Turk J Elec Engin, VOL.12, NO.2 2004, © TÜBİTAK

The 7 C's for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering

Mehmet AKŞİT
University of Twente, Department of Computer Science,
Postbox 217, 7500 AE, Enschede, The Netherlands
e-mail: M.Aksit@ewi.utwente.nl

Abstract

This article proposes the 7 C's for realizing quality-oriented software engineering practices. All the desired qualities of this approach are expressed in short by the term living software. The 7 C's are: Concern-oriented processes, Canonical models, Composable models, Certifiable models, Constructible models, Closure property of models and Controllable models. Each C is explained by the help of a set of definitions, a short overview of the background work and the problems that software engineers may experience in realizing the corresponding C. Further, throughout the article, a software development example is presented for illustrating the realization of the 7 C's. Finally, this article concludes by infomally justifying the necessity of the 7 C's.

1. An Example: Car Dealer Management System

Assume that we would like to adopt a process for buying a car. This process consists of the following steps:

- Define the characteristics of the car to be purchased.
- Select the car models that fulfill these characteristics.
- Select a model from the candidate car models.
- Make sure that the car can be financed.
- Purchase the car and complete all the necessary formalities.
- Maintain the car and make sure that the necessary formalities remain fulfilled.

Although various different refinements and iterations are possible, this process is likely to be agreed upon as a basic guideline by most customers and car dealers. There are several parameters, however, which influence this process considerably. Consider, for example, the following list:

- Personal preferences.
- Social context.
- Income.

- Purpose of use.
- Interest rate.
- Amount of tax to be paid.
- Cost of maintenance, etc.

There are a number of model and/or dealer specific features, which may also play an important role:

- Characteristics of the cars available in the market.
- Service quality offered by a car dealer.
- Proximity of a car dealer, etc.

This is not an exhaustive list of all parameters. However, it is likely that a group of experts may reach a consensus on a list of say, 10 parameters. The values of these parameters will continuously change over time. Therefore, any optimal decision at a given time may cease to be optimal soon after.

Assume that we would like to design a software system, which will support the car-dealers and their customers in purchasing and maintaining cars. Such a system must fulfill the following two basic requirements:

- Maximize the satisfaction of the customers in time.
- Maximize the profit of the car dealer in time.

This system must continuously monitor the customer's current situation, and whenever necessary, it must give the right advice in purchasing and/or maintaining the customer's car. Similarly, for a car dealer, the system must continuously demonstrate its benefit in supporting the dealer's business. These objectives can only be achieved by creating a *living car dealer management system*.

2. Problems with Current Software Systems

Although software systems are the fundamental assets of many businesses of today, software projects frequently fail in achieving their objectives [1]. As reported by the Standish Group [2], in 1995, U.S.A. government and businesses spent about 59,000,000,000 Dollars for budget overruns and 81,000,000,000 Dollars on canceled software projects. It was also stated that in the United States, only about one-sixth of all projects were completed on time and within budget, nearly one third of all projects were canceled outright, and well over half were considered "challenged." Of the challenged or cancelled projects, the average project was 190 percent over budget, 220 percent behind schedule, and contained only 60 percent of the originally specified features. Based on these survey results we can assume that:

- Most commercial software systems are very complex, and therefore hard to understand, and hard to modify.
- A considerable number of commercial software projects are cancelled, overdue and over budget.

We can therefore conclude that most commercial software systems are hardly *alive*. In the following sections, we will investigate the necessary conditions for making software more *living* than today.

There are at least 7 properties of software models we consider important for creating living software. These are Concern-oriented design processes, Canonical models, Composable models, Certifiable models, Constructible models, Closure property of models and Controllable models. These properties will be discussed in the following sections.

3. Concern-Oriented Design Process

3.1. Definitions

A software development process is a problem solving activity, which transfers a set of problems into a set of executable solutions. The important properties of a concern-oriented software development process can be summarized as follows:

Refinement levels: Software development processes consist of a set of refinement levels.

Starting point: The top-level is defined by the most abstract problem to be solved. In commercial projects, the problems at this level have to be derived from the business requirements.

Problem solving: Each level is defined by:

- A set of problem domain concerns and solution domain concerns. The term concern refers to a relevant issue to be considered.
- A problem solving process, which transforms the problem domain concerns into the solution domain concerns.
- A verification process, which checks the solution domain concerns against the problem domain concerns.

Retaining the quality of solutions: In general, during a multi-level refinement process, a set of incremental solution models are defined at different abstraction levels. These solutions may show different quality characteristics. It is important to retain the desired quality factors while applying refinement processes.

Problem-solution-problem chain: The solution domain concerns of a level become the problem domain concerns of its sub-level.

Stopping criteria: Each level has to be refined into a sub-level until the following two conditions are met:

- The problem domain concerns are solved satisfactorily.
- The solution domain concerns can be directly mapped onto processor architecture[1] .

Some levels can be standardized for commitment and interchange. Typical examples are analysis, design and implementation phases of software development methods [3] and distributed system standards[2] such as CORBA [4].

[1] Processor architecture can be implemented in hardware, firmware, as a virtual machine, etc.

[2] In distributed systems, the term *layers* is used instead of *levels*. In a concern-oriented process, levels may correspond to abstraction levels of a refinement process as well as to software system layers.

3.2. Motivations

The possible advantages of a concern-oriented software development process may be summarized as follows:

Reduced complexity: Complexity is reduced by dividing the software design problem into levels of simpler problems.

Goal-directedness: The design process progresses from abstract problem specifications to concrete solutions.

Explicit reasoning: At each level, the context, the problems and solutions associated with it are made explicit. This creates explicit reasoning about the design process and improves documentation.

Quality of solutions: Problem solving requires systematic application of the relevant knowledge. The quality of solutions is, therefore, directly related to the maturity of the available scientific knowledge. Since scientific approaches are reliable and are based on well-established principles, the resulting software solutions will also be of high quality.

Uniformity of approach: Since both functional and non-functional requirements are considered as design problems to be solved, at the level of solution models, every requirement is treated equally. For example, creating an adaptable solution requires application of specific techniques for this purpose.

Reduced costs through sharing levels: Certain levels can be standardized and shared among multiple systems. For example, most distributed applications require a common set of services such as naming and addressing, data serialization and transfer. Instead of repeatedly implementing these services each time an application is developed, it is more economic to provide them by the underlying system [4]. Obviously, this reduces the development costs enormously.

3.3. Software development methods

There are some differences among methods in the interpretation of the analysis, design and implementation levels. For example, Object Modeling Technique [5] suggests a detailed specification of operations and attributes at the analysis level, whereas in Unified Process [3], these specifications are deferred to the design level.

Most commercial software development methods do not consider the software development process as a problem solving activity [6]. For example, in Unified Process, software is derived from external functional specifications, which are mainly related to problem domain concerns. Software is designed by mapping problem domain concerns into software abstractions. Synbad is a synthesis-based software architecture design method, which adopts problem-solving principles [7,8]. Software patterns [9,10] can be used in mapping problems into solutions.

At the design level, the problem identification and solving processes are generally implemented by the use of design heuristics. At the implementation level, these processes are generally realized through interpretation and/or compilation [11].

3.4. The concerns in the car dealer management system example

The problem domain concerns of a car dealer management system can be specified as supporting the sale, taxing, registration, repair and maintenance processes, and managing the client and stock data.

To deal with the complexity of problems, concerns can be decomposed into sub-concerns. For example, the concern stock data management can be refined into a new problem: *how to minimize the costs through sharing the spare parts among dealers?*

Figure 1 shows a level in the process of designing the car dealer management system. Here, a problem solving process transforms the problem domain concerns into a set of solution domain concerns. The solution domain concerns are checked against the problems through a verification process. To solve the problem of how to share spare parts among multiple dealers, the following solution domain concerns are identified:

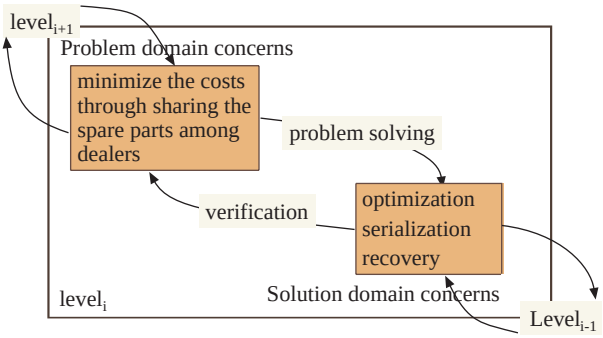

Figure 1. A level in the process of designing the car dealer management system.

- Optimization techniques for allocating spare parts to the distributed stocks. Optimization algorithms compute the best location for a part within a given set of costs [12].

- Serialization techniques for sharing distributed spare part data among dealers. Data may become inconsistent, if two or more processes simultaneously read and write or write and write on it. Serialization techniques are used to order the processes such that the processes execute as if they do not overlap in time [13].

- Recovery techniques for dealing with failures while transferring data between the distributed stocks. Data may become inconsistent if a process crashes while updating it. Recovery techniques either set back the initial value of the data before the crash or to repair the crashed process so that it can be completed safely [13].

Within the context of the design of the car dealer management system, in the following sections, the serialization and recovery techniques will be refined further.

3.5. Problems with current software development processes

In the following, we list some of the significant problems that we have identified by analyzing the support of current software development methods in fulfilling the properties of a concern-oriented process explained in this section. For each problem description, the corresponding property is shown between the parentheses.

Problem dependency of refinement levels (refinement levels): The required refinement levels for solving a given software development problem are defined by the characteristic of the problem being solved and therefore the levels cannot be precisely defined for an arbitrary software development problem. Most methods, however, define a predetermined process for dealing with arbitrary software design problems [3, 5].

Difficulty of identifying relevant requirements (starting point): Identifying a relevant set of requirements in a dynamically changing business context is a difficult problem. In general, a software system can be marketed as a product or as automation for supporting business processes. Determining the relevance of software products requires successful policies in market analysis, consumer behavior understanding and

forecasting, organization analysis, market and technology forecasting, portfolio definition, product development and management, and marketing [14]. Determining the relevance of automation requires a detailed cost-benefit analysis of the intended automated process.

Difficulty of dealing with multiple stakeholders (starting point): Requirements are generally formulated by multiple persons and therefore there may be discrepancy in the definition of requirements. It may be sometimes even impossible to meet all the requirements. Software development methods, therefore, must provide a means to deal with the requirements of multiple stakeholders.

Difficulty of aligning business and software development processes (all): In order to affectively react to changing business demands, the software development processes must be optimally aligned with the business processes. Current tools and methods, however, are generally optimized only for a certain phase in the overall process.

Difficulty of mapping requirements into problems (problem solving): Defining a systematic process for mapping requirements into problem definitions is not an easy task. This is because to identify the right set of problems, generally software engineers must have knowledge on the necessary solution domains.

Non-separable problems/solutions (problem solving): To find adequate solutions for complex problems, one has to map requirements as much as possible into an orthogonal set of problems so that each problem can be solved independently. Due to the complexity of the requirements, lack of knowledge, etc., software engineers may find it difficult to decompose requirements into orthogonal problems/solutions. In [15], this problem was termed as the decomposition problem.

Lack of problem solving techniques (problem solving): Most popular methods derive solutions from requirement specifications without considering requirements as problems to be solved. Deriving solutions directly from requirement specifications has two major drawbacks. First of all, solutions may become too sensitive to requirement specifications. Second and most importantly, even if all the relevant requirements are specified, it may still be difficult to identify the solutions. This is because the identified problem domain concerns can be very different from the required solution domain concerns and deriving solutions directly from problem domain concerns may not result in a solution or the solution may be of a poor quality. This problem has been discussed in detail in our related publication [6].

As an example, consider the problem and solution domain concerns of the car dealer management system shown in Figure 1. The solution domain concerns *optimization*, *serialization* and *recovery* cannot be directly derived from the problem domain concern *cost minimization through sharing* unless a problem solving process is adopted.

Dependency among refinement actions (problem solving): In a typical software development process, multiple problems must be solved together. Solving one problem may hinder solving another. A typical example is the early partitioning problem, which has been defined in our related publication [16]:

> *"The dilemma here is that if the software engineer does not identify subsystems before starting with object identification, then the project probably becomes unmanageable. On the other hand, if the software engineer identifies subsystems prior to object-identification, then the defined subsystem boundaries may not be optimal."*

Current software development methods generally advise a predefined order in dealing with multiple problems, and therefore, they implicitly prioritize certain problems.

Lack of awareness and/or lack of solutions (problem solving): Obviously, to find an adequate solution for a given problem, there must a known solution for the problem and one has to be knowledgeable about

the solution. In [16], for example, we have made the following statement:

> *"Quite often, underlying theories of large systems are not completely understood, and it is difficult to define reusable hierarchies for these types of systems. One may not expect software engineers to organize inheritance hierarchies any better than their understanding of the classifications within the theory itself."*

Dealing with large solution spaces (problem solving-retaining the quality of solutions): In general, a problem may be solved in many ways. However, the quality factors of solutions, such as adaptability, reusability, performance, etc. may be different from each other. Software engineers, therefore, have to deal with two important problems: dealing with large solution spaces, and selecting a solution that satisfies the desired quality requirements in an optimal way. Unfortunately, in current methods, both of these problems are not addressed very well [17].

Not properly defined refinement levels (problem-solution-problem chain): One difficulty of solving a complex problem is to deal with multiple levels of problems and solutions. A solution in a level becomes a problem at a lower level. In projects this is generally a source of confusion because project members may unconsciously refer to different levels but may assume that they are talking about the same level.

Difficulty of determining the level of detail (stopping criteria): One of the problems in a refinement process is to decide if the problem has solved satisfactorily and no further refinement is required.

4. Canonical Models

4.1. Definitions

A canonical model is a compact representation of the common features of a representative set of instances. A canonical solution domain model has the following properties[3]:

Knowledge-driven: A canonical solution model can be derived by analyzing the relevant and objective solution domain knowledge.

Generic solution abstraction: A canonical solution model defines the common properties of a set of solution instances. A canonical solution model is a generic abstraction with respect to its instances, since every instance complies with the definition of the model. An emerging instance, which cannot longer be expressed by a canonical solution model, is called a genericity-breaking instance[4] .

Succinct: Canonical solution models do not include redundant or irrelevant abstractions.

First-class property of concerns: The abstractions of the canonical solution model[5] correspond to the solution domain concerns.

4.2. Motivations

Defining canonical solution models may have the following advantages:

Reduction: Searching for a canonical solution domain model may help the designers to identify and specify the essential concerns and relations of a solution. This reduces complexity and makes the solution more manageable.

[3] Although the focus of this section is mainly on solution domain models, in a concern-oriented process, canonical problem domain models can be used for identifying a relevant set of problems.

[4] This is similar to the terms *symmetry* and *symmetry breaking,* which are defined within the context of patterns [61].

[5] In case of a canonical model, the terms *concerns* and *canonical abstractions* can be used interchangeably.

Comparison: Canonical modeling helps the designers to compare the alternative solutions.

Invariance-variance: Canonical modeling makes it easier to design adaptive and/or evolving solutions since a canonical model defines the invariant properties of the solution. The variant properties of a solution can be derived from its canonical model through specialization and/or instantiation.

Prediction: The properties of a solution may be predicted by inspecting the definition of its canonical model.

Reduced costs through deriving solutions from common models: Since canonical models are generic, they can be reused in deriving solutions across multiple applications. This results in reduced development costs.

A canonical modeling step can be considered as a part of refinement process. Identification of solution domain concerns is easier if canonical models are used. A disadvantage of adopting a canonical model may be the increased cost due to the extra work necessary for identifying and defining the model.

4.3. Background work

Our definition of canonical solution domain models is quite similar to Alexander's architecture patterns [18]. We think however that not all the software patterns [9, 10] can be classified as canonical solution models.

Object-oriented modeling techniques [19] and languages [20] introduce the class abstraction as a means of expressing the common properties of a set of instances. A canonical solution model is different from a class abstraction for two reasons. First, not every class necessarily represents a canonical solution. Second, some canonical solution models may be better defined as a set of classes or as other abstractions such as operations and constraints.

One of the main objectives of domain engineering techniques [21], product-line architectures [22, 23] and application frameworks [24] is to define a generic model that can be specialized for different needs. For example, in one of our publications [6], we defined software architecture as follows: "Architecture is a concept representing a set of abstractions and relations, and constraints among these abstractions". Here the term 'concept' means a canonical model. In other words, a canonical solution model is also the architecture of a solution.

In addition to defining canonical models for architectures, there is a need for canonical models for solving problems in various different areas, levels of abstractions and granularities. In [25], for example, canonical models are presented for expressing synchronization and real-time constraints in object-oriented programming languages. Obviously, defining canonical models remains a difficult problem.

4.4. Canonical models in the design of the car dealer management system

In section 3.4, serialization and recovery techniques were considered relevant for solving the problem of sharing spare parts among multiple dealers. Our next concern is to implement these techniques.

In the literature, atomic transactions are proposed for serializing concurrent executions and for recovering from failures [26]. There are, however, a considerable number of different atomic transaction implementation techniques. Each technique may perform better than the others depending on the context of execution and the size of the shared data. We have two alternatives: either to select a transaction technique and implement it directly or to define a canonical model for atomic transactions and derive an implementation from the canonical model. We have decided to define a canonical model for at least two reasons. First, it is expected that the context of the car dealer management system will vary considerably

and therefore to maintain the performance the implementation is likely to be adapted. Second, the system has to be designed for long term needs and therefore it must be prepared for evolution.

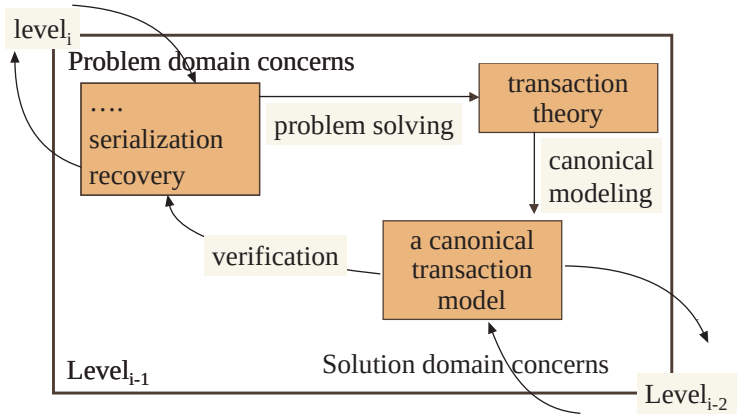

Figure 2. An example of mapping problems into canonical solutions.

As shown in Figure 2, serialization and recovery concerns are now considered as the problem domain concerns of the level i-1. Transactions are proposed as a solution to the problem of implementing serialization and recovery techniques. In contrast to Figure 1, here the solution domain model is derived from the theory domain, i.e. the transaction theory.

To identify a canonical model of atomic transactions, it is necessary to evaluate a considerable number of relevant atomic transaction techniques [8, 7]. By comparing 8 different publications, we have discovered a canonical model with the following components:

- *Transactional code*: represents the critical code in the application.
- *Transaction manager*: is responsible for initiating and controlling the overall transaction process.
- *Policy manager*: determines the implementation strategy to be taken.
- *Data manager*: coordinates interactions with the shared data.
- *Scheduler*: manages concurrent accesses to the shared data.
- *Recovery manager*: takes actions in case of a crash or error.

These six components are the abstractions and the concerns of the canonical solution model. We found necessary to continue with the refinement process and canonical modeling for the following reasons:

- The problem domain concerns are not yet solved satisfactorily. For example, the alternative implementation of the components and their adaptation strategy are yet to be determined.
- The solution domain concerns are not yet detailed enough for mapping onto existing language concepts and/or processor architecture.

In [8], it is shown how the components of the transaction model can be refined into sub-canonical models. In this way, the transaction system can be verified, adapted and reused at various levels of detail. In the following sections, we will continue with refining the transaction system model.

4.5. Problems in Defining and/or Utilizing Canonical Models

In the following, we summarize a number of problems that software engineers may experience while they are defining and/or utilizing canonical models. For each problem description, the corresponding canonical model property is shown between the parentheses.

Lack of support for canonical models (all): Most commercial software development methods are not suitable for defining canonical solution models, since they aim at a single software solution [3, 5]. Domain engineering techniques [21], product-line architectures [22, 23] and application frameworks [24] aim at developing family of software systems, but however, they generally do not emphasize the canonical property of models as defined in this article.

Lack of awareness (knowledge-driven): To identify a canonical model, designers have to gather sufficient number of instances in the domain [16]. If the designer has no access to the required information, then practically it will be impossible to identify a canonical model.

Lack of available knowledge (knowledge-driven): The quality of a canonical model is directly related to the number of known instances in the domain. In emerging disciplines, obviously, there may not be sufficient number of instances necessary for describing a (high-quality) canonical model.

Difficulty of extracting information (knowledge-driven): Knowledge may be available in many forms and it may not be always easy to extract the necessary information for defining a canonical model. This may be due to excessive number of information and/or due to the inappropriate representation of information.

Difficulty of knowing when genericity may break (generic solution abstraction): When the genericity of a canonical model breaks, preferably a new canonical model has to be introduced. If one can estimate when and how the genericity of a canonical model may break, then it could be possible to prepare a new canonical model. It is, however, difficult to reason about emerging instances.

Difficulty of determining succinctness (succinct): To be able to reason about the succinctness of a model, generally one has to reason about the necessity and sufficient conditions for the model. For complex models, this may not always easy to determine.

Lack of expressiveness (first-class property of concerns): The adopted modeling language must be rich enough to directly represent the features of the canonical model. Since canonical models can be very diverse, certain features of canonical models may be difficult to express directly in a general purpose modeling language.

5. Composable Models

5.1. Issues in designing composable models

5.1.1. Process perspective

There are two basic processes in composing solutions in software development:

Integration of solutions: Typically, an integration of solutions process is applied to integrate peer-to-peer level compositions. Example composition operators at the programming level are code-inline [11], operation call and/or aggregation [19, 20]. Complex systems can be integrated by using a sophisticated integrator such as CORBA [4].

Evolution of solutions: Here, typical composition operators at the programming level are aggregation, inheritance and/or delegation [19, 20, 27].

5.1.2. Separation of concerns perspective

The separation of concern principle may exhibit the following four characteristics:

Balancing quality values: The concerns of problems and solution models must be separated from each other to achieve the desired quality values, such as reduced complexity and increased reusability. Evolution of problems and demands for different quality values may enforce conflicting strategies for separating concerns. The ideal decision of which concerns must be separated from each other requires finding the right compromise for satisfying multiple (quality) requirements.

Composition of crosscutting concerns: A special case of integration of solutions is the composition of *crosscutting concerns*. Typical programming level composition operators are aspect weaving [28] and/or superimposition [29].

Probabilistic compositions: Due to uncertainties in the evolution of requirements, generally the problems and consequently the solutions cannot be precisely determined. This may lead to probabilistic problem specifications. It may be preferable to define and optimize the solutions and the composition of solutions with respect to the probabilistic problem definitions [30].

Fuzzy compositions: In a design process, if a problem can be solved in a number of alternative ways, and if it is not possible (or desired) to commit to a single alternative, then each alternative may be assigned to a fuzzy set [31]. A fuzzy set may express the degree of relevancy of a solution. By this way, a candidate solution may not be eliminated too early. Along with the refinement process, and/or due to the change of context, the values of fuzzy sets can be re-computed. This requires special composition operators that can reason about fuzzy sets. At a given refinement level, when a non-fuzzy solution is desired, defuzzification operation can be applied.

5.2. Motivations

A composable model may provide the following advantages:

Reduced complexity: A common approach to solving a complex problem is to decompose it into simpler sub-problems. Each sub-problem may be solved separately and then combined together to obtain a composite solution.

Increased adaptability and evolvability: Separating different concerns from each other and composing them using explicit composition operators increase adaptability and reusability.

Reduced costs through composing solutions from predefined ones: Commonly required (sub) solutions can be designed and stored and reused across different applications through composing them together.

We also would like to ephasize that canonical modeling supports composability for at least two reasons. First, since canonical models are succinct, they help in identifying the necessary set of composition operators. Second, the abstractions of a canonical model also determine the granularity of compositions.

5.3. Background work

Software development methods [19, 5] and programming languages [20] offer various composition mechanisms such as operation call, aggregation and inheritance. In addition, methods introduce abstractions such as collaboration and sequence diagrams, generalization/specialization, uses relations and associations. Only a few languages adopt a delegation mechanism; examples are Self [27], DARWIN/LAVA [32] and Composition Filters [33].

A number of researchers have been working on composition anomalies mainly within the context of synchronization inheritance [34]. Several publications [15, 25] generalized the definition of synchronization anomalies to other concerns such as real-time.

Various languages and systems aim at modeling crosscutting concerns [35] such as Adaptive Methods [36], HyperJ^{TM}[37], AspectJ [28] and Composition Filters [33, 38, 29].

Most composition operators at the design level are used to specify relations among the software artifacts. Design for composition has not been explored very well. This involves modeling design heuristics [39], and defining design operations for supporting integration, evolution and refinement processes [17, 8].

Probability theory has been extensively applied in modeling random processes. Generally, probabilistic models require specific reasoning techniques [40]. However, using probabilistic techniques within the context of software development methods has not been explored yet [30].

The application of fuzzy-logic techniques [31] is a new direction in software engineering. In other engineering disciplines there have been some applications of fuzzy sets and logic [41].

5.4. Composition concerns in the car dealer management system example

In the previous sections, we have refined the initial requirements for a car dealer management system into a canonical model for transactions. The following list shows the incremental solution models identified so far:

Car dealer management system requirements → cost reduction through sharing parts among stocks → serialization and recovery techniques → adopting transaction theory → defining a canonical model for transactions

Due to space limitations, we only show part of the refinement process.

To consider the composition for integration of solutions, we refer to the canonical components of the transaction model presented in section 4.4. The integration of most components here can be realized by operation calls. The integration of the transaction code and other components, however, needs some consideration. If the transaction code in the application program is expressed by keywords such as begin-end transaction, a preprocessor or interpreter can be used to integrate the application and the transaction system. This will result in a layered implementation of architecture.

We will now discuss the integration of solutions and the evolution of solutions.

Assume that two new requirements are provided. First, it is now estimated that the number of competing transactions is not expected to be too high. Second, as a new feature, all the incoming and outgoing calls on the modules are required to be monitored as well.

The first requirement may result in refining two components in the canonical model: scheduler and recovery manager. For example, optimistic scheduling and recovery techniques [26] give better results if the number of competing transactions is low. This new requirement should not cause any problem because in the canonical model, both scheduler and recovery manager have been identified as separate components. Instantiation and/or inheritance techniques can be used to specialize the transaction model.

As shown in Figure 3, the implementation of the monitor, however, can be problematic since every transaction component has to be extended to register the incoming and outgoing calls. Monitoring here is a *crosscutting concern*. If the adopted language cannot express crosscutting concerns, monitoring will increase the complexity of software and will possibly result in a composition anomaly.

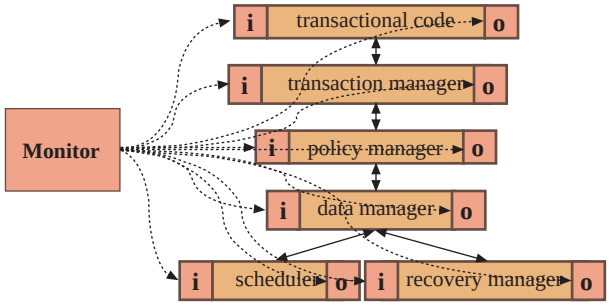

Figure 3. Monitoring the components of the transaction system.

Figure 4 shows the effect of the new requirements on the transaction model. The original requirements were serialization and recovery, which were solved by the application of the transaction theory and resulted in the canonical transaction model. A new requirement gives a new context to the transaction system, which is an estimation that less competition among the transactions will occur. This can be dealt with by a sub-theory in the transaction theory, namely the optimistic scheduling and recovery theory. This results in a refinement of the canonical model. Applying a simple monitoring technique such as logging can solve the monitoring requirement. The composition of the monitor, however, requires a special composition operator. This is because monitoring is a crosscutting concern. The shaded composition operator in the figure denotes this.

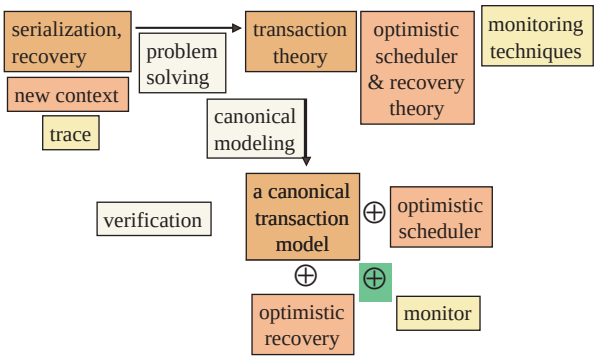

Figure 4. The affect of the new requirements.

5.5. Problems in designing composable models

In the following, we list some of the significant problems that software engineers may experience in designing composable models. For each problem description, the corresponding composable model property is shown between the parentheses.

Functional (in)compatibility (integration of solutions): Solution models cannot be integrated unless the following three criteria are satisfied [42]:

- *Relevance*: This is the most basic requirement. Relevance of sub solutions means that all sub solution models are relevant for solving the total problem. For example, according to the transaction model defined in section 4.4, the components Transactional code, Transaction manager, Policy manager, Data manager, Scheduler and Recovery manager are relevant for designing a transaction system.

- *Semantic (in)compatibility*: All sub solution models must be relevant and semantically consistent. Having relevant sub solutions does not always mean that the solutions are semantically compatible.

For example, in a transaction system if the scheduler component implements an optimistic scheduling algorithm whereas the recovery component implements a strict recovery algorithm, then these sub solutions are likely to be semantically incompatible.

- *Synchronous*: Time-dependent behavior of all sub solutions must be synchronous with each other.

Procedural (in)compatibility: If sub solutions are functionally compatible, but not procedurally, cooperation can only be obtained by introducing a suitable adaptation module, which translates between the interaction procedures used by the different solutions [42]. The translation mechanism can be implemented as a wrapper around sub solutions, as an abstraction for converting operation names and/or attributes, or as an interpreter/compiler for translating calls between sub solutions.

Difficulty of defining a common composition operator (integration & evolution of solutions): Certain solutions may demand dedicated composition operators. On the other hand, the composition operators of the adopted language must be general enough to compose solutions from multiple domains. This problem is similar to the arbitrary composition problem, which has been defined in our related publication [15]:

> *"Arbitrary composition is the difficulty in composing components if some or all aspects of components are generated from specifications and if the specifications cannot be composed by using the composition mechanism of the component model. This problem can be experienced mainly in designing application generators and constraint systems".*

Composition anomalies (evolution of solutions): If a relevant and correct composition cannot be expressed, for instance, due to the lack of a composition operator and/or non-separable concerns of the solution models, it is called a composition anomaly. Generally, an anomaly requires redefinition of the previously defined solutions, although this should be unnecessary. Composition anomalies may manifest in various forms. For example, in [15] we have discussed five example cases of composition anomalies:

- *Composition versus real-time specifications*: is the difficulty of reusing or extending real-time specifications of components. This problem can be experienced in designing components with real-time behavior.
- *Composition versus synchronization*: is the difficulty of reusing or extending synchronization code of components. This problem can be experienced in designing concurrent components with explicit synchronization.
- *Multiple views*: is the difficulty of adapting the interface of a component based on its context.
- *Sharing behavior with state*: is the difficulty of sharing a common behavior among components if the behavior is affected by a common state.
- *Lack of support for dynamic composition*: is the difficulty of adapting composition structures (such as inheritance or delegation) at run-time.

Tyranny of dominant decomposition (balancing quality values): Most modeling techniques favor a certain decomposition scheme. This may cause two problems: First, due to the predefined decomposition scheme, certain quality factors may be over emphasized. Second, in case of changing business context, it may be difficult to (re)organize the decomposition to fulfill the new requirements [43].

Implicit quality values (balancing quality values): Current software development techniques do not provide explicit means to compare the quality factors of alternative decompositions and therefore they are not suitable in leveraging composition alternatives with respect to the obtained quality values [17].

Lack of support for crosscutting concerns (composition of crosscutting concerns): Languages like Java and C++ and current software development methods do not provide means for modeling crosscutting concerns.

Lack of support for probabilistic concerns (probabilistic compositions): In general requirement specifications have a probabilistic character. Current requirement analysis techniques, software artifacts and composition operators do not provide means to model probabilistic requirements.

Lack of support for fuzzy concerns (fuzzy-logic compositions): During the analysis and design phases, it is generally difficult to define artifacts with 100 percent certainty. Fuzzy artifacts and composition operators can provide means to model uncertainties. However, current software development methods and notations do not provide means to express fuzzy artifacts and composition operators.

6. Certifiable Models

6.1. Definitions

Certification is the process of confirming that a system or component complies with its specified requirements and acceptable for operational use [44].

Verification is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [44].

Certification includes concepts such as certification authority, one or more verification activities, documentation of certificates and certification methods. Certification authority is an impartial organization possessing the necessary competence to operate a certification activity. Verification is a necessary sub-process for certification. In the following section we will focus on the verification component of certification.

As shown in Figure 5, a verification process consists of a number of sub-processes. On the left hand side of Figure 5, a problem solving process is depicted where a problem is transformed into a solution. This represents a refinement process, as explained in section 3.

Quality requirements are part of a problem specification and they specify the composite of the required quality values of a solution. By adopting a certain modeling technique, quality requirements are expressed using problem quality-feature models. Different problem quality-feature models may be used if necessary. The types of quality requirements might include usability, quantitative performance, functional behavior, etc.

Only in very simple cases, the necessary quality attributes of a solution can be directly obtained from the description of a solution. Generally, by using a certain modeling technique, a solution is mapped onto a solution quality-feature model, which enables complex measurement and certification processes[6] . As shown in Figure 5, there may be several quality-feature models for the same solution. Obviously, problem quality-feature and solution quality-feature models must be comparable with each other.

Both problem and solution quality feature models are expressed within a value system. A value system is a system of numbers, symbols and/or operators with well-defined (measurement) rules [45]. As indicated in the figure, depending on the variety of quality requirements, there may be more than one value system

[6]In the literature *solution model* is generally referred to as *system* and *quality-feature model* as *system model*. Since a solution can be specified at various abstraction levels and is itself a model of a solution, we prefer to use the terms *solution model* and *quality-feature model*.

for a single solution model. Through a measurement process, the attributes of a solution quality-feature model are mapped to one or more quality values by which numbers or symbols are assigned. Generally, measurement processes are tailored with respect to the characteristics of the desired quality values.

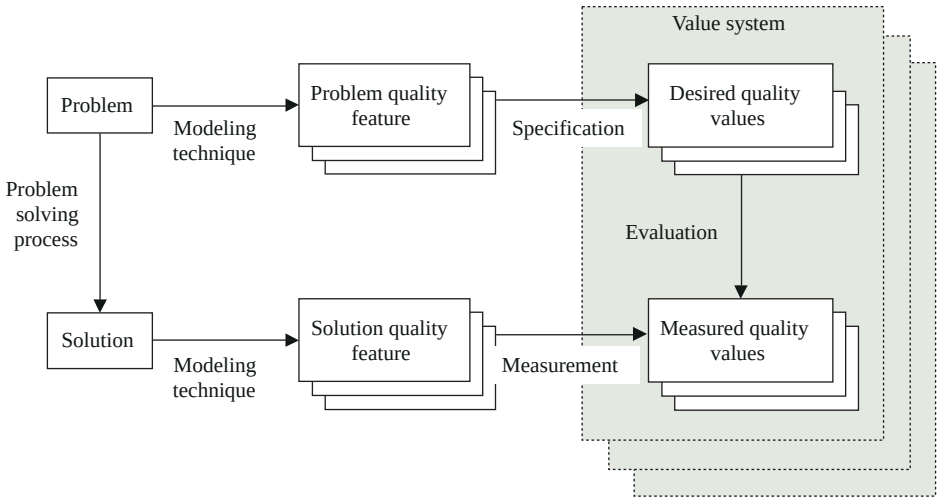

Figure 5. The concepts of a canonical verification process.

An evaluation process compares the desired quality values with the quality values of a solution within the context of a value system.

6.2. Kinds of verification activities

There are a number of issues to be considered within the context of a verification activity:

Verification of solution models in levels: Generally, a complex problem has to be solved at a number of levels. This requires incremental verification of solutions at different abstraction levels.

Verification of canonical models: In case of canonical modeling, every solution instance must fulfill the invariant properties of its canonical model. This requires verification of the canonical model and verification of the instances against the model. A number of certified canonical solution models might already exist in the project repository. These models can be directly used in the design and verification processes, if necessary.

Verification of integration of solutions: In case of integration of solutions, a verification activity must be carried out at the levels of sub-solutions and composite solution.

Verification of evolution of solutions: Both the initial and incremental solutions must be considered for verification. The invariant properties of a solution must be preserved after each evolution step.

6.3. Motivations

Certification may bring the following advantages:

Critical systems: There are a number of critical systems where failures in fulfilling certain quality requirements can have catastrophic results.

Marketing perspective: In the market, a certified product is likely to be more preferable to uncertified ones.

Verification of refinement: As shown in section 3.1, verification is one of the main activities in a problem solving process.

Concise specification of the design process: The quality requirements express the ultimate goals of a software development process. The progress of a design can be better monitored at the quality level than at the detailed implementation level.

Quality control: A fundamental requirement for software quality control is to determine and verify the actual quality values of the artifacts being designed.

Justification of the adopted techniques: Many tools and methods aim at enhancing quality features of software products. Verification can help in evaluating these tools and methods to determine whether they fulfill the promises or not.

In case of canonical modeling, solution quality-feature models can be directly derived from the canonical models. Note also that the composability property of a model is also a quality feature of that model.

A disadvantage of certifying models may be the increased cost due to the extra work necessary for certification.

6.4. Background work

The international standardization organization ISO has published the product quality model, which defines a set of hierarchically organized quality features [46]. The Capability Maturity Model (CMM) was proposed by the Software Engineering Institute as a process quality model to assist the Department of Defense in assessing the quality of its contractors [47].

There is a general consensus that a very broad spectrum of approaches to quality verification is needed [48]. Based on the concepts of verification shown in Figure 5, we will now refer to a wide range of verification techniques.

There are examples of analytic, simulation and testing based techniques for verifying the quantitative performance of systems. In all these techniques, the quality requirements can be expressed using expressions such as "the system must respond within a specified time threshold", "the average response time must not exceed a given value", "the availability of a system must be greater than a certain fraction", etc. The desired quality values are the fulfilment of the conditions. The measured quality values can be expressed for example as numbers or distributions. What makes these performance verification techniques different are the ways by which the solution quality-feature models are constructed and evaluated. For example, analytic techniques could adopt Markov chains or queuing networks [49], simulation based techniques adopt behavioral and statistical simulators [50] and testing-based techniques adopt measurement models [51].

We will now discuss three examples of behavioral verification techniques [52, 48]: model checking, theorem proving and dynamic testing. Example problem quality-feature models are temporal logic and predicate calculus formulas. The desired quality values are the satisfaction of expressions in these techniques. The major difference is in how solution quality-feature models are defined and evaluated.

In model checking [52], the quality-feature model of a solution is a finite-state model and checking is performed as a state-space search. The challenge is in devising algorithms and data structures that are suitable for handling large state spaces [52]. The advantage of model checking is that it is completely automatic. The disadvantage is the state explosion problem.

In theorem proving, the solution quality-feature model is expressed as formulas in some formal logic [52]. Theorem proving is the process of finding a proof of a property from the axioms of the system, for example through predicate transformations. In contrast to model checking, theorem proving can deal with infinite state spaces. Depending on the characteristics of the system, sometimes user interaction may be

necessary for providing additional information. There are recent attempts in combining model checking and theorem proving techniques together [52]. In both model checking and theorem proving, the evaluation of the fulfilment of requirements is trivial since the measurement process always results in true or false. However, the system can provide, for example counter examples, in case the verification result is "false".

In dynamic testing [48], programs are executed and the results are examined with respect to the specifications. To increase the efficiency and effectiveness of the testing process, a dedicated testing suite can be generated from the desired properties and the characteristics of the system being tested. Program output data can be saved and interpreted. A good test suite includes expected test outcomes against which the actual outcomes can be compared automatically.

Usability evaluation method [53] is by nature quite different from the previous verification techniques. Usability evaluation can be defined as the act of measuring the usability quality features of a system with respect to a set of users and user tasks [53]. Here, the concept problem quality-feature model may serve two purposes: to gain insight into the behavior of a system and its users in actual usage situations to improve usability, or to validate that usability has been improved. In the first case, the measurement data is used for supporting the design process and in the latter case it may be used to determine the quality features such as learnability, efficiency, memorability, errors, and satisfaction. The desired quality value in the first case is to gather reliable usage information about the system being considered and in the second case, to maximize the usability features. Forming a solution quality-feature model generally includes collecting a wide range of usage data in various formats. These data are then correlated, interpreted and expressed in a set of usability measures. The evaluation of the data is generally carried out by experts.

Recently, a number of new techniques have been introduced for verification and design of some important but neglected quality features such as adaptability [17], relevance [31] and cost of design policy [30].

In the Design Algebra based approach, solution quality-feature models are defined by mapping canonical solution models onto a canonical adaptability model, thereby creating a space of all possible adaptable models [17, 8]. Based on the preferences of the designer, a relevant set of alternatives is ordered and mapped to a value domain. The designer selects the alternative that fulfils the adaptability and other quality criteria.

The fuzzy-set based approach is used to determine the relative relevance of alternative solutions [31]. Here, the solution quality-feature model is represented as a fuzzy system. Each alternative has a relevance attribute, which is defined as a fuzzy-set. During the refinement process, when more information is available, the relevance attributes are re-calculated. Measurement is a fuzzy-inference process, which computes a set of fuzzy quality values. These values are compared with the desired fuzzy-conditions given by the designer. For this purpose, approximate reasoning or defuzzification can be used, if necessary.

The quality requirement can be quite diverse, for example, to minimize the cost of software development in case of evolving user requirements [30]. Here, the requirements are defined using probabilistic change cases. Using this information, the solution quality-feature model is defined as a Markov decision process. By using dynamic programming techniques, the optimal software development policy is determined and its cost is calculated. The designer can then compare the cost value with the desired one and decide on the policy to be adopted.

An important concern is to consider the quality factors during the design process rather than verifying the quality values of software delivery. Design for quality or quality-oriented software engineering [54] is a new research area and aims at controlling the design process based on the required quality values. In fact, this article proposes the 7 C's for this purpose.

6.5. Verification of the transaction subsystem

We will discuss the verification of the transaction system from the following three perspectives:

Verification of the refinement process: This requires a proof that the resulting solution is indeed a solution of the specified problem. During the refinement and canonical modeling phases of the transaction system example, the following two assumptions were made:

- Cost saving can be achieved through sharing spare parts among multiple dealers. This can be realized using optimization techniques.
- Sharing may cause inconsistencies. Transaction concurrency control techniques can be adopted for this purpose, because transactions provide serializability and recoverability.

The first assumption can be verified by analyzing the optimization algorithms published in the literature [12]. Concerning the second assumption, in the computer science literature, many publications have addressed the problems caused by concurrent accesses to shared data. As a solution to these problems, serialization and/or recovery techniques have been proposed. Within the context of atomic transaction techniques, numerous publications have proved the serializability and recovery characteristics of various transaction algorithms [8]. These publications can be used for at least three purposes. First, they can be utilized for defining a canonical transaction model. This was discussed in section 4.4. Second, they provide valuable information about the correctness of transaction algorithms. Third, the pre and post conditions of the operations in a solution model can be directly derived from the canonical transaction model [8]. In addition, class invariants may be specified as assertions.

Design for performance: Assume that the car dealer management system will be operational in very different contexts. For example, the transaction system may execute within a local dealer or across multiple dealers, the shared data can be small or large, the access conflicts can be rare or frequent. Within these constraints, the transaction system is expected to deliver the best quantitative performance.

Since none of the transaction techniques fulfill all these conditions, depending on the context of execution, an optimal transaction technique must be selected. For this purpose, we have decided to implement the policy manager as a control system. In this case, the policy manager continuously monitors the transaction context and based on the heuristics, it selects the optimal transaction implementation technique for maintaining the performance. The transaction architecture defined in section 4.4 does not have to be modified since the policy manager has been already identified as a separate component.

In our approach [55], first, analytic performance models were used for analyzing the performance of the alternative transaction techniques. Second, based on this analysis, an initial set of heuristic rules was defined. Third, using on the canonical model defined in section 4.4, the transaction system was implemented. Fourth, the environment of the transaction system was simulated. Finally, the performance characteristics of the transaction system were measured and the validity of heuristic rules was verified. If necessary, the parameters of the rules were adapted accordingly.

Verification of the integration of solutions: To improve the performance, the components of the transaction system can be dynamically selected from a set of alternative implementations. This requires a validation of the composite structure in all possible configurations. There have been a number of publications about composing transaction systems from various scheduling and recovery components. Indeed, in certain combinations of schedulers and recovery techniques, structural deadlock situations may arise [56, 57]. In the literature, theorem-proving techniques have been used for verifying the correctness of composition of various

scheduling and recovery techniques [56, 57]. In our case, in the implementation of the control system, these restrictions were explicitly taken into account.

It is clear that using canonical solution models have important advantages in achieving and verifying quality factors. The solution quality-feature models of canonical models can be certified and archived together with the canonical models. These models can be incorporated during the refinement and verification phases. In the implementation level, the quality-feature models can help in deriving the pre and post conditions of the software modules.

6.6. Problems of certifying models

In the following, we present a number of problems that may be experienced in certifying models. For each problem description, the corresponding certifiable model property is shown between the parentheses.

Lack of support for design for quality (all): The term design for quality means that quality requirements are explicitly taken into account in mapping requirements into solutions. Current software development methods, however, do not provide adequate means to derive solutions from quality requirements.

Trusting the authority (certification authority): One of the important issues in certification if that the certification authority is trustable and/or the certificate is real.

Lack of problem/solution quality feature models (problem/solution quality feature): One of the crucial issues in verification is to be equipped with a modeling technique that is rich enough to express the desired problem and solution domain quality features. In addition, there must be adequate means to derive the model parameters of the problem and solution quality features from requirement specifications and solution models, respectively.

Difficulty of measurement (measurement): One of the most difficult process in verification is to map the solution quality feature model parameters to a corresponding value system.

Difficulty of evaluation (evaluation): For certain quality factors such as usability, it is very difficult to automatically determine whether the measurements fulfill the requirements or not.

Difficulty of defining a suitable value system (value system): For certain quality factors such as adaptability, it may not be easy to define a value system that fulfills software engineers' intuition.

Difficulty of defining common/compatible value systems (value system): To be able to balance various quality factors of solutions, there is a need of a common and/or compatible value systems for different quality models. Obviously, this may be difficult or sometimes even impossible.

Level dependency (problem-solution): What is correct at a level may be incorrect at its sublevel. Assume for example that a large amount of data is pushed into a communication channel. The data must be received within a certain time limit. At an abstract level, we may assume that the channel bandwidth is large enough to pass high amounts of data within a specified time limit. Assume now that at a lower level, the channel is implemented as a bounded buffer and at certain times, the time limit may be exceeded. While verifying the property of solutions, it is therefore important to specify the level of verification in an explicit way.

Difficulty of determining the genericity-breaking instance (verification of canonical models): Most proof systems assume closed word assumption. It is therefore difficult to verify when an emerging instance may break the genericity of a canonical model.

Difficulty of verification of relevance (verification of integration of solutions): Verification of integration of solutions requires verification of the relevance of sub solutions. As explained in 3.5, identifying a

relevant set of requirements in a dynamically changing business context is a difficult problem. Moreover, if the solutions are integrated dynamically and if the order of integration influences the semantics of the total solution, it may be difficult to verify the integrated solution.

Difficulty of prediction of evolution (verification of evolution of solutions): Verification of evolution of models requires a fulfillment of two demands: A prediction of the evolution process, and checking the invariant properties of the models after each evolution step. Similar to the problem above, if more than one concern of a solution evolve dynamically and simultaneously and if the order of composition influences the semantics of the total solution, it may be difficult to verify the composed solution.

7. Constructible Models

7.1. Computation perspective

The computational aspects of constructing solution models in software can be summarized under the following three items:

Constructibility is an ability to create a computable solution model that retains its desired quality values.

Expresiveness: Instead of programming bare-hardware architecture, software engineers generally express their solutions as computer programs. Every language provides a set of first-class abstractions that are directly supported by the mechanisms of that language. A first-class abstraction may be passed as an argument of a call, it may be returned as a result of a call, it may be stored or retrieved, etc. The first class abstractions of a language are important factors in evaluating that language because they define the direct support that a programming language offers for expressing solutions.

Layered architectures: As stated in section 3, processor architecture can be implemented in hardware, firmware or as a virtual machine. It is of course possible to define a stack of virtual machines, where virtual machines at a lower layer provide services to the virtual machines at a higher layer. For example, distributed systems are organized in a similar way as a stack of protocols systems. In this article, unless otherwise stated, the term processor architecture may refer to hardware, firmware, virtual machine or stack of virtual machine (or protocol) implementations.

7.2. Process perspective

Refinement for computation: An ultimate goal of a concern-oriented process is to refine abstract problem specifications towards solutions that are constructible.

Computation for refinement: Computed Aided Software Engineering (CASE) environments aim at sporting software development activities by providing a set of software tools. Computation for refinement considers implementation of concern-oriented processes as a problem to be solved and applies concern-oriented processing principles for solving it.

7.3. Constraints perspective:

A concern-oriented process has to compromise alternative solutions within the context of the following two constraints:

Abstractness constraint: The abstractions of a programming language must be abstract enough to match the abstractions of solution models but also must be concrete enough to match the abstractions

(or components) of processor architectures. We can generalize this constraint as: The abstractions of an incremental solution model must be abstract enough to match the abstractions of the higher-level solution models but also must be concrete enough to match the abstractions of the lower-level solution models.

This constraint aims at minimizing the effort in refining solution models.

Standardization constraint: Abstractions (or components) of processor architectures must be standardized to ease sharing among multiple solution models but must be different enough to match the needs of multiple solution models.

This constraint aims at reducing costs through sharing implementations.

7.4. Background work

One of the major achievements of the theoretical computer science research is the introduction of the concept of computability [58, 59]. In the literature, computability is defined as a property of a function that can be computed by a Turing machine [59][7]. In this article, we prefer to use the term constructibility, which does not only refer to the computability property of programs but also to the underlying pragmatics used in creating them.

Programming languages can be classified based the paradigms (or computation models) that they adopt. Functional, logic, and imperative languages are the three basic paradigms and they all have equivalent expression power [60]. This means that any computable program can be expressed in one of these paradigms. In practice, however, the main reason why a programming language might be selected is more based on the pragmatic factors such as availability of the tools and skills rather than the paradigm of the language. Moreover, most commercial languages are not based on a single paradigm but they borrow features from multiple paradigms.

Programming languages can be divided into two categories: general purpose or domain specific. General-purpose languages aim at providing generic abstractions and mechanisms for a large category of domains. Languages such as C, C++ and Java are general-purpose languages. A domain-specific language offers abstractions and mechanisms that are particularly suitable for a given problem domain. An overview of domain-specific languages can be found in [61]. The issue of selecting a general-purpose or a domain-specific language is related to satisfying the constraints discussed in the previous section.

Programs can be automatically mapped onto processor architecture using compilation and/or interpretation techniques [11]. There are a number of dedicated generators available such as the stub generators for distributed systems [62]. Further, by using open compilers [63] and compiler-compilers [64], one can build dedicated generators.

7.5. Construction of the transaction system for car dealer management

Based on the canonical model of section 4.4, a prototype implementation of the transaction system was built in the Smalltalk language [8]. The controller part of the system was implemented later [55]. A more detailed description of the controller will be given in section 9.

[7] In [77], interactive model of computation is proposed as an alternative to Turing machines.

7.6. Problems of constructing solutions in software

The following list shows a number of problems that may be experienced in constructing solutions in software. For each problem description, the corresponding constructible model property is shown between the parentheses.

Difficulty of determining the quality values of software at run-time (constructibility): There may be numerous factors that influence the quality of executing software. It is generally difficult to reason about the properties of software at run-time, especially in an open and distributed computing environment.

Lack of expressiveness (expressiveness): This problem is similar to the expressiveness problem discussed in section 4.5. The adopted programming language must be rich enough to directly represent the abstractions of the solution models. The problem of lack of expressiveness may manifest in various forms. For example, in [15] we have discussed the following three lack of expressiveness problems:

- *Fixed message passing semantics*: is the difficulty of defining and reusing tailored message passing semantics of interacting components.

- *Lack of support for coordinated behavior*: is the difficulty of representing and reusing coordination among components especially if the coordination is implemented as multiple messages.

- *Unmatched layer functions* (layered architectures): This problem indicates a mismatch between the needs of the solution models and the functions provided by the realization layer. The interface of layers must be fixed to achieve compatibility and portability. On the other hand, due to the dynamic context of applications, the solutions models will likely to evolve continuously. Generally, this requires extensions to the interface of the realization layers. Changing interfaces, however, may conflict with the compatibility and portability objectives.

Inefficiency of mapping high-level solution concerns to computation models (refinement for computation): To express certain behavior precisely, solution models may adopt high-level language constructs that are very inefficient to implement using popular programming languages.

Difficulty of automating certain processes (computation for refinement): Certain processes, such as software development, are very complex. Software engineers, therefore, need tools to assist them with their task. Software development is also a human activity. Therefore, knowledge of software development is not generally available as a mathematical theory, but in the form of human knowledge, i.e. in the form of rules and heuristics, formulated by experts in the field. These rules and heuristics are necessarily imprecise, and formulated in terms of imprecisely defined concepts. Tools for software development, which deserve to be called intelligent, will have to incorporate (part of) this knowledge.

Difficulty of leveraging realization levels (abstractness constraint): In complex systems, different concerns of a solution may demand different levels of realization. Generally, once a programming language or execution environment is selected, software engineers do not have any means to leverage the level of realization.

Difficulty of standardizing the computation models (standardization constraint): Different solution models may demand different computation models. Therefore, it is generally not possible to define an ideal computation model for all kinds of problems.

8. Closure property of Models

8.1. Definitions

Given a set of models, if an operation applied to any member of the set produces a model that is also in the set, then it is called the set is closed under that operation. We call this the closure property of models [59]. The closure property of models can be analyzed from the following three perspectives:

- Closure for manipulating models requires a constant set of operations on models so that models can be manipulated in a uniform manner.
- Closure for process support requires a constant set of process operations for mapping problem domain concerns into solution domain concerns.
- Closure for self-manipulation requires a constant set of reflective operations. A reflective system is a system, which incorporates models representing (aspects of) itself. This self-representation is causally connected to the reflected entity, and therefore, makes it possible for the system to answer questions about itself and supports actions on itself. Reflective computation is the behavior exhibited by a reflective system. The term reflection was introduced by [65] as a technique to structure and organize self-modifying procedures and functions. In [66] reflection was applied within the context of object-orientation.

8.2. Motivations

Easier to define clients: The availability of a constant set of operations on models helps in defining generic client software that utilizes these models.

Easier to define methods and tools: The availability of a constant set of process operations helps in defining generic methods and tools.

Easier to define self-evolving models: The availability of a constant set of reflective operations helps in defining generic self-evolving models.

8.3. Background

The concept of closure has been studied in various fields, such as set theory, formal languages and relational databases [58, 59, 67].

The composite pattern [10] aims at creating hierarchically nested object models with a constant set of operations for each element in the composite.

The term reflection was introduced by [65] as a technique to structure and organize self-modifying procedures and functions. In [66], reflection was applied within the context of object-orientation. A considerable amount of work has been done in reflection techniques, for example, in concurrent computation, distributed system structuring and middleware, programming language design, real-time systems, and in network design [68, 69].

The OMG has defined various standards for specifying meta-models, such as Metadata Interchange Specification, Software Process Engineering Metamodel Specification and Metaobject Facility Specification [4].

8.4. Problems of defining closure properties

The following list indicates the problems that may be experienced in defining solution models with the desired closure properties. For each problem description, the corresponding closure model property is shown between the parentheses.

Difficulty of defining uniform and cooperating solutions (closure for manipulating models): To be able to define a constant set of operations on models, the required solution model must be partitioned into a uniform and cooperating sub solution models. This may not be an easy task. Moreover, a uniform decomposition may be preferred for certain quality factors such as stability of software but may be undesired for other quality factors such as extensible behavior.

Difficulty of formalizing design as a uniform and cooperating processes (closure for process support): Similar to the problem above, to be able to define a constant set of operations on processes, the required process model must be partitioned into uniform and cooperating sub processes.

Lack of reflection (closure for self manipulation): A considerable amount of research work has been done in reflection techniques, for example, in concurrent computation, distributed system structuring and middleware, programming language design, real-time systems, and in network design [68, 69]. Conventional programming languages, however, do not provide adequate support for reflective system development. Moreover, closure for self-manipulation requires identification of a set of standard reflective operations that do not change after each self-manipulation.

9. Controllable Models

9.1. Basic components of a feedback control system

Figure 6 depicts an example feedback control system architecture, which consists of the following eight components:

Controlled system: A solution model, a concern-oriented process or combination of these. We distinguish two kinds of controlled systems:

- Abstract: A controlled system may be an abstract solution as defined by a problem solving process that maps problems into abstract solutions.
- Concrete: A controlled system may be a constructible solution or a process (interpreter/compiler), which maps a constructible solution into another constructible solution.

Abstract controlled systems generally require human assistance, whereas the concrete ones may be implemented completely in software.

Sensor: A software, hardware or human input that provides the actual information about the controlled system.

Feedback loop: A channel or interpreter that connects the sensor to the model of the controlled system.

Model of the controlled system: A model that is suitable for reasoning about the actual properties of the controlled system.

Reference model: A model of the controlled system that specifies its desired properties.

Comparator: An evaluator that compares the model of the controlled system with the reference model. Error is the difference between the desired and the actual properties of the controlled system.

Compensator: Determines the necessary controlling actions on the controlled system so that the controlled system fulfils the desired properties.

Actuator: Implements the controlling action on the controlled system.

In Figure 6 the term control system refers to all 8 components. The term controller includes all the components except the component Controlled system. A control system thus consists of a controlled system plus a controller.

There are similarities between the verification model of Figure 5 and the feedback control architecture of Figure 6. The components Solution, Solution quality feature, Problem quality feature models of Figure 5 correspond to the components Controlled system, Model of the controlled system and Reference model of Figure 6, respectively. In fact, verification of models is a necessary requirement for controlling models. In addition to verification, controlling aims at correcting the errors of models.

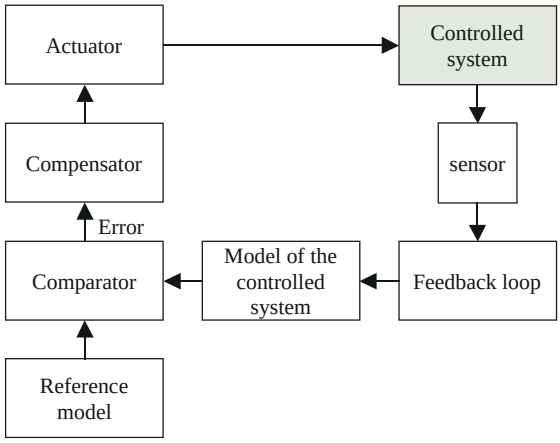

Figure 6. A feedback controller.

9.2. Control architectures

Fixed-structure fixed-parameter control architectures: These control systems assume that the structure and/or parameters of the controlled system do not vary [70, 71]. The architecture of Figure 6 is typical example of this category. Other known examples are hierarchical and/or distributed controllers[8] [72].

Adaptive control: If the structure and/or the parameters of the controlled system vary, it may not be possible to affectively control the system unless the control system adopts a strategy to deal with this variation. In adaptive control, depending on the variations of the parameters of the controlled system, the control system is adapted. The adaptation can be realized, for example, by changing the transfer function of the feedback loop, the reference model, the implementation of the compensator, etc. [70].

Optimal control: In optimal control, the control system implements a strategy to minimize a predefined performance index (also called cost function). An optimal controller requires that the performance index is represented as a mathematical expression and it is measurable [70]. Obviously, the performance index must be specified as simple as it can be otherwise the mathematics can be too difficult to solve.

[8] Adaptive and/or optimal control architectures can be considered as extensions to these architectures.

9.3. Quality of control

Controllability and observability: This is the fundamental property of a control system. A system is controllable if it is possible to cause its state vector to move from any initial value, to any value, in a finite time. A system is observable if it is possible to reconstruct the state-vector completely from measurements made. Within the context of this article, the state vector of a system refers to the relevant properties of a solution model or concern-oriented process. The concept of measurement was defined in section 6. The term controllable model, which is the title of this section, means that the solution model/concern-oriented process, which is also called the controlled system, satisfies the controllability quality factor.

Stability: If the properties of a controlled system can be controlled within the specified amount of time, then it is called a stable system.

Performance criteria: Stable control systems can be analyzed in more detail with respect to certain performance criteria. Examples are:

- Steady-state error: The difference between the actual and desired properties of the controlled system that are no longer observable by the controller and therefore no attempts are made to correct it.
- Rise time: The shortest time to achieve some specified percentage of the desired property values of the controlled system.
- Setting time: The time taken for the controlled system to reach and remain within some specified range of its final property values.

9.4. Motivations

Feedback control systems are based on the assumption that it is easier to correct the errors of a system during its operational phase rather than designing the system to be ideal at the creation time. Controlling the quality of software processes and products have many obvious advantages, such as improved client satisfaction, complexity reduction etc. However, designing an efficient and affective control system for this purpose is not trivial and demands solutions to several important and open research questions that are discussed in this article.

9.5. Background work

Control systems have been successfully applied in almost all areas of engineering and extensive literature exists in this field [70, 71]. However, the formalisms adopted in traditional control systems [70, 71], such as differential equations, are generally not suitable for controlling software products and processes. In the literature, intelligent controllers have been introduced for controlling complex systems, which cannot be expressed using mathematical models such as differential equations [73, 74]. By intelligent controller, we mean the application of soft computing techniques[9] to the design of control systems. Most intelligent control systems, however, have been applied to other disciplines than software engineering, such as artificial vision, thermal processes, target identification, etc.

Within the area of distributed systems, several researchers have experimented with the so-called quality-aware middleware systems [75, 76]. These systems generally adopt control architecture to monitor and improve the quality of service parameters of the middleware systems.

[9] Also called computational intelligence. Currently, computational intelligence techniques are based on fuzzy-logic, neural-networks and genetic algorithms.

Although software quality has been extensively studied in the literature [46, 47], controlling quality of software products and processes based on the principles of control theory has not been explored yet.

9.6. Controlling the transaction subsystem of the car dealer management system

To optimize the performance of the transaction system, the policy manager component of the transaction system has been implemented as a control system [55]. The policy manager continuously monitors the transaction context and based on its heuristics, it selects the scheduling scheme that provides the best performance. In our prototype implementation, we have experimented with three kinds of scheduling schemes: two-phase locking, timestamp-ordering and optimistic.

The controller aims at optimizing the throughput for a number of concurrently executing transactions between 0 and 100. The following heuristic rules have been extracted from various publications on atomic transactions [55]:

IF # concurrent_transactions $<$ lowerT

THEN select optimistic scheduler

IF # concurrent_transactions $\geq$ lowerThreshold and # concurrent_transactions $\leq$ upperT

THEN select two-phase locking scheduler

IF # concurrent_transactions $>$ upperT

THEN select timestamp-ordering scheduler

The variables *lowerT* and *upperT* represent the threshold values for selecting a different scheduler to increase the throughput of the system. Obviously, these heuristics rules may only be effective if the right threshold values have been defined. Since it is difficult to statically determine the ideal values, we have implemented a self-tuning adaptive control system that adjusts the threshold values after each decision point in time. In order to experiment with the system, the environment of the transaction system was simulated. The simulator generated random calls on the transaction system based on predefined simulation properties. Figure 7 shows the change in the threshold values, where the X-axis corresponds to the number of simulated decision points and the Y-axis shows the threshold values. In this example setting, the threshold values were stabilized after about 50 decision points.

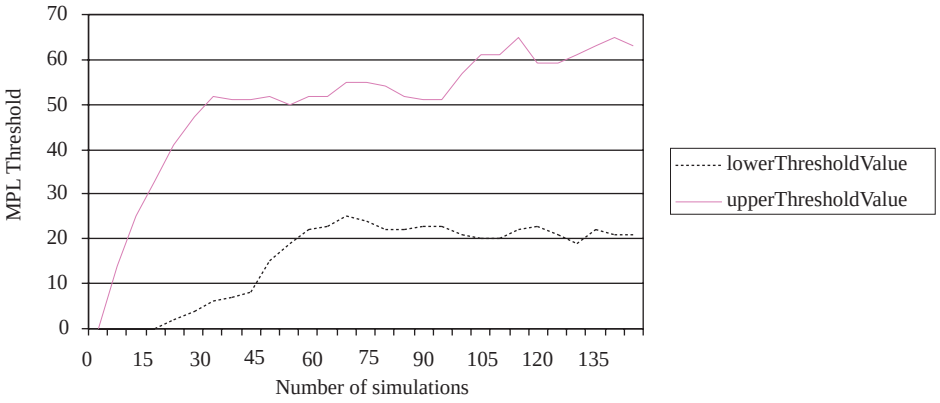

Figure 7. Tuning the threshold values.

To evaluate the affect of the controller, an experiment was carried out with the following situations:

- Fixed scheduler with a two-phase locking concurrency control scheme.
- Fixed scheduler with a timestamp ordering concurrency control scheme.
- Fixed scheduler with an optimistic concurrency control scheme.
- Run-time adaptive controller based scheduler as explained above.

Figure 8 shows the result of the simulations. From this figure we can conclude that for lower number of concurrent transactions the optimistic scheduler implementation performs better than the timestamp ordering and the two-phase locking scheduler implementations. For a higher degree of concurrency the throughput for the optimistic scheduler implementation decreases significantly and the timestamp-ordering scheduler implementation then provides a better throughput. The run-time adaptable scheduler implementation nearly follows the optimal values of the different schedulers.

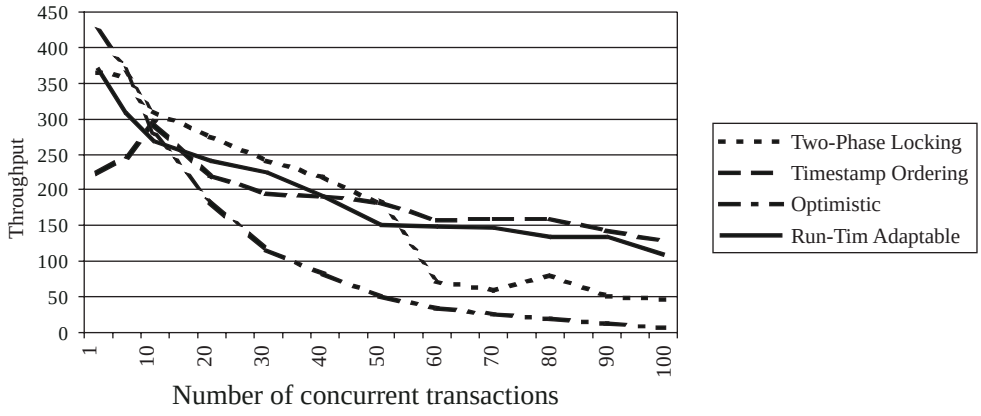

Figure 8. Performance values for the fixed scheduler implementations and the run-time adaptable scheduler implementation.

9.7. Problems in defining controllable models

The following list describes a set of problems that one may experience in designing controllable models. For each problem, the related properties of control systems are indicated between the parentheses.

Difficulty of obtaining the actual parameters of controlled systems (sensor): The actual parameters of a controlled system may not be easily obtained by just calling on its interface operations. This problem is also related to the quality factor observability, which was defined 9.3. Moreover, an ideal measurement process should not influence the controlled model in any means.

Semantic gap between the sensed data and the model of the controlled system (feedback loop): If the model of a controlled system is defined at a much higher level of abstraction than the system being controlled, then it may be difficult to derive the model parameters directly from the measured parameters.

Difficulty of defining the model of the controlled system (model of the controlled system): The quality and the effectiveness of a control are directly related to the expression power of the model of the controlled system. Obviously, for a complex system, it may be difficult or impossible to define a suitable model.

Difficulty of defining an ideal reference model (reference model): This problem is similar to the problem defined above. In addition, in some cases, it may be difficult to define an ideal model, since there may be more than one model that fulfills the definition of what is called the "ideal".

Difficulty of defining an automated comparator (comparator): If there is a semantic gap between the model of the controlled system and the reference model, it may be difficult to implement the comparator without a human assistance.

Difficulty of defining an automated compensator (compansator): To be able to control, there must be known strategies for influencing the controlled system so that it fulfils the desired characteristics. For certain complex systems, there may not be a known strategy at all. Moreover, due to changing context, it may be difficult to predefine a strategy. In addition, for certain abstract problems such as software deign, it may be difficult to implement a strategy without the need for a human assistance.

Difficulty of applying the desired control actions (actuator): This problem is related to the term controllability and observability, which were defined in section 9.3. To be able to control, the controlled system must provide the right handlers (interface). For certain cases, the desired interface may not be available.

Difficulty of defining an optimal control architecture (control architectures): In general, there is no general design strategy to synthesize the ideal control architecture for a given control problem [70, 71]

Difficulty of determining an adaptation and/or optimization strategy (control architectures): Generally, adaptive control systems are equipped with an adaptation strategy, which may be implemented as a meta-control system that controls the components of the base-level control system. Designing a meta-level control system may involve all the issues discussed in this article. To be able to define an optimal control architecture, the optimization criteria must be formalized and implemented as an efficient computation.

Lack of means to compute the quality factors of control (quality of control): Due to the lack of precise mathematical models, in general, the quality of control parameters, such as stability, steady-state error, rise time, setting time, etc. cannot be determined analytically.

10. Justifying the 7 C's and Conclusions

After an extensive study, in [chapter 2, 8] it has been shown that controlled problem solving processes form one of the basic reasoning mechanisms of both traditional engineering disciplines and philosophy. The definition of controlled problem solving process as defined in [8] conforms the definition of controllable concern-oriented process given in sections 3 and 9 of this article. We will now introduce the concept of quality-oriented software engineering, which is based on the principle of controllable concern-oriented processes:

Quality-oriented software engineering: aims to solve the problems of stakeholders by developing and/or selecting optimal products along the software engineering process.

Stakeholder: Any person or representative of an organization who has a stake-a vested interest-in the product or whose opinion must be accommodated. A stakeholder can be an end user, a purchase, a contractor, a developer, or a project manager.

Problems of stakeholders: A set of concerns to be solved through the utilization of products. In other words, products are used to solve the problems of the stakeholders. Features, properties or behavior that are necessary to solve the problems can be described as product requirements.

Optimal product: A product that satisfies the constraints imposed by its context. A product is generally a software system. In embedded systems, however, software and hardware products naturally

co-exist. Products may have different granularity; its may be a library such as collection classes, or a large distributed system, such as a workflow management system.

Context of a product: The quality factors imposed by the market, stakeholders, enabling technology, financial conditions, personal skills, available knowledge and scientific developments define the context of a product. The quality factors define the dimensions of the relevant characteristics of products. Examples of quality factors are correctness, relevancy, adaptability, performance, reusability, changeability, traceability, reliability, low-cost, etc. An optimization process balances the quality factors of a product within a given context.

We will now informally justify the 7 C's using the following reasoning:

Concern-oriented processes: To create and/or select an optimal set of products, there must be a satisfactory solution for the problems of the stakeholders. To be able to provide a solution to a given problem, problem-solving techniques must be adopted.

Controllable models: To satisfy the product constraints imposed by its context, the properties of the products must be continuously monitored and improved if necessary. This requires controllable solutions.

Constructible models: To create a software product, the product must be constructible as a software system.

Closure property of models: To be able to design a stable controller, the software system (product) must provide a constant set of controlling operations to the controller. This requires that the software system be closed under the controlling operations.

Certifiable models: To be able to decide on a controlling action, the desired properties of the software system must be verified. This requires that the software system is certifiable.

Composable models: To be able to carry out the necessary adjustments, the software system must be composable.

Canonical models: To be able to identify the composable parts, the software system must be derived from a canonical model.

In this article, we have presented and motivated the 7 C's to overcome the problems in current software design and implementation practices. For each C, we have identified a set of open problems that may inspire the researchers. The 7 C's may also help the practicioners to be concious about the fundemantals of the difficulties that they possibly experience.

References

[1] L.J. May, major Couses of Software Project Failures, http://stsc.hill.af.mil /crosstalk/1998/jul/causes.asp, 1998.

[2] The Standish Group, Chaos, http://www.standishgroup.com /chaos.html, 1995.

[3] I. Jacobson, G. Booch, and J. Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999.

[4] OMG, http://www.omg.org.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. Object-Oriented Modeling and Design, Prentice Hall, 1991.

[6] B. Tekinerdogan and M. Aksit, Classifying and Evaluating Architecture Design Methods, in Software Architectures and Component Technology, M. Aksit (Ed.), Kluwer Academic Publishers, pp. 3 - 27, 2002.

[7] B. Tekinerdogan and M. Aksit, Synthesis Based Software Architecture Design, in Software Architectures and Component Technology, M. Aksit (Ed.), Kluwer Academic Publishers, pp. 143 - 173, 2002.

[8] B. Tekinerdogan. Synthesis-Based Software Architecture Design, PhD Thesis, Dept. Of Computer Science, University of Twente, March 23, 2000.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1999.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Software, Addison Wesley, 1995.

[11] A.V. Aho, R. Sethi and J.D. Ullman, Compilers Principles, Techniques and Tools, Addison-Wesley, 1986.

[12] D.P. Gaver and G.L. Thompson, Programming and Probability Models in Operations Research, Brooks/Cole Publishing Company, 1973.

[13] A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Transactions on Database Systems, 8(4): 484-502, 1983.

[14] K. Blois, The Oxford Textbook of Marketing, Oxford University press, 2000.

[15] M. Aksit and L. Bergmans, "Guidelines for Identifying Obstacles when Composing Distributed Systems from Components, in Software Architectures and Component Technology", M. Aksit (Ed.), Kluwer Academic Publishers, pp. 29-56, 2001.

[16] M. Aksit and L. Bergmans, "Obstacles in Object-Oriented Software Development", Proceedings OOPSLA '92, ACM SIGPPLAN Notices, Vol. 27, No. 10, pp. 341-358, October 1992.

[17] M. Aksit and B. Tekinerdogan, "Deriving Design Alternatives Based on Quality Factors, in Software Architectures and Component Technology", M. Aksit (Ed.), Kluwer Academic Publishers, pp. 225-257, 2001.

[18] C. Alexander, The Timeless Way of Building, Oxford University Press, 1979.

[19] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, 1999.

[20] B.J. Holmes, D.T. Joyce, Object-Oriented Programming with Java, Jones and Bartlett Publishers, 2001.

[21] K. Czarnecki, U.W. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[22] M. Aksit (Ed.), Software Architectures and Component Technology, Kluwer Academic Publishers, 2001.

[23] H. Obbink, R. van Ommering, J.G. Wijnstra and P. America, "Component Oriented Platform Architecting for Software Intensive Product Families, in Software Architectures and Component Technology: The State of the Art in Research and Practice", M. Aksit (Ed.), Kluwer Academic Publishers, pp.99-142, 2001.

[24] M.E. Fayad, D.C. Schmidt and R. Johnson (Eds), Building Application Frameworks, Wiley, 1999.

[25] L. Bergmans and M. Aksit, "Composing Synchronisation and Real-Time Constraints", In Journal of Parallel and Distributed Computing, Vol. 36, No. 1, pp. 32-52, 1996.

[26] P.A. Bernstein and E. Newcomer, Principles of Transaction Processing, Morgan Kaufman Publishers, 1997.

[27] D. Ungar and R. B. Smith, "Self: The Power of Simplicity", In Proceedings OOPSLA'87, ACM SIGPLAN Notices, Vol. 22, No. 12, pp. 227-242, December 1987.

[28] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, Getting Started with AspectJ, CACM, Vol. 44, No. 10, pp. 59-65, October 2001.

[29] L. Bergmans and M. Aksit, "Composing Crosscutting Concerns Using Composition Filters", CACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[30] J. Noppen, A methodological framework for coping with evolving software systems, MSc. Thesis description, TRESE Group, University of Twente, department of Computer Science, Enschede, The Netherlands, September 2001.

[31] M. Aksit and F. Marcelloni, Deferring Elimination of Design Alternatives in Object-Oriented Methods, Concurrency and Computation: Practice and Experience, Vol. 13, pp. 1247-1279, John Wiley & Sons, Ltd, 2001.

[32] G. Kniesel, "Type-Safe Delegation for Run-Time Component Adaptation", ECOOP'99 Conference Proceedings, LNCS 1628, pp. 351-366, 1999.

[33] M. Aksit, L. Bergmans and S. Vural, "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", ECOOP '92, LNCS 615, Springer-Verlag, pp. 372-395, 1992.

[34] S. Matsuoka and A. Yonezawa. Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, In Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner and A. Yonezawa (Eds.), MIT Press, Cambridge, MA, pp. 107-150, October 1993.

[35] T. Elrad (moderator), M. Aksit, G. Kiczales, K. Lieberherr and H. Ossher (panelists), "Discussing Aspects of AOP", CACM, Vol. 44, No. 10, pp. 33-38, October 2001.

[36] K. Lieberherr, D. Orleans and J. Ovlinger, "Aspect-Oriented Programming with Adaptive Methods", CACM, Vol. 44, No. 10, pp. 39-41, October 2001.

[37] H. Ossher and P. Tarr, "Using Multidimensional Separation of Concerns to (re)Shape Evolving Software", CACM, Vol. 44, No. 10, pp. 43-50, October 2001.

[38] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, "Abstracting Object-Interactions Using Composition-Filters", In object-based distributed processing, R. Guerraoui, O. Nierstrasz and M. Riveill (Eds.), LNCS, Springer-Verlag, pp. 152-184, 1993.

[39] B. Tekinerdogan and M. Aksit, "Providing Automatic Support for Heuristic Rules of Methods", in Object-Oriented Technology, S. Demeyer and J. Bosch (Eds.), LNCS 1543, ECOOP'98 Workshop Reader, Springer Verlag, pp. 496-498, July 1998.

[40] R. Greiner, C. Darken and I. Santoso, "Efficient Reasoning", ACM Computing Surveys, Vol. 33, No. 1, pp. 1-30, March 2001.

[41] R. Schleiffer, H-J. Sebastian, E.K. Antonsson, "Genetic Algorithms in Fuzzy Engineering Design", Proceedings of DETC'99, September, 1999.

[42] G. von Bochmann, "Concepts for distributed systems design", Springer-Verlag, 1983.

[43] H. Ossher and P. Tarr, Multi-Dimensional Separation of Concerns and the Hyperspace Approach, in Software Architectures and Component Technology, M. Aksit (Ed.), Kluwer Academic Publishers, pp. 293 - 323, 2002.

[44] Software Engineering Terminology (1999), IEEE Std 610.12-1990. In: IEEE Standards Software Engineering, 1999 Edition. Volume One: Customer and Terminology Standards (ISBN 0-7381-1559-2).

[45] F.S. Roberts, Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences, Addison-Wesley, 1979.

[46] ISO 9126, Information Technology, Software Product Quality – Part 1: Quality Model, International Organisation for Standardisation, FCD 1998.

[47] M.C. Paulk, et al., The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley (ISBN 0201546647) 1994.

[48] L. Osterweil et al, "Strategic Directions in Software Quality", ACM Computing Surveys, Vol. 28, No. 4, pp. 738-750, December 1996.

[49] L. Kleinrock, Queueing Systems Volume 1: theory, John Wiley & Sons, 1975.

[50] A.M. Law and W.D. Kelton, Simulation Modeling and Analysis, 3e, McGraw-Hill, 2001.

[51] D. Ferrari, Computer System Performance Evaluation, Prentice-Hall, 1978.

[52] E.M. Clarke and J.M. Wing, et al, "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, Vol. 28, No. 4, pp. 626-643, December 1996.

[53] D.M. Hilbert and D.F. Redmiles, Extracting Usability Information from User Interface Events, ACM Computing Surveys, Vol. 32, No. 4, pp. 384-421, December 2000.

[54] Quality-Oriented Software Engineering, http://trese.cs.utwente.nl/quality_ oriented_se/

[55] F. Schopbarteld, Dynamically Tuning Transaction Behavior in a Distributed Environment, M.Sc. Thesis, Dept. of Computer Science, University of Twente, 1999.

[56] R. Guerraoui, "Atomic Object Composition", In Proceedings of the European Conference on Object-Oriented Programming, LNCS 821, Springer Verlag, pp. 118-138, 1994.

[57] N. Lynch, M. Merritt, W. Weihl and A. Fekete, Atomic Transactions, Morgan Kaufmann Publishers, 1994.

[58] D.C. Kozen, Automata and Computability, Springer-Verlag, 1997.

[59] A. Parkes, An Introduction to Computable Languages and Abstract Machines, International Thomson Computer press, 1996.

[60] K.C. Louden, Programming Languages: Principles and Practice, PWS-Kent Publishing Company, 1993.

[61] A. van Deursen, P. Klint and J. Visser, Domain Specific Languages: An Annotated Bibliography, CWI Report, 1998.

[62] G.F. Coulouris and J. Dollimore, Distributed Systems: Concepts and Design, Adison-Wesley, 1988.

[63] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura, OpenJIT: "An Open-Ended, Reflective JIT Compiler Framework for Java", ECOOP'2000 Conference Proceedings, LNCS 1950, pp. 362-387, 2000.

[64] Webgain, JavaCC homepage, http://www.webgain.com/products/java_cc/

[65] B.C. Smith. Reflection and Semantics in a Procedural Language. MIT-LCS-TR-272, Mass. Inst. of Tech. Cambridge, MA, January 1982.

[66] P. Maes, "Concepts and Experiments in Computational Reflection", In Proceedings OOPSLA'87, ACM SIGPLAN Notices, Vol. 22, No. 12, pp. 147-155, December 1987.

[67] C.J. Date, An Introduction to Database Systems, Vol. 1, Addison-Wesley Company, 1986.

[68] A. Yonezawa (Ed.), "Reflection and Meta Level Architecture", Proceedings of IMSA'92, Tokyo, November 1992.

[69] P. Cointe (Ed.), "Meta-Architectures and Reflection", Springer Verlag LNCS 1616, St Malo, May 1999.

[70] K. Dutton, S. Thompson and B. Barraclough, The Art of Control Engineering, Addison-Wesley, 1997.

[71] B.C. Kuo, Automatic Control Systems, Prentice-hall Inc., 1995.

[72] A.J.N. van Breemen, Agent-Based Multi-Controller Systems, Ph.D. Thesis, Twente University Press, May 2001.

[73] M.M. Gupta and N.K. Sinha (Eds), "Intelligent control Systems, Theory and Applications", IEEE Press, 1996.

[74] M.M. Gupta and N.K. Sinha (Eds), Soft Computing and Intelligent Systems, Theory and Applications, Academic Press, 2000.

[75] G.S. Blair, A. Andersen, L. Blair, G. Coulson and D. Sánchez Gancedo, "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform", IEE Proceedings - Software, Vol. 147, Issue 01, February 2000.

[76] L. Bergmans, A. van Halteren, L. Ferreira Pires, M. van Sinderen and M. Aksit, "A QoS-Control Architecture for Object Middleware", IDMS'2000 Conference Proceedings, LNCS 1905, Springer Verlag, pp. 117-131, October 2000.

[77] P. Wegner, Interactive foundadtions of Computing, "Theoretical Computer Science", Vol. 192, pp. 315-351, February 1998.