Handling Complexity in Software Design

Anders Törne

Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
ato@ida.liu.se

Introduction

Contemporary and future engineers face increasing demands from society and laymen on design quality. A durable quality entails many things - safeness, reliability, functionablity , operational and extensional simplicity, low production and maintenance costs, certified properties, environmental acceptability, and an aesthetic appearance are but some of the qualities asked for. Some of these demands may result in contradictory requirements. A part of the engineering task is actually to balance these requirements, so that satisfactory levels of quality are achieved for each one.

The number of design decisions increase with the number of requirements. This means that every extension of such requirements will result in extra design decisions and designs that will become more difficult to comprehend for the individual engineer and certainly for the layman. This incomprehensibility is "complexity".

Two lines of thought appear from this - "What is a complex design?" - and "How can the engineer be supported in his task when complexity increases?" In the introductory chapter we discuss some measures of complexity and how to apply this to design. A chapter follows on complexity in software systems and a presentation of how this complexity can be handled. Finally there is a chaper on a specific project adressing the issue of tools and methods for handling complexity in system design, where software is only a part, i.e., systems with embedded software.

Complexity in Design

Measures of Complexity

Spontaneous generation of complexity is observed as a natural phenomena, e.g., the generation of life, or the change from sunny and clear weather to thunderstorms within a few hours. A measure of complexity must relate to models of the phenomenon or object in question, since there is no other way to access it other than by observation[1]. A model is a view of an object for a specific observer and every observer has a individual intention with the model - to understand, to explain, to use or just to communicate.

Example 1: Two crossing lines on a white piece of paper are just that for an engineer, but for an art gallery visitor, recognizing the signature, they might be something more significant. The art-lover will probably have a much more complex model of the object than the engineer (no offence intended).

Assume that the observer model, the observer and the object (or phenomena) correspond to an imagined computer program, a computer user and an image on the screen, respectively (see Figure 2). The minimal imagined universal program (in bits) known to generate the image is the *algorithmic complexity*[2] of the object. This measure of complexity is equal for different observers and corresponds to a measure of the information contents in the object.

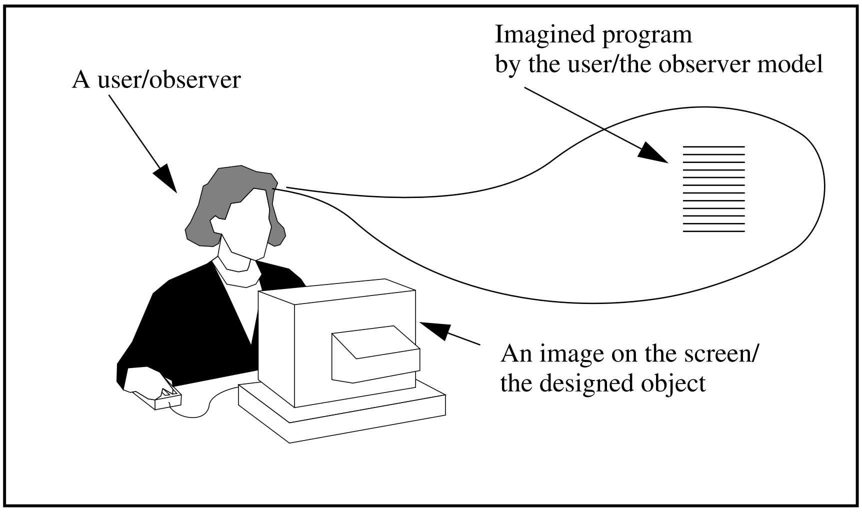

Figure 1 Relating designs, observers and the observers model to a screen image, a computer user and the user imagined generating program.

On the other hand many different imagined programs can generate the same image. A simple extension is to consider the complexity of each observer model as the length of the corresponding imagined program. However, this is counterintuitive, since a program making a simple output of a stored bit pattern would correspond to a rather complex model (at least for high resolution color screens). Such a bit pattern output program would not be considered a complex model by the observer. A more intuitive measure would be the *logical depth*[3] or the computation processing time for the imagined program. The observer will recognize complex objects as objects that take a long time to compute; like the artistic process, where a piece of art is the result of trial and error and a maturing process.

This does not mean that the creator (artist) program is identical to the observer model; rather, logical depth measures the complexity of the imagined generation process in the mind of the observer. The observer wants to understand the image and thereby imagine a program as close to the "artists" as possible - somewhat like running the computation backwards.

Both measures are usable, particularly in the context of engineering, where comprehensibility relates both to the designer(s) abilities to generate the design and the later "observer" trying to figure out why the design looks the way it does.

Design

To support the continued presentation, consider the following simplified design process. A set of design *intentions* is the origin of a new design - mostly broad and unclear, external and internal influences on/in the organization where the design is going to take place. A first and difficult step is the *system requirement analysis* which generates a *system requirement specification*. The *initial design* step arrives at a *system design*. The system design normally tries to partition the design into *design objects*, so that the design of each object is as *decoupled* as possible from the others. For each object there is an *object requirement specification*.

Each object is modelled using an adequate notation and the *object design* is to adjust the model so that requirements are satisfied. The resulting model is a *design specification* for the object. A parametrization of a model denotes a set of models where each parameter set corresponds to a model instance. From the design specification more *detailed designs* can be generated. In the following we will distinguish between the *early design* steps - the initial design - and the *actual design* - the object design and the detailed object design steps.

Information Contents and Design Depth

Relating to the discussion about algorithmic complexity and logical depth above, we define two important quantities for complex design - *information contents* and *design depth*. Information contents is the sum of the *initial information* and the *engineering knowledge* necessary to run the design process until a resulting design has been achieved. Initial information is typically the design intentions and external requirements. In the following the system requirement specification is also considered as initial information. Engineering knowledge is typically resident in *synthesis and analysis tools*, *design methodologies* and in the *engineers* themselves. Engineering knowledge is evolving more slowly and can be considered as static in comparison with initial information, which changes from one design to the next.

Typical for a design with low information contents is that it contains few, decoupled design objects, with simple models and parametrizations. This results in simple design specifications. The information increases, for example, if there are many design objects that are not easily decoupled or if the models have threshold effects and/or non-linearities.

Design depth is a measure of the amount of time the design process takes. Typical for a shallow design is that there is a straightforward method of adjusting the object models to fit the requirements and to generate the detailed object designs from the object specifications. A shallow design process is easier to automatize. For deep design the process is more evolutionary, i.e., design instances have to be generated in order to be analysed for conformance with the requirements. Sometimes the process to generate new designs is so complicated and expensive that the organization avoids it and instead makes incremental changes to older designs. The result is that old design architectures and solutions are conserved. This is, however, not the same as *reusing* parts of older designs, when it is clear that they satisfy the requirements.

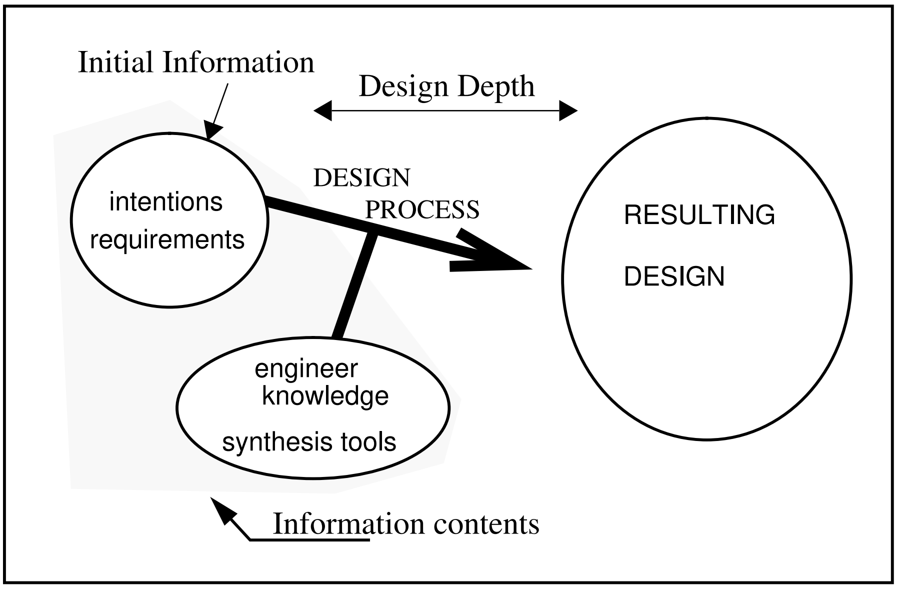

Figure 2 A schematic conceptual illustration on information contents and design depth

Design Reliability

Design reliability is a measure of the agreement between the design intentions and the resulting design. This agreement is also extended over the life time of the designed product/system, which means that hidden errors in the design are also considered as failures of the design to agree with the intentions.

A design can be made more reliable by refining the initial information (including system requirement specifications) or the "static", engineering knowledge. Errors in initial information can be avoided by having transparent specification languages and analysis tools for consistency and property checks. Faults in engineering knowledge can be avoided by education and experience, and by the use of tools known to be reliable, i.e., known not to introduce errors in the final design.

Even if initial information and engineering knowledge is reliable, errors can still be introduced during the design process - there will then be a mismatch between the resulting design and the correct design represented by the early specifications.

The design process has a macro and a micro perspective, where the macro perspective is represented by project plans and management, and the micro perspective is related to design teams and/or individual design tasks. To avoid the introduction of errors it is important that the macro scale methodology reflects a step-wise refinement of the design objects, where the eventual long range interactions with other design objects are kept under control.

Example 2: Take the issue of physical space in the engine compartment of a vehicle - changes in the electrical control system might lead to changes in the physical size or the distribution of the control circuitry. This might have effects on the geometrical requirements of the steering rods or the suspension - physical size and weight are typical global requirements that cause interaction between design objects.

Other ways to avoid errors in the design process are to introduce redundancy in the process and the design, to use synthesis tools on the microscale with built in knowledge about interaction effects and the use of project management tools to support the macro process.

Design Robustness

Another important factor is design robustness - or the design sensitivity to changes in requirements. Let $D = \{d_1, d_2, \ldots\}$ be the set of possible designs and $\mu(d_i, d_j)$ a subjective measure of the distance between two possible designs. Consider a small perturbation in the initial information or a mutation in the "static" design knowledge, Δ. Let d_i denote a design and $d_{i\Delta}$ denote the resulting design after the perturbation/mutation.

The design is robust if $\mu(d_i, d_{i\Delta})$ is small when Δ corresponds to a subjectively small perturbation/mutation. A design process is robust if it generates robust designs.

Robustness is a requirement for the reuse of old designs, i.e., design by modification of old designs, and for iterative design (generate/modify/test design). On the other hand a design process which is robust for larger Δ will result in the use of obsolete/unfit design concepts and failure to recognize the necessity to redesign from scratch.

Structural design - architecting[4] - is a comparably unrobust process, while parametric design is comparably robust.

Characteristics of Complex Systems

A complex system can only be comprehended by humans through *decomposition* and *abstraction*. Decomposition means that the system is decomposed into parts and that the function/behaviour of the whole can be understood by understanding the parts and their relations. Abstraction can be used in two dimensions - as different views of the design where un-needed information for understanding the specific features is suppressed (*design views* - like an electric circuit diagram only showing circuit connections) and/or as simplification where details in the whole design are suppressed (*focused* - like a mechanical drawing of a component, where the surrounding objects are only sketched).

These mechanisms are used in the design process to make communication between engineers and reliable design possible. Decomposition is difficult, however, since the long range design dependencies will remain, particularly for complex systems. Decomposition and abstraction are also connected. Depending on view - geometrical, electrical function, data flow - different decompositions gives best decoupling of the design.

A third property of a complex design is self-similarity. The system model consists of "similar" components used in different combinations, e.g., electrical circuits or programming language statements. For human made artefacts this is caused by striving to make the design comprehensible.

Furthermore a complex system is very difficult to design from scratch. Normally parts of old designs are modified and the experience of old designs is built into current engineering knowledge. The complexity of computers, cars etc. has evolved from simpler designs, accumulating experience and know-how.

To relate to the earlier discussion of complex design, a complex system is typically the result of high information contents to begin with and considerable design depth in the design process.

Software Design Complexity

Even if some of the functionality of modern complex products and systems could have been realized using means other than software, it is software technology which has enabled designs with extreme complexity. Due to the rapid progress in computer performance, evolution towards higher and higher complexity is fast increasing - the software crisis will therefore probably never end - the tools for software development of today are the solution to yesterdays crisis.

At the same time the user/operator wants the interaction with the system to be simple. In order to facilitate interaction a simplified model of the system must be presented. Normally this corresponds to familiar earlier systems - i.e., a complex car containing 20 microprocessors should be no different to drive than a 1966 Volkswagen. This places extra requirements on the design which therefore becomes more complicated. Consider, for example, modern word processing systems. They correspond to 5-10 MBytes of code which is about 10 times more than the corresponding systems had 5 years ago - functionality has been added - but still the interaction and functionality should be understood by computer novices only asking for an advanced typewriter.

Since the technology is developing so quickly it is difficult to “catch” the requirements. User expressed requirements often correspond to 3-5 year old technology. This results in requirement specifications which have been technology driven - the designer adds on functionality which is considered probably useful.

This applies mostly to pure software products. If the system only contains embedded software, the requirements are requirements on the whole system - a car, or a nuclear power plant. In these cases the software requirements can be derived, although it may still be a difficult task.

Often larger software systems typically consist of 1 - 10 million lines of code. Each line contains perhaps 3 - 10 decisions on the average. This amount of design decisions is significant and compares well with complex mechanical designs like building complexes, cars, ships or power plants.

There is also an extreme sensitivity to “bugs” or minor human errors in the design. The software engineering process is therefore unrobust compared to traditional engineering in mechanics or electromechanics. Minor human errors, like typing a semicolon instead of a dot on a mechanical drawing, are normally caught, since the design is interpreted by a human. Software is automatically compiled and if the compiler does not catch the error, it will be present in the executing system - an older version of Fortran was used for the Viking space probe and an erroneous dot instead of a comma caused disaster.

Furthermore, software offers ultimate flexibility, since no significant production step is needed to observe the behaviour of the design.[5] This gives the uninitiated an illusion of simplicity and strong demands are therefore put on software designers and programmers to make late changes to the design when requirements happen to change. This tends to make the design more complex since the result might be a mismatch between original architecture and added requirements - late changes are made at the cost of introducing “fixes” - which can be obscure.

Another characteristic is that long range or implicit interactions are common. Often this is caused by some shared resource - a CPU or a common database for different processes or a free variable[8] in sequential programs.

Abstraction and Decomposition

Abstraction and decomposition are general methods used to control complexity. Abstraction is the human ability to discriminate between significant and insignificant factors for the specific task at hand. This task may be a sub-task of the design or the analysis of a special view of the design. Abstraction means that the details of the insignificant factors are suppressed, either totally - e.g., dropping geometrical placement when designing the electronic function - or by using simplifications by making models - e.g., in the design of the computer communication protocol in a distributed multicomputer system by assuming simplified models of the actual communication which will take place.

Abstraction enables the engineers to describe, analyse and communicate their ideas about a specific design task. A simplified model of the factors which interact with the subsystem to be designed, will become a requirement for the engineers designing these other parts of the system. A well-defined abstraction can therefore be used as a barrier between a refined/detailed design and an external view (the abstraction) of the same design object

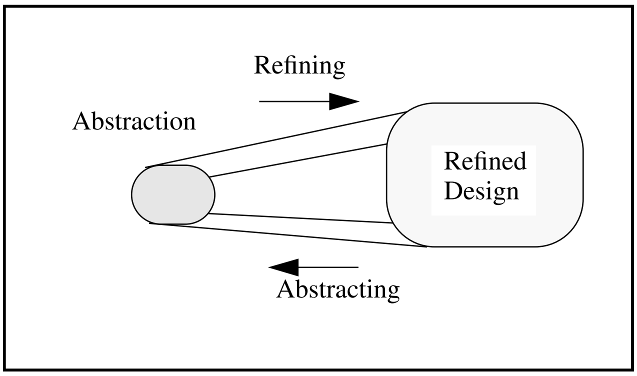

Figure 3 An illustartion of abstraction

This principle is well-established in software design to control complexity. Encapsulation is a concept which means that the actual way in which certain operations are performed on a data object is not accessible - neither can the actual data object be directly accessed. Encapsulation therefore is a realization of abstraction. Given a data object of which the detailed actual implementation is unknown, the external "user" of the object can only perform certain abstract operations for which the detailed implementation is unknown. The abstraction consists of these abstract data object and the associated abstract operations.

Example 3: Take the concept of a queue. Given a "queue" the external "user" can perform insertions, removals and queries on the "queue", without having to know whether it is implemented as a pointer structure or an array. Of course an empty "queue" can also be created, so the "user" has something to start with.

Relating to the discussion about information contents, the abstraction has less information than the implementation. The abstraction corresponds to many possible implementations (designs) and when one (or a true subset if the design uses other abstractions) of these is selected the information content increases.

This way the design process can be viewed as a step-wise refinement process, where for each step more information is put into the design. The added information is of course derived from the "static" knowledge (cf. above)

An orthogonal way of controlling complexity is by decomposition - partition the design and perform the design for each part. This presupposes that the other parts have a well-defined abstraction (or model) which can be used to model the external interaction with the part to be refined.

If the decomposition and abstraction mechanism are combined, an ideal way of controlling complexity is achieved. This assumes that later refining of a design does not interfere with the assumed abstraction of that part. However, as was described above, one of the properties of complex systems is that there are long range interactions or indirect influences between subparts. This influence can be ignored temporarily but has to be considered sooner or later.

Another way of stating the essence of the discussion above is that early decomposition in the design process puts a burden on the integration phase, where long range effects are noticed.

Another way is to say that the early specifications have to be as close as possible to the final result, i.e., the abstractions or early simplified models have to be as correct as possible. As will be presented in the following chapter, this is one of the motivating factors for the presented project.

The argumentation above applies well to software design, where the abstraction/decomposition duality comes under the labels abstraction /modularization.

Methodology and the Design Process

A methodology consists of a notation/language for communication, a description of the design process and different methods and tools for manipulating, analysing and synthesizing (intermediary) designs.[7]

The earliest methodologies (60ties & 70ties) in software design are the *structural design methodologies* where the idea is to focus the control flow and the operations that are to be performed. These methods were developed for algorithmic software, for example for calculations, and are based on procedural abstraction or what is commonly labelled subroutine calls.

Somewhat later (70ties & 80ties) methodologies for data modelling or *data driven design* were developed. They correspond to the kind of software where data objects and their relations are complicated, like in databases. For the “same” operation there are different implementations depending on the data object in question.

Object-oriented design (80ties & 90ties) [8] is an effort to merge these two views of system development. An object is an abstract entity with state and specific operations associated with it. Procedural abstraction is achieved within each object by letting an operation call other operations on the same object or other objects. At the same time the implementation of the operation is possible to be define locally - only for the class of objects in question. The implementation is hidden for the “user” of the object. Objects oriented design therefore both contains abstraction and decomposition by letting objects be aggregations of sub-objects.

Object oriented design comes with concepts like *inheritance* - a class of objects is a subset of another set and inherits data and operational definitions - if the class does not *shadow* what is being inherited.

Notation and language

Engineering needs a language to express and communicate designs and this is true for all sectors of engineering. One distinction to be made is between formal and informal languages. Informal languages are typically spoken/written language, maps, circuit diagrams etc., where the receiver or interpreter of the material is human. Informal languages are evolving all the time and may contain a possibility to express ambiguities and contradictions. If a machine like a computer is going to interpret a language to analyse or translate it to another form, it has to be formal. A formal language is precise, with precise syntax and semantics. Syntax is the form of the language or a description of the legal constructions in the language and semantics is what it means. A programming language has to be a formal language since it has to be compiled. A design or specification in a formal language can be automatically analysed for completeness or evaluation of properties.

A design methodology may contain formal and informal languages. Many software design methodologies have informal languages for the specification phases while formal languages are used in detailed design. During the last 10 years a development of formal languages for specification has taken place, since such offers obvious advantages. There are, however, also disadvantages. There are several efforts in industry aimed at evaluating formal languages in specification and one of these projects is presented below.

In software engineering there are several notations or languages which are common:

- Data-flow diagrams, which resemble circuit diagrams, where data flows along wires and operations are components with in- and out-connections.
- Data model diagrams, where data objects are broken down into primitive pieces and the relation between data objects and their parts are described by arrows.
- State transition diagrams, which is an annotated graph with nodes and transitions between nodes. This is used to represent how operations on data objects affect the state of the object.
- etc.

In traditional engineering there are others:

- Differential equations and their computerization - difference equations, which is the traditional way to represent behaviour.
- Surface & volume models, to represent geometrical form.
- Digital circuit diagrams, for electronics.
- etc.

Many of these languages have formal definitions but they are often used in an informal way in the design process, i.e., the property of being a formal language is not used. For example, differential equations are translated into a formal programming language to be simulated - the formally used model is the simulation program which is compiled end executed. However, there is more and more agreement on that the usage of these languages also should be formal, so that, for example, translation can be done automatically.

The design process

The design process has to balance the requirement for management control, progress measurement and predictability, with the need for creativity and innovation. An appealing process model for design - at least for management - was the waterfall model, where the design process consisted of a sequence of well defined design steps. Experience has shown that this model is probably only applicable to the macro level of the design close to management and project planning. Closer to design teams and individual engineers the process contains iterations - the Boehm spiral model[9] - where the design is refined by redoing earlier steps when needed, and increasing the information contents for each iteration. Prototyping is one manifestation of this way of doing design.

Another valuable insight is that there is probably no perfectly, rational design process[10], i.e., the actual design process in each case will deviate from the current favourite methodology. However, as a starting point and as a design quality guide it is very valuable.

Tools & Methods

The design of complex systems can be supported by computerized support tools and methods for analysis and synthesis. These tools and methods can be divided into four different types:

- project management and planning tools to support the methodology itself.
- support for design information management. Design information contains large, structured and interconnected data objects which has to be stored for analysis and retrieval. Version handling and configuration management are other important functions. Often this takes the form of a database where simple syntax and constraint checking can be performed.
- synthesis tools and methods. These tools support the generation of a refined design given an abstract specification. Typical synthesis tools are CAD-systems, where the synthesis consists of composing library components together to form the design. The tools may also be (semi)-automatic. Fully automatic synthesis would mean that the design is derived from the specification. The construction of automatic synthesis tools from specification to design (corre-

sponding to compilers for programs) is a challenge to the research community. For the automatic synthesis of complex systems it is possible that no better alternative than variants of a 'generate and test' strategy is possible.

- analysis tools and methods. These tools assume that the design or refined specification has been generated either manually or by (semi)-automatic methods. Analysis is to check whether the design meets the more abstract specification or to analyse aspects of the design which could not be taken care of in the synthesis process. Typical tools are timing analysis of generated programs, simulators and formal property checking.

Methodology for Complex Software Design?

For the software design process there is presently no well-established general "static" engineering knowledge. This means that only the simplest possible synthesis tools are possible, at least for early specification stages.

Secondly, the general analysis problem is undecidable, which means that no algorithm exists for the problem given a general language and a general property to analyse. Even for restricted languages and properties the existing algorithms scale very poorly for large designs - they often grow exponentially in time when the size of the design increases. This means that the engineer has to fall back on the last resort, i.e., to observe the behaviour of the implemented program and compare with the requirements - what is normally labelled testing.

There are many approaches to these problems, but we will not elaborate further on this. The important insight is that it is difficult to construct tools and languages for software design. In the following chapter this question is extended to systems where only a significant part is software - so called systems with embedded software.

The COHSY project

This chapter reports on a collaboration project aimed at the problem of handling some of the complexity issues in design. The title of the project is *Specification, Synthesis, Modelling and Verification of Complex, Heterogeneous Systems* and it involves the following partners: from LiTH, Linköping University - dept. of Mechanical Engineering (IKP), dept. of Electrical Engineering (ISY) and dept. of Computer and Information Science (IDA); from industry - Saab Scania AB/Saab Military Aircraft, Volvo Aero Corporation and VOAC Hydraulics AB.[11]

The main industrial concern in this project is the increased cost for system validation and correction during prototype testing. In particular this cost is significant for heterogeneous systems, where the design process involves parallel design efforts by engineers of different disciplines. This puts a heavy burden on system integration and final validation exactly as was discussed above. A strict application of a design methodology involving analysis and simulation tools early in the design will result in more reliable designs and design decision traceability.

Another motivation for the work is for the academic partners to found basic research in methods for systems engineering of complex systems on industrial problem settings and for the industry partners to investigate alternatives and possible future developments within the area in the perspective of their own design method efforts.

The work within this project is concerned with tools and methods for the development process for complex, heterogeneous systems. It targets modelling for simulation/verification, distributed simulation, formal representations of discrete and/or continuous models, database technology and computer based tools for specification, synthesis and (semi-)automatic verification. The basic idea is to develop and get experience from tools applied to realistic industrial examples of the design of complex heterogeneous systems. From this work an adequate methodology will be developed which integrates formal analysis and simulation in the early design process, thus enhancing the quality/reliability of the final designs before implementation/manufacturing.

A heterogeneous system is a system composed of, e.g., mechanical, electrical, and/or hydraulic components, in interaction with a software-based control system. The considered systems are typically vehicles, aircraft, large machinery etc. The software often performs controlling operations, even if a large part is purely supervisory in character. The software design is a significant part of the total design task.

The focus is on the system development phase, which in the terminology above corresponds to the early specification phases from requirement specification to complete design object specifications. In this phase the early, main design decisions are taken - the system is architectured. In our framework, these decisions are based on the analysis/simulation of abstract models of the design, depending on the function and situation of system usage. The design of the software is an important part, but is viewed in integration with the design of the hardware and the environment with which it interacts.

The project has one academic part and one industrial. The industrial work is directly connected to company specific efforts within the same domain. Only a brief presentation of the industrial effort will be made in this paper. The project has five academic subprojects:

- *Discrete and Continuous Distributed Simulation*, where the possibilities of using natural modularization to distribute simulations over a network of workstations and to use database technology in the pre and postprocessing phases are studied.
- *Verification of Discrete Dynamic Systems*, where polynomial representations for discrete description languages of system behaviour are investigated.
- *Hybrid Modelling and Verification*, where discrete and continuous variables are used in the model and symbolic model checking can be used. Presently simplified models of industrial heterogeneous subsystems are studied.
- *Human Factors in Engineering Design*, where work on the evaluation of computer based tools or aids for engineering design will be done. One significant problem is to evaluate to what extent the computer aids are truly helpful to the engineering task, and to what extent they can be improved.
- *Tool Integration for Complex, Heterogeneous Design*, involving object relational database technology for storage and domain specific mediators for integration. Specifically this also involves the intermediary representation for different kinds of models and the integration of analysis tools.

Industries in Cohsy

Modern military aircraft have about 50 different computer processors on board. They are distributed from navigational equipment over central processors to control the engine and weapon delivery. From the experiences with JAS 39 at Saab Military Aircraft it is clear that the next generation of aeroplanes will have drastically increased complexity just for the hardware and software. Therefore new techniques to control this complexity have to be developed.

Volvo Aero Corporation delivers jet engines to the Swedish Air Force. The area of interest is more focused to simulation and specifically hybrid simulations, where the engine is simulated and the developed control system is connected to the simulation. Simulation is today the most well-used method in analysing early specifications.

VOAC Hydraulics AB is the largest manufacturer of hydraulic components in Sweden. The use of software control in these products is increasing. VOAC are also interested in the use of simulation for early design analysis of hydraulics and control software.

Participation from Linköping University

The research programme of the Fluid Power group at IKP, lead by Prof. Jan-Ove Palmberg, covers central areas in the field of Fluid Power Systems and Technology. The main research can be divided into three areas: *System modelling and simulation*, *Design and control of fluid power components and systems* and *Methods for reduction of noise and vibrations in fluid power systems*. In the first area there are several on-going projects which represent a firm basis for the project work; efficient algorithms for simulation of stiff equations, real time simulation including hardware and man in the loop, parallel processing and optimization for system design.

The Division of Industrial Ergonomics at IKP (Prof. Martin Helander) taks major responsibility in the planned undergraduate education for Aerospace Engineers (Flygteknik). Courses on Man-Machine Interaction, Human-Computer Interaction, and Human Information Processing are given at the Division. This type of expertise forms the background for the proposed evaluation of engineering software. Prof. Martin Helander has been with the division since spring 1994.

The Automatic Control group lead by Prof. Lennart Ljung at ISY carries out research related to control and signal processing. This includes *System identification and signal processing*, *Non-linear modelling and control, Discrete systems computations*, and *Hybrid modelling.* It is primarily the work in the last two areas which relates to the project problem domain.

The Real-Time Systems laboratory (Dr. Anders Törne) at IDA is a newly formed laboratory which focuses on the design of real-time software systems for control and supervision. Specifically the research focuses on tools, method and architecture for real-time system design with applications in automation and embedded systems. Associated to the project work is Prof. Tore Risch from the laboratory for engineering databases.

The laboratory for knowledge representation in logic (Prof. Erik Sandewall) at IDA performs research on the representation of system behaviour using extended logics. The work which is connected to the project deals with the modelling of hybrid systems, where one part of the model is continuous and the other is discrete.

Design of complex, heterogeneous systems

The work in this project requires integration of specification, synthesis, modelling and analysis/verification techniques rooted in different disciplines; e.g. various engineering domains (hydraulic, mechanical, electronics, automatic control), and computer science (software development methods, formal methods, database technology). We refer to systems requiring such an integration of design methods from different fields as *heterogeneous* systems.

Complexity manifests itself in two ways:

- many design objects which are complicated and highly structured, i.e., large amounts of information
- abundant non-local functional/design dependencies between the design objects, i.e. changes to a design in one object might result in a necessary global revision of all or many other design objects

In the heterogeneous case, complexity is also enhanced by the proliferated formats for the representation of the design objects.

The complexity issue focuses attention on the scale-up properties of design tools. Simulation and analysis tools must have higher performance if the models are made at detailed level. If these tools cannot handle the total detailed model, abstractions (or simplified models) must be used, possibly in a hierarchy of abstractions. Complexity can in this way be controlled and modelling/analysis can be carried out for each level. The catch is that it is possible for the final design to be a sub-optimization, since there is detailed design information lacking in the abstractions. For example, network communication is modelled for assumed worst case utilization, which might be overestimated because the communication needs of the detailed designs are not known.

However, the following issues are identified as critical for the aims of the project and the industrial concern:

- performance in simulation and analysis
- adequate computer-based modelling, specification and synthesis tools for the engineer
- tool integration and interoperability
- support for the reuse of designs and models
- congruence between the mental process of the design engineer and the computer-based support tools

Heterogeneous System Design Methodology

A presentation is made below of two approaches to real-time software development: the Design Approach for Real Time Systems (DARTS), which is a development of Real Time Structured Analysis and Design (RTSAD) and the Synchron approach, which is based on the idea of synchronous sampling and cyclic execution. The Synchron approach and variants of RTSAD are presently investigated within the project framework.

DARTS

DARTS[7] is an extension of Real Time Structured Analysis and Design (RTSAD). The Yourdon method, investigated by Volvo Aero Corporation, is also a variant of RTSAD.

The steps in the DARTS design method are: System Specification Development (essentially RTSA, Real Time Structured Analysis), Task Structuring, Definition of Task Interfaces, Task Design (essentially SD, Structured Design). The last step is a direct application of Structured Design for a sequential program.

- System Specification Development

  The first thing to specify is a context diagram, where the system interface to the environment is described - input, output, events etc.

  In the second step the (software) system, shown in the context diagram as one data transformation, is structured into functions and the interfaces between the functions are defined as data flows or control flows. This decomposition is hierarchical.

  Specify the behavioural characteristics of each control transformation with State Transition Diagrams (or by corresponding means)

  Specify each data transformation in, for example, Structured English

  Develop the data dictionary which defines all data flows, event flows, and data stores

- Task Structuring

  Use the criteria for task structuring to merge functions into parallel tasks in a task architecture diagram. Typically parallel tasks can be allocated to distributed processors in the implementation stage. Tasks can be of state transition type (triggered by events), of periodic type (timer triggered) or of asynchronous type (triggered by demand)

- Definition of Task Interfaces

  Map data flow interfaces to either loosely or tightly coupled communication mechanisms. Map event (control) flows to event signals. Update the task architecture diagram with the task interfaces.

  Perform an eventual timing analysis - i.e. investigate the response times to external events

- Task Design

Structure each task into modules and design each module internals using, e.g., Structured Design

Synchron

The synchron technology[12] is based on synchronous data flow languages for data transformation specification (LUSTRE or Signal) and Esterel for control transformation specification. The languages have precise mathematical semantics which lends the design to formal proof and verification of both control and data flow. The specifications are executable for simulation of the software and controlled system. Code can be generated directly. Typically the use of these tools corresponds to the Task Design phase in the DARTS method.

These tools are complemented with ASA, a graphical tool for modelling and system specification. ASA is a graphical tool, which can be used to statically decompose the system into modules hierarchically, where dynamics is defined in the leaves by using extended state machines. Each module is a data transformation with control input for mode selection and can be further decomposed. The usage of ASA corresponds to the System Specification Development phase of the DARTS method above.

Saab Military Aircraft are currently investigating and testing the Synchron technology.

Relation to Complex System Design

The two presentations above give complementary views of the software development issue. DARTS is a generic *methodology,* while Synchron is a set of tools to use in a method. From the brief presentation above it can be concluded that the software development methods and tools have to be extended to take the design of complex, heterogeneous systems into perspective.

The extensions have to integrate early modelling and simulation/verification into the method and provide tools for such . At the bottom line the set of tools also have to be integrated into the methodology and the design information handling system. The last part is essential for the acceptance of any tool to be used in production.

System Design at Saab Military Aircraft

Saab MA is presently testing tools for system development - several are connected with Synchron technology. A framework for system development is used which separates the specification phase (cf. DARTS above) into three levels: project specification, system specification and detailed specification.

The outputs of the project specification are functional/non-functional requirements on the total system, and after partitioning into subsystems also the derived requirements on the subsystems. A subsystem is related to a coherent set of functions, like hydraulic power distribution. Sometimes particular requirements on the implementation or design solution are also given. A subsystem can be contracted for delivery by a subcontractor.

During system specification the functions of a (sub)system are further divided into suitable design objects and the function specifications are given in detailed form (preferably mathematical). The context diagram (cf. DARTS above) for the (sub)system is described and implementation technologies - software, electronics, hydraulics etc. are allocated to the design objects. For a software implementable design object, the detailed specification corresponds to the output of the system specification development phase in the DARTS method.

Tool Integration for Complex Heterogeneous Design

Framework

Complex designs means necessarily large amounts of information. In the project we limit this to design information for the considered class of heterogeneous systems, in particular specification/modelling tool output, like formal models for analysis and simulation models, functional and non-functional specifications, test case specifications, results (possibly intermediary) from analysis and simulation. We envision a common database as a design information repository.

An object oriented database will, e.g., support multiple access, hierarchical decomposition, object oriented system modelling, ad hoc querying over the design and a common data model for design information representation. Local user databases will hold data in update. The common data model is instantiated for a specific set of tools used.

The base technology for this is presently developed by Prof. Tore Risch, EDSLAB, IDA and his graduate students in a basic research project, AMOS. In the AMOS architecture the mediator concept provides a way to interface a plethora of different application tools - the so called domain mediators. The main issue to be addressed in the research is the common data model - generation and representation and the tool integration (the mediators) to the common data model. The mediators may be generated from a description of the representation used by the tool - a type of compiler-compiler technology.

The basic idea has similarities to the concept of a "program database" and the Masterscope facilities in the Interlisp-system from the 70-ties. In our case, we use an OO DB which affords larger technical possibilities.

Conclusion

The complexity of software design can be handled from several perspectives:

- increase design process reliability. The goal is to have the final design satisfying the design specification.
- increase the reliability of initial information. Many of the errors are introduced due to bad early design, which makes it necessary to iterate designs or to introduce fixes.
- increase the reliability of engineering knowledge, by eduaction and experience.

The paper reports on a project aimed primarily at the second perspective for a specific class of software systems, namely systems with embedded software. Even if it is too early to make any firm conclusions regarding this work, it is clear that a pure software engineering perspective is unsatisfactory. Software is designed in conjunction with hardware and mechanics, for example, and cannot be methodologically separated from the design of these subsystems. Even if methods for real time software development is applicable, they have to be complemented by including for example, hardware in the loop simulation.

Complexity in itself is not the signifying factor for this, but pushes the issue of scale-up of the methods and tools to be used to the forefront.

Acknowledgement

Financial support is gratefully acknowledged from NUTEK (Swedish National Board for Industrial and Technical Development) and CENIIT (Center for Industrial Information Technology, Linköping University).

In the presented project there are many participants, from Linköping University and industry. I would like to acknowledge the individual effort of each one who has been involved. In particular I want to mention the project leader, Leif Larsson, Saab Military Aircraft, who has been enthusiastic about the project from the beginning. My own role is to coordinate the academic work. The project will continue at least until 1996, and the results will be presented at workshops and by international publication.

Notes and References

1 An engineering perspective. I will not touch on the issues of intrinsic complexity of natural phenomena, physical laws, or Gods intentions with the world.

2 Also called Solomonoff-Kolmogorov-Chaitin complexity,
see Chaitin, G.J. Algorithmic Information Theory, Cambridge Univ. Press, 1987

3 Slight modification of the presentation by Bennet,
Bennet, C.H. “How to Define Complexity in Physics, and Why”, in Complexity, Entropy, and the Physics of Information, SFI Studies in the Sciences of Complexity, vol. VIII, Ed. W.H. Zurek, Addison-Wesley, 1990.

4 Eberhardt Rechtin, “Systems Architecting - Creating & Building Complex Systems”, Prentice Hall 1991.

5 Compiling and copying is the production step of software engineering.

6 A free variable is a “common” variable in a subprogram which is not a formal parameter, that may also be used by other subprograms .

7 H. Gomaa, “Software Design Methods for Concurrent and Real Time Systems”, Addison Wesley 1993.

8 Grady Booch, “Object Oriented Analysis and Design”, Benjamin/Cummings Publishing, 1994.

9 Boehm, B, “A Spiral Model of Software Development and Enhancement”, *IEEE Computer*, 21 (5), 61-72, May 1988

10 Parnas, D, , Clements, P, ”A Rational design Process: How and Why to Fake it”, *IEEE Trans. Software Eng.*, Vol-SE-12 (2), Feb 1986.

11 Financial support from Nutek under grant 8323-93-10266 is gratefully acknowledged.

12 A. Benveniste, G. Berry, “The Synchronous Approach to Reactive and Real Time Systems”,
Proc. IEEE, 79, (9) sept 1991.