1

The Science of Software Development

Software engineering is a discipline concerned with developing correct software products. We begin by introducing paradigms relied upon in the software engineering community to model the software development process. These paradigms describe the path from an abstract description of the problem to be solved by a software system (the specification) to its implementation (the program). In this book, we subscribe to a view of the software development process maintaining that the specification is a formal theory and that the program is obtained from this theory by applying correctness-preserving transformations.

1.1 Software Engineering as Theory Construction

We want to liken the activity of the software engineer to the job of a scientist. Part of the task of the scientist is to describe phenomena. Scientists do so by formulating theories, i.e., sets of sentences of an appropriate language, intended to capture that part of the world under consideration.[1] What makes a description of a phenomenon a good scientific theory?

Consider the following description of the motion of a (large) pendulum (with only a small angle of deviation from the resting position). It is well known that the motion of such a pendulum can be approximated by a one-dimensional linear harmonic oscillator. A one-dimensional linear harmonic oscillator is a system consisting of a single mass constrained to move in one dimension only. Taking its rest position as origin, the total energy of the system is

$$H = T + V = \frac{p^2}{2m} + \frac{1}{2}kx^2, \text{ where } p = m\frac{dx}{dt}.$$

[1] Obviously scientists do not only describe the world, but their theories should also explain the phenomena. Various accounts have been given of what makes a certain description of a phenomenon an explanation of that phenomenon. Suggestions range from the claim that an explanation of a phenomenon has to license a prediction of the phenomenon, to the claim that to explain a phenomenon is to derive the sentence describing it from the theory and sentences stating initial conditions. This issue will not concern us here. The claim that any scientific theory will be a description of the phenomenon is uncontroversial.

The development of the system over time is given by solutions to the following equations of motion:

$$\frac{dx}{dt} = \frac{\partial H}{\partial p'} \text{ and } \frac{dp}{dt} = -\frac{\partial H}{\partial x}.$$

Let these equations define a theory Θ. Surely, a theory will not be good (under any reasonable interpretation of "good") unless its theorems are true. The theorems are true if there is a model that makes them true. Assume standard interpretations of theoretical terms like mass, position, etc., then let the model that makes Θ true be the linear harmonic oscillator.[2] The linear harmonic oscillator exhibits all the characteristics specified in the theory Θ. For example, the mass of the linear harmonic oscillator exhibits perfectly sinusoidal motion.

A real-life pendulum is *not* a linear harmonic oscillator since it does not exhibit the exact behavior specified by Θ (for instance, its motion exhibits dampening). Thus, such a pendulum is not a model for the theory Θ. However, we think the pendulum can be described by Θ. What makes Θ a good theory of the pendulum is the existence of a model (namely the linear harmonic oscillator) approximately isomorphic to it. By approximate isomorphism, we mean that the two models are isomorphic in specified respects to a specified degree. For example, we might claim that all quantities in the pendulum system remain within ten percent of the quantities in the linear harmonic oscillator for the first minute of operation. When a theory adequately describes a phenomenon we call the theory *"valid."*

Definition 1.1 *A theory Θ of a phenomenon is valid if and only if there exists a model for Θ which is isomorphic to the subset of the domain containing the phenomenon in respects and to degrees specified by Θ.*

We want to claim that the software engineer is involved in a similar process. The software engineer also wants to describe a domain through a theory (the software system). The scientist's descriptions generally constitute discoveries whereas the software engineer's descriptions do not make any claim to novelty. This difference is one of purpose, but not a difference in the semantics of descriptions. For the software engineer, the theory (software system) is not formulated in natural language or set theory but in a programming language. Just as for the theories a scientist constructs, the notion of praise for a software system is that of validity.

Example 1.1 Consider a software system managing all the student records of Chicago State University. If the software system states, there exists a student A.N. Other with social security number 123-45-6789, then the system will not be correct unless there is indeed a student attending Chicago State University which can be mapped onto the object that is the interpretation of A.N. Other, having a social security number

[2]More precisely, the linear harmonic oscillator is a class of models, obtained by specifying unique values for all parameters of the system.

that can be mapped onto 123-45-6789. Conversely, for any student the model of the software system should contain an object that can be mapped onto the student.

Software engineering studies the development process of software systems. It mainly studies methodologies to produce theories describing a domain in a programming language.

Construction of a theory (a software system) begins by identifying the subset of domain which is to be described. This isolated subset of the domain will be the model that bears an approximate isomorphism to a model of the software system. We will call this model the domain model for the software system.

Definition 1.2 *The domain model for a software system is a subset of the domain that the software system is intended to describe.*

The domain model is that subset of the domain which the model of the software system bears an approximate isomorphism to if the software system is a valid description of the domain. The model of the software system which is approximately isomorphic to the domain model will be termed the conceptual model of the software system.[3]

The conceptual model is described by the requirements specification (requirements theory) of the software system. It forms the starting point for the software system building process. Software engineering studies and provides methodologies for the efficient and reliable transformation of the requirements into the implemented system. It is the study of the process that advances from a theory merely describing the conceptual model to a theory which, besides describing the conceptual model, is efficiently executable (the program/software system). The view that the software building process is but a process of theory transformation has been advanced by other researchers as well [138].

1.2 Software Engineering Paradigms

In the late sixties, software engineers and system designers were faced with what was then termed the "software crisis." This crisis was the direct result of the introduction of a new generation of computer hardware. The new computers were substantially more powerful than hardware available until then, making large applications and software systems feasible. However, the strategies and skills employed in designing software for the new systems did not match the new hardware capabilities. The result

[3]This terminology deviates from standard usage. At the stage in the development of a software system at which one (the prospective users, the system designers) acquires a vague idea of what the software system should be able to do and how it should perform, one is often described as having a "conceptual model" of the software system. In this sense, "conceptual model" bears no relation to semantics; a conceptual model resembles a scale model of the software system. According to our usage, however, the conceptual model is the semantic entity which models the software system.

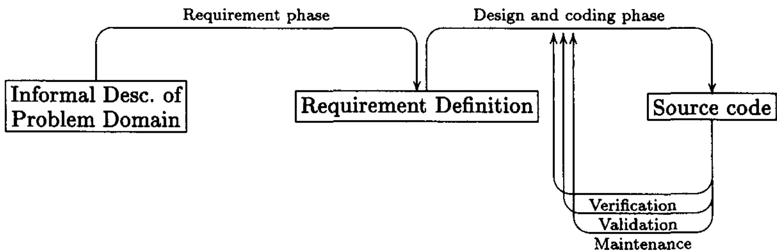

Figure 1.1. Software development life-cycle paradigm.

was delayed projects (often for years), considerable cost overruns, as well as unreliable and poorly performing applications. The need arose for new techniques and methodologies to design large software systems.

The methodology proposed as an answer to the software crisis was the Software Development Life-Cycle Paradigm (also known as the "Waterfall" Paradigm). This methodology was first proposed by Royce [273] and later modified by [33, 108, 160, 222] and others (see Fig.1.1). The life-cycle paradigm has seen many variations, amendments, and deletions, but all these share the basic assumption that software is developed in a sequence of distinct stages.[4]

- *Requirements phase* — The requirements phase (analysis phase) is concerned with understanding what the system is designed to accomplish. Starting from an informal description of the problem and the system's requirements, the designer formulates a system to solve the problem and defines it in terms of "what" it does. The definition of the requirements treats the system as a black box describing all required characteristics of the external behavior as well as the constraints on the system ("what"), but no characteristics of its internal structure that will generate that behavior ("how").

- *Design and coding phase* — The internal structure of the software system is determined in the design specification. The designer decomposes the system into abstract software components which shall produce the behavior demanded by the requirements. The interfaces between software components are frozen. Although the relationships between these abstract software components are defined, the components themselves are again treated as black boxes. In top-down fashion, a detailed design is formulated for each abstract software component in terms of simpler abstractions, until the problem is divided into many small,

[4]The U.S. Department of Defense currently requires contractors to follow the "Waterfall" model of software development (DOD Standard 2167A).

easily understood and implemented pieces. The design phase determines the optimal structure of the software components and how they interface. Ideally, the design specification should be complete enough to reduce the implementation effort to little more than a translation to a target programming language.

The design is implemented in a programming language. This involves the realization of the internal mechanism of the abstract software components as a (set of) program(s) in some programming language. Knowledge of both the design and target environment is incorporated to produce the final system software. All the physical aspects of the system are addressed during implementation.

- *Verification phase* — Information from the previous three phases is used in verifying the software system. Test plans can be derived from requirements and design specifications. Verification confirms that the software conforms to requirements and design specifications and that the code is correct.
- *Maintenance and validation phase* — Bug fixes and adaptations which result from experience with the software are activities of the maintenance phase. The software is now being used. Users will come across errors and discover where the system does not meet their expectations. Maintenance involves the correction of errors which were not discovered in early stages, improving the implementation, and enhancing the services the system renders.

During the software engineering process the system builder is confronted with the following two questions:

- Are we building the *correct system*?
- Are we *building* the system *correctly*?

Correspondingly, validation determines whether the services and functions rendered by the system (as specified in the requirements definition) comply with the customer's informal requirements. Verification determines whether the system under construction meets the requirements definition. In the terminology introduced in the previous section, validation attempts to determine whether the conceptual model is isomorphic to the domain model. Verification determines whether the conceptual model is a model of the implemented software system.

Obviously, a software engineer must be able to answer both questions affirmatively. At what stage can this be determined to be the case? One of the major criticisms of the software life-cycle paradigm applies here. Although most explications of the paradigm provide for some feedback loops between stages, validation and verification cannot be performed conclusively until after the coding stage. At this point, most of the effort has been applied. Detection of errors can send the system back into the requirements stage. The system may have to be redesigned (or even re-specified) should severe misunderstandings of the users' requirements have occurred. As indicated in Fig.1.1,

however, this is typically not the case. Often, the system is merely adjusted to meet the users' requirements by changing its source code. This results in a system with obsolete requirements specification.

Software development usually begins with an attempt to recognize and understand the users' requirements and then proceeds to implement a software system which satisfies those requirements. The users' requirements specification is formulated in a dialogue between users and system analysts. Typically, the requirements definition reflects the developers' interpretation of the users' needs. Where the communication of these needs has been distorted, either by preconceptions or by general unfamiliarity on either side, it is unlikely that misunderstandings become apparent until the users test (examine) the near-ready product. Thus, maintenance is performed at the implementation level (at the source code). At this point, the programmer has applied considerable skill and knowledge to optimize the code. But optimization spreads information: it takes advantage of what is known elsewhere in the system and substitutes complex but efficient realizations for the simple abstractions of the specification. The result being a system more difficult to understand due to increased dependencies among its components and scattering of information [15]. Correcting errors deriving from the requirements phase during software maintenance becomes exceedingly expensive [34].

The main assumption of the life-cycle paradigm is that it begins with well-understood requirements and that those requirements, and thus the design specification, are fixed. Tools supporting this paradigm usually enforce this rigidity by static type-checking and interface descriptions. In practice, though, standard techniques do not allow one to arrive at exact specifications. Often, the users do not know, and cannot anticipate their exact requirements. (The users' informal requirements description is more aspiration than specification.) Since the users have no experience on which to ground those aspirations, it is only by exploring the properties of some putative solutions that the users can find out what is really needed.

These problems are serious, and many critics have called for a replacement of the software life-cycle methodologies by new paradigms for software engineering—e.g., [4, 15, 119]. Three new methodologies emerged:

- Rapid Prototyping [5, 54, 133, 278, 302, 328].
- Executable Specification [4, 17, 53, 298, 379, 380].
- Transformational Implementation [14, 26, 58, 166, 247, 369].

Rapid prototyping is the process of building a working model of a software system (or part of a system). Using this model one can validate the requirements or perform feasibility studies. An operational specification is a system model that can be evaluated or executed to generate the behavior of a software system at an early software development stage. Transformational implementation is an approach to software

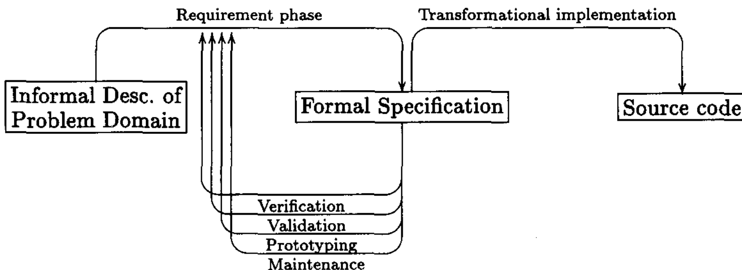

Figure 1.2. Modified software life-cycle paradigm.

development that uses automated support to apply a series of transformations that change a specification into a concrete software system. These new software engineering paradigms all strive to avoid errors in the requirements phase by demonstrating to the users the system behavior during an early stage of the software development process.

Although each of these methodologies addresses a slightly different problem with the conventional life-cycle methodology, they all share (or can share) an underlying paradigm, as depicted in Fig.1.2.

The requirements specification is a theory of the domain model (the subset of the domain under consideration), and is interpreted by the conceptual model. It states the problem to be solved by the software system including the constraints it must meet. The new paradigm insists that maintenance, verification, and validation be performed as close to the conceptual model as possible.

- *Requirements analysis* — The requirements definition of the life-cycle paradigm generally is a description of the system's behavior in natural language. It may be constrained by structure or supplemented by pictures, diagrams, etc. This type of description is psychologically distant from the conceptual model and from the final system itself. The same is true for requirements definitions in terms of automata. According to the new paradigm the end product of requirements analysis is a formal specification which is executable (prototype, operational specification). It is so possible for the users to be exposed to a "working model" of the system at a very early stage in its development. Misinterpretations can be revealed before the coding phase. The users may also gain a better understanding of their informal requirements and the conceptual model in turn resulting in a better specification.
- *Validation and verification* — The requirements specification is then subject to careful evaluation. The users can validate the system's requirements by inter-

acting with the rapid prototype/operational specification. These representations are much closer to the users' view of the final system than the diagrams and natural language descriptions of traditional requirements specifications. Misunderstandings of the users' needs, or misunderstandings that the users may have of their own needs, quickly become apparent. In addition, if the requirements specification language provides a mechanism to infer consequences of the specification these can also be compared to the domain model. Furthermore, the requirements specification may be subject to formal checks attempting to show the absence of typical sources of incorrectness.

- *Coding* — When the system designers obtain the final formal specification, they then possesses a formal description of the system which is modeled by the conceptual model. It expresses all the requirements and constraints on the system (if the conceptual model is indeed approximately isomorphic to the domain model). By adhering to the life-cycle paradigm possession of such a specification does not guarantee correctness of the final system, since errors could still be introduced in the coding phase. Implementing the specification must not change the functionality of the system. The new paradigm wants to eliminate such a possibility. The assumption of the new paradigm is that progress from specification to program is made by applying only transformations known to preserve the correctness of the involved constructs. These transformations may choose data structures, replace algorithms with equivalent algorithms, change control structures, etc., but they do not change the behavior of the system itself. The final product is then guaranteed to behave as the specification did.

- *Maintenance* — Most maintenance costs result not from system errors but from changing system requirements after the system has been put into operation. Whereas the users generally have little difficulties incorporating such changes into their conceptual model of the system, these changes usually have a massive effect at the implementation level due to the scattering of information after optimization. If it is the specification that is maintained, such changes are kept at a level before information is spread throughout the system. Similarly, the discovery of errors is far easier before implementation has taken place.

According to these paradigms, the software building process begins with the construction of a theory (requirements specification) that has a model (the conceptual model) which is isomorphic to the domain model. One then proceeds to transform this theory into another theory formulated in a programming language (source code, implemented system). Coding is then, in a sense, merely a linguistic exercise: the replacement of expressions of one theory with semantically equivalent expressions of another theory. This transformation must not affect the semantics of the theory, i.e., the transformation must be model-preserving. The testing phase of the life-cycle methodology should establish that the conceptual model does indeed model the implemented system. Whereas the life-cycle methodology has no means of ensuring that

the conceptual model is also a model of the implemented system, according to the new paradigm the transformations applied are correctness-preserving and thus also preserve the model. Note also that the verification and validation phases have moved to a much earlier stage in the lifetime of the software system.

Any software development process based on the new paradigms builds on formal languages for representing system requirements. As will be pointed out below, such a language must meet several desiderata to be effective in supporting the new software engineering paradigms.

1.3 The Path from Problem to Program

The new paradigm for software engineering presented in the previous section splits the path from problems to programs into two essentially different steps. Given the objects of the problem domain and the relations they bear to each other, plus constraints pertinent to the domain model (the subset of problem domain under consideration), we want to obtain a valid linguistic description. It is generally referred to as "specification;" we would prefer the terminology "requirements theory" but shall not depart from common usage. The minimum requirement is that this linguistic description have a model (the conceptual model) isomorphic to the domain model. In a second step, the linguistic description is transformed into the executable program. The linguistic description of the conceptual model should be (if very inefficiently) executable (operational specification, rapid prototyping) or at least formal and capable of validation.

At either of these steps, knowledge enters massively into the software engineering process. Representing the conceptual model through the requirements specification involves knowledge of the problem domain itself. The second step, from requirements specification to program, requires knowledge of programming. Due to the involvement of knowledge, the software engineering process lends itself to artificial intelligence techniques, which can be applied at either of these steps.

Current research most often considers the second step. Extensive work has been done on gradually refining a requirements specification into a program, on obtaining the explicit knowledge to facilitate these transformations, and on representing knowledge allowing these transformations to be performed automatically.[5]

Until recently, little thought has been given to the first step mentioned. Currently requirements specifications are mostly functional specifications of the program's behavior and are normally not in a form allowing them to be automatically transformed into a program.[6] Requirements specifications often do not capture the problem domain adequately, in the sense that their conceptual model is not isomorphic to the

[5] For an overview consult [130], the relevant section in the collections [266] and [301].

[6] For current work in this area, refer to [41]. Recently, a trend developed to specify nonfunctional requirements [106].

domain model. Others have shared this sentiment: *We do not primarily need a specification of what the program should do, but a theory that is valid for the problem domain*–see [17, 40, 50, 271, 370, 377].

Specifying system requirements, as currently practiced, unnecessarily mixes programming and domain knowledge: To produce the requirements specification programming knowledge is needed; to transform it into a program requires domain-knowledge beyond that conveyed by the requirements specification. We feel that these sets of knowledge are, and should be, separate:

- A valid requirements specification is a theory describing the problem domain that should be reflected in the program completely (since it has a model isomorphic to the domain model), and therefore, no further domain knowledge should be necessary to transform it into the program. The requirements specification captures all knowledge of the domain needed by the programmer and system designer.

- The requirements specification describes only the problem, not its solution, thus no knowledge of programming should be required to formulate a problem into a requirements specification. Information about algorithms or the specification of any nonabstract data types is particularly foreign to the spirit of a requirements specification.

Both claims are standard desiderata; yet two remarks are in order here.

We are not arguing that to describe the problem domain in a requirements specification no knowledge about how to solve the problem is necessary. There is a strong connection between describing and solving a problem. A formal description of a problem is already one way in which to solve that problem. Consider a formal description of the property "list y is a sorted version of list x":

$$\forall x \cdot \forall y \cdot \mathsf{sort}(x, y) \equiv \mathsf{permutation_of}(x, y) \wedge \mathsf{ordered}(y)$$

The predicate $\mathsf{sort}(x, y)$ expresses the above property and describes it in the following way: $\mathsf{sort}(x, y)$ is true if and only if y is a permutation of x, and y is ordered.[7] This obviously captures the meaning of $\mathsf{sort}(x, y)$. This description also presents us with a method of computing a sorted version of a list: look for permutations of the original list until you hit upon a permutation which is ordered. This method is, of course, hopelessly inefficient (taking time 2^n to sort a list of length n), but even this innocent looking specification is a method of solving the given problem. The following explication of $\mathsf{sort}(x, y)$ is extensionally equivalent to the one above, but presents a far better method.[8]

---

[7] We will ignore how to express permutation_of and ordered.

[8] Let a list be represented in familiar list-notation [65].

$$\begin{aligned}\forall x \cdot \forall x \cdot \mathsf{sort}(x, y) \equiv&\\ &x = [\,] \wedge y = [\,]\\ &\vee (\exists z \cdot \exists zs \cdot x = [z \mid zs] \wedge\\ &\quad \exists u \cdot \exists u' \cdot \exists v \cdot \exists v' \cdot\\ &\qquad \mathsf{partition}(z, zs, u, v) \wedge \mathsf{sort}(u, u')\\ &\qquad \wedge \mathsf{sort}(v, v') \wedge \mathsf{append}(u', [x \mid v'], z))\end{aligned}$$

Provided that $\mathsf{partition}(z, zs, u, v)$ is true if and only if u is the list of all members of list zs less than or equal to z, and v is the list of all members of zs greater than z, the latter description also gives the meaning of $\mathsf{sort}(x, y)$. The method of solving the problem given is the "quicksort"-algorithm. These two descriptions are equivalent to each other in the sense that both predicates will be true for the same pairs of terms. The point is that for the sake of a requirements specification it should not matter which of these descriptions is chosen. The requirements specification does not commit us to a particular method of solving the problem, although it, necessarily, presents us with some such method. "No information about how to solve the problem" should read "no information about how to do it efficiently" instead.

Furthermore, in the transformation phase domain knowledge can be used to reduce the complexity of the search space for a program implementing the requirements specification. For example, consider the problem of determining the amount of hydrocarbons in a ground formation [23]: since hydrocarbons are not uniform (the density of gas varies depending on temperature, depth, etc., to a considerable degree), the effect of hydrocarbons on measurements is difficult to capture precisely. Human experts interpreting oil well logs have developed a simple heuristic: "Since light hydrocarbons are uncommon all calculations should be performed assuming there are no light hydrocarbons. If the results are implausible, consider the possibility that light hydrocarbons are present." [23] In Barstow's system, this heuristic is reflected in two subproblems: porosity analysis and hydrocarbon correction. During program synthesis domain specific knowledge enables one to reduce a complex problem into two simpler subproblems.

Another area where domain knowledge enters the program transformation process is the selection of alternative possible implementations. For example, various techniques exist to represent real numbers. The knowledge that a sensor reads temperatures to within an accuracy of, say, +0.03 centigrade would have a direct bearing on how we choose to represent temperature to the software system. Or consider the problem of solving a complex system of nonlinear polynomial equations: From a mathematical point of view this may not be tractable due to the multiple solutions for the same unknown. However, only one solution with a physically plausible range of values may exist [23]. If an approximating numeric technique is necessary, domain knowledge might allow us to predict the number of iterations required to achieve the desired accuracy. Using the temperature sensor above, if two iterations result in 4-digit accuracy, nothing more is required.

The programming knowledge that is applied at the second step enables us to trans-

form the requirements specification into a working program. It is during this step that efficiency considerations and implementation issues are addressed: How should the representations of the requirements specification be implemented in the programming language, i.e., how are the abstract data types of the requirements specification to be reduced to data types of the programming language? Which algorithms should be employed to solve the problem?[9]

How should requirements specifications be formulated? Any software engineering methodology requires that there be some correspondence between the domain model, the requirements specification, and the program. According to the new paradigm it is insisted that

- The domain model is isomorphic to a model of the requirements specification (the conceptual model).
- The program is a theory obtained from the requirements specification by application of model-preserving transformations only.

Therefore,

- The conceptual model is a model of the program.
- The domain model is isomorphic to a model of the program.

It is the insistence on the isomorphism between conceptual model and domain model that makes the application of the new paradigm possible. It ensures the correctness of the program as well as the possibility of the transformation.

First, the transformation from requirements specification to program is carried out through model-preserving transformation steps only. Thus, the conceptual model is also a model of the program. Since validity of a theory was defined in terms of an isomorphism between a model of the theory and the domain, the validity of a program consists in this isomorphism between the model of the program and the domain model.

Second, the transformations are based on knowledge of the domain. This knowledge must, therefore, be specified in the requirements specification. If the model of the requirements specification is isomorphic to the domain, then for every relevant fact of the domain (these are just the facts in the domain model), there exists a sentence that is entailed by the requirements specification, such that the fact under consideration can be mapped onto the interpretation of this sentence. Consequently, the description of every relevant fact of the domain can be derived from the requirements specification.

[9]Although the requirements specification gave us some algorithm, the system is not committed to this algorithm, but can replace it by any algorithm which is functionally equivalent.

1.4 Knowledge-Based Software Development

Various knowledge-based systems geared to assisting the users in producing software have been developed. These systems tackle different phases of the software life-cycle. Knowledge-based techniques can help in elicitation and formalization of requirements, they can aid in software design and implementation as well as assist during testing and debugging phases.

1.4.1 Requirements Elicitation and Formalization

KBRA [74] (Knowledge-Based Requirements Assistant) is a component of the KBSA (Knowledge-Based Software Assistant) developed by Sanders Associates, Inc. KBSA is intended to support software development spanning a system's life-cycle from requirements to code.

KBRA provides computer assistance from the project's beginning, presenting multiple views of the system being specified. These views support capturing and enforcing ramifications of design decisions, handling reusability at the requirements level, and critiquing and automatically completing certain aspects of requirements specifications (e.g., acquisition, analysis, and communication).

Czuchry [74] identified knowledge representation issues associated with requirements acquisition and analysis. He also employed artificial intelligence techniques to provide consistent reasoning for the intelligent assistant: inheritance of properties from generic object types, automatic classification based on discriminators, and constraint propagation for processing ramifications of requirements decisions.

KB/RMS (Robert Binder Systems Consulting, Inc.) [31] is a knowledge-based requirements definition assistant. It uses rule-based inference and natural language processing to create a high-level system model from a collection of natural language statements. KB/RMS also assists the software developers in refining the requirements specification.

KB/RMS is independent of any particular application domain, software development method or implementation platform. It identifies the objects in the problem and solution spaces. The so obtained model also serves as the logical schema for the KB/RMS database.

Requirements Apprentice (RA) [265] is designed to assist the analyst in the creation and modification of software requirements which focus on the boundary between informal and formal specifications. RA has been developed in the context of the Programmer's Apprentice project, whose goal is to automate the analysis, synthesis, modification, specification, verification, and documentation of software systems. RA consists of three modules.

- CAKE, a hybrid knowledge representation and reasoning system. CAKE supports dependency-directed reasoning aiding in disambiguation of informal specifications and in contradiction detection.

- EXECUTIVE, an aid for interacting with the analyst and providing high-level control of the reasoning performed by CAKE.
- A repository for storing domain-independent and domain-dependent information.

RA attempts to resolve the ambiguities and incompleteness inherent in informal specifications, as well as over-generalizations and inconsistencies, by resorting to a library of prior domain knowledge or "clichés." A cliché is a representation of commonly occurring structures of the domain. Mechanisms are provided to instantiate a cliché in a particular situation. RA also contains a requirements knowledge base and a knowledge-based editor (KBEmacs) which automatically implements a program from algorithmic fragments.

WATSON (AT&T Bell Labs.) [168] is an artificial intelligence-based software development environment for reducing the complexity of formal specifications elicited for new features in telephone switching software.

WATSON uses a variety of design knowledge to bridge the gap between informal English "scenarios" and an executable finite-state skeleton:

- Domain specific knowledge of telephone hardware, telephone network protocols, expected end-user etiquette, and principles of finite-state automata design.
- WATSON relies on novel heterogeneous knowledge representation techniques incorporating goal-directed plans (generalized from the English scenarios), temporal logic, and relational approximations to finite-state automata. These representation techniques are interrelated through theories to allow for internal consistency checking.
- Background knowledge is embedded in prefabricated plan templates, axioms of temporal logic, and constraints on finite-state automata.

Designers of new telephone features present a natural-language scenario. Internally scenarios are represented in the form of finite-state automata with temporal logic constraints on transitions. WATSON expands these scenarios to resolve inconsistencies and incomplete information.

1.4.2 *Software Design and Implementation*

ASPIS (Application Software Prototype Implementation System) [258] encourages a more flexible and effective software-development life cycle by smoothing the transition between users' needs, analysis, and design.

ASPIS consists of several assistants:

- The analysis assistant and design assistant are used directly by the developers of a particular application. They embody knowledge about both the particular software development methodology employed and the application domain.

- Once defined, the specifications can be executed by the prototype assistant, which verifies the system's properties.

- The reuse assistant aids developers in reusing specifications and designs.

The design schemas in *IDeA* (Microelectronics and Computer Technology Corp.) provide a refinement paradigm of software development and support a uniform view of software specification, design, and prototyping. Reusable design information is abstracted in the form of domain-oriented design schemas applicable to various design families. Constraint information, such as design dependencies and consistency requirements of a users' specifications are also included in the design schema. In addition, rules for design specialization and refinement are included to support a top-down refinement-based methodology. Some support is also provided in the form of bottom-up design through planning, rapid prototyping, and goal management.

IDeA (Intelligent Design Aid) [203, 204] is a prototype design environment integrating knowledge-based support for software reuse, analysis, and testing. The design schema representation in IDeA is based on data-flow modeling. This representation supports the techniques of structured analysis and data-flow design and facilitates the use of process graph structures for simulation.

KIDS (Kestrel Interactive Development System) [295] formalizes and automates various sources of programming knowledge and integrates them into a highly automated environment for developing formal specifications into correct and efficient programs.

KIDS provides an open architecture for experimenting with the automated development of formal specifications into correct and efficient programs. The system has components for performing algorithm design, deductive inference, program simplification, partial evaluation, finite differencing optimizations, data type refinement and other development operations. Although their application is interactive, all of KIDS operations are automatic except the selection of an algorithm design tactic. Design tactics provided by KIDS are divide-and-conquer, global search, and local search.

Users of KIDS develop a formal specification into a program by interactively applying a sequence of high-level transformations. During development, the users view a partially implemented specification annotated with input assumptions, invariants, and output conditions.

All KIDS transformations are correctness-preserving and perform significant, meaningful steps from the users' point of view. Their intent is to provide a collection of program transformations that can be composed through tactics or a meta-programming language to yield higher-level and domain-specific transformations.

Marvel [18, 164] is a programming environment that provides early error-checking and answers questions about a program under development. Marvel has a certain understanding of the systems being developed and how tools are used to produce software.

Marvel supports two aspects of an intelligent assistant: it provides insight into the system and it actively participates in development through opportunistic processing.

Marvel consists of two key components:

- An object data base stores data represented as objects. This data base defines object classes and the relationships among objects.
- A process model imposes structure on programming activities. The model is an extensible collection of rules specifying conditions that must exist so that particular tools may be applied to an object. Rules are relevant only when users invoke a tool or when the environment initiates processing.

MicroScope (Hewlett-Packard) [9] is a knowledge-based programming environment designed to improve quality and productivity of software development. The MicroScope program analysis system helps programmers in comprehending and modifying programs.

MicroScope provides a framework for navigation through different views of programs to help programmers focus on the parts they want to understand. MicroScope contains rules for reasoning about program properties which enable it to advise the programmer during code evolution (debugging and modification).

Programmers can extend or change the rule base and request explanations of MicroScope's reasoning. MicroScope stores program information in a central knowledge base, so different services have access to the same information, saving programmers from using separate tools having different levels of programming knowledge. MicroScope provides a common user interface among its services and allows programmers access to each service at any time.

1.4.3 Software Testing and Debugging

To support fault-driven bug localization, Falosy [285] incorporates a knowledge base of heuristic associations between output discrepancies and possible causes. Each fault model in the knowledge base relates fault symptoms to hypotheses regarding possible faults. Function-driven fault localization relies on relations between less specific fault hypotheses and functions. In addition, Falosy's knowledge base contains functional prototypes that describe implementation alternatives.

Given the program to be debugged and a list of output discrepancies, Falosy checks whether any expected faults listed in the fault model are present (fault-driven localization), and identifies the discrepancies between the functional prototype and the closest matching section of the code as bugs (function-driven localization).

Falosy uses a straightforward fault-localization strategy. On the basis of output discrepancies, it decides which fault-localization tactic (fault-driven or function-driven) it will initially pursue. The output discrepancies are then matched against the symptoms of which the system is aware.

Tsai proposed a noninterference architecture [338, 339] to collect the program execution history of a target system without interfering with its execution. It guarantees

the preservation of timing requirements as well as the reproduction of errors. His system detects synchronization errors and timing related errors. To eliminate redundant information in the collected execution history, a post-processing mechanism to organize the information necessary for testing and debugging is introduced. A knowledge-based debugging aid [337] automates the examination of collected data and assists the users in localizing errors.

MTA [141] it is intended to help debug process-structured programs with message-based interprocess communication. MTA examines the message trace and outputs a list of suspect processes and anomalies discovered in the communicated messages. MTA identifies illegal message sequences in the trace and determines the processes at fault.

Proust [161, 162] uses a less-formal problem description in the form of a list of goals to be satisfied and a description of objects the program manipulates. Proust utilizes programming plans to understand programs. Programming plans contain statements and subgoals, and show how particular programming goals can be realized.

A problem in the matching process arises when no known plans for the currently considered goal exactly match the code. Proust then decides whether it is faced with an unknown implementation or with buggy code. Plan-difference rules are triggered by discrepancies between the plan being matched and the code, and may either transform the program code into a correct form or explain the differences as a bug.

Program-analysis based debugging compares the program to specifications, e.g., Laura [3] or output assertions, e.g., PUDSY [208]. Comparison typically proceeds through several stages: standardization transformations, graph matching (to bind the corresponding variables, nodes, and arcs), and error detection. The program to be debugged must implement the same algorithm as the specification, or derive the same output assertions, respectively.

1.4.4 Roadmap to This Book

We have developed a software development environment encompassing the whole life-cycle from requirements specification to testing and debugging [336, 337, 340–342, 346–348, 350–352]. This environment adheres to the new software engineering paradigm as presented above. It is based on a requirements specification language based on the representational constructs of frames and rules, appropriately termed FRORL (this acronym stands for "Frame- and Rule-oriented Requirements specification Language").

Software system development starts from the formation of informal requirements by the client according to the problem domain. The requirements may include functional operations requirements, reliability requirements, performance requirements, communication protocols, interface/environment accommodations, or timing constraints. Starting with the client's informal requirements the developers build up a conceptual model of the proposed software system.

The developers form the FRORL specification from the conceptual model by follow-

ing the six steps of the frame-and rule-oriented development methodology as described in Section 2.5. This methodology guides the users to construct specifications systematically that appropriately model the problem domain. Now the developers must check the correctness of the specification. The FRORL analyzer performs various consistency checks on the specification and attempts to detect desirable and undesirable properties of the specification. These include static and dynamic functional properties (i.e., properties concerned with the functional aspects of the systems), temporal properties, and timing constraints consistency (see Chapter 6). Inference mechanisms are provided to compare consequences of the requirements specification to the domain model, and to determine whether important facts of the domain model hold in the requirements specification.

The requirements specification is executable (i.e., the specification is a rapid prototype). During prototype evaluation clients and developers cooperate to validate the prototype by inspecting its behavior and by comparing consequences of the requirements specification with the domain model. The abstractions of the requirements specification are then implemented by a transformation system in a conventional programming language.

This is *not* a book about the aforementioned software development environment. Instead, it introduces applications of artificial intelligence and knowledge-based techniques applied to the software development process. During recent years this field has expanded considerably. Many different approaches of solving subproblems have been presented, often based on radically different theoretical and philosophical foundations. A book of this length could not completely cover the field of software engineering and still do justice to the approaches presented. We have opted not to attempt to cover the field of software engineering in its entirety. We have selected important aspects of the software development process and discuss in-depth particular approaches of applying knowledge-based techniques to those selected aspects. Survey sections at the beginning of relevant chapters relate the presented techniques to other approaches.

For the sake of continuity we decided to base the discussion of presented techniques on our FRORL requirements specification language [348, 350, 352]. However, we have excluded material that is unique to the development of software systems specified in FRORL, except where it proved necessary to maintain coherence of the text. We intend this book as a guide to developing software systems starting from formal specifications (provided that the specification language has a clean, well-understood semantic foundation).

The next chapter begins by presenting demands a requirements specification language should meet. We argue that any requirements specification language should provide freedom from implementation concerns, allow as natural a representation of the system's requirements as possible, and provide mechanisms for verification and validation of the requirements specification against the domain model. After giving a short survey of types of requirements specification languages, we present FRORL (a "Frame- and Rule-Oriented Requirements specification Language") which we rely

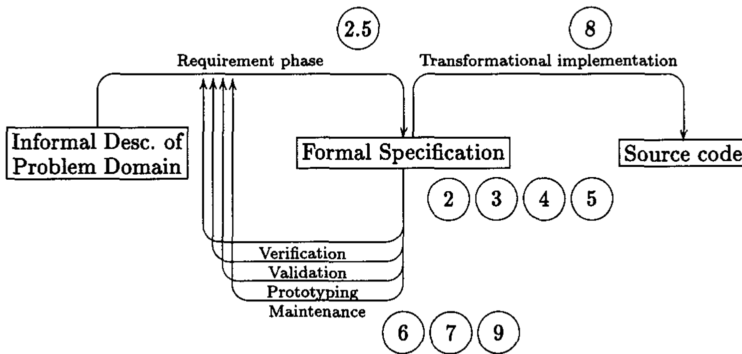

Figure 1.3. Relationship between chapters and phases of the software life-cycle.

on throughout this book. A discussion of a methodological approach to constructing requirements specifications concludes this chapter.

If the requirements specification language is expected to allow for thorough verification beyond mere syntactic checks and execution of the specification, then a sound formal basis is necessary. In Chapter 3, we present the formal foundations of FRORL together with schemes for translating constructs of FRORL into its underlying logic. To support features such as default inheritance and exceptions, we base the FRORL requirements specification language on a nonmonotonic variant of Horn-clause logic. In this chapter, we also prove this logic to be both sound and complete.

Real-time distributed systems require additional features to be represented completely: Multiple processes and inter-process communication must be supported. Mechanisms are necessary to express timing constraints and nondeterministic behavior. After discussing the special aspects of real-time distributed systems modeling, we present various real-time specification languages. Chapter 4 also introduces the features of FRORL peculiar to the specification of real-time distributed systems.

To permit the formal verification of specifications of real-time distributed systems the formal foundation of the requirements specification language must also include the concept of time. In Chapter 5, we present the temporal fix-point calculus which serves to verify the time-dependent aspects of the specification. Through model checking, we can determine whether a temporal sentence expressing some timing constraint on the system holds for a given specification. We show how the timing-related constructs of FRORL can be translated into the temporal fix-point calculus.

Chapter 6 is devoted to the verification of a requirements specification. Verification may apply to both functional and time-dependent aspects of a specification. The functional aspects of the specification are concerned with ensuring the functionality

of the system. Dynamic analysis may determine liveness, reversibility, consistency, and similar properties. The time-dependent aspects of the specification are concerned with timing-constraints imposed by the system's environment. Timing consistency analysis determines whether it is possible to meet these constraints. Through model checking, various important temporal properties of a specification, such as safety properties (partial correctness, mutual exclusion, deadlock freedom), liveness properties (total correctness, guaranteed accessibility), precedence properties, and fairness are established.

In Chapter 7, we discuss the development of knowledge-based systems and show that it can be guided by the same principles that assist in developing conventional software. We develop verification techniques for the specification of knowledge-based systems and argue that the paradigms underlying this book are equally well applicable to the development of knowledge-based systems.

The implementation of a specification in conventional programming languages is discussed in Chapter 8. We focus on the technique of transformational implementation: the step-by-step transformation of constructs of the specification into constructs of the implementation language by correctness-preserving transformation rules. Two major aspects of implementing a logic-based specification are data dependency and flow analysis and the implementation of the backtracking control mechanism inherent in the specification. For one, constructs of a logic-based specification are assumed to be logical descriptions and can be used, procedurally speaking, to compute multi-directionally. On the other hand, all possible solutions of a given logic-based specification can be computed by way of backtracking. Neither of these features is present in conventional programming languages. We present techniques to determine the intended direction of the flow of computation and to eliminate unnecessary nondeterminism from a specification. The spirit of transformational programming is illustrated by the example of canonicalizing transformations which are a necessary first step for the data flow analysis techniques presented.

The debugging and testing process is indispensable to producing a good program. It becomes more difficult as the complexity of the program increases and yet more difficult when programs are written in concurrent languages where several tasks may be executing in parallel. Traditionally, debugging is performed in the implementation phase of the software life cycle. The executability of a specification makes it possible to apply debugging techniques earlier, during validation of the requirements specification which will exhibit errors in the specification. In Chapter 9 we discuss techniques that aid in locating such errors and correcting it.

The appendices contain two examples of system specifications using FRORL (the alternating bit protocol and the specification of part of the subscriber-line controller in a telephone exchange). We present the formal grammar of FRORL, as well as theoretical results governing fix-points relied upon throughout this book.

Fig.1.3 relates the phases of the software life-cycle to parts of this book. As pointed out earlier, because the techniques presented in this book are discussed in-depth it

is not possible to cover all the presented issues in complete breadth. For example, although Chapter 8 discusses the implementation of FRORL specifications in conventional programming languages, no mention is made of how the inheritance hierarchy of a FRORL specification is converted into data structures of the implementation language. Neither do we discuss data flow analysis as it applies to the concurrent subparts of specified systems. This book should not be interpreted as a complete discussion the steps necessary for arriving at a correct implementation starting from a FRORL specification. Instead, this book attempts to present, for each of the phases of the software life-cycle, particular applications of knowledge-based techniques to aid software developers.

1.4.5 Acknowledgments

The research presented here would have been impossible without the assistance and contributions of our graduate students involved in the development of the FRORL software development environment. H. Jang contributed to the implementation of the verification mechanisms discussed in Chapter 6. T. Moritz implemented the development and verification environment for knowledge-based systems presented in Chapter 7 and developed the potential-conflict backtracking scheme relied upon in this environment. R. Sheu and B. Li have conceived and implemented the data dependency and data flow analysis techniques of Chapter 8. A. Liu and K. Nair have contributed to the discussion of specification debugging in Chapter 9.

We are grateful to Fujitsu America, Inc., National Science Foundation under Grants CCR-8809381 and CCR-9106540, IEEE and Engineering Foundation under a Grant RI-A-88-11, and the Science and Technology Agency (STA) of Japan, for supporting the research reported in this book.