

Technology Briefing Report

System Modelling

RENAISSANCE Esprit Project 22010

Identifier	D5.1b System Modelling
Type	Internal result
Activity	WP5.1b
Date	15 th September 1996
Status	Final
Partner	SINTEF / Thomas Kresken
Availability	Restricted

1. Overview

System modelling is a technique to express, visualise, analyse and transform the architecture of a system. Here, a system may consist of software components, hardware components, or both and the connections between these components. A system model is then a skeletal model of the system.

System modelling is intended to assist in developing and maintaining large systems with emphasis on the construction phase. The idea is to encapsulate complex or changeable aspects of a design inside separate components with well-defined interfaces indicating how each component interacts with its environment. Complete systems are then developed by composing these components. System modelling can increase reliability and reduce development cost by making it easier to build systems, to reuse previous built components within new systems, to change systems to suit changing requirements such as functional enhancement and platform changes, and to understand systems. In this way, a system model can satisfy different requirements such as documenting the system, providing a notation for tools such as consistency checkers and can also be used in the design stage of system development.

Thus, system modelling is used to ensure that a developing piece of software evolves in a *consistent* manner and that the task of integrating software components is *simplified*.

1.1 Modelling framework and language

For system modelling, we need a conceptual framework and a system modelling language, textual and/or diagrammatic. Besides textual notations like tables or prose, diagrammatic notations like graphs are common today. Within these diagrams, there are symbols representing the parts of the system, e.g. objects and groups of objects, and other symbols visualising the connections between these parts. The number of symbols for each purpose differs noticeably between notations.

During the past decades four main conceptual frameworks have evolved:

- Design Methods
- Module Interconnection Languages (MILs)
- Software Architectures
- Design Patterns

Design methods focusing on program language modules date back to the early seventies; modelling systems with objects is a technique of the late eighties. The first MIL was presented 1975, and in the following years a lot of different MILs have been developed. In contrast, software architectures and design patterns are more recent techniques and are each only about five years old.

1.2 Design methods

Design methods consist of a concept, a language, and a design process. The concept defines which parts of the program are to be represented by the components of the system model. The language is used to describe the system model. To carry out the design of the system according to the software lifecycle or a part of it, a step-by-step process is given to be followed by the designer.

When modelling a system with a *modular design method*, program modules are used as the components of which the system is composed. Modules include related functions of a specific program language and may keep the data used by these functions. Further, a module offers an interface with the functions that may be used from this module.

An interrelationship between two modules is established by a function contained in one module calling a function contained in another. Thus, this modelling method describes systems in terms of function calls and composition of modules containing functions.

A commonly used example is Structured Design (SD) introduced by Constantine and Yourdon at the end of the Seventies; Modular Design (MD) is an extension of SD, subdividing the system to be designed into modules which can be developed independently of each other.

Object-oriented design methods, on the other hand, do not focus on the functional aspect. An object component includes data and functions. These functions constitute an external interface, and represent the only possibility

to manipulate and/or access the data within an object. Another type of components are classes, which serve as templates from which objects can be instantiated, i.e. derived.

The connections between the objects of a system model are manifold, e.g. instantiation of an object from a class, composition, and use. Interacting objects often are not viewed as calling each other's functions, but requesting each other via messages to perform a desired action on their data. An object is responsible that its representation is hidden from other objects. Thus, the object-oriented modelling method adds behaviour over a mere structural solution.

A design method on the verge to object-oriented design is MD++, mainly MD with an object-oriented extension. Other methods focus on a particular implementation language, e.g. HOOD (Hierarchical Object-Oriented Design) is used to develop systems with the ADA implementation language. Today's widely used object-oriented methods include the Object Modelling Technique (OMT) by Rumbaugh, the Object-Oriented Design and Analysis (OODA) by Booch, the Object-Oriented Design Language (OODLE) by Shlaer, and Ooram (Object-Oriented role modelling) by Reenskaug.

For both modular and object-oriented design methods there exist textual and diagrammatic notations coming with the specific design methods, with diagrammatic notations becoming common during the last years.

Besides these two main design methods there exist design methods for special purposes like task design, which is focusing on processes and their communications, and real-time systems design.

1.3 Module Interconnection Languages

Creating program modules and connecting them to form large systems are different design efforts. Module interconnection languages (MILs) are means to support the connection effort of large systems.

A MIL is separate and distinct from an implementation language. Its purpose is to formally express the system designer's intent regarding system structure. In the absence of a MIL, this information is buried within the implementation language modules, linker commands and informal documentation; thus, the information is scattered, unreliable and doesn't enforce connectivity.

The main features of MILs are:

- a separate language to describe the system,
- static intermodule type checking,
- design and construction information in a single description, and
- version control.

A MIL can be considered a structural design language because it states what the system modules are and how they fit together to implement the system. However, MILs are *not* concerned with what the system does (specification information), how the major parts of the system are embedded into the organisation (analysis information), or how the individual modules implement their function (detailed design information).

The components of MIL notations are modules (computational units) with well-defined interfaces, through which the modules import and export resources (named implementation-language elements). The connection of the modules is based on definition/use bindings (i.e. each module defines a set of facilities that are available to other modules, and uses facilities that are provided by other modules).

Given that a MIL offers a formal notation to express the system model, this model can be used by specific tools, e.g. a compiler, to check the completeness and consistency of the system described. A compiler for a MIL ensures that if one module uses a resource that another provides, the types of the resources match; that if a module declares to offer a resource, it actually does and so on. Reasoning about the correctness of a MIL description might be able in terms of pre- and post-conditions.

As mentioned before, without a MIL, the information how the system is to be build from its components and how these interact with each other, is scattered. A MIL description is a single document which contains explicitly all the necessary information to construct the system it describes. MIL system models are written descriptions of the system design which must be followed. Thus, a MIL can prohibit programmers from changing the system's architectural design during evolution and maintenance without an explicit change in the MIL description.

During the evolution and maintenance of a system, modules are changed and, hence, new versions of modules are obtained. Further, there are several reasons for building different versions of modules, e.g. if the system is provided for different hardware platforms. In the latter case, different configurations of the system are to be supported. Some MILs offer a version control system to determine which version of which component should be used to form a particular version of a particular configuration.

Module Interconnection Languages are means to explicitly describe the structure of a system and provide a basis for enforcing that structure. On the other hand, MIL descriptions are cost-effective only for the decomposition of larger systems. Further, they do not promote reuse of components or reusable patterns of composition. Usually, MILs are described in a textual notation.

More recently, a number of component-based languages has been proposed. They describe systems as configurations of modules that interact in specific, predetermined ways (e.g. remote procedure call, messages, events) or enforce specialised patterns of organisation among components in large systems. These languages are usually oriented around a small, fixed set of communication paradigms and implementation-level descriptions, or enforce a specialised, single-purpose organisation, and are thus inappropriate for expressing a broad range of system architectures.

Well known module interconnection languages include Tichy's InterCol, introducing version management to MILs, and Somerville's and Thomson's System Structure Language (SySL). The Proteus Configuration Language (PCL), a 'new-generation MIL', has recently been developed within the Proteus research project.

1.4 Software Architectures

Both design methods and module interconnection languages force the system designer to use a relatively low level of abstraction. System decomposition is described in terms of modules, objects, or procedures, i.e. in terms of implementation-language items. With increasing size and complexity, the specification and design of the overall system structure become more important than the choice of algorithms and data structures. To meet these requirements, software architectures are being developed.

There exist some different definitions on what the term 'software architecture' should include besides (computational) components and their interrelationships. Some developers include the possibility to describe constraints on the relationships, others focus on different connections between components, yet others develop rationales which demonstrate that the components, connections, and constraints define a system that satisfies the given requirements.

1.4.1 Software Architectures vs. MILs

The main difference between MILs and software architectures is the increasing level of abstraction for system modelling with software architectures and the lack of a clean separation between architectural-level issues and those related to the implementation level with MILs.

Where MILs describe the connections between system components through definition/use bindings, i.e. in terms of the implementation language, software architectures use *styles* like remote procedure calls, pipelines, and client/server organisation, without regarding how these connections are implemented in a specific programming language. Using software architectures to model a system, the level of abstraction is significantly increased.

Further, MILs are not capable of expressing connections of specific architectural styles explicitly, if these connections are not expressible in the implementation language, i.e. mainly procedure calls and data sharing. For example, the protocol between components organised in a client/server style is buried within the functions contained in the implementation modules. MILs only show how those modules are connected but fail to make the protocol itself obvious. Architectural abstractions are forced to be translated into low-level primitives provided by the implementation language. Hence, system models built with software architectures treat connections as elements in the same way that components are used, regardless of how these components and connections will be implemented.

1.4.2 Software Architectures vs. Design Methods

From the software architecture's point of view, modular and object-oriented notations are appropriate for special kinds of architectural decomposition. But they, like MILs, force designers to use a lower level of

abstraction than is regarded appropriate. Design methods, too, do not separate cleanly between the architectural level of a system's design and the implementation level. All in all, the argumentation is the same as in section 1.4.1.

1.4.3 State of the Art

Recently, software architectures have become an important field of study for software engineering researchers and practitioners. There is work going on in areas such as module interface languages, domain-specific software architectures, software reuse, formal methods for architectural design, and architectural design environments.

Although there are many architectural styles, e.g. pipelines, layered systems, client/server organisations, they are usually described only in an idiomatic way and applied in an ad-hoc method. To lend formal rigor to system development, research focuses on various ways to formally specify the architecture of a system. Compared to MILs, software architectures do not provide yet means to *enforce* the overall structure of a system's architecture when implementing the system. Further, tools developed for system design with software architectures do not intend to provide version control yet.

As seen, neither module interconnection languages nor implementation languages are able to express a system's decomposition as required by the discipline of software architecture. Thus, architecture description languages (ADLs) are being developed due to the need for higher-level languages specifically oriented to the problem of software architectures. At the present time, various ADLs and architecture-driven implementation environments are used in research projects. However, there exists no wide industrial experience with these systems yet.

The notations used to describe and visualise software architectures range from textual notations to informal diagrams and graphical notations provided by ADLs and their supporting tools.

1.5 Design Patterns

Compared to the three conceptual frameworks discussed so far, design patterns are the most theoretical approach to model a system and its behaviour.

Design patterns were introduced by the architect Christopher Alexander for the design of buildings and cities at the end of the Seventies. Since the beginning of the Nineties, software engineering researchers have become more and more interested in design patterns. Most research effort has been put into finding object-oriented design patterns, but the use of design patterns is by no means restricted to object-oriented software development.

There exists no formal definition of the term 'design pattern', but the smallest common denominator seems to be 'a reusable solution to a problem in a particular context'.

Patterns are applicable in fields where certain problems occur over and over again and describe the core of the solution to those problems. To reuse a pattern, knowledge about the decisions and trade-offs that led to the particular solution are important. Design patterns are, thus, described in a textual notation, either informal (i.e. prose) or formal (i.e. a particular formalism or implementation language), often accompanied by a graphical notation (i.e. sketches or diagrams).

Usually, a description of a design pattern consists of the pattern *name*, the *problem description*, the *context* in which the problem occurs, the *solution* to the problem using the pattern, the sometimes conflictive *forces* influencing the problem and the solution, and at least one *example*. (As there exist plenty of different definitions of the term 'design pattern', there exists also no consensus on how to describe a given pattern. The items of the above enumeration can, again, be viewed as a smallest common denominator.) Given a description like that, a design pattern can be used to solve problems even by people who are not experienced enough to develop a similar solution. All in all, design patterns ease and customise the reuse of successful solutions to frequent problems.

Transferred to the field of software development, design patterns can assist in solving problems at different levels of abstraction, from the choice of algorithms and data structures via choosing the appropriate composition of objects and functions to overall architectural designs.

Design patterns usually describe software solutions used by professional, experienced designers and programmers in their software, making it easy to (re)use those proven solutions by anyone facing the same problem. When using design patterns this way, they act as building blocks of a system. Thus, design patterns aid in making a system or parts of it reusable.

Design patterns are, like software system architectures, independent of a specific implementation language or a language concept like modular or object-oriented programming. However, the most important value of patterns today is that they are used to complement existing design methods, solving problems that are beyond the reach of those methods, especially object-oriented design methods.

An object-oriented design pattern identifies the participating classes and instances and the way they interact. Each design pattern focuses on a particular object-oriented design problem using a description similar to that mentioned above. The example provided with the pattern is written in an object-oriented programming language. Here, the choice of implementation language is important, because it determines what can and cannot be implemented easily.

Design patterns must not be seen as a recipe of how to develop systems or parts of it. They represent a more generic description of how to build parts of a system. Patterns are not designed to be executed by computers but to be applied by humans in situations where the patterns seem appropriate.

The benefits of design patterns are reuse of previous patterns and software abstractions, guidance during the development of systems, and communication between the people involved in the development of systems.

2. Role in evolution process

2.1 Software lifecycle

The first phase when constructing a system is the requirement analysis phase. Here the main system components and their connections are determined. In the following design phase, these decisions are refined, general components are added (e.g. human interface components), and implementation decisions made. One part of the design phase is to make a system architecture model to visualise the components and their connections.

This model can also be used after the implementation of the system: During testing, the system model helps to determine where to use certain kind of tests.

2.2 Maintenance

When maintaining the system, the model aids in understanding the system. Various surveys have shown that 40-90% of the maintenance time is used for this purpose, depending on the specific system and the nature of the changes to be made. System models could reduce this amount of time significantly. Especially if main changes are to be made later, the model will help to show side-effects of contemplated changes. Provided that the model is kept updated, it represents an invaluable part of the system's documentation. In addition, retaining the designer's intentions about the composition of the system will help maintainers to preserve the integrity of the system's design.

The four conceptual frameworks discussed in the Overview chapter act at different levels of abstraction and, thus, aid in a different way to the understanding of the system.

Software architectures describe the overall system structure, without regarding how the components and their interconnections are implemented in a specific programming language. Module interconnection languages are not concerned about the implementation of the modules at the algorithmic level, although the system composition is expressed with the use of implementation-language items. Design methods define how the components and their interconnections are implemented in terms of the programming language paradigm used. Finally, design patterns can be used to describe system components and interconnections at any desired level of abstraction. When used to complement design methods, patterns are, of course, on a lower level of abstraction than design patterns describing the system's architectural structure.

2.3 Analysis of existing systems

The purpose of analysing existing systems is to understand *how* the system does *what* it does. Whether the system is maintained or it is prepared to be transformed, e.g. to another hardware platform, the emphasis of system analysis lies on understanding of which components it is built and how they interact.

A system model as the result of this analysis, again, aids to the understanding of the system and the side-effects of main changes.

2.4 Transformation of systems

System modelling eases the technical transformation of systems, i.e. from one hardware platform to another or from one architectural style to another, without changing the overall functionality of the system.

Using the model of the existing system, the benefits are the same as during maintenance: Understanding the overall structure of the system and understanding the side-effects of the changes to be made.

3. Relevance to RENAISSANCE

For many legacy systems there is no actual or updated documentation of the system's architecture. To retrieve information about its structure, the legacy systems will have to be reverse engineered and then documented (see the technology briefing report on reverse engineering by Engineering). This information will be used to build a model of the system's architecture. Whether it is possible to build this system model automatically or if this has to be done manually from the data of the reverse engineering process, depends on the tools available.

Such a model of a centralised system can be transformed, step by step, into one describing a more and more decentralised system exploiting a client/server architecture. Using a method with a diagrammatic notation will help the developer to better understand the system, i.e. which changes are more appropriate to transform the old system, and how changes of one component of the system may affect other components.

There may be several ways to visualise and transform an existing system. The developer will have to use his experience to decide on the most promising method here.

4. Available support

The available support differs significantly between the four conceptual frameworks of system modelling approaches explained in the Overview chapter.

Both **design methods** are very well supported by literature and tools. For every major system modelling language there exists at least one book by the developers, describing the method and the process associated with it. Further, a lot of books and articles about specific modelling languages or comparisons of them have been and are still published. Some of the modelling languages have been revised and refined since they came out.

Modular design methods are still in use, but object-oriented design methods have gained more and more ground during the past years.

If a tool vendor claims to support a certain method, this does not mean that the tool supports the entire process or all the concepts of that method. Nonetheless, there are a lot of tools supporting a method entirely.

The support for **Module Interconnection Languages** is rather poor. Although a lot of articles and books have been published on that topic, implementations and especially the industrial use of MILs can be neglected. Besides the Xerox system modeller there has been no nameable industrial tool support.

Software system architectures and **design patterns** are still topics of research. A lot of books and research articles are published on both, but finding throughout accepted 'standard' literature is still difficult.

The development of tools to support the Architecture Description Languages is in progress but there exists almost no industrial use besides in projects of the U.S. Department of Defense.

As design patterns are not meant to be executed by computers, there will be no tools for this purpose. But a lot of effort is used to find object-oriented patterns. So patterns may be used in object-oriented design methods and tools in the future.

Table 1 gives an overview over the support offered by some modelling languages according to the following requirements:

1. **Integrated system modelling:** Modelling all aspects of the product in one formalism, i.e. incorporate descriptions of and interrelationships between software and hardware, including network and distribution aspects.
2. **Multiple structural viewpoints:** Be able to express and show several viewpoints of the same system, e.g. its interface, its logical composition and its run-time structure.
3. **Structural variability:** The ability to define variability in the logical composition of a system, in interfaces and in relationships in which an entity participates.
4. **Component variability:** The ability to represent variability in the concrete system, e.g. versions, and to allow intentional version selection.
5. **Object-oriented modelling:** The extent to which the language uses the concepts provided in object-oriented formalisms, such as classification, inheritance and encapsulation.

Framework	Language	Int. Mod.	Mult. VP.	Struct. V.	Comp. V.	OO Mod.
Design methods	SD	Limited	None	None	None	None
	MD++	Limited	Limited	??	Limited	Limited
	HOOD	Limited	Limited	??	Limited	Limited
	OMT/Rumbaugh	Limited	Good	Good	None	Good
	OODA/Booch	Limited	Good	Good	None	Good
	OODLE/Shlaer	Limited	Limited	Good	None	Good
	OOram/Reenskaug	Limited	Good	Good	None	Good
MILs	InterCol/Tichy	Limited	None	None	Limited	Limited
	SySL/Sommerville	Good	None	Limited	None	Good
	PCL/PROTEUS	Good	Good	Good	Good	Good
Software Architectures	Various ADLs	Limited/Good	Limited	Good	None/Limited	Limited/Good
Design patterns	N/A	Good	Good	Good	Good	Good

Table 1: Support offered by modelling languages

The assessment of the software architecture's ADLs depends partly on the specific architecture description languages, as there exist ADLs for different domains and purposes.

5. Maturity assessment

The maturity for the methods discussed in the Available support chapter is assessed as follows:

Conceptual framework	Modelling language	Maturity
Design methods	SD	high
	MD++	medium - high
	HOOD	high
	OMT/Rumbaugh	high
	OODA/Booch	high
	OODLE/Shlaer	medium - high
	OOram/Reenskaug	medium
MILs	InterCol/Tichy	low - medium

	SySL/Sommerville	low - medium
	PCL/PROTEUS	medium
Software architecture	Various ADLs	low
Design patterns	N/A	low

Table 2: Maturity assessment

6. Inapplicability

The most important disadvantage with existing modelling frameworks is that they are meant for developing new systems, not re-engineering existing systems.

This is obvious especially with the processes of design methods. Without adapting the design process to re-engineering, almost no design method seems to be usable. Thus, the use of some tools available will be problematic because they guide the user step by step through the predetermined design process.

Today's object-oriented modelling languages are not supporting real-time systems, concurrent and distributed systems very well. Especially the last aspect may be a problem for the RENAISSANCE project. The lack of concepts and design process steps does not necessarily mean that it is impossible to model distributed systems. But it would be an advantage to have method and tool support for e.g. quality assurance of distributed systems.

Although there exist a few object-oriented modelling languages for real-time systems, they have other disadvantages like immaturity or sparse tool support.

Some of the design methods have been developed to be used with certain programming languages. They therefore lack generality both in their design process and in their concepts.

There exist a few reports on using design patterns in the development of large software systems. But still, sufficient material has not been published yet to allow decisions about the benefits of design patterns in large projects.

7. Future development

7.1 Design methods

Developers have become aware of the importance to support distributed and concurrent systems. All major method and tool developers are working on extensions for distribution and concurrency. How comprehensive these extensions will be cannot be evaluated at the moment. However, there seems no interest in changing the processes accompanying the various design methods to support re-engineering next to the development of new systems.

7.2 Unified Modeling Language

The Unified Modeling Language (UML), a third-generation object-oriented design method, represents the unification of three major second-generation object-oriented design methods (Rumbaugh's OMT, Booch's OODA and Jacobson's Objectory). UML is meant to be a method for modelling *systems*, not just software, and designed to be usable by both developers and development tools.

As a modelling language, UML is not only meant to support OMT, OODA and Objectory. UML is influenced by other methods and their developers, too, and developed to be a unified *notation* rather than a unified *process*. There will be no standardised process, although UML will probably be accompanied by an architecture-driven, incremental, and iterative development process. However, the developers of UML regard the processes of all widely used design methods to be sufficient for the use of the UML notation. Moreover, processes are context-dependent. Sticking to a special kind of process will not work in every domain and for every project, respectively. Thus, in the eyes of its developers, UML would miss its goal to become a standard object-oriented design method.

The UML notation is a union of the graphical syntax of OMT, OODA, and Objectory, including use cases (Objectory), class diagrams (OMT, OODA), state-machine diagrams (OODA, OMT) next to other

representations found in OMT, OODA, Objectory, and other methods. With respect to the modelling language, UML is no departure from its predecessors.

Because UML will be more of a unified notation than of a unified process, the development of UML should be followed. Thus, for the re-engineering of legacy systems, it is possible to either choose the most suitable process or to develop a particular, application-tailored process.

Compared to currently used object-oriented design methods, UML provides additional modelling elements. Some weight is put on the modelling of concurrent and distributed systems (particular for modelling systems built with CORBA and OLE/COM, and distribution as found in Java applets). The UML notation is not meant to be restricted to those technologies, but they are expected to be of great importance. Further, the modelling of threads and processes is planned to be integrated into UML.

The goal of the UML developers is to bring more stability to the object-oriented market. With respect to this, it should be mentioned that even the Fusion group at Hewlett Packard, which has a similar (process) approach, is considering the adoption of the UML notation.

UML is expected to be standardised by the OMG late this year. Rational Software Corporation, the company responsible for UML, claims that there will be tools available from several vendors by early 1997.

7.3 Tools to support design methods

Some vendors are announcing more 'automated' tools supporting the development of a system. These tools will ease the implementation part of the development process. The vendors claim that if the developer changes the source code generated by the tool, the tool will be able to adapt the system model from the source code.

Other vendors offer a linguistic knowledge-based development tool, where natural language utterances of the end users or customers are analysed and stored in a knowledge base. Requirements documents, user manuals and even program source code are claimed to be produced by so-called generators which exploit the knowledge base.

7.4 Software architectures

Some Architecture Description Languages have been developed during the last years. Although there have been some projects using ADLs and tools to support their formalisms, tools for industrial use cannot be found at the moment. However, efforts are made to develop such tools in the near future.

7.5 Design Patterns and Object-Oriented Design

As mentioned before, design patterns themselves are not meant to be executed by computers, i.e. by tools aiding the implementation of a system. Their primary benefit lies in guiding the developer and improve communication between them. Some of the developers of object-oriented design methods show interest in patterns. It is expected that the benefits of patterns will be used to improve object-oriented design methods. Although, when using design patterns within object-oriented design methods, the developer might no longer be following much of the method beyond its notation. But this, on the other hand, could be a benefit for the re-engineering of legacy systems (s. Inapplicability chapter).

8. Other comments

Although many system modelling languages of the design method framework are meant to support the development of new systems, it should be possible to adapt the step-by-step processes so that they support the RENAISSANCE method. Interesting enough, there seems to be no experience in using existing system modelling languages for the re-engineering of large systems, which are on the other hand widely used for the development of new systems.

Some methods and tools have only been made by companies for developing their own and their customer's systems. It is therefore hard to judge on their quality and, especially, on the maturity.

All in all, the support for software engineering may be rather poor. Some design methods do not consider reuse of components, others do not provide modelling language constructs to support the development of large

systems. Many methods need to be revised regarding extensibility, modifiability and maintainability. Often a method supporting some of these requirements, is poor in supporting others.

Despite their common roots, the different system modelling design methods are not well integrated. Although the processes show some similarities because they are supporting more or less the same phase of the software lifecycle, the notational concepts differ noticeably between methods. This means in particular that a model written in one modelling language cannot be altered using another.

Summary

System modelling methods to be used with the re-engineering of large systems cannot be found 'off the shelf', but have to be adapted to suit the specific needs of re-engineering.

Tools claiming to support a particular design method have to be thoroughly reviewed, to verify how far this support goes and which programming languages are available with this tool.