3 Specification of Requirements Models

Ricardo J. Machado, Isabel Ramos and João M. Fernandes

Abstract: The main aim of this chapter is to present and discuss a set of modeling and specification techniques, in what concerns their ontology and support in the requirements representation of computer-based systems. A systematic classification of meta-models, also called models of computation, is presented. This topic is highly relevant since it supports the definition of sound specification methodologies in relation to the semantic definition of the modeling views to adopt for a given system. The usage and applicability of Unified Modeling Language (UML) diagrams is also related to their corresponding meta-models. A set of desirable characteristics for the specification methodologies is presented and justified to allow system designers and requirements engineers to more consciously define or choose a particular specification methodology. A heuristic-based approach to support the transformation of user into system requirements is suggested, with some graphical examples in UML notation.

Keywords: Modeling, Specification, Meta-Models, Requirements, Model transformation.

3.1 Introduction

Computer-based systems integrate, as information processing sub-systems, one or more computing systems able to capture, store, process, transfer, present and manage information. Within the design of computer-based systems, this justifies the need for the incorporation of several technological entities: (1) software, firmware, and (analog and digital) hardware, to process and store information; (2) communication network services to transport information; (3) sensors and actuators to interact with the physical environment; and (4) human-machine interfaces to exchange information with human operators. Although computer-based systems can be strictly based on computer technologies, they normally include other entities such as human operators, organizational subsystems, documentation, and manuals.

Since computer-based systems are, by nature, heterogeneous, modeling and specifying their requirements demands a holistic approach.

A requirement can be defined as "something that a client needs." From the point of view of the system designer or the requirements engineer, a requirement could also be defined as "something that must be designed." The IEEE 610 standard [21] defines a requirement as: (1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents; (3) a documented representation of a condition or capability as in (1) or (2).

Clients and developers (system designers and requirements engineers) have, naturally, different points of view towards requirements, which imply that requirements can be divided into two different categories: user and system requirements.

User requirements result directly from the requirements elicitation task (see Chap. 2 for further details on requirements elicitation techniques), as an effort to understand the clients' needs. They are, typically, described in natural language and with informal diagrams, at a relatively low level of detail. User requirements are focused in the problem domain and are the main communication medium between the clients and the developers, at the analysis phase. System requirements result from the developers' efforts to organize the user requirements at the solution domain. They, typically, comprise abstract models of the system, at a relatively high level of detail, and constitute the first system representation to be used at the beginning of the design phase. The correct derivation of system requirements from user requirements is an important objective because it assures that the design phase is based on the effective clients' needs. This also guarantees that no misjudgment is arbitrarily introduced by the developers during the process of system requirements specification.

The aim of this chapter is to present and discuss a set of modeling and specification techniques, in what concerns their ontology and support in the requirements representation of computer-based systems. This chapter is not intended to be used as an exhaustive survey and summary of existing modeling approaches. It provides some guidelines to system designers and requirements engineers so that they select the modeling approach that best fits their problems. The intended audience of this chapter is system designers and requirements engineers who wish to expand their background knowledge on meta-modeling and improve their development strategy options.

Section 3.2 discusses the differences between the modeling and the specification activities. In this chapter, specification is only related to models, and not to other possible forms. Sect. 3.3 presents a systematic classification of meta-models as a key issue for the semantic definition of the modeling views to adopt for a given system. Some authors use the term "modeling techniques", instead of "meta-models". Sect. 3.4 describes a set of desirable characteristics for specification methodologies, so that system designers and requirements engineers can more consciously define or choose a particular specification methodology. Sect. 3.5 briefly describes a heuristic based approach to support the transformation of user into system requirements. This section shows that model continuity is a key issue and highlights the importance of having a well defined process to relate, map and transform requirements models.

3.2 Modeling vs. Specification

The first decision of developers, when they want to specify a system, is to select which part of the system they wish to take into account. The selection of that part defines the system view, i.e., the system perspective that needs to be represented

[5]. This view has a merely conceptual existence in the human mind, and, according to an unstructured and informal representation, at least at the conscious level of the developers.

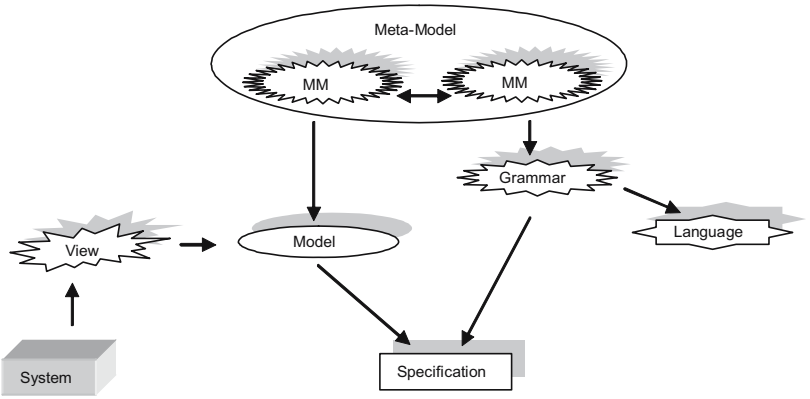

Fig. 3.1 Specification of systems

The formalization of the system view occurs when it originates a model. This model consists in a representation, still conceptual, of the view of the system, according to a particular meta-model. This meta-model corresponds to a set of (functional or structural) composition elements and of composition rules that permits to build a model representing the system view. This model serves the purpose of explaining and sharing the conceptual view held in the human mind. In this way, developers make their view available to the judgment of others and to further reformulation.

The accuracy of a particular modeling approach depends on its capability to select the meta-model that semantically supports the characteristics of the system to be modeled. The selected meta-model defines the semantic limits of the system representation at the model level. Meta-models characterization is of central importance due to its impact on the systems modeling accuracy.

Although the system model is already the result of a formalization effort of the system view, its existence is still at the conceptual level. To become "tangible" it must be transformed into a concrete representation called "specification", i.e., a real representation of the system model in a given language [41]. The conceptual model adopted in the definition of the language corresponds to the language meta-model, which allows the description of the system model by means of a graphical, textual or other kind of representation; see Fig. 3.1.

According to the terminology used here, the difference between modeling and specification, activities that are often misunderstood, is now clearer. Modeling corresponds to the activity of selecting a meta-model to formalize, at the conceptual level, a given system view, while specification is related to the adoption of a language to make a system model tangible. Obtaining a specification that ade-

quately represents the system depends both on the characteristics of the selected meta-model for the modeling activity and on the meta-model of the chosen representation language. Thus, to avoid semantic mismatches, the two adopted meta-models must be compatible. Whenever possible, the language meta-model should be the same as the one used in the system modeling activity. In this context, it becomes clear that the characterization of meta-models is a fundamental issue for accomplishing both the modeling and the specification activities.

3.3 Meta-Models Categories

Although the two meta-models involved in the construction of a system specification may not be exactly the same, one can assume, for simplification purposes, that the representation language has been consciously selected taking into account the characteristics of its meta-model (which is not always true).

Ideally, representation languages should allow the specification of the desired system characteristics, in a non ambiguous way. This is possible, if the meta-model of the language is: (1) formal (accurate, rigorous), to avoid ambiguities in the interpretation of the system representation; (2) complete, to allow the construction of a representation that totally describes the system view. These are not absolute properties, since they depend on the particular system to be specified. In [17], Gajski *et al.* organize the most common meta-models into five distinct groups. A brief description of each meta-model category is presented next.

3.3.1 State Oriented Meta-Models

State oriented meta-models allow modeling a system as a set of states and a set of transitions. The transitions between states evolve according to some external stimulus. These meta-models are adequate to model systems in which temporal behavior is the most important aspect to be captured. Finite state machines (FSMs), finite state machines with data paths (FSMDs), StateCharts and Petri nets are examples of state oriented meta-models.

FSMs [32], also known as “finite state automata”, correspond to the most used meta-model in the description of control systems, since the temporal behavior of these systems is naturally represented in the form of states and transitions between states. The two basic alternatives to construct state machines (Mealy or Moore) differ only in the output function. On Mealy machines the output function depends both on the state and the inputs, while on Moore machines the output function depends only on states. Graphical diagrams that represent state machines are usually called “state transition diagrams” (STDs).

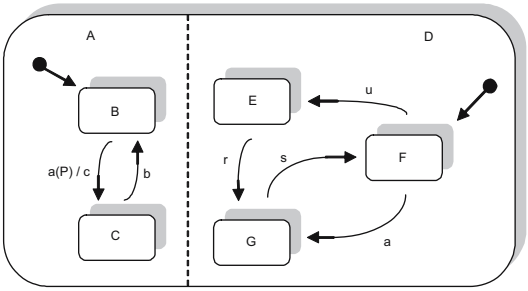

Fig. 3.2 Example of a StateChart

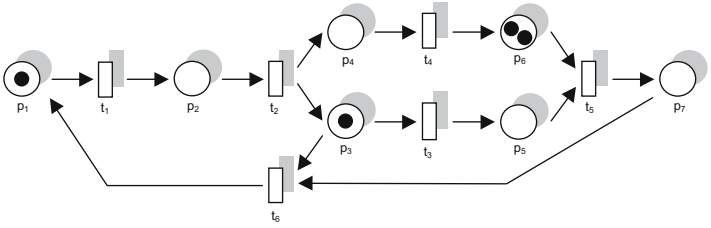

Fig. 3.3 Example of a Petri net

FSMDs [16] are an evolution of FSMs to solve, in a simple way, the problem of state explosion. FSMDs extend FSMs by using integer or floating variables to replace thousands of states in the corresponding FSM. While FSMs can only represent control systems, FSMDs are also able to represent computing systems. These meta-models are not able to capture complex behaviors, since they lack the ability to deal with concurrency and hierarchy.

HCFSMs are another FSM extension, since they support the representation of concurrency and allow the construction of hierarchical models. HCFSMs are relatively limited in dealing with complex data structures. The meta-model behind HCFSMs is the same as Harel's StateCharts graphical representation language [18]; see Fig. 3.2. UML's state diagrams have their origins in Harel's StateCharts.

Petri nets [34, 35] constitute another state oriented meta-model. Petri nets are appropriate to model concurrent actions, since they can deal with parallelism, synchronization, resource sharing and memorization; see Fig. 3.3. Petri nets enclose a solid mathematical base, enabling models to be formally analyzed. Additionally, Petri nets are one of the meta-models that offer more extensions, allowing an enormous variety of utilizations, from system specification and performance analysis to system synthesis and implementation. Several Petri net extensions include powerful semantic mechanisms, such as hierarchical approaches and object orientation, allowing to cope with complex system modeling [24, 31]. There are some languages that directly support some of the existing Petri net extensions [25, 28].

3.3.2 Activity Oriented Meta-Models

Activity oriented meta-models allow modeling a system as a set of activities related by data or by execution dependencies. These meta-models are well suited to model systems where data are affected by a sequence of transformations at a constant rate. Data flow diagrams (DFDs) and flowcharts are two examples of activity oriented meta-models.

A DFD [10], also known as a "data flow graph" (DFG), consists in a set of interconnected activities or processes with arcs representing the data flow among them. DFDs support hierarchy, since each activity can be further detailed by another DFD. DFDs can not express temporal behavior, or action control. UML does not have any kind of diagram based on this meta-model [12]. Neither UML's use case diagrams nor UML's activity diagrams are DFDs, although some developers argue that there are some graphical resemblances.

Flowcharts [9], also known as "control flow graphs" (CFGs), model control flow among activities. While in FSMs transitions are activated by external events, in flowcharts transitions are activated as soon as an activity is complete. This meta-model is suitable for modeling systems with well defined activities and that do not depend on external stimulus, allowing the representation of sequences of activities related by control flow. UML's activity diagrams are essentially based on this meta-model. However, fork and join primitives of activity diagrams are inspired by Petri net transitions.

3.3.3 Structure Oriented Meta-Models

Structure oriented meta-models allow the description of system physical modules and their interconnections. These meta-models are dedicated to the characterization of the physical composition of a system, instead of its functionality. Block diagrams, also called "component-connectivity diagrams" (CCDs), are the most frequently used structure oriented meta-model. UML's deployment and component diagrams are based on this meta-model.

3.3.4 Data Oriented Meta-Models

Data oriented meta-models allow modeling a system as a collection of data related by some kind of attribute. These meta-models dedicate more importance to the organization of data than to the system functionality. UML does not have any kind of diagram exclusively based on these meta-models, since it favors object oriented systems and does not promote the usage of diagrams mainly dedicated to data modeling. Nevertheless, it is possible to argue that UML's class diagrams are partially data oriented meta-models.

Data oriented meta-models are, typically, used within methodologies based on the traditional structure analysis and design techniques [46]. Entity relationship diagrams (ERDs) and Jackson's structured diagrams (JSDs) are two examples of

data oriented meta-models. ERDs [6] describe a system as a collection of entities and the existing relationships among them. Each entity corresponds to a unique type of data with one or more specific attributes. This meta-model is useful when developers want to organize complex relationships between different data types. ERDs cannot model functional or temporal characteristics.

JSDs [42] model the structure of each data type, through subtype decomposition. Decomposition is performed in a tree structure in which the leaves correspond to the basic data types and the other nodes to the composite data, obtained through various operations such as composition (AND), selection (OR), and iteration (*). While ERDs are suitable to model different data entities with complex inter-relations, JSDs are adequate to model complex data structures. The limitations of JSDs are similar to the ones referred for ERDs.

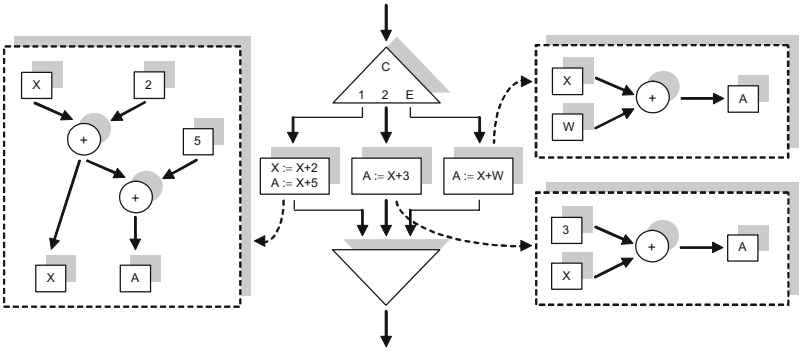

Fig. 3.4 Example of a control/data flow graph

3.3.5 Heterogeneous Meta-Models

Heterogeneous meta-models allow the usage, in the same system representation, of several characteristics from different meta-models, namely the four categories described before. These meta-models are a good solution when relatively complex systems must be modeled. Control/data flow graphs (CDFGs), object process diagrams (OPDs) and program state machines (PSMs) are examples of heterogeneous meta-models.

CDFGs [16] embody DFDs (to model data flow between system activities) and flowcharts (to impose the sequence of DFDs execution). CDFGs succeed in modeling, in a single representation, data dependencies and system control sequence, simultaneously benefiting from DFDs and flowcharts advantages; see Fig. 3.4.

Within the Object Process Methodology (OPM), the combined usage of objects and processes is recommended [11]. An OPD can include both processes and objects, which are viewed as complementary entities that together describe the structure and behavior of the system. Objects are persistent entities and processes transform the objects by generating, consuming or affecting them. In addition, states are also integrated in OPDs to describe the objects.

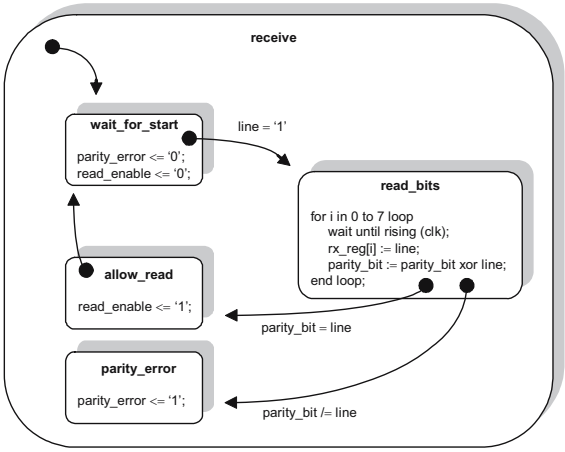

Fig. 3.5 Example of a PSM model specified in the SpecCharts language

PSMs [33] allow the integration of HCFSMs with a textual programming language. This meta-model basically consists in a hierarchy of program states, in which each state represents a distinct computation mode. At any instant, only a subset of the program states is simultaneously executing their computations. PSMs are more powerful than HCFSMs to model systems that possess complex data structures, since they are able to incorporate, in a unique model, data, activities and states. HCFSMs and programming languages delimit the two opposite extremes of using PSMs. A program may be considered a PSM with only one specified state, and a HCFSM may be viewed as a PSM in which none of their states possess descriptions in the programming language. SpecCharts is a representation language for the PSM meta-model; see Fig. 3.5.

If PSMs are considered a heterogeneous meta-model, it is also acceptable to consider programming languages as a meta-model themselves. There exits a considerable number of developers that make use of programming languages to specify systems, usually, their behavior and data structures. This approach to specification imposes a considerable amount of design and implementation decisions at the analysis phase, which can have an undesired effect on the specifications.

Programming languages allow the modeling of data structures, activities and control. The modeling "style" imposed by a particular programming language is called paradigm in computer science terminology. The meta-model behind a programming language is its paradigm and not the language itself. Programming languages should be considered representation languages at the implementation level.

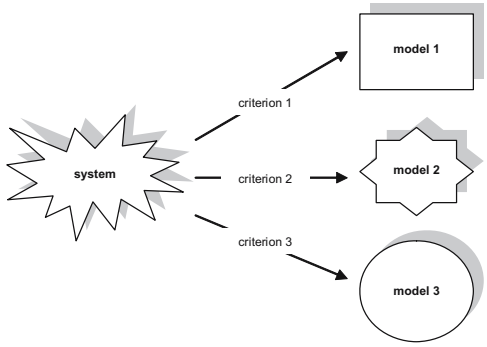

Fig. 3.6 The multiple view approach

Historically, there are two different meta-models (paradigms) for programming languages: imperative and declarative. The imperative paradigm (where C and Pascal are included) follows von Neumann's computational model, since it adopts the sequential execution of the computing primitives. The declarative paradigm (where Lisp and Prolog are included) does not define an explicit order of execution of the primitives, focusing in defining the target of computation, through functions and logic rules declaration. More recently, the object oriented paradigm has emerged, which is based on the heterogeneous object oriented meta-model. Object oriented meta-models evolved from data oriented meta-models, being characterized by its tendency in describing the system as a collection of cooperating objects. Each object consists in a data collection and in operations to transform its data. This meta-model supports data abstraction (information hiding), through encapsulation of data in each object, making data invisible to other objects. They can easily represent concurrency, since each object coexists with the others and can potentially execute its tasks in parallel with tasks in other objects.

3.3.6 Multiple-View Approach

With the increasing complexity of systems, the use of different meta-models to represent different kinds of system characteristics is becoming a common practice. A system is modeled by a set of different models, each one corresponding to a different view of the system, devoted to represent a well delimited set of the system characteristics, see Fig. 3.6, where the criteria shown are related to the characteristics each view is intended to capture. This multiple view approach does not correspond to the usage of a heterogeneous meta-model, since the information in different views may not be explicitly related through common information structures. On the contrary, in a heterogeneous meta-model the different views must hold common information structures within a unique integrated representation. UML notation permits the adoption of multiple view approaches.

Multiple view modeling can adopt orthogonal views: (1) the function view is responsible for representing the processes of the system and UML's activity diagrams can be used to support this view; (2) the data view defines system information, that can be supported by UML's class diagrams; (3) the control view characterizes the system dynamic behavior that can be described by UML's state diagrams. Several authors have defined different multiple view approaches where views are vehicles for separation of concerns [1, 14, 27, 29].

3.4 Specification Methodology

Formal description, comparison, and construction of methods and techniques for systems development are the main goals of the method engineering community [19]. Meta-models of the development process are also called "meta-process models" and meta-models of the development products, or deliverables, are called 'meta-data models' (in this chapter we call these just "meta-models"). Some well known approaches to the method engineering are: ISO/IEC 12207 [22], OPEN [15] and PIE [8].

The act of defining our own specification methodology is called "situational method engineering" [44] and it is in this context that it is important to take into consideration the following three key issues [39]: specification language, complexity control, and model continuity.

3.4.1 Specification Language

Specification languages must allow the representation of a particular system view, without ambiguities. This is the main purpose of specification languages, and their relation with the meta-models has already been discussed. Additionally, specification languages must offer support for analyzing and reasoning about the specification. The available analysis mechanisms depend on the specification language itself. However, there are essentially two different kinds of mechanisms: formal analysis and specification execution. Formal analysis is important to verify if a specification is incoherent, but its existence is only possible if the specification language owns a solid mathematical base. Executable specifications allow an early testing of system prototypes for requirements validation, rendering a more robust and understandable specification process.

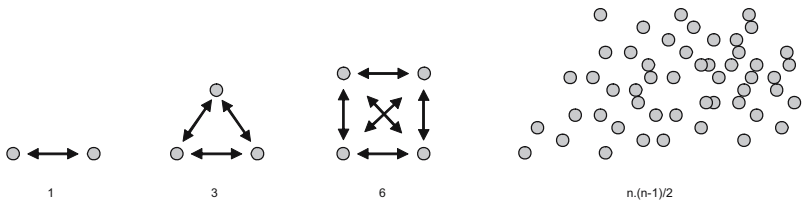

Fig. 3.7 Complexity

3.4.2 Complexity Control

The control of the complexity of the specification process can be carried out within two different dimensions: representational complexity and development complexity. The complexity of a system does not only depend on the cardinality of its parts, but mainly on the way its parts interact among them; see Fig. 3.7, where systems are represented by circles and interactions by arrows.

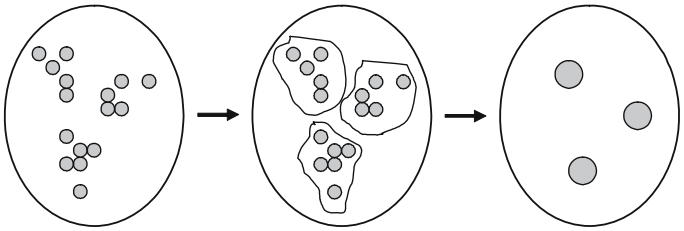

Fig. 3.8 Abstraction levels

The first dimension of complexity control refers to the representational complexity. It essentially depends on the specification language and, if correctly managed, permits concise and comprehensible specifications to be obtained. Complexity control at the representation level can be achieved by making use of three different techniques: hierarchy, orthogonality, and representation scheme. Developers must be able to decide the appropriate abstraction level to be used. Typically, the adoption of higher levels of abstraction improves the understanding of the system as a whole, while details are being hidden.Model hierarchization corresponds to grouping similar (structural or behavioral) system parts together into a new element that represents the group; see Fig. 3.8. Model orthogonalization consists in describing a set of system behaviors independently from each other (whenever possible). In what concerns the representation scheme, complexity control effort can decide either for textual representations or for graphical representations. Graphical representation schemes imply visual formalisms where both syntactic and semantic interpretations are assigned to graphical entities. Graphical ap-

proaches are usually easier to understand than textual ones and thus improve the readability and the understandability of system view. UML adopts a graphical approach.

The second dimension of complexity control (development complexity) refers to the control of the evolution of the system specification from initial conceptualization of requirements. This control can be accomplished by deferring certain details to the next phases of system development and by adopting different specification evolutions throughout the specification process (top-down, bottom-up or middle-out).

3.4.3 Model Continuity

Models obtained in the initial phases of the development must be persistent, avoiding their rewriting at each step. To support design and implementation methodologies, this model continuity concern must assure conformity in models evolution throughout the whole development process. This is possible by allowing models to be refined through the inclusion of new behavioral and structural attributes acquired along the design and implementation phases; see Fig. 3.9.

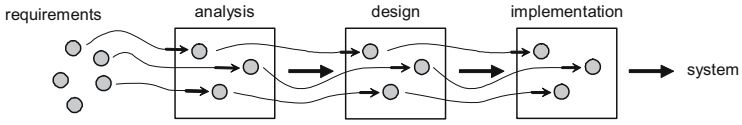

Fig. 3.9 Model continuity

The first model must be independent of implementation, allowing developers to focus in the system behavioral modeling. When constructing the first specification, design or implementation decisions and unnecessary restrictions should be avoid. Within a full model continuity approach, it is desirable that the automatic synthesis of the solution is completely based on the system specification. This synthesis technique, carried out at the system level, is not yet sufficiently efficient. It is usually based on the structural characteristics of the specifications and it has the disadvantage of limiting the design space exploration, generating non-optimal solutions for system implementation.

3.4.4 Non-Functional Requirements

Non-functional requirements limit the design space exploration, since they typically impose, at early stages of development, particular design and implementation solutions. This kind of requirements can be classified into three different groups: design objectives, design decisions, and design constraints.

Design objectives are related to general requirements of qualitative system performance. Typical design objectives appear in the form of “it must be as fast as possible,” “it must be cheap” or “it must be easy to adapt.” Although, these design

objectives are not really requirements, they can be transformed into design constraints if some metrics can be devised. Otherwise, design objectives should only be used to select amongst functional equivalent alternatives, when there is no firmer criterion for the decision; see Chap. 12 for further details on decision support in requirements engineering.

Design decisions can be related, for example, to the inclusion of the system in a given family of commercial products or with the incorporation into a bigger product. These non-functional requirements can affect the technological decisions or interfere with the functionality of the system, so they should always be questioned and justified. UML's OCL (Object Constraint Language) can be used to describe architectural or functional design decisions. Design constraints include, for example, performance, reliability, cost and size. Timing requirements can be classified as reply time, repetition rate and correlation time. This kind of non-functional decisions is typically quantifiable and syntactically incorporated in the system models as tagged values or object stamps. UML's sequence diagrams can support the inscription of timing and performance requirements.

In [7, 36] non-functional requirements are thoroughly treated both on how to discover and on how to specify them.

3.5 Requirements Transformation

The problem of obtaining system requirements models from user requirements that can be directly used within the design phase is not simple and easy and faces several difficulties [26]. Generically, it involves several decisions that can not be made by a method or a tool, due to the natural discontinuity between functional and structural models. Holland and Lieberherr consider that the identification of objects and the description of the relationships between them are two of the three challenges of object oriented design in the construction of object oriented models [20].

There are many authors that propose solutions to tackle this problem, namely by guiding the transformation of use case models into object/class models [2, 3, 23, 37]. Some approaches [30, 38] propose a use case rationale based on goal identification and can be used to better support the transition for the architectural design issues. However, they lack an explicit scenario framework for capturing the semantic intentionality of each use case. This could be incorporated by adopting some scenario based requirements engineering techniques, such as those suggested in [43, 45]. See Chap. 5 for further details on requirements interdependencies.

In this section, we describe an approach for defining the system objects based on use cases and their respective textual descriptions. The strategy uses the object categories (interface, data and control) defined in [23] and incorporates some mechanisms that allow each object to be related to the use cases that gave origin to it. Due to the relatively weak support of UML 1.5 to component based design, UML object concept was chosen to represent system level entities or components.

UML 2.0 was not used here since its final approval as an ISO standard was not taken at the time of writing.

3.5.1 User Requirements Modeling

The identification of the system components requires the definition of a model to capture the system functionalities offered to its users. Use cases are one of the most suitable techniques for that purpose, since they are simple and easy to read. In fact, they only include three main concepts (use cases, actors and relations). This low number of concepts is a fundamental characteristic for involving non-technical stakeholders in the requirements capture process.

Although use cases are used in several object oriented projects, they do not hold any intrinsic characteristic that can be classified as "pure" object oriented. However, there is a large consensus on the recognition that use cases are a proper technique for object oriented projects [4], namely for discovering (and later specifying) the behavior of the system, during the analysis phase. This is also highlighted by the fact that use cases are part of UML. Thus, adopting use cases for user requirements is undoubtedly a valid technique, but poses the problem related to the transformation of use cases into objects or components.

The requirements for the case study used in this chapter were acquired using requirements engineering techniques, and the end-result was a collection of artifacts, including UML diagrams. Some of the artifacts are presented in Fig. 3.10-3.11. After identifying all the use cases of the system, the next step is to describe their behavior. There are some alternatives for describing use cases, namely informal text, numbered steps with pre- and post-conditions pseudo code and activity diagrams [40]. As an example, the description of the top level use case {U0a.1} with informal text is presented. Similar descriptions were created for the other top-level use cases.

{U0a.1} send alert: Send domain alert or disseminating domain information to the users informing of domain related events and situations or unexpected domain situations that are happening in the region. Only users that have previously subscribed this e-service will receive the alert messages (subscription made via {U0a.4} user profile subscription). This is an asynchronous e-service. If technically possible, the system acquires user context raw information (location, time, etc) from external context sources. Also, a contextualization process will assist the system in making the level of granularity of the information adequate to the geographic location of the user context (geographic location context, time context and activity context). Examples: an alert of a dangerous hole in a street should only be sent to the users geographically located in that street; an alert of a street obstructed should be sent to the users geographically located in that street or in any of the incident streets; an alert of weather storm should be sent to all the users in the region. The information associated to the alert should always be up-to-date and match the user-specific request, excluding any extra information or undesired advertisements. For those users that require personalized information, a subscription must be made via {U0a.4} user profile and e-service subscription.

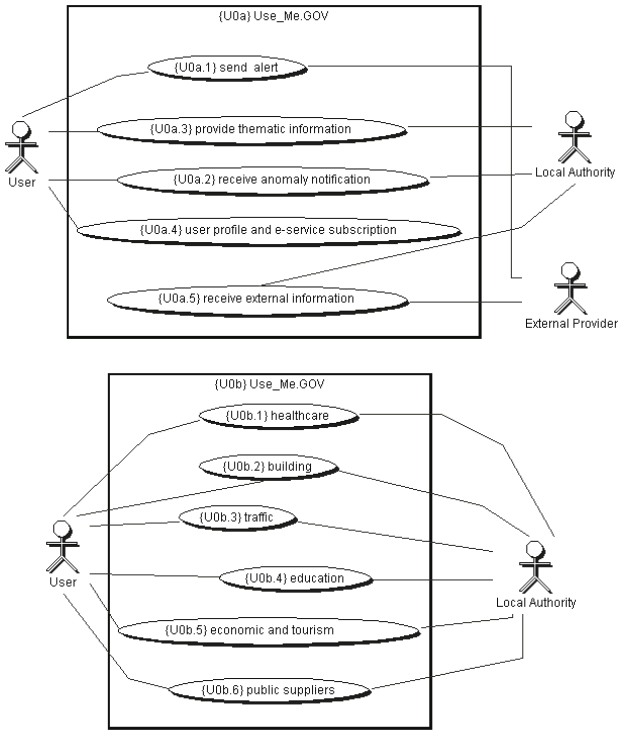

Fig. 3.10 UML top level use case diagram according to two orthogonal criteria; top: functionality criterion; bottom: domain criterion

3.5.2 4SRS Technique

Transforming use cases into architectural models representing system requirements is a difficult task. A technique called 4 step rule set (4SRS) was proposed to help with that task in [13]. The 4SRS technique is organized as four steps to transform use cases into objects: object creation (step 1), object elimination (step 2), object packaging and aggregation (step 3) and object association (step 4).

In step 1 (object creation), each use case must be transformed into three objects (one interface, one data, and one control). Each object receives the reference of its respective use case appended with the suffix (i, d, c) that indicates the object's category (in this approach, object references start with an "O"). This is a fully "automatic" step, since there is no need to any kind of particular decisions or rationale for the specific context of each use case. From this step on, there are only objects as design entities. Use cases are still used in the following steps to allow the introduction of requirements into the object model.

In step 2 (object elimination), it must be decided which of the three objects must be maintained to fully represent, in computational terms, the use case, taking into account the whole system and not each use case in isolation. These decisions must be based on the textual description for each use case. This step aims at deciding which of the objects created in step 1 must be kept in the object model. It also eliminates redundancy in the user requirements elicitation and detects missing requirements. Object elimination is the most important step of the 4SRS technique, since the definitive system level entities are decided here. To cope with the complexity of the step, it has been decomposed into seven micro-steps: use case identification (micro-step 2i), local elimination (micro-step 2ii), object naming (micro-step 2iii), object description (micro-step 2iv), object representation (micro-step 2v), global elimination (micro-step 2vi) and object renaming (micro-step 2vii). The description of these micro-steps is out of the scope of this chapter.

In step 3 (object packaging and aggregation), the remaining objects (those that were maintained after step 2) for which there is an advantage in being treated in a unified way should give origin to aggregations or packages of semantically consistent objects. This step supports the construction of a truly coherent object model, since it introduces an additional semantic layer at a higher abstraction level, that works as a "functional glue" for the objects.

Packaging is technique that can introduce a very light semantic cohesion among the objects. This cohesion can be easily reversed within the design phase whenever needed. This means packaging can be flexibly used to obtain more comprehensive and understandable object models. In the opposite way, aggregation imposes a strong semantic cohesion among the objects. The level of cohesion in aggregations is more difficult to reverse in subsequent stages, which suggests a more scrupulous approach in using this kind of functional glue. Thus, aggregation should only be used when it is explicitly assumed that the set of considered objects is affected by a conscious design decision. Typically, aggregation is used when there is a part of the system that constitutes a legacy subsystem, or when the design has a pre-defined reference architecture that constricts the object model.

Step 4 (object association) of the 4SRS technique supports the introduction of associations in the object model, completely based on the information from the use case model and generated in micro-step 2i. Regarding the information in the use case model, if the textual descriptions of use cases possess hints on the kind of sequences use cases are inserted in, this information must be used to include associations in the object model.

Alternatively, the use case model can include other kinds of information to support associations, when there are UML relations between use cases. As an example, use case {U0a.1.1} «uses» use case {U0a.1.2}, which justifies the association between objects {O0a.1.1.d} and {O0a.1.2.c}, and between objects {O0a.1.1.i} and {O0a.1.2.d}; see Fig. 3.12.

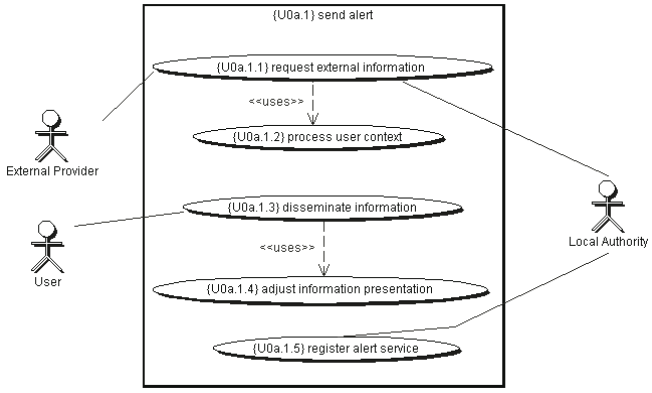

Fig. 3.11 Refinement of UML use case {U0a.1}

3.5.3 System Requirements Modeling

The system architectural model expresses the system requirements, but also an informal description of the objects. 4SRS helps to define a logical architecture for the system by capturing all its functional requirements and its non-functional intentions. The former gives origin to textual descriptions for each object in the model and the later has been classified as design decisions and design constraints. Design objectives are not allowed at system requirements models generated by the 4SRS technique.

The generated object model shows how significant properties of a system are distributed across its constituent parts. The 4SRS technique generates a raw object diagram that identifies the system level entities, their responsibilities and the relationships among them. Its purpose is to direct attention at an appropriate decomposition of the system without delving into details. Each one of the used packages defines one different decomposition region that contains several tightly semantically connected objects. Within the next design phases, these packages must be further specified concerning its architectural structure, by using design patterns.

The resulting raw object diagram can be used in the following development phases to support the definition of specific sub-projects, by using collapsing and filtering techniques. These techniques allow the redefinitions of the system boundary, giving origin, for instance, to the database project, services formalization, or platform pattern analysis. Fig. 3.12 shows the collapsed object diagram that was obtained from the raw object diagram by hiding packages details. Therefore, associations appear at a higher level of abstraction and the resulting object diagram is more readable.

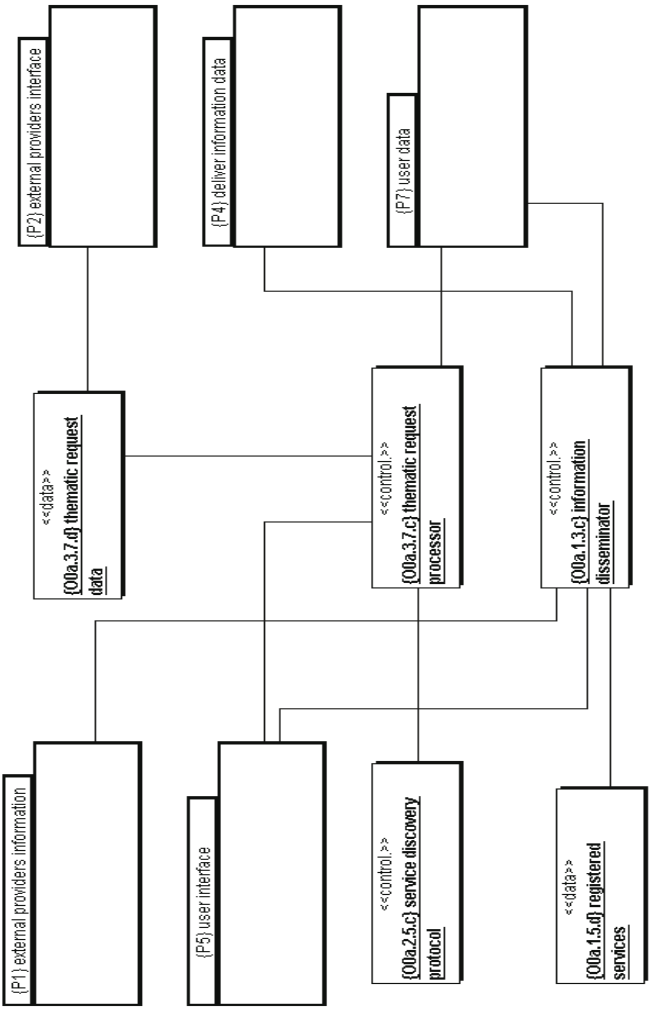

Fig. 3.12 Collapsed UML object diagram representing system requirements

3.6 Conclusion

The correct derivation of system requirements from user requirements is an important topic in requirements engineering research. This activity assures that the design phase is based on the effective clients' needs without any misjudgment arbi-

trarily introduced by the developers during the process of system requirements specification. One approach to support this derivation is by transforming user requirements models into system requirements models, by manipulating the corresponding specifications. User requirements are, typically, described in natural language and with informal diagrams, at a relatively low level of detail and are focused in the problem domain. System requirements comprise abstract models of the system, at a relatively high level of detail, and constitute the first system representation to be used at the beginning of the design phase.

This chapter deals with the characteristics of different modeling techniques for the specification of systems requirements. It presents various classes of modeling and specification techniques that can be used in different circumstances during development projects. Here, meta-models play an important role, since they define the semantic capability of the modeling views to adopt for a given system. The chapter ends with a brief description of a heuristic based approach to support the transformation of user into system requirements. This transformational approach shows that model continuity is a key issue and highlights the importance of having a well defined process to relate, map and transform requirements models.

The topics presented in this chapter emphasize the fact that system design is a highly abstract task that focuses on the functional and non-functional requirements of computer-based systems. Both system designers and requirements engineers benefit from a model based approach to requirements specification to allow the correct evolution of system representations during development projects.

References

1. Ainsworth M, Cruickshank AH, Groves LG, Wallis PJL (1994) Viewpoint specification and Z. Information Software Technology, , February 36: 43–51
2. Back RJ, Petre L, Porres I (1999) Analyzing UML use cases as contracts: Beyond the standard. In: Proceedings of 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, USA, pp.518–33
3. Becker LB, Pereira CE, Dias OP, Teixeira IM, Teixeira JP (2000) MOSYS: A methodology for automatic object identification from system specification. In: Proceedings of 3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000), Newport Beach, CA, USA, pp.198–201
4. Booch G (1996) Best of booch: Designing strategies for object technology. SIGS, New York, NY, USA
5. Calvez JP (1996) A system specification model and method. In: High Level System Modeling: Specification and Design Methodologies. Waxman R, Bergé JM, Levia O, Rouillard J. (Eds.), Kluwer Academic, Dordrecht, The Netherlands
6. Chen PS (1977) The entity relationship approach to logical data base design. Q.E.D. Information Sciences, Wellesley, MA, USA
7. Chung L, Nixon B, Yu E, Mylopoulos J (2000) Non-functional requirements in software engineering. Kluwer Academic, Boston, MA, USA
8. Cunin PY, Greenwood R, Francou L, Robertson I, Warboys B (2001) The PIE methodology: Concept and application. In: Proceedings of 8th European Workshop on Software Process Technology, Witten, Germany, pp.3–26

9. Davis WS (1983) Tools and techniques for structured systems analysis and design. Addison-Wesley, Reading, MA, USA
10. De Marco T (1979) Structured analysis and system specification. Yourdon Press, New York, NY, USA
11. Dori D (2002) Object-process methodology: A holistic systems paradigm, Springer, Berlin, Germany
12. Fernandes JM, Lilius J (2004) Functional and object-oriented views in embedded software modeling. In: Proceedings of 11th International Conference on the Engineering of Computer Based Systems (ECBS 2004), Brno, Czech Rep., pp.378−87, IEEE CS Press, May
13. Fernandes JM, Machado RJ (2001) From use cases to objects: An industrial information systems case study analysis. In: Proceedings of 7th International Conference on Object-Oriented Information Systems (OOIS'01), Calgary, Canada, August pp.319−28
14. Finkelstein A, Kramer J, Nuseibeh B, Finkelstein L, Goedicke M (1992) Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering, 2: 31−57
15. Firesmith D, Henderson-Sellers B (2002) The OPEN process framework: An introduction. Addison-Wesley, Harlow, UK
16. Gajski D, Dutt N, Wu A, Lin S (1992) High level synthesis: Introduction to chip and system design, Kluwer Academic, Boston, MA, USA
17. Gajski D, Vahid F, Narayan S, Gong J (1994) Specification and design of embedded systems. Prentice Hall, Englewood Cliffs, NJ, USA
18. Harel D (1988) On visual formalisms. Communications of the ACM, 31(5): 514−30
19. Henderson-Sellers B (2003) Method engineering for OO systems development. Communications of the ACM, 46(10): 73−8
20. Holland IM, Lieberherr KJ (1996) Object-oriented design. ACM Computing Surveys, 28(1): 273−5
21. IEEE (1990) IEEE Standard glossary of software engineering terminology, 610.12-1990
22. International Standards Organization (1995) Information technology: Software lifecycle processes (ISO/IEC12207). Geneva, Switzerland
23. Jacobson I, Christerson M, Jonsson P, Overgaard GÄ (1992) Object-oriented software engineering: A use case driven approach. Addison Wesley, Reading, MA, USA
24. Jensen K (1997) Colored Petri nets: Basic concepts, analysis methods and practical use. Vol.1, Basic concepts. Monographs in Theoretical Computer Science, Springer, New York, NY, USA
25. Jensen K, Christensen S, Huber P, Holla M (1992) Design/CPN: A reference manual. MetaSoftware Corporation
26. Kaindl H (1999) Difficulties in the transition from OO analysis to design. IEEE Software, 16(5): 94−102
27. Kotonya G, Sommerville I (1992) Viewpoints for requirements definition. Software Engineering Journal, 7(6): 375−87
28. Lakos C, Keen C (1994) LOOPN++: A new language for object oriented Petri Nets. In: Proceedings of European Simulation Multi-conference, Barcelona, Spain, pp.369 74, Society for Computer Simulation
29. Leite JCSP, Freeman PA (1991) Requirements validation through viewpoint resolution, IEEE Transactions on Software Engineering, 12(12): 1253−1269

30. Liang Y (2003) From use cases to classes: A way of building object model with UML. Information and Software Technology, 45: 83–93
31. Machado RJ, Fernandes JM (2001) A Petri Net meta-model to develop software components for Embedded Systems. In: Proceedings of 2nd IEEE International Conference on Application of Concurrency to System Design (ACSD'01), Newcastle, UK, pp.113–22, IEEE CS Press
32. Moore EF (1964) Sequential machines: Selected papers. Addison Wesley, Reading, MA, USA
33. Narayan S, Vahid F, Gajski D (1991) System specification and synthesis with the speccharts language. In: Proceedings of International Conference on Computer-Aided Design (ICCAD '91), Santa Clara, CA, USA, pp.266–9, IEEE CS Press
34. Peterson J (1981) Petri Net theory and the modeling of systems. Prentice Hall, Upper Saddle River, NJ, USA
35. Reisig W (1985) Petri Nets: An introduction, EATCS Monographs on Theoretical Computer Science, Vol.4, Springer, Berlin, Germany
36. Robertson S, Robertson J (1999) Mastering the requirements process, Addison Wesley, Reading, MA, USA
37. Rosenberg D, Scott K (1999) Use case driven object modeling with UML: A practical approach. Addison Wesley, Reading, MA, USA
38. Saeki M, Kaiya H (2003) Transformation based approach for weaving use case models in aspect-oriented requirements analysis. 4th Workshop on AOSD Modeling with UML, within the UML 2003 Conference, San Francisco, CA, USA, October
39. Sarkar A, Waxman R, Cohoon J (1995) Specification modeling methodologies for reactive systems design. In: High Level System Modeling: Specification Languages. Bergé JM, Levia O, Rouillard J. (Eds.), Kluwer Academic, Dordrecht, The Netherlands
40. Schneider G, Winters JP (1998) Applying use cases: A practical guide. Addison Wesley, Reading, MA, USA
41. Stevens R, Brook P, Jackson K, Arnold S (1998) Systems engineering: Coping with complexity. Prentice Hall Europe, Hertfordshire, UK
42. Sutcliffe A (1988) Jackson system development. Prentice Hall, Hertfordshire, UK
43. Sutcliffe A, Maiden M, Minocha S, Manuel D (1998) Supporting scenario-based requirements engineering. IEEE Transactions on Software Engineering, 24(12): 1072-88
44. ter Hofstede AHM, Verhoef TF (1997) On the feasibility of situational method engineering. Information Systems, 22(6/7): 401-22
45. van Lamsweerde A, Willemet L (1998) Inferring declarative requirements specifications from operational scenarios. IEEE Transactions on Software Engineering, 24(12): 1089–114
46. Yourdon E, Constantine L (1978) Structured design: Fundamentals of a discipline of computer program and systems design. Yourdon Press, New York, NY, USA

Author Biography

Ricardo J. Machado is an assistant professor of Software Engineering and coordinator of the Software Engineering and Management Research Group (SEMAG) at the Department of Information Systems, Universidade do Minho (Guimarães, Portugal). He holds a PhD and an MSc degrees in Informatics and Computer Engi-

neering (both from U.Minho), and a DEng degree in Electronics and Computer Engineering (from FEUP). He is the president of the Portuguese technical committee responsible for analyzing the documents produced by JTC1/SC7 from ISO/IEC and by TC311 from CEN/CENELEC in the software and system engineering domain, and he is one of the founding members of IFIP WG10.3 SIG-ES special interest group. He is a regular scientific reviewer of IEEE Transactions on CAD and IEEE Transactions on Software Engineering. He acted as general chair of ACSD'03 conference, as co-organizer of MOMPES series workshops, and has been appointed as general chair of DIPES'06 conference. He has also served as a PC member of ETFA'03, ACSD'03/'04/'05, MOMPES'04, and INDIN'05. His current research interests include software engineering, embedded software, and pervasive information systems.

Isabel Ramos holds a doctorate degree in Information Technologies and Systems, specialization in Information Systems Engineering and Management, since 2001. She also holds a master degree in Informatics for management. Isabel Ramos is an assistant professor at the Department of Information Systems, Universidade do Minho (Guimarães, Portugal). She is a researcher in the Algoritmi Research Center and coordinates the interest group in Knowledge Management of the department. She is also responsible for the Requirements Engineering modules in the Master course on Information Systems. She integrates the steering committee of a Master on Business Information. Isabel is author of several scientific publications presented at international conferences and published in scientific and technical journals. Her main areas of interest are requirements engineering, knowledge management, organizational theory, sociology of knowledge, history of science, research methodology.

João M. Fernandes is an assistant professor at the Department of Informatics, Universidade do Minho (Braga, Portugal). He received a DEng degree in Informatics and Systems Engineering in 1991, MSc degree in Computer Science in 1994, and a PhD degree in Computer Engineering in 2000, all from U. Minho. From Sep/2002 until Feb/2003, he was a post-doctoral researcher at the TUCS Embedded Systems Laboratory (Turku, Finland). He is a (co-)author of several scientific publications with peer revision on international conferences, journals and chapters of books. He has already served as a scientific reviewer for an Addison-Wesley book, for several international conferences, and for IEEE, Elsevier, and Springer international journals. He has also served as a member of the Program and Organizing Committees of international workshops and conferences, namely DSOA 2004, CPN 2004, MOMPES 2004, ETFA 2003, and ACSD 2003. His research interests focus on embedded software, hardware/software co-design, methodologies for system development, software modeling, software process and management, and history of computing.