A Declarative Approach to Software Requirement Specification Languages*

Jeffrey J.P. Tsai, Thomas Weigert, and Mikio Aoyama+

Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, IL 60680

Abstract

It is a paradigm of software engineering which is especially popular among researchers involved in automatic programming that a complete representation of the problem domain (called *requirement theory*) is repeatedly refined by correctness preserving transformations into the program itself. The design of a requirement theory has to support such a design methodology. In this paper, we present a declarative approach to a software requirement specification language which is capable to specify requirements for software systems under the emerging methodology. The language presented here is based on first-order predicate logic, but augments standard first-order logic by introducing hierarchies and exceptions to its generalizations, in order to allow for a more natural description of the problem domain. The proposed requirement language is valid, and can be determined to be internally consistent. A theorem prover which can interpret the language is also implemented.

Keywords - Software engineering paradigm, requirement theory, requirement specification language, horn-clause logic, non-monotonic logic, multiple inheritance, exception.

§1 Introduction

Until recently software engineering methodologies have adhered to what came to be known as the "software life-cycle paradigm". Although several of these methodologies (e.g., Yourdon-Constantin, Warnier, Jackson design methodologies) are nowadays very widespread in the software engineering community, the software life-cycle has met heavy criticisms [1, 4, 5]. New methodologies, such as rapid prototyping [23], operational specification [26], transformational implementation [3], have emerged and try to address the alleged shortcomings of the life-cycle paradigm. These new methodologies can be seen as sharing a new software engineering paradigm.

On either paradigm, the software engineering process begins with the construction of a theory (the requirement specification, requirement theory) that has a model which is isomorphic to the subset of the the domain the software system is targeted at and then involves the transformation of this theory into another theory formulated in a programming language (the source code, implemented system). The new paradigm differs from the life-cycle paradigm in that

(i) the requirement theory is known to have a model isomorphic to the subset of the domain of interest, and
(ii) the transformations applied to the requirement theory are correctness preserving and therefore also preserve the model.

Demand (i) guarantees that the software engineering process begins with a valid theory of the user's requirement. Methodologies subscribing to the new software engineering paradigms might provide various means of determining that this demand is in fact met. If the requirement theory is capable of being operationally interpreted, the user can be exposed to a "working model" of the software system at an early stage of the development, which will reveal obvious misunderstandings about systems requirement (and the invalidity of the requirement theory). Secondly, if an inference mechanism is available for the requirement theory and it can be checked whether these theorems are satisfied in the model of the software system. Finding theorems which are not satisfied in the model reveals the invalidity of the requirement theory.

Demand (ii) ensures that one cannot derive an invalid program from a valid requirement theory. If all transformations preserve the model, the model of the requirement theory will also be the model of the software system (program) itself.

This paper provides a language which is powerful enough to describe requirements for a software system while meeting the demands of the new software engineering paradigms. We will claim that such a language has to

(i) provide the appropriate syntactic constructs to allow for a natural representation of the problem domain;
(ii) provide a mechanism to check the validity of the requirement theory against the problem domain;
(iii) be capable of operational interpretation;
(iv) provide freedom from implementation concerns; and
(v) be sound and complete.

Languages which are currently available to express requirement theories do not meet all these desiderata. It is especially desideratum (ii) which is largely sidestepped. Currently available requirement specification languages rely on the executability of the requirement theory to determine the validity of the theory. Obviously though such will reveal only partly possible discrepancies between the model of the requirement theory and the problem domain. The absence of an inference mechanism for the requirement specification language makes it impossible to completely validate the requirement theories. The absence of an inference mechanism is explained by the fact that these languages are not clearly understood as far as their semantics is concerned. Specifically, none is known to be sound and/or complete, as desideratum (v) demands. Desideratum (ii) and (iv) are met to varying degrees.

The unavailability of a requirement specification language meeting all the above demands greatly hampers the attempt to utilize the new paradigm in software system design. In particular, the possibility of the validation of the requirement system is crucial to the new paradigm.

*This research is currently supported by Fujitsu American, Inc. under contract 052-80498.
+ Aoyama is with Fujitsu Limited in Japan and currently visiting University of Illinois at Chicago.

The goal of this paper is to propose a language which can serve as a requirement specification language for the new paradigm. To provide an inference mechanism the language presented here is based on first-order predicate logic, but augments standard first-order logic by introducing hierarchies and exceptions to its generalizations, to allow for a more natural description of the problem domain (the difficulty of describing domains easy and naturally in first-order languages has been the main criticism of logic as a requirement specification language).

§2 The Path from Problem to Problem

The new paradigm for software engineering splits the path from problems in the real world to programs into two essentially different steps. Given the objects of the problem domain and the relationships these stand in, plus constraints pertinent to the domain model (the subset of the world/problem domain under consideration), we want to obtain a linguistic description that is valid. The minimum requirement is that this linguistic description has a model (the conceptual model) that is isomorphic to the domain model. In a second step, this linguistic description is transformed into the executable program. The linguistic description of the conceptual model should be (if very inefficient) executable (operational specification, rapid prototyping) or at least formal, and capable of validation. It is generally referred to as specification. For reasons soon to be apparent we will deviate from the standard terminology and refer to it as the *requirement theory.*

At either of these steps knowledge, enters massively into the software engineering process. To represent the conceptual model through the requirement theory involves knowledge about the problem domain itself. The second step from the requirement theory to the program requires knowledge about programming. Because of the involvement of knowledge the software engineering process lends itself to artificial intelligence techniques. Artificial intelligence can enter software engineering at either of these steps [27].

Current research has given consideration mostly to the second step, the move from requirement theory to executable program. Extensive work has been done on how to gradually refine a requirement theory into a program, how to obtain the explicit knowledge to facilitate these transformations, and how to represent the knowledge as to allow for these transformations to be performed automatically.

Until recently little thought has been given to the first step mentioned. Requirement theories currently are mostly functional specifications of the program's behavior and are normally not in a form that would allow it to be automatically transformed into a program. They often do not capture the problem domain adequately, in the sense that their conceptual model is not isomorphic to the domain model. Others have shared this sentiment, that we do not primarily need a specification of what the program should do, but rather, a theory that is valid for the problem domain [2, 25, 17, 10].

Requirement specification as currently popular is a curious mix: to produce it programming knowledge is needed, to transform it into a program requires extensive knowledge about the domain, beyond the information that is conveyed by the requirement theory. We feel that these sets of knowledge are, and should be, separated:

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

b. The requirement theory describes only what the problem is, not how it is to be solved, thus no knowledge about programming should be required to formulate a problem in terms of the requirement theory. In particular, information about algorithms or the specification of any non-abstract data types is foreign to the spirit of requirement theory.

Both claims are standard desiderata: nevertheless, two remarks are in order here.

We are not arguing that to describe the problem domain in a requirement theory, we do not need any knowledge about how to solve the problem. There exists a strong connection between describing a problem and solving the problem. A formal description of a problem will already be one way how to solve the problem. Consider a formal description of the property "list y is a sorted version of list x":

$$\forall x \forall y\ (\mathrm{SORT}(x,y) \equiv \mathrm{PERMUTATION\_OF}(x,y)\ \&\ \mathrm{ORDERED}(y))$$

The predicate SORT(x,y) expresses the above property and describes it in the following way: SORT(x,y) is true if and only if y is a permutation of x, and y is ordered. This obviously captures the meaning of SORT(x,y). In addition, this description gives us also a method of finding 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 in order to sort a list of length n), but, nevertheless, even this innocent looking specification is a method of solving the given problem. The following explication of SORT(x,y) is extensionally equivalent to the one above, but gives a far better method.

$$\forall x \forall y\ (\mathrm{SORT}(x,y) \equiv ((x=[]\ \&\ y=[])\ v\ (\forall z \forall zs(x=[z|zs]\ \&$$

$$\forall u \forall u^* \forall v \forall v^* (\mathrm{PARTITION}(z,zs,u,v)\ \&\ \mathrm{SORT}(u,u^*)$$

$$\&\ \mathrm{SORT}(v,v^*)\ \&\ \mathrm{APPEND}(u^*,[x|v^*],z))))))$$

; let a list be represented in familiar List-notation.

The latter description also gives the meaning of SORT(x,y), if we let PARTITION(z,zs,u,v) be true iff u is the list of all members of list zs which are less than or equal to z, and v is the list of all members of zs which are greater than z. The method of solving the problem given in the latter description 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 requirement theory it should not matter which of these descriptions is chosen. The requirement theory does not commit us to a particular method of solving the problem, although it necessarily gives us some method. "No information about how to solve the problem" thus really should read "no information about how to do it efficiently".

Secondly, in the transformation phase domain knowledge can be used to reduce the complexity of the search space for a program which implements the requirement theory. As an example, consider the problem of determining the amount hydrocarbon in a ground formation [6]: 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, and the theories are hard to deal with mathematically. Human experts interpreting oil well logs have developed a simple heuristic:

"Since light hydrocarbons are relatively uncommon, do all of the calculations assuming there are no light hydrocarbons; if the results are implausible, consider the possibility that light hydrocarbons are present." [6]

In Barstow's system, this heuristic is reflected in two subproblems: porosity analysis and hydrocarbon correction. During program synthesis the domain specific heuristic allows to reduce a complex problem to two simpler subproblems.

Another area where domain knowledge enters the program transformation process is the selection of alternative possible implementations. For example, there are various techniques to represent real numbers. The knowledge that a sensor reads temperatures to within an accuracy of, say, ±0.03 C would have a direct bearing on how we choose to represent the temperature to the software system. Or consider the problem of solving a complex system of non-linear polynomial equations. From a mathematical point of view this may not be tractable due to the multiple solutions for the same unknown possible. However, there may be only one solution with a physically plausible range of values [6]. Or, 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 will result in 4-digit accuracy, no more will be required.

The programming knowledge that enters at the second step enables us to transform the requirement theory into a working program. It is at this step that efficiency considerations and implementation issues are addressed: How should the representations of the requirement theory be implemented in the programming language, i.e., how are the abstract datatypes of the requirement theory to be reduced to datatypes of the programming language? Which algorithms should be employed to solve the problem?

§2.1 Demands on A Requirement Theory

In this paper we will be concerned only with the requirement theories and ignore the second step (the transformation of the requirement theory into the program) of the software engineering process. How should requirement theories be formulated?

Any software engineering methodology has to require that that there is some correspondence between world (that is, the domain model), requirement theory, and program. But on the new paradigm it is insisted that

- The world (the domain model) is isomorphic to a model of the requirement theory (the conceptual model).
- The program is obtained from the requirement theory 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 requirement theory to program proceeds through model-preserving transformation steps only. Thus the conceptual model will also be the model of the program. Since validity of a theory was defined in terms of an isomorphism between a model of the theory and the world, the validity of a program consists in this isomorphism between the model of the program and the domain model.

Second, the transformations that take place will be based on knowledge about the domain. This knowledge must therefore be specified in the requirement theory. If the model of the requirement theory is isomorphic to the domain model, then for every relevant fact of the world (these are just the facts included in the domain model) there exists a sentence that is entailed by the requirement theory such that the fact under consideration can be mapped onto the interpretation of this sentence. As a consequence, the description of every relevant fact of the world can be derived from the requirement theory.

§2.2 Demands on A Requirement Specification Language

Traditionally, requirement theories have predominantly been stated informally. At best, requirement theories have been formulated in some graphical design language, e.g., SADT, SREM. On the new paradigm, the requirement theory plays an essential role in software development. Placing requirement theory into this central position also involves putting heavy demands on languages that express the theory (requirement specification languages).

The following demands can be imposed upon a language that is to be used to express the requirement theory:

- Since the specification language describes the intended behavior of the software system, without prescribing a particular method of solving the problem, it must provide freedom from implementation concerns. This freedom comprises: finding a method for solving the problem, providing the data required for that method, and making the program efficient [2].
- Since the requirement theory must have a model isomorphic to the conceptual model, the specification language has to provide a mechanism to determine whether this is in fact the case. We must be able to check the validity of the requirement theory.
- The specification language has to allow the domain to be described easy and naturally. It has to be possible to represent both the objects and relations of the domain, as well as the constraints on them and rules about them, in a straightforward manner.
- The specification language must be known to be both sound and complete.
- Since the requirement theory should be able to be executed as rapid prototype, we have to be able give the expressions of the specification language interpretations in terms of observable behavior.

§2.3 Languages to Formulate Requirement Theories

Unlike a conventional software specification, a requirement theory with interpretation of its language constructs can exhibit the behavior of the proposed software system. To achieve operationality, the requirement theory is expressed in a language or a form that allows it to be evaluated or interpreted to show system behavior.

An operational requirement theory addresses a key shortcoming of the conventional life-cycle paradigm, in which users and developers must wait well into the design phase before they have linguistic constructs (in this case, procedures and modules) that are producing system behavior. By that time, design commitments have been made. If the system behavior is not acceptable, unraveling these commitments is costly. If the requirement theory is executable, user feedback can be gathered early in the software development process.

The design errors that are exposed by the rapid prototyping design methodology result in the requirement theory defining behavior differently from the intended behavior. A system designer that committed errors of this kind selected a

different subset of the world as the conceptual model than the user had in mind. In a sense, a software system with such design errors is correct in that the requirement theory is valid. It just happens to be the wrong system, due to the selection of a wrong conceptual model.

Two approaches to make specification languages executable are possible:

a. Wide Spectrum Languages - These languages include both high-level primitives like sets, mappings, relations, as well as low-level constructs. Wide spectrum languages are capable of being compiled into executable form, or into a language that can be executed.

The V Language [20] can be viewed as comprising a hierarchy of descriptions. The lowest level is a procedural subset, a strongly-typed block-structured language which can be directly compiled into LISP. All high-level constructs can be mapped onto low-level structures.

SETL [19] is a programming language providing a multitude of high-level data structures, but can be compiled and executed. It is implemented with value-semantics rather than pointer-semantics.

b. Interpreted Languages - These specification languages themselves are purely declarative, but their constructs can be given operational interpretations.

OBJ [13] is based on the idea of equationally defined abstract data types which are given an operational interpretation by viewing them as rewrite rules.

In GIST [2] all possible behaviors of the system are characterized through symbolic evaluation techniques. Symbolic execution allows many different execution paths to be examined simultaneously.

FDL [22] translates the specification into a Prolog form. It is then executed in conjunction with a set of modelling functions which provide an operational interpretation of the semantics of FDL.

§3 Horn-clause Logic as the Basis of a Specification Language

We pointed out above that any statement of a problem in terms of a theory will also be a method to solve the problem. Let us call the point of view that looks at a theory as the statement of a problem to be solved ("what to do") the declarative point of view, while referring to the "how to do it" aspect as the procedural point of view. The above approaches all share that they look at a design problem more from the declarative point of view than from the procedural point of view. (There is a continuum of languages with logic at one end of the spectrum, which takes only the declarative point of view. At the other end of the spectrum of course are programming conventional programming languages which allow only the procedural point of view.) Many of the procedural aspects of specifications drawn up using these languages are brought out only by the process of interpreting the constructs as operations, etc. Nevertheless, none of the specification languages above are completely empty of explicit procedural information. For example, SETL and V use most of the standard programming constructs like while-loops, if-then-else constructions, etc., to express control information. The difference to conventional programming languages is that these specification languages provide many very high-level constructs, such as iteration over a set, or universal quantification, which are also present in a purely declarative theory and thus allow the designer to assume the declarative point of view more frequently.

This paper is an attempt to make requirement theories even more declarative by proposing a specification language based on the Horn-clause subset of first-order predicate logic.

As Kowalski has repeatedly pointed out [15] Horn-clause logic allows for two different semantical interpretations of its sentences:

- The declarative semantics of Horn-clause logic is based on the standard model-theoretic semantics of first-order logic. The intersection of all models of a theory is also a model for the theory and is known as the minimal model. The minimal model is the declarative meaning of a theory expressed in Horn-clause logic.
- The procedural semantics is a way of describing procedurally the meaning of a theory. The procedural meaning of a theory is the set of ground clauses that are instances of queries that are solved by the theory using an abstract interpreter.

Execution of theories expressed in Horn-clause logic relies on the interpretation of expressions of the form

A if B and C and ...

as procedures

to do A, do B and C and ...

Given this interpretation, a requirement theory expressed in Horn-clause logic will be executable, using the mechanism of an abstract interpreter (as in a Prolog program). In contrast to other specification languages, Horn-clause logic has well understood semantics and is known to be both sound and complete. Together with the Prolog theorem-proving mechanism, the validity of the requirement theory can be checked for. Horn-clause logic and its interpretation by Prolog provides most of the freedoms demanded from a requirement specification language. Specifications need not be efficient to be executable (efficiency freedom). All data in a Prolog program is global, in addition a Prolog data base can be treated as a relational data base model (data access freedom). Unfortunately, Horn-clause specifications do not completely grant complete freedom of finding a method for solving the problem. Some care has to be taken to avoid enumeration of infinite data structures, lest the execution will not terminate. Also some control information is provided by the ordering of the clauses in the data base and the presence of the cut.

The procedural interpretation of the axioms of requirement theories expressed in Horn-clause logic also provides the inference mechanism required for the validity checking of the requirement theory against the domain model. One can formulate goal-clauses expressing some fact about the software system and determine whether this fact can be derived from the requirement theory. Horn-clause logic here provides an additional advantage over standard first-order logic. Any theory with a large number of axioms formulated in a standard first-order language will have a huge number of theorems, most of them irrelevant for the validation of the requirement theory (think only of the many disjunctions one can derive using standard first-order inference). Horn-clause logic is not susceptible of this problem since the procedural interpretation of Horn clauses treats conditionals as having only one inferential direction. Using Horn-clause logic as the logical foundation of the requirement specification language enables to escape the complexity problem validation would be victim to in a standard first-order framework.

In its original form Horn-clause logic does not meet the demand of naturalness of representation. A commonplace in software design is that the objects of the world should be

considered to stand in some hierarchy to each other. Properties and constraints can be inherited down this hierarchy, and provide default information about objects. On the other hand, property inheritance can be overridden in exceptional cases. Standard Horn-clause logic provides neither a hierarchical structure with property inheritance nor an exception mechanism to its rules and constraints. The proposed requirements language HCLIE (Horn-Clause Logic with Inheritance and Exception) will be augmented with these two mechanisms.

§4. Language: HCLIE

The syntax of HCLIE is a super-set of ordinary Prolog. It adds the syntactic category of common nouns. Common nouns can take any place predicates can take, and are distinguished from predicates by the prefix 'kind'. For example, *kindBird, kindStudent, kindElephant* are common nouns. The denotation of common nouns are the objects which are instances of that kind.

A HCLIE program consists of a set of standard Prolog Horn clauses, and a set of non-Horn clauses which, for lack of a better terminology, we will refer to as HCLIE -clauses.

a. Horn Clauses - A Horn clause is of the form

$$\text{Head} \leftarrow \text{Body}_1 \ \& \ \text{Body}_2 \ \& \ ... \ \& \ \text{Body}_n.$$

where either the head or the body may be empty. Horn clauses are understood to be implicitly universally quantified.

b. HCLIE-clauses - The general form of a HCLIE-clause is

$$\text{Sort: Head} \leftarrow \text{Body}_1 \ \& \ \text{Body}_2 \ \& \ ... \ \& \ \text{Body}_n.$$

The head, body, and sort of the HCLIE-clause may be empty. HCLIE-clauses are understood to be implicitly quantified with the common noun-prefixed quantifier. For example,

(1) kindBird(X) : fly(X) ← has-wings(X).
should be read as implicitly quantified
(2) (∀bird,X) has-wings(X) ⊃ fly(X).
The common noun-prefixed quantifier ranges only over the variables mentioned in the sort. Thus,
(3) kindBird(X) : eats(X,Y) ← nut-shaped(Y).
has to be understood to mean
(4) (∀bird,X)(∀Y) nut-shaped(Y) ⊃ eats(X,Y)

Only common nouns can take the place of sorts. This example also shows the syntactic form of a common noun: an identifier preceded by 'kind'. Clauses can be quantified by more than one common-noun prefixed quantifier. Common nouns can be combined by '.', as is *kindAnimal.kindCircusPerformer*.

The difference between Horn clauses and HCLIE-clauses is that while Horn clauses hold for any object, HCLIE-clauses can only be instantiated by objects which are instances of its sort. In the above examples, the definition of "fly" and "eats" apply only to objects that are "birds". Therefore, the examples should be read as

"Birds with wings fly", and
"Birds eat anything nut-shaped", respectively.

Common nouns can also form the head of a HCLIE-clause. It is so possible to create hierarchies of sorts. One may wonder what the difference between Horn clauses and HCLIE-clauses amounts to. Both

(5) fly(X) ← bird(X).
(6) kindBird(X) : fly(X) ←.

seem to say the same thing, namely that "a thing X flies, if it is a bird". Horn clauses and HCLIE-clauses differ in the way they interact with other clauses in the program. The 'if' ('←') in the Horn clause is interpreted as the standard first-order conditional. Thus any object satisfying *bird(X)* will also satisfy *fly(X)*. This is not always true for HCLIE-clauses. If both the sort and head are common nouns, the 'if' (':') between them represents something like the IS-A link and is hierarchy forming. If the head is a predicate, the 'if' represents predication of a kind. E.g., an instance of *kindBird(X)* will only satisfy *fly(Y)* if it does not at the same time satisfy ¬ *fly(X)*. The interpretation of the 'if' in such HCLIE-clauses is that of the default conditional. Predicates defined by HCLIE-clauses can be inherited down hierarchies, but only as long as they are not overridden by conflicting '¬'-statements. The predicate '¬' is not to be confused with the standard built in predicate 'not'. While 'not' is defined by the negation as failure rule, this is not the case for '¬'. These two uses of negation can both be used in a program, but care has to be taken to differentiate them and they cannot be interdefined.

§5 The Inheritance Mechanism of HCLIE

There are two basic ways to view the description one associates with a kind such as *kindBird*. The first is that the description simply characterizes a prototypical instance of the kind, in this example a prototypical bird. If we hold this view of kinds, then some or all of the descriptions associated with the kind may be contradicted by an instance of that kind. The second view treats the descriptions associated with a kind as necessary conditions which must be fulfilled by all its instances [9]. These two views of what kinds are leads to two different inheritance mechanisms:

a. Default Inheritance - If a (kind or individual) object C is an instance of kind B, i.e., C is lower down in the hierarchy than B, and B has a set of properties A1...An, then any Ai can be overridden in the definition of C. Most current AI-researchers accept this interpretation of inheritance, cf. Smalltalk [11], KRL [7], FRL [18], etc.
b. Strict Inheritance - If a (kind or individual) object C is an instance of kind B, then necessarily C must have all the properties A1,...,An that B has. This view is adopted in Simula [12], TAXIS [16], and semantic data models.

It has been argued extensively that strict inheritance is too inflexible for representing real-world domains [24]. The real world contains exceptions to most of its generalizations. If we required the properties of a kind (an abstraction for purposes of representation) to be present in all the instances of the kind, inheritance would loose most of its usefulness, since very little properties could be had by kinds. Nowadays, most inheritance systems allow instances to override the properties of its kind that do not apply to them.

Another basic question distinguishes inheritance systems: can particular instances (kinds or individuals) be instances of more than one kind, i.e., can particular instances inherit from more than one kind? Sometimes it is the most natural view to think of an object to be an instance of more than one kind. For instance, if we defined a *kindZooAnimal*, with a set of properties, and a *kindPenguin* , with its respective properties, it makes sense to view *fred* , a penguin in Brookfield Zoo, as an instance of both kinds. It can so inherit properties from either kind, which otherwise would be impossible and force duplication of information. Multiple inheritance seems advantageous, but it does not come without problems. Consider the theory in [Fig.5.1]. If we allow *fred* to inherit from both *kind Penguin* and *kindZooAnimal*, *fred* will have contradictory properties. A slightly different problem is shown in [Fig.5.2]. In light of these diffi-

culties, some inheritance systems don't support multiple inheritance, e.g., multiple inheritance was not provided for Smalltalk-80 until the most popular book on Smalltalk-80 was written, and is only a added on feature. One might attempt to solve the second problem by excluding redundant statements from the theory. But it seems strange to exclude obviously true statements from a theory [24], and the example shows that such statements may not always be dispensable. For instance, birds might be happy for other kinds of reasons than that they fly, thus penguins would be happy, although they do not fly. So the statements "birds are happy", although it can be deduced from the theory, is not redundant at all. If the inheritance system performs inferences according to some distance ordering [24], this problem will be avoided, but not the first. Although *fred* is inferentially more distant from *kindAnimalUsedtoHumans* than from *kindPenguin*, it may well be that this distance is completely irrelevant for the question whether *fred* is shy or not.

```
kindPenguin(X) : shy(X) <-.
kindPenguin(fred) <-.
kindZooAnimal(fred) <-.
kindZooAnimal(X) : kindAnimalUsedtoHuaans(X) <-.
kindAnimalUsedtoHuaans(X) : ¬shy(X) <-.
```

[Fig.5.1] Problem for multiple inheritance.

```
fly(X) <- happy(X).
kindBird(X) : fly(X) <-.
kindBird(X) : happy(X) <-.
kindPenguin(X) : kindBird(X) <-.
kindPenguin(X) : ¬fly(X) <-.
kindPenguin(fred) <-.
kindBird(fred) <-.
```

[Fig.5.2] Another problem for multiple inheritance.

HCLIE allows for multiple inheritance, thus avoiding the duplication of information. In cases like [Fig.5.1] there is not way of telling whether *fred* is shy or not. Unless other information is provided, HCLIE will not make a choice either way. In situations as in [Fig.5.2] HCLIE deduces theorems according the inferential distance ordering.

Inheritance is provided by HCLIE through the assumption that conditionals which have common nouns both as consequent and antecedents will hold exceptionless. Exceptions are possible to HCLIE-clauses that are quantified with a common noun-prefixed quantifier, and where the head is not a common noun. In a sense the predicate asserted in the head of such a clause is very similar to the slot-assertions of frame based systems. An HCLIE-clause with both sortal and head being formed by a common noun, i.e., a conditional between common nouns, corresponds very closely to the ISA-link of frame systems. HCLIE-clauses with empty sortal are also thought to hold without exceptions. The syntax grammar of HCLIE is summaried in Def.5.1. We have elsewhere shown that the proposed language is sound and complete.

§6 Specification through Prototype Hierarchies as Design Methodology

Borgida [9] introduced a methodology to design software systems he termed *taxonomic specification*. The key idea of this methodology is that a theory can be constructed by describing first, in terms of classes, the most general kinds in the problem domain, and then proceeding to deal with sub-cases through more specialized kinds. Theories constructed through such a process have their objects structured in a taxonomy. For example, when building a student enrollment system for a university, one might consider first the general kinds 'student' and 'course' and the task of enrolling a student for a course. Later, the designer can consider graduate and undergraduate students and courses, full- and part-time students, elective and non-major courses, and the rules and regulations that apply to these sub-cases.

The obvious advantage of organizing descriptions into taxonomies is the notion of inheritance. Instances of a sub-kind are generally also instances of all its super-kinds, so there is no need to repeat the information specified in the description of a kind for each of its sub-kinds, and for all their sub-kinds, in turn. The result is a more concise description, and some clerical errors can be avoided by reducing the amount of repetitions.

Our approach to achieve a naturalness in the description of the problem domain is to represent the objects of the problem domain as standing in a hierarchy of prototypes. Prototype hierarchies go beyond taxonomies by adopting the default interpretation of the link between a particular instance and the kinds it is an instance of. Properties specified in the kind are only inherited by an instance if the property is not overridden here, i.e., if the description of the instance does not assert otherwise. The adoption of the default interpretation allows to deal with over-abstractions that will be necessarily introduced in the process of forming the theories. Although certain properties might hold for a kind, they do not necessarily hold for the instances of this general kind. Default inheritance allows to ignore such differences during the process of constructing the hierarchy.

The construction of a prototype hierarchy is simply another abstraction methodology. The abstraction involves factoring out the commonalities in the description of several concepts into the description of a more general concept, and the refinement process reintroduces these details by specifying the ways in which a more specialized concept differs from the more general one.

In the following, we illustrate the design of a prototype hierarchy using the language HCLIE as specification language with the example of a student enrollment information system. In its most general form, the enrollment information system deals with two kinds of objects, *kindStudent*, and *kindCourse*, and the *enroll*-relationship which holds between them. Assume that for the general case the following happens when a student is enrolled for a course: first certain preconditions have to be met, namely, the course has to be offered, the course must not be full, i.e., the size of the course must be less than its limit, and the student must not have taken that class before. If these conditions are met the student is added to the roster for this class, the class is added to the student's record, and the class- size is incremented. [Fig.5.3] shows this description of the *enroll*-relation.

The methodology supported by HCLIE allows us to view this general relation between students and courses as a prototype for all *enroll*-relations. Specialization (through the notion of inheritance) allows us to describe sub-cases by specifying only the additional details necessary, and the differences which arise. The *enroll*-relation can be specialized by introducing additional constraints for kinds that stand further down in the hierarchy, or by introducing additional actions to be performed.

In order to introduce further detail about the *enroll*-relation, we describe a taxonomy of students and courses, and introduce the sub-kinds *kindGradStudent, kindUndergradStudent, kindGradCourse,* and *kindUndergradCourse* as shown in [Fig.5.4].

The *enroll*-relation can now be inherited down the taxonomy pictured in [Fig.5.4]. At lower level additional prerequisites may be introduced, or further actions may be added. If additional prerequisites are not met at a kind further down in the hierarchy, the *enroll* -relation will be cancelled. [Fig.5.5] shows several plausible specializations.

```
kindStudent(X).kindCourse(Y)  :
                    enroll(X,Y)   <- (Y not full?) &
                                     (Y offered?) &
                                     (X did not take Y before?) &
                                     (add X to Y-roster) &
                                     (add Y to X-record) &
                                     (increment Y-size).
```

[Fig.5.3] Most general form of the *enroll*-relation.

```
kindGradStudent(X)  :  kindStudent(X)  <-.
kindUndergradStudent(X)  :  kindStudent(X)  <-.
kindGradCourse(X)  :  kindCourse(X)  <-.
kindUndergradCourse(X)  :  kindCourse(X)  <-.
```

[Fig.5.4] Instances of *kindStudent* and *kindCourse*.

```
kindGradStudent(X)  :  enroll(X,Y)  <-
                          (Y-level ≥ 3rd year?) &
                          (infora graduate school of X's enrollment).
kindGradStudent(X)  :  ¬enroll(X,Y)  <-
                          (1st year course?).
kindGradStudent(X)  :  ¬enroll(X,Y)  <-
                          (too many courses: gradmax?).
kindUndergradStudent(X)  :  ¬enroll(X,Y)  <-
                          (after deadline?).
kindUndergradStudent(X)  :  ¬enroll(X,Y)  <-
                          (too many courses: undergradmax?).
kindGradCourse(X)  :  enroll(X,Y)  <-
                          (issue X key to library) &
                          (give X unlimited computer account).
kindUndergradStudent(X).kindGradCourse(Y)  :  ¬enroll(X,Y)  <-
                          (no permission?).
kindUndergradStudent(X).kindUndergradCourse(Y)  :  ¬enroll(X,Y)  <-
                          (X excluded?).
kindUndergradStudent(X).kindUndergradCourse(Y)  :  ¬enroll(X,Y)  <-
                          (X not enough preparation?).
kindUndergradStudent(X).kindUndergradCourse(Y)  :  ¬enroll(X,Y)  <-
                          (too many 1st year courses?).
```

[Fig.5.5] Specializations of *enroll*.

The obvious advantage of this methodology is that we need not, when designing the overall system, take the exceptional cases into consideration. The over-abstraction introduced at the most general level can be corrected at the subkind level without affecting the representation of the system.

Specification through prototype hierarchies and HCLIE are similar to the object-oriented design strategy as described in [21], [8]. Object oriented techniques also describe the world in terms of objects standing in a taxonomy. These objects are interpreted as actors, each with its own set of applicable operations. These operations can be inherited down the hierarchy, and can be cancelled at lower levels. The benefit of this technique is again that by grouping objects into sets of related objects, we factor out common properties and so localize design decision for all instances of an object-kind. Booch [8] argues that through this technique we can achieve increased modifiability, efficiency, reliability, and understandability, since the abstractions of the design are very close to the abstractions of the problem domain. The complexity of a software solution is decreased, the less distance we have between the abstractions of the solution and the real-world abstraction. [Fig.5.6] shows the design for a graphical simulation which moves objects on a display.

[Fig.5.6] defines the most general object, *kindDisplayObject* , and the general operations applying to this object. Any instance of the kind *kindDisplayObject* can inherit these operations. [Fig.5.5] describes a sub-kind of *kindDisplayObject, kindMovingObject,* for which a different move-operation is required (which depends on the object's velocity).

```
kindDisplayObject(X)  :  draw(X)  <-
                            drawCircle(X).
kindDisplayObject(X)  :  drawCircle(X)  <-
                            findDesc(X,Desc) &
                            circle(Desc).
kindDisplayObject(X)  :  undraw(X)  <-
                            undrawCircle(X).
kindDisplayObject(X)  :  undrawCircle(X)  <-
                            findDesc(X,Desc) &
                            eraseCircle(Desc).
kindDisplayObject(X)  :  moveTo(X,NewX,NewY)  <-
                            undraw(X) &
                            updatePos(X,NewX,NewY) &
                            draw(X).
```

[Fig.5.6] Description of *kindDisplayObject*.

```
kindMovingObject(X)  :  kindDisplayObject(X)  <-.
kindMovingObject(X)  :  ¬moveTo(X,NewX,NewY)  <-.
kindMovingObject(X)  :  move(X,Velocity)  <-
                            undraw(X) &
                            updatePos(X,Velocity) &
                            draw(X).
```

[Fig.5.7] An instance of *kindDisplayObject*.

An instance of the kind *kindMovingObject* inherits the *draw-* and *undraw*-operations from *kindDisplayObject* , but overrides its *moveTo*-operation. Further specializations might modify the *draw*-operations, etc. as shown in [Fig.5.8].

```
kindMovingEllipse(X)  :  kindMovingObject(X)  <-.
kindMovingEllipse(X)  :  ¬drawCircle(X)  <-.
kindMovingEllipse(X)  :  draw(X)  <-
                            drawEllipse(X).
kindMovingEllipse(X)  :  drawEllipse(X)  <-
                            findDesc(X,Desc) &
                            ellipse(Desc).
kindMovingEllipse(X)  :  ¬undrawCircle(X)  <-.
kindMovingEllipse(X)  :  undraw(X)  <-
                            undrawEllipse(X).
kindMovingEllipse(X)  :  undrawEllipse(X)  <-
                            findDesc(X,Desc) &
                            eraseEllipse(Desc).

kindZeppelin(X)  :  kindMovingEllipse(X)  <-.
kindZeppelin(X)  :  draw(X)  <-
                            drawFins(X).
```

[Fig.5.8] *MovingEllipse* overrides and augments the (un-)*draw*-relation.

§7 Conclusion

In this paper, we link the activity of software engineering to the job of a scientist and treat the process of software development as a process of theory construction. We argue that there is a demand of a requirement theory to support new software engineering paradigms. We claims that the current available language intended to yield requirement theory can not meet the demand. A requirement language using Horn-clause logic argumented with multiple inheritances and exceptions was proposed in order to support new paradigms. We have elsewhere shown the proposed language is valid, sound, and can be determined to be internally consistent. A theorem prover which can interpret the language is also implemented.

References

[1] W.W. Agresti, *New Paradigms for Software Engineering*, IEEE Computer Society Press, Washington D.C., 1986.

[2] R. Balzer, N. Goldman, D. Wile, "Operational Specification as the Basis for Rapid Prototyping," *ACM Software Eng. Notes*, 7, 5, 1982.

[3] R. Balzer, "Transformational Implementation: An example," *IEEE Trans. on Software Engineering*, SE-7, 1, 1981.

[4] R. Balzer, T.E. Cheatham, C. Green, "Software Technologies in the 1990's: Using a New Paradigms," *Computer*, Nov. 1983, pp. 3-16.

[5] B. Boehm, "A Spiral Model for Software Development and Enhancement," *IEEE Computer*, May 1988.

[6] D.R. Barstow, "Domain-Specific Automatic Programming," *IEEE Trans. on Software Engineering*, vol. SE-11, No. 11, Nov. 1985, pp. 1321-1336.

[7] D.G. Bobrow, T. Winograd, "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, 1, 1, Jan. 1977, pp. 3-46.

[8] G. Booch, *Software Engineering with Ada*, Benjamin Cummings, Menlo Park, 1983

[9] A. Borgida, J. Mylopoulos, and H. Wong

"Generalization/Specialization as a Basis for Software Specification," in *On Conceptual Modelling*, (eds. M. Brodie, etc.), Springer-Verlag, New York, 1984.

[10] A. Borgida, S. Greenspan, J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications," *IEEE Computer*, April 1985.

[11] A.H. Borning, D.H. Ingalls, "Multiple Inheritance in Smalltalk-80," *Proc. AAAI-82*, 1982, pp. 234-237.

[12] O.J. Dahl, C.A.R. Hoare, "Hierarchical Program Structures," in *Structured programming*, (eds. O. Dahl, etc.), Academic Press, New York, 1972, pp. 175-220.

[13] J. Goguen, "Rapid Prototyping in the OBJ Executable Specification Language," *ACM Software Engineering Notes*, 7, 5, Dec. 1982, pp. 75-84.

[14] A. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Readings, 1983.

[15] R. Kowalski, "The Relation between Logic programming and Logic Specification," in *Mathematical Logic and Programming Languages*, (eds. C.A.R. Hoare, etc.), Prentice-Hall, Englewood Cliffs, 1985, pp. 11-24.

[16] J. Myopoulos, P.A. Berstein, H.K.T. Wong, "A Language Facility for Designing Interactive Database-Intensive Applications," *ACM Trans. on Database Systems*, 5, 2, June 1980. pp. 185-207.

[17] C.V. Ramamoorthy, A. Prakash, W.T. Tsai, Y. Usuda, "Software Engineering: Problems and Perspectives," *Computer*, pp. 191-209, Oct. 1984.

[18] R.B. Roberts, I.P., Goldstein, *The FRL Manual*, AI Memo No. 409, MIT AI Lab, Cambridge, 1977.

[19] E. Schonberg, J. Schwartz, M. Sharir, "On Automatic Technique for Selection of Data Representation in SETL Programs," *ACM Trans. on Programming Languages and Systems*, 3, 2, 1981.

[20] D. Smith, G. Kotik, S. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute." *IEEE Trans. on Software Eng.*, SE-11, 11, Nov. 1985.

[21] I. Sommerville, *Software Engineering*, Addison-Wesley, Reading, 1985.

[22] R. Tavendale, "A Technique for Protoyping Directly from a Specification," *Proc. 9th Software Engineering Conf.*, 1985, pp. 224-229.

[23] T. Taylor, T.A. Standish, "Initial Thoughts on Rapid Prototyping," *ACM SIGSOFT*, 1982, pp. 160-166.

[24] D.S. Touretzky, *The Mathematics of Inheritance Systems*, Morgan-Kaufmann Publishers, Los Altos, 1096.

[25] R. Yeh, R. Mittermeir, "Conceptual Modelling as a Basis for Deriving Software Requirements," *Int. Computer Symp.*, Taipei, Taiwan, 1980.

[26] P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Commun. ACM*, Vol. 27, No. 2, Feb. 1984.

[27] I. Zualkernan, W.T. Tsai, and D. Volovik, "Expert Systems and Software Engineering: Ready for Marriage?," *IEEE Expert*, Vol. 1, No. 4, Winter, 1986, pp. 24-31.

[Def.5.1] Context free grammar for HCLIE

```
‹program›          ::= ‹clause›*
‹goal›             ::= ‹structure› |
                       ‹common_noun_functor› ( ‹term› {, ‹term› }* )
‹clause›           ::= ‹horn_clause› | ‹hclie_clause›
‹horn_clause›      ::= ‹head› <— ‹body›.
‹hclie_clause›     ::= ‹sort› : ‹head› <— ‹body›.
‹body›             ::= Ø | ‹word› {& ‹word›} *
‹head›             ::= Ø | ‹word›
‹sort›             ::= Ø | ‹common_noun› { . ‹common_noun›}*
‹word›             ::= ‹constant› | ‹structure›
‹common_noun›      ::= ‹com_noun_functor›( ‹variable› { , ‹variable› }*)
‹com_noun_functor›
                   ::= kind‹letter›‹letter›*
‹constant›         ::= ‹atom› | ‹number› | ‹string›
‹structure›        ::= ‹functor› ( ‹term› { , ‹term› }* ) |
                       ‹term› ‹functor› ‹term› |
                       ‹functor› ‹term› | ‹term› ‹functor›
‹functor›          ::= ‹atom›
‹term›             ::= ‹variable› | ‹structure› | ‹constant›
‹variable›         ::= ‹up_let›‹letter›* | _‹letter›*
‹atom›             ::= ‹name› | ‹special_symbol› | '‹char›‹char›*'
‹name›             ::= ‹low_let›‹letter›*
‹special_symbol›   ::= ‹special_char›‹special_char›*
‹number›           ::= ‹integer› | ‹real›
‹letter›           ::= ‹up_let› | ‹low_let› | _
‹char›             ::= ‹letter› | ‹special_char›
```