
<type_id>pipe_material</type_id>
<underlying_type>
```

<enumeration>
<enumeration_id>cupper</enumeration_id>
<enumeration_id>aluminium</enumeration_id>
<enumeration_id>stainless_steel</enumeration_id>
</enumeration>
</underlying_type>
</type_decl>

<entity_decl>
<entity_id>pipe</entity_id>
<explicit_attr_block>
<explicit_attr>
<attribute_id>length</attribute_id>
<base_type>
<real/>
</base_type>
</explicit_attr>
<explicit_attr>
<attribute_id>diameter</attribute_id>
<base_type>
<real/>
</base_type>
</explicit_attr>
<explicit_attr>
<attribute_id>material</attribute_id>
<base_type>
<type_ref>pipe_material</type_ref>
</base_type>
</explicit_attr>
</explicit_attr_block>
</entity_decl>

</schema_decl>
</express_driven_data>

In early binding the actual names and structures from the EXPRESS schema are embedded into the DTD definition. The early bound DTD for the same example looks as follows:

```
<!ELEMENT pipe_material EMPTY>
<!ATTLIST pipe_material
source_express_concept_type CDATA #FIXED "enumeration-type"
express_name CDATA #FIXED "pipe_material"
value (copper | aluminium | stainless_steel) #REQUIRED>

<!ELEMENT pipe (pipe.length, pipe.diameter, pipe.material)>
<!ATTLIST pipe id ID #REQUIRED
source_express_concept_type CDATA #FIXED "entity-data-type"
express_name CDATA #FIXED "pipe">

<!ELEMENT pipe.length (real)>
<!ATTLIST pipe.length source_express_concept_type CDATA #FIXED "attribute"
express_name CDATA #FIXED "length">

<!ELEMENT pipe.diameter (real)>
<!ATTLIST pipe.diameter source_express_concept_type CDATA #FIXED "attribute"
express_name CDATA #FIXED "diameter”>

<!ELEMENT pipe.material (pipe.material)>
<!ATTLIST pipe.material source_express_concept_type CDATA #FIXED "attribute"
express_name CDATA #FIXED "material”>
```

The following is an example of an XML document or an instance of a pipe according to early bound DTD. If the fixed attributes are not given in the XML the values are taken from the DTD.

```
<pipe>
<pipe.length>
<real>2.45</real>
</pipe.length>
<pipe.diameter>
<real>0.25</real>
```

</pipe.diameter>
<pipe.material>
<pipe_material value=”stainless_steel”/>
</pipe.material>
</pipe>

Appendix B: An example of the usage of the data model

The following example process is described using the GML format. The purpose of the example is to illustrate different mechanisms in the data model.

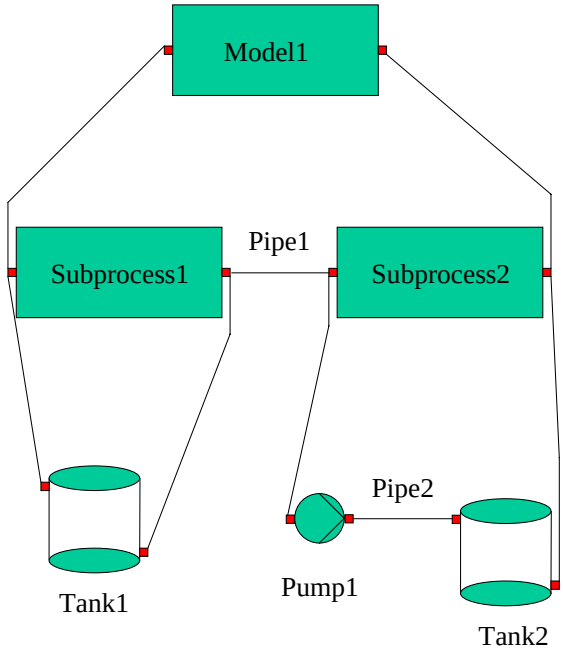

Figure B.1. Example process.

The component type descriptions of the model components are shown below. In a real case the component types would have more property infos. In order to keep the example compact, only one property info for the parameter values per component type is created. Also the ids have been changed from GUIDs to more intuitive strings.

Sub process component type with one value mapping property and two terminal properties (one start and one end terminal, both flow type). The value mapping

property is defined to be optional i.e. the instance components do not have to contain it:

```
<ComponentType id="subprocess component type" name="Subprocess" locked=”false” runnable=”false” loadable=”false” liftValues=”true” liftTerminals=”true” composed=”true”>

<PropertyInfo id="tank level mapping property info of subprocess" name="Tank level" necessity="optional">
<VectorInfo type="reference" allowedSize="1"/>
<Constraint name="valueMapping"/>
</PropertyInfo>

<PropertyInfo id="subprocess input terminal property info" name="Input" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="endTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

<PropertyInfo id="subprocess output terminal property info" name="Output" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="startTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

</ComponentType>
```

Tank component type description with one value property (liquid level) and two terminal properties (one start and one end terminal, both flow type):

```
<ComponentType id="tank component type" name="Tank" locked=”false” runnable=”false” loadable=”false” liftValues=”false” liftTerminals=”false” composed=”false”>

<PropertyInfo id="tank liquid level property info" name="Liquid level" necessity="implied">
<VectorInfo type="double" allowedSize="1" unit="m"/>
```

```
<Default>2.0</Default>
</PropertyInfo>

<PropertyInfo id="tank input terminal property info" name="Input" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="endTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

<PropertyInfo id=" tank ouput terminal property info" name="Output" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="startTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

</ComponentType>
```

Pump component type description with one value property (running information) and two terminal properties (one start and one end terminal, both flow type):

```
<ComponentType id="pump component type" name="Pump" locked=”false”
runnable=”false” loadable=”false” liftValues=”false” liftTerminals=”false”
composed=”false”>

<PropertyInfo id="pump running property info" name="Running" necessity="implied">
<VectorInfo type="boolean" allowedSize="1"/>
<Default>false</Default>
</PropertyInfo>

<PropertyInfo id="pump input terminal property info" name="Input" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="endTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>
```

```
<PropertyInfo id=" pump output terminal property info" name="Output"
necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="startTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

</ComponentType>
```

Pipe component type description with one value property (length) and two terminal properties (one start and one end terminal, both flow type):

```
<ComponentType id="pipe component type" name="Pipe" locked=”false”
runnable=”false” loadable=”false” liftValues=”false” liftTerminals=”false”
composed=”false”>


<PropertyInfo id="pipe length property info" name="Length" necessity="implied">
<VectorInfo type="double" allowedSize="1"/>
<Default>5.0</Default>
</PropertyInfo>


<PropertyInfo id="pipe input terminal property info" name="Input" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="endTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>


<PropertyInfo id="pipe output terminal property info" name="Output" necessity="implied">
<VectorInfo type="reference" allowedSize="2">
<Default/>
<Default/>
</VectorInfo>
<Constraint name="startTerminal">
<Constraint name="flow"/>
</Constraint>
</PropertyInfo>

</ComponentType>
```

Terminal typing is done here on a very general level. There are only flow type terminals in the model and all connections are possible. The terminal rules for these component types could look like the following:

```
<TerminalRules>

<ReferenceRule startSubConstraint=”flow”>
<EndSubConstraint name=”flow”/>
</ReferenceRule>

</TerminalRules>
```

The example process can be described using these component types and terminal rules. It has to be pointed out that all this XML data is processed programmatically. The model configurators and model users do not have to deal with the XML. The kernel developers and providers describe the component types and terminal rules but they do not have to handle instance XML either. By closely examining the example one can notice that the tank level property is statically mapped from tank2 to the upper levels and the running information of pump1 is dynamically mapped to the upper levels. The separator for vector elements in a reference type of vector is ‘~’. The connection is the first element of the reference vector of a terminal and mapping is the second element.

```
<!-- Model1 component with statically mapped liquid level from tank2. Input terminal is mapped to the input terminal of subprocess1and output terminal is mapped to the output terminal of subprocess2.  The running property of pump1 is dynamically mapped. -->
<Component id="model1 component" name="Model1" typeId="subprocess component type" tracked="false">

<Property id="statical mapping property of model1 for tank level" infoId="tank level mapping property info of subprocess">
<Vector>statical mapping property of subprocess2 for tank level</Vector>
</Property>

<Property id="model1 input terminal" infoId="subprocess input terminal property info">
<Vector>~subprocess1 input terminal</Vector>
</Property>

<Property id="model1 output terminal" infoId="subprocess ouput terminal property info">
<Vector>~subprocess2 output terminal</Vector>
</Property>
```

<Property id="dynamical mapping property of model1 for pump running property" infoId="pump running property info" lifted="true">
<Vector>dynamical mapping property of subprocess2 for pump running property
</Vector>
</Property>

<!—Subprocess1 component does not contain the optional mapping property defined in the type description. Input terminal is mapped to the input terminal of tank1and output terminal is mapped to the output terminal of pipe1. The output terminal is connected to the input terminal of the pipe1 -->

<Component id="subprocess1 component" name="Subprocess1" typeId="subprocess component type" tracked="false">

<Property id="subprocess1 input terminal" infoId="subprocess input terminal property info">
<Vector>~tank1 input terminal</Vector>
</Property>

<Property id="subprocess1 ouput terminal" infoId="subprocess output terminal property info">
<Vector>pipe1 input terminal~tank1 output terminal</Vector>
</Property>

<Component id="tank1 component" name="Tank1" typeId="tank component type" tracked="false">

<Property id="tank1 liquid level property" infoId="tank liquid level property info">
<Vector>3.76</Vector>
</Property>

<Property id="tank1 input terminal" infoId="tank input terminal property info">
<Vector>~</Vector>
</Property>

<Property id="tank1 output terminal" infoId="tank output terminal property info">
<Vector>~</Vector>
</Property>

</Component>

</Component>

<Component id="pipe1 component" name="Pipe1" typeId="pipe component type" tracked="false">

<Property id="pipe1 length property" infoId="pipe length property info">
<Vector>2.5</Vector>
</Property>

<Property id="pipe1 input terminal" infoId="pipe input terminal property info">
<Vector>~</Vector>
</Property>

<Property id="pipe1 output terminal" infoId="pipe output terminal property info">
<Vector> subprocess2 input terminal~</Vector>
</Property>

</Component>

<!—Subprocess2 component with statically mapped liquid level from tank2. Input terminal is mapped to the input terminal of pump1 and output terminal is mapped to the output terminal of tank2. The running property of pump1 is dynamically mapped. -->

<Component id="subprocess2 component" name="Subprocess2" typeId="subprocess component type" tracked="false">

<Property id="statical mapping property of subprocess2 for tank level" infoId="tank level mapping property info of subprocess">
<Vector>tank2 liquid level property</Vector>
</Property>

<Property id="subprocess2 input terminal" infoId="subprocess input terminal property info">
<Vector>~pump1 input terminal</Vector>
</Property>

<Property id="subprocess2 output terminal" infoId="subprocess output terminal property info">
<Vector>~tank2 output terminal</Vector>
</Property>

<Property id="dynamical mapping property of subprocess2 for pump running property" infoId="pump running property info" lifted="true">
<Vector>pump1 running property</Vector>
</Property>

<Component id="pump1 component" name="Pump1" typeId="pump component type" tracked="false">

<Property id="pump1 running property" infoId="pump running property info">
<Vector>true</Vector>

```
</Property>

<Property id="pump1 input terminal" infoId="pump input terminal property info">
<Vector>~</Vector>
</Property>

<Property id="pump1 output terminal" infoId="pump output terminal property info">
<Vector>pipe2 input terminal~</Vector>
</Property>

</Component>

<Component id="pipe2 component" name="Pipe2" typeId="pipe component type"
tracked="false">

<Property id="pipe2 length property" infoId="pipe length property info">
<Vector>4.2</Vector>
</Property>

<Property id="pipe2 input terminal" infoId="pipe input terminal property info">
<Vector>~</Vector>
</Property>

<Property id="pipe2 output terminal" infoId="pipe output terminal property info">
<Vector>tank2 input terminal~</Vector>
</Property>

</Component>

<Component id="tank2 component" name="Tank2" typeId="tank component type"
tracked="false">

<Property id="tank2 liquid level property" infoId="tank liquid level property info">
<Vector>1.45</Vector>
</Property>

<Property id="tank2 input terminal" infoId="tank input terminal property info">
<Vector>~</Vector>
</Property>

<Property id="tank2 output terminal" infoId="tank output terminal property info">
<Vector>~</Vector>
</Property>
```

```
</Component>

</Component>

</Component>
```

The graphichs under the svg element of subprocess 2 would look like the following. The symbol elements for the components and terminals are copied from the symbol library. The terminal symbol refers to the terminal type that it represents. Component symbols refer to the corresponding component types. The use elements under the component symbol that represent the terminals of the component refer to the corresponding property info under the component type description. The use elements that represent the actual component instances refer to the corresponding components and the connection lines refer to the start terminal property, the starting point of the connection line.

```
<svg width="1000px" height="200px" viewBox="0 0 1000 200"
xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">

<defs>

<symbol id="flow terminal symbol" gml:tid=”flow” gml:type=”terminal”>
<rect x="0" y="0" width="3" height="6"
style="fill:rgb(255,64,64);stroke:rgb(0,0,128);stroke-width:1"/>
</symbol>

<symbol id=”pump symbol” gml:tid="pump component type" gml:type=”component”>
<ellipse cx="26" cy="25.5" rx="18" ry="18.5"
style="fill:rgb(192,192,255);stroke:rgb(0,0,128);stroke-width:1"/>
<line x1="44" y1="25.875" x2="26.75" y2="7.125" style="fill:none;stroke:rgb(0,0,0);stroke-
width:1"/>
<line x1="25" y1="43.625" x2="44" y2="26.375" style="fill:none;stroke:rgb(0,0,0);stroke-
width:1"/>
<use id="pump input terminal property info reference" x="5" y="23" xlink:href="#flow
terminal symbol"/>
<use id="pump ouput terminal property info reference" x="44" y="23" xlink:href="#flow
terminal symbol"/>
</symbol>

<symbol id=”tank symbol” gml:tid="tank component type" gml:type=”component”>
<ellipse cx="27.5" cy="10" rx="24.5" ry="8"
style="fill:rgb(192,192,255);stroke:rgb(0,0,128);stroke-width:1"/>
```

```
<ellipse cx="27.5" cy="46" rx="24.5" ry="8"
style="fill:rgb(192,192,255);stroke:rgb(0,0,128);stroke-width:1"/>
<line x1="52" y1="10" x2="52" y2="46" style="fill:none;stroke:rgb(0,0,0);stroke-width:1"/>
<line x1="3" y1="10" x2="3" y2="46" style="fill:none;stroke:rgb(0,0,0);stroke-width:1"/>
<use id="tank input terminal property info reference" x="0" y="14" xlink:href="#flow
terminal symbol"/>
<use id="tank output terminal property info reference" x="52" y="40" xlink:href="#flow
terminal symbol"/>
</symbol>

</defs>

<g id="pump1 component reference">
<use x="30" y="6" xlink:href="#pump symbol"/>
</g>

<g id="tank2 component reference">
<use x="200" y="15" xlink:href="#tank symbol" />
</g>

<g id="pipe2 component reference">
<line x1="77" y1="32" x2="200" y2="32" style="fill:none;stroke:rgb(0,0,0);stroke-
width:1"/>
</g>

</svg>
```

In order to extend the symbol element with its own attributes following DTD extension declarion has to be added to the beginning of the svg document before reading it to the SVG renderer:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd"
[
<!ATTLIST symbol
  xmlns:gml CDATA #FIXED "http://www.simulationsite.com/gml"
  gml:tid CDATA #REQUIRED gml:type CDATA #REQUIRED>
]>
```

Appendix C: Type library examples of Prosim and Apros

Prosim and Apros type libraries contain the component type descriptions for the co-use case represented in Chapter 5. This Appendix lists the pump type description for both type libraries. One should notice from the description how differently Prosim and Apros deal with the process components. This is due to the different simulation schemes used for simulating the components.

Prosim pump type:

<ComponentType locked=”false” runnable=”false” loadable=”false” liftValues=”false” liftTerminals=”false” composed=”false” name="Pump" id="a251F0FF0-A60D-4090-ADCD-65ABC0215A5D">

<PropertyInfo name="Is flow copied from outlet" necessity="implied" id="a12DAEFB2-C96C-44e5-BEBE-13F73FA3A408">
<VectorInfo type="boolean" allowedSize="1">
<Default>true</Default>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="Efficiency" necessity="implied" id="a63D18836-31C9-489a-8DDA-342FD596C605">
<VectorInfo type="float" allowedSize="1" min="0" max="1">
<Default>0.8</Default>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="Losses" necessity="implied" id="a561149A2-ACD7-4a5c-9B76-A1F293B3AE4D">
<VectorInfo type="float" allowedSize="1" min="0" max="100" unit="%">
<Default>5</Default>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="Power" necessity="optional" id="a9D0999F8-D5F2-439e-9588-F49B2D70E37F">
<VectorInfo type="float" allowedSize="1" unit="kW"/>
</PropertyInfo>

<PropertyInfo name="Flow in" necessity="optional" id="aCB296406-EAE4-4a49-B990-26C104CB0560">

<VectorInfo type="reference" allowedSize="2"/>
<Constraint name="endTerminal">
<Constraint name="Flow in"/>
</Constraint>
</PropertyInfo>

<PropertyInfo name="Flow out" necessity="optional" id="a21BCAC02-A322-41bd-88E3-592A40E798F7">
<VectorInfo type="reference" allowedSize="2"/>
<Constraint name="endTerminal">
<Constraint name="Flow out"/>
</Constraint>
</PropertyInfo>

<PropertyInfo name="Connection to shaft" necessity="optional" id="a366B1579-C937-402e-8F8E-0E5B41E04C45">
<VectorInfo type="reference" allowedSize="2"/>
<Constraint name="endTerminal">
<Constraint name="Mechanical coupling"/>
</Constraint>
</PropertyInfo>

</ComponentType>

Apros pump type:

<ComponentType name="BASIC_PUMP_TYPE" locked=”false” runnable=”false” loadable=”false” liftValues=”false” liftTerminals=”false” composed=”false” id="a1a24ebad-9635-4690-b17e-c553bafd1e89">

<PropertyInfo name="PU11_ACCURACY_LEVEL" id="a3c6a325d-16e0-46ab-a581-1eac9efb3a2e" necessity="optional">
<VectorInfo type="a04C57C7B-4AB9-4840-B313-36B685AD564F" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_AREA" id="aa192fde1-fc3d-463a-a976-7f2306519409" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_BUSBAR_NAME" id="a36262968-4b8f-42be-933b-5fda884d5f0c" necessity="optional">
<VectorInfo type="reference" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_COAST_DOWN_TIME" id="a6c6b3505-ebdc-4a11-b0c0-eb760f592315" necessity="optional">

<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_CURRENT" id="a89882e43-2599-4060-a6f0-3aa890215109" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_CURVE_TYPE" id="a6d35a2e3-edc4-4cd9-b1bc-2b10dc40b30c" necessity="optional">
<VectorInfo type="integer" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_DENS_NOMINAL" id="a523d925e-fe5c-48c5-b51f-49f090f85672" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_FLOW_NOMINAL" id="adc3347ce-c107-4b46-a170-ec89f3a8a30a" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_HEAD_CURVE" id="a4cb11975-497a-40b0-b4e5-f72063078335" necessity="optional">
<VectorInfo type="vector" allowedSize="2">
<VectorInfo type="float" allowedSize="10"/>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="PU11_HEAD_MAXIMUM" id="a0ec303d4-2dad-4cdf-997d-032b56a42c1f" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_HEAD_NOMINAL" id="ac86ca869-ba53-4ac4-966b-8877a45d9da0" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_HEAT_CALCULATED" id="af7ba266f-e9e0-4cc3-a1c7-ebc58532e179" necessity="optional">
<VectorInfo type="boolean" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_LENGTH" id="a6ba5c846-ec57-488e-ae23-f7ff53a47e6a" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_LOSS_COEFF_STOPPED" id="a131855b3-618a-4a78-8944-28d15851d131" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_MALFUNCTION" id="ae0319b01-701a-4367-a5c3-4dc2a89b0a03" necessity="optional">
<VectorInfo type="integer" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_MIX_MASS_FLOW" id="aa5322846-0642-476f-bf01-984a3309f61d" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_MOTOR_EFFICIENCY" id="ab83631c9-fa19-4299-8919-bae68db07fab" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_NPSH" id="a47416b19-59fd-478b-9b5a-2dc8125db80a" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_POWER_COEFF" id="a4b3be30a-d459-4658-b21c-4985f5d867b1" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_PUMP_EFFICIENCY" id="a3b7af37c-5899-4eb8-adb5-953cefbfc666" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_SECTION_NAME" id="adcda06fd-ab72-408e-917b-cede84d3f8eb" necessity="optional">
<VectorInfo type="a3A307B86-A098-4B05-88CA-551920B709E5" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_SPEED" id="a796273b0-72c7-425c-a842-4d1b0baffac7" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_SPEED_NOMINAL" id="a94ee633c-0df1-4636-9097-c2ba33336cf4" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="PU11_SPEED_SET_POINT" id="a2fc5d02b-6df2-4650-8b19-07f958a959c4" necessity="optional">
<VectorInfo type="float" allowedSize="1"/>
</PropertyInfo>

<PropertyInfo name="MODULE_NAME" id="a52ebf398-6d34-493f-a28e-49733fc40679" necessity="optional">
<VectorInfo type="reference" allowedSize="2">
<Constraint name="endTerminal">
<Constraint name="endReference"/>
</Constraint>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="PU11_CONNECT_POINT_1" id="ae771f3d3-792c-4db8-b4b0-256ff810d8c6" necessity="optional">
<VectorInfo type="reference" allowedSize="2">
<Constraint name="startTerminal">
<Constraint name="conPoint_POINT"/>
</Constraint>
</VectorInfo>
</PropertyInfo>

<PropertyInfo name="PU11_CONNECT_POINT_2" id="ac3bef610-6f12-4df4-8ce1-c4c76702d650" necessity="optional">
<VectorInfo type="reference" allowedSize="2">
<Constraint name="startTerminal">
<Constraint name="conPoint_POINT"/>
</Constraint>
</VectorInfo>
</PropertyInfo>

</ComponentType>

Published by

Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
Phone internat. +358 9 4561
Fax +358 9 456 4374

Series title, number and report code of publication

VTT Publications 479
VTT–PUBS–479

Author(s)
Karhela, Tommi

Title

A Software Architecture for Configuration and Usage of Process Simulation Models

Software Component Technology and XML-based Approach

Abstract

Increased use of process simulation in different phases of the process and automation life cycle makes the information management related to model configuration and usage more important. Information management increases the requirements for more efficient model customisation and reuse, improved configurational co-use between different simulators, more generic extensibility of the simulation tools and more flexible run-time connectivity between the simulators and other applications.

In this thesis, the emphasis is on large-scale dynamic process simulation of continuous processes in the power, pulp and paper industries. The main research problem is how to apply current information technologies, such as software component technology and XML, to facilitate the use of process simulation and to enhance the benefits gained from using it. As a development task this means developing a new software architecture that takes into account the requirements of improved information management in process simulation. As a research objective it means analysing whether it is possible to meet the new requirements in one software architecture using open specifications developed in information and automation technologies.

Process simulation is analysed from the points of view of standardisation, current process simulation systems and simulation research. A new architectural solution is designed and implemented. The degree of meeting the new requirements is experimentally verified by testing the alleged features using examples and industrial cases.

The main result of this thesis is the design, description and implementation of a new integration architecture for the configuration and usage of process simulation models. The original features of the proposed architecture are its openness, general distribution concept and distributed extensibility features.

Keywords
Process simulation, software architecture, XML, software component technology, model configuration

Activity unit
VTT Industrial Systems, Tekniikantie 12, P.O.Box 1301, FIN–02044 VTT, Finland

ISBN	Project number
951–38–6011–6 (soft back ed.) 951–38–6012–4 (URL:http://www.inf.vtt.fi/pdf/)	

Date	Language	Pages	Price
October 2001	English	129 p. + app. 19 p.	C

Series title and ISSN	Sold by
VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.inf.vtt.fi/pdf/)	VTT Information Service P.O.Box 2000, FIN-02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374