Using a Framework for Domain Theory Structure and Evolution to Evaluate Knowledge Acquisition Tools

Arthur Alexander Reyes
Department of Information and Computer Science
University of California at Irvine
Irvine, CA 92697-3425 U.S.A.
`http://www.ics.uci.edu/~artreyes`

Technical Report 96-52
12 November 1996

Abstract

Domain-specific approaches to software engineering require the automation and evolution of domain knowledge. As automatic, domain-specific design synthesis techniques improve, emphasis will shift to automatic generation of domain-specific development environments themselves. These environments will need to be generated from domain expert-evolvable representations of knowledge. The discipline of Knowledge Acquisition has produced tools to automatically generate domain-specific expert systems from domain expert-evolvable representations of knowledge. Thus Software Engineering's emerging era of domain-specificity calls for many of the same capabilities that spurred the discipline of Knowledge Acquisition. The discipline of Algebraic Specification has produced correctness-preserving specification structuring mechanisms to build large specifications in a methodical and understandable way. In an effort to discover a framework to define tools to support Software Engineering's new era, we apply algebraic specification structuring mechanisms to evaluate several exemplary knowledge acquisition tools. Algebraic specification structuring mechanisms provide an architectural metalanguage into which knowledge acquisition tools are recast. This highlights how knowledge acquisition tools support domain theory structure and evolution. This evaluation provide insights into requirements for tools to automatically generate effective, efficient, domain-specific development environments from domain expert-evolvable representations of knowledge.

1 Introduction

Software Engineering enters an *era of application domain-specificity*. This era is heralded by the confluence of research in Domain-Specific Software Architectures (DSSAs) [21]; Domain Modeling, Domain Analysis, and Domain Engineering [25]; Domain-Oriented Development Environments (DODEs) [7]; Domain-Specific, Knowledge-Based Software Engineering (KB-SE) [19];and Domain-Specific Languages [12]. Domain-specificity offers many advantages over domain-independent approaches to software construction and evolution, the chief advantage being the collection, representation, evolution, and automation of domain knowledge. This knowledge remains as a permanent source of organizational memory which can be used by automated tools to work in a variety of problem classes within the domain, e.g., automated program synthesis [19] and end user tutoring [1].

Domain knowledge can be represented in many forms, such as software library components, class hierarchies, databases, semiformal domain models (DMs) in machine-processable notations (e.g., the emerging Unified Modeling Language (UML) for Object-Oriented Development [2]), and formal, declarative representations such as domain theories (DTs). Once automated in these forms, the domain knowledge can be used to answer queries and guide synthesis. These representations can become organizational centerpieces and the focus of considerable refinement and evolution.

As Keller points out, domain-specificity also has disadvantages [16]. The chief disadvantage is the amount of effort required to evolve domain models and domain theories in response to continual real-world application domain evolution. Application domains themselves evolve with the discovery of new knowledge, extension to reusable component libraries, and obsolescence of old technologies and adoption of new technologies. Domain artifacts such as documents, test cases, and tools must evolve in response to both evolution of the domain itself and evolution of our understanding of the domain. If Software Engineering's new era is to scale up effectively, domain models and domain theories must easily evolve with their associated application domains.

The discipline of Knowledge Acquisition (KA) [3] has been associated mostly with the development and evolution of expert systems [10]. A few authors, such as Eriksson [6], have discussed the relationship between software engineering and knowledge acquisition, specifically with regard to requirements analysis, specification, and design. Software Engineering's era of domain-specificity calls for many of the same capabilities as those that spurred the creation of the knowledge acquisition discipline. These capabilities include enabling domain experts (who are responsible for the creation and evolution of domain models and other key domain artifacts) to easily evolve domain models, without first acquiring skills in knowledge representation, artificial intelligence, and automated inference. Well-integrated knowledge acquisition tools automatically synthesize new runtime, end user environments (often referred to as "performance systems") from domain expert-evolved knowledge.

While approaches to knowledge acquisition such as Inductive Logic Programming (ILP) [22] appear promising, we believe the knowledge acquisition discipline could benefit from insight into what the space of possible knowledge acquisition approaches is. Given that knowledge acquisition is concerned with the evolution of domain models and domain theories, a sufficiently general and formal framework for investigating this evolution might be able to provide insights into the boundaries of the space of knowledge acquisition approaches and their limitations. Unexplored areas of this space could indicate a more interesting and broad research agenda for domain model and domain theory evolution and provide insight into the kinds of knowledge acquisition tools needed to work in different areas of this space.

The discipline of Algebraic Specification [28] (especially the Joseph Goguen school of research) believes the purpose of a specification language is not merely to construct sentences in a formal language, but rather to provide an infrastructure for building modular, easily-understood, and well-structured specifications. Researchers in algebraic specification have discovered specification structuring mechanisms which work independently of the particular syntax and logical systems of the individual, component specifications (i.e., regardless of the languages in which the component specifications are written).

Given that domain theories can be represented via algebraic specifications, and that knowledge acquisition tools provide processes and user interfaces to evolve domain models, we recast domain models as domain theories and use algebraic specifications and their structuring mechanisms to define an architectural metalanguage for knowledge acquisition tools. This architectural metalanguage can be used to represent both the static structure of domain theories and domain theory evolution. Armed with this metalanguage as a framework for domain theory structure and evolution, this survey answers the following question.

How does the architecture of a knowledge acquisition tool support domain theory structure and evolution?

Superficially, our evaluation method is similar to "software architectural analysis method" (SAAM) [15]. SAAM seeks to produce more credible comparisons among software tools than is possible using feature-oriented, taxonomic approaches only. SAAM seeks to understand how the structure of a tool supports quality attributes of interest to the human evaluator within the framework of a reference architecture [21] for the tools being evaluated. The architectural metalanguage/framework for domain theory structure and evolution used in this survey (based on algebraic specifications and structuring mechanisms) is designed to specifically address issues of interest to our research and bears little resemblance to SAAM's architectural metalanguage (based on computational components and data/control flow between them). Additionally, a reference architecture for knowledge acquisition tools was not available at the start of our evaluation.

This Survey uses a "What versus How" exposition of domain theory structure and evolution. The "what" branch of domain theory structure and evolution is the syntax, structure, and semantics of domain theories, constructors to incrementally build domain theories, and the criteria which must be satisfied during domain theory evolution. The "what" branch is defined using concepts from the discipline of algebraic specification. The "how" branch of domain theory structure and evolution is the collection of processes and user interfaces needed to generate and evolve domain theories. The "how" branch is defined using concepts from the discipline of knowledge acquisition. We can think of the "what" branch as providing a declarative specification of the goals to be accomplished and the "how" branch as providing processes satisfying the declarative specification.

1.1 Domain Theory Structure And Evolution: WHAT

This survey introduces a tentative framework for the investigation of domain theory structure and evolution. This framework represents domain theory evolution as progression through a state space of domain theories. Each evolutionary step a domain theory takes maps the old domain theory to a new domain theory. In this framework, each evolutionary step is represented by the application of one or more domain theory *constructors*. Because this framework is based on constructor application, the framework can represent both initial development and post development evolution of domain theories.

Our framework represents a domain theory by a specification in a suitable (usually algebraic) specification language. Many domain theory constructors defined in this framework correspond to well-understood algebraic "specification-building operations" [27]. We currently categorize domain theory constructors into instance constructors, sort constructors, sentence constructors, specification constructors, and institution constructors. Application of one or more constructors from any of these categories constitutes a domain theory evolutionary step.

To be practical, domain theory evolution must be constrained in some way. For example, an evolutionary path that routinely invalidates many important, previously developed domain artifacts (e.g., test suites, software library components, problem specifications, tools, etc.) would not be cost effective. Thus we believe that certain *invariants* must hold during every evolutionary step a domain theory can take. The discovery of these invariants will be a subject of further research.

1.2 Domain Theory Structure and Evolution: HOW

We attempt to treat domain theory constructors as formal specifications of processes. Domain theory constructors assume their arguments already exist and that processes exist which can map arguments to correct results. Domain theory evolution is realized by processes enacted by humans and computers which produce the arguments to the constructors and compute the results of constructor application. These processes require user interfaces by which domain experts can view and evolve domain theories. Knowledge acquisition tools provide such processes and user interfaces.

1.3 Comparison of Knowledge Acquisition Tools Within the Framework

This survey examines the body of work on knowledge acquisition tools. Representative, example knowledge acquisition tools presented in this survey were selected using the following criteria.

- Fully-integrated, end-to-end process support: The tool acquires knowledge from domain experts and automatically synthesizes (and possibly operationalizes) a new, domain-specific performance system.
- Domain experts need apply only existing skill set during knowledge acquisition: The knowledge acquisition tool acquires knowledge from domain experts using representations with which the domain expert is already familiar. In other words, the knowledge acquisition tool does not require the domain expert to gain new skills in knowledge representation, automated reasoning, etc.
- Performance systems solve synthesis problems: Domain-specific performance systems synthesized by the knowledge acquisition tool themselves solve synthesis (i.e., design, configuration, etc.), rather than analysis (i.e., classification, diagnosis, etc.) problems within the domain.

Only a small number of representative example knowledge acquisition tools were found that satisfy these criteria. This survey evaluates Kipps's DTAS/LOLA [17], Iscoe's Ozym [13], Marcus's and McDermott's SALT [20], Musen's PROTÉGÉ [23], and Gruber's ASK [9] knowledge acquisition tools. For each knowledge acquisition tool surveyed, the tool is recast in our architectural metalanguage/framework. Once recast, the architecture of the knowledge acquisition tool indicates how domain theory structuring and evolution are supported. Support is indicated in two ways. The first way a knowledge acquisition tool can support domain theory structuring and evolution is whether or not specific domain theory constructors (i.e., for instances, sorts, sentences, specifications, and institutions) are implemented in the tool. The second way a knowledge acquisition tool can support domain theory structuring and evolution is how the tool's architecture allocates its structure to our reference architecture for knowledge acquisition tools.

1.4 Organization of Survey

In order to ground our discussion and provide a better appreciation for our viewpoint, Section 2 on page 5 provides an introduction to automatic deductive program synthesis in Amphion [19] and describes the structure and evolution of Amphion-style domain theories. The Amphion system introduced us to the need for effective domain theory structure and evolution and has influenced our thinking considerably. Section 3 on page 12 describes an architectural metalanguage/framework for domain theory structure and evolution in terms of atomic domain theory constructors. This provides important background knowledge for Section 4 on page 19, in which we present the formal reviews of the surveyed knowledge acquisition tools. Section 5 on page 57 provides a summary of knowledge acquisition tool evaluations. Section 6 on page 58 presents our conclusions.

2 Fundamentals of Automatic Deductive Program Synthesis in Amphion

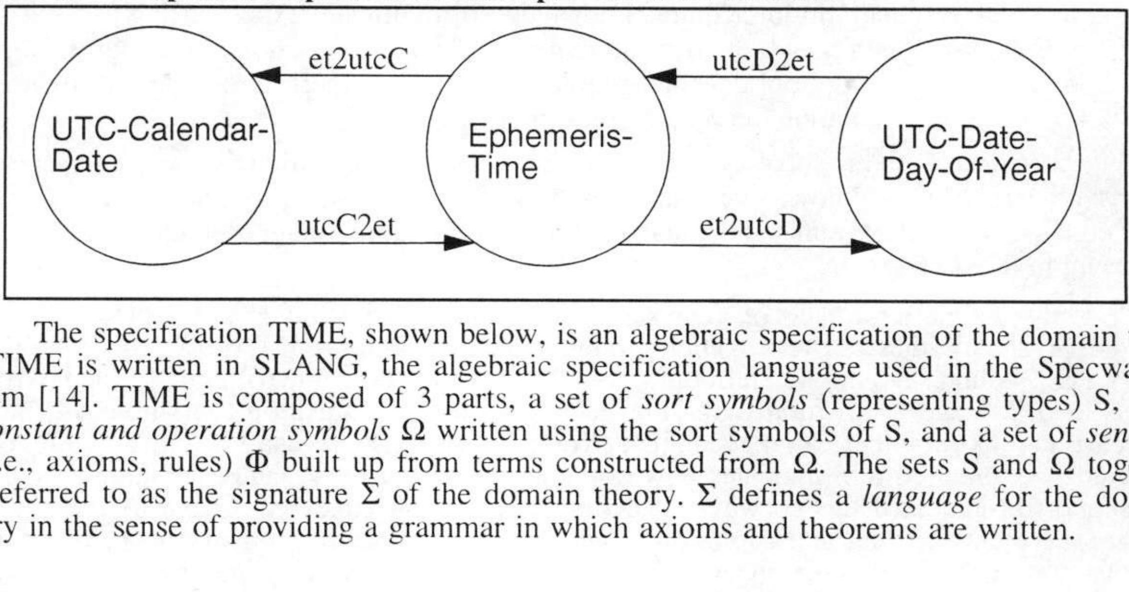
This section provides an introduction to automatic deductive program synthesis as exemplified by the Amphion system [19]. This will provide insight into our viewpoints and motivation for the need for effective domain theory structure and evolution.

2.1 Example Domain Theory

This section describes an example domain theory. This domain theory permits the specification and satisfaction of problems dealing with conversions between 3 systems for time measurement.

Imagine an application domain in which 4 functions (implemented as FORTRAN subroutines) exist to perform conversions between time units as shown in the graph below. Nodes represent time units and arcs represent conversion subroutines between time units.

Figure 2.1 Graphical Depiction of Example Problem Domain



The specification TIME, shown below, is an algebraic specification of the domain theory. TIME is written in SLANG, the algebraic specification language used in the Specware™ system [14]. TIME is composed of 3 parts, a set of *sort symbols* (representing types) S, a set of *constant and operation symbols* Ω written using the sort symbols of S, and a set of *sentences* (i.e., axioms, rules) Φ built up from terms constructed from Ω. The sets S and Ω together are referred to as the signature Σ of the domain theory. Σ defines a *language* for the domain theory in the sense of providing a grammar in which axioms and theorems are written.

```
spec TIME is

sorts TIME, TIME-COORDINATE, TIME-SYSTEM

const Ephemeris-Time : TIME-SYSTEM
const UTC-Calendar-Date : TIME-SYSTEM
const UTC-Date-Day-Of-Year : TIME-SYSTEM

op utcC2et : TIME-COORDINATE -> TIME-COORDINATE
op et2utcC : TIME-COORDINATE -> TIME-COORDINATE
op utcD2et : TIME-COORDINATE -> TIME-COORDINATE
op et2utcD : TIME-COORDINATE -> TIME-COORDINATE

op coordinates-to-time : TIME-SYSTEM,TIME-COORDINATE -> TIME

axiom UTCC-TO-ET is
(equal
   (coordinates-to-time UTC-Calendar-Date tc)
   (coordinates-to-time Ephemeris-Time (utcC2et tc)))

axiom ET-TO-UTCC is
(equal
   (coordinates-to-time Ephemeris-Time tc)
   (coordinates-to-time UTC-Calendar-Date (et2utcC tc)))

axiom UTCD-TO-ET is
(equal
   (coordinates-to-time UTC-Date-Day-Of-Year tc)
   (coordinates-to-time Ephemeris-Time (utcD2et tc)))

axiom ET-TO-UTCD is
(equal
   (coordinates-to-time Ephemeris-Time tc)
   (coordinates-to-time UTC-Date-Day-Of-Year (et2utcD tc)))

end-spec
```

The sort TIME (not to be confused with the enclosing specification TIME) corresponds to the basic quantity of time[1]. The sort TIME-COORDINATE corresponds to a general unit of time, which is interpreted to a specific unit via an instance of sort TIME-SYSTEM[2]. Operations utcC2et, utcD2et, et2utcC, and et2utcD are functions that convert a time between units, while preserving the "abstract value" (i.e., unitless value) of time. These operations represent FORTRAN subroutines performing the same task in the real world. The operation coordinates-to-time is an *abstraction function*, which when given an instance of TIME-COORDINATE and an instance of TIME-SYSTEM, returns the abstract value of time. The reasons for structuring the domain theory in this manner are described in Section 2.5 on page 9.

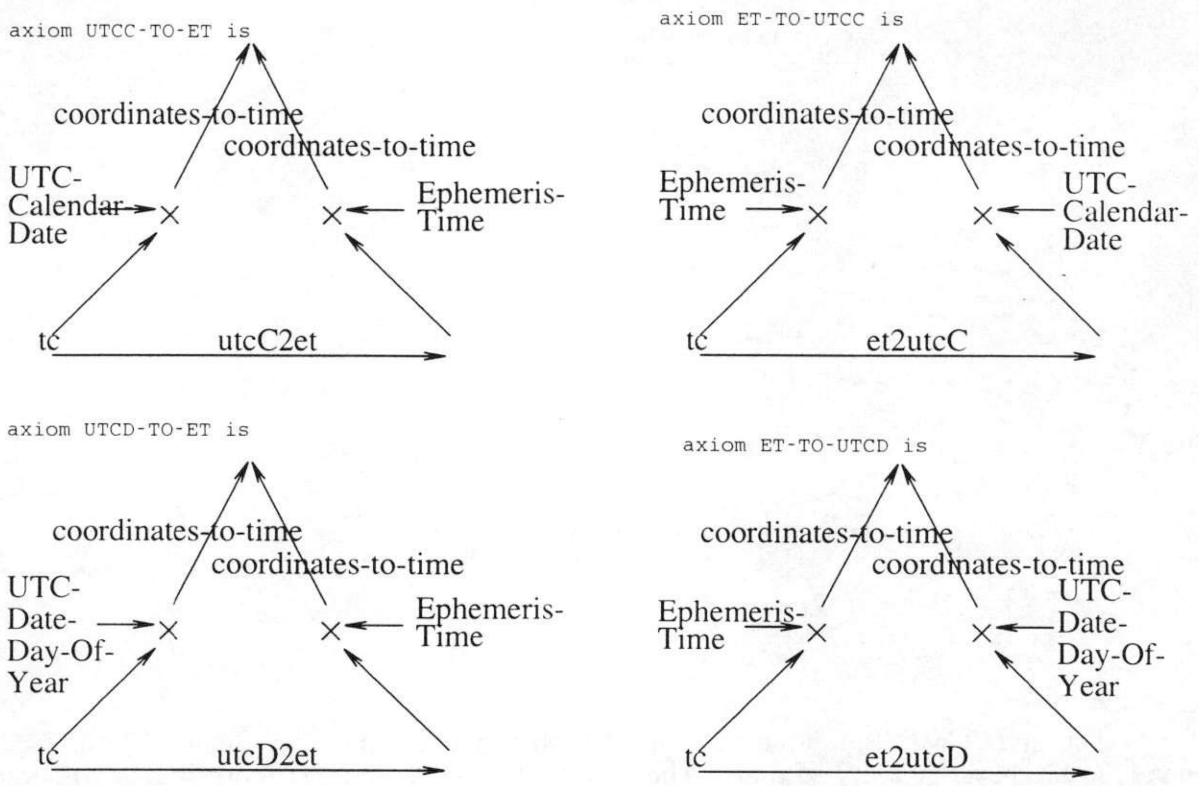
The four axioms of the specification TIME tell us that each conversion operation preserves the abstract value of time, i.e., the conversion subroutines do not change the origin of the clock. The axioms show that if we wish to convert a concrete time value measured in UTC-Calendar-Date to an equivalent time value measured in UTC-Date-Day-Of-Year, we must first convert to and then from an equivalent time value measured in Ephemeris-Time, as indicated by the graph provided earlier. Unless indicated otherwise, all variables are universally quantified.

When designing Amphion domain theories, it is common practice to draw an axiom as a commutative diagram, because most axioms take the form of equalities between terms. Commutative diagrams representing the 4 axioms of the specification TIME follow.

1. Basic quantities in the SI system of measurement are time, length, mass, temperature, current, substance, and luminosity [24].

2. Note that other formulations of this domain theory are possible.

Figure 2.2 Graphical Depiction of TIME's Axioms

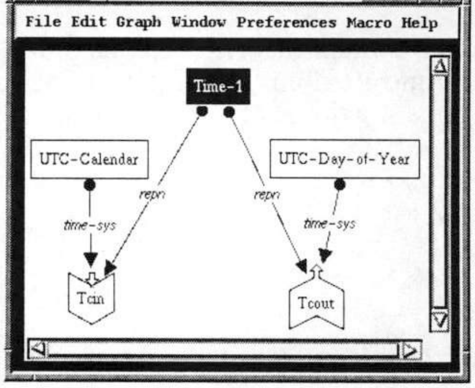


Amphion is a generic knowledge-based program synthesis (KBPS) system that is specialized to a particular application domain by a declarative domain theory, such as TIME. Let us imagine that a new KBPS system, Amphion/TIME, has been so instantiated using TIME and the declarative domain theory.

2.2 Problem Specification

This domain theory can be used to specify and solve (very) simple problems in the application domain of time unit conversions. The diagram below, a screen shot from the specification editor used by Amphion/TIME, specifies a simple problem.

Figure 2.3 Amphion/TIME's Specification Editor


Amphion/NAIF: [ROOT] Utc-C-to-Utc-D
File Edit Graph Window Preferences Macro Help
Time-1
UTC-Calendar
UTC-Day-of-Year
repn
repn
time-sys
time-sys
Tcin
Tcout

This problem specification asks "given a time value measured in UTC-Calendar-Date units (the input), what is the equivalent time value measured in UTC-Date-Day-Of-Year units (the output)?" We say that this problem specification is represented here using a *surface syntax*.

Amphion/TIME converts this graphical specification into an equivalent predicate written using the signature of TIME. We say that this predicate representation of the problem is written using the *abstract syntax* provided by the signature of the domain theory.

```
conjecture UTC-C-TO-UTC-D is
(fa (tcIN)
   (ex (tcOUT time-1)
    (and
      (equal (coordinates-to-time UTC-Calendar-Date tcIN) time-1)
      (equal (coordinates-to-time UTC-Date-Day-Of-Year tcOUT) time-1))))
```

Thus Amphion/TIME's domain theory provides a ready-made *domain-specific specification language*. The Amphion/TIME specification editor is automatically generated from the signature of the Amphion/TIME domain theory.

2.3 Program Synthesis via Resolution Theorem Proving

Amphion/TIME takes the problem specification predicate and attempts to prove that it is a *theorem of the domain theory*. A successful resolution proof yields variable substitutions for the output variable(s), i.e., a *witness* for the proof. These substitutions are applicative terms that are syntactically translated into the concrete syntax of the solution language, which in our case is a FORTRAN program that makes calls to subroutines.

We negate the problem specification predicate as follows.

```
(not
(fa (tcIN)
   (ex (tcOUT time-1)
    (and
      (equal (coordinates-to-time UTC-Calendar-Date tcIN) time-1)
      (equal (coordinates-to-time UTC-Date-Day-Of-Year tcOUT) time-1))))
```

Amphion/TIME next converts the negated problem specification predicate into clause form as follows.

```
(or
 (not (equal
        (coordinates-to-time UTC-Calendar-Date tcIN!)
        time-1))
 (not (equal
        (coordinates-to-time UTC-Date-Day-Of-Year tcOUT)
        time-1)))
```

tcIN!, which represents the input, is now treated as a *constant*.

Resolving this clause first with axiom UTCC-TO-ET and secondly with axiom ET-TO-UTCD produces the empty resolvent and the following substitution for tcOUT.

$$\{tcOUT \leftarrow (et2utcD\ (utcC2et\ tcIN!))\}$$

During deductive synthesis, Amphion/TIME uses a strategy that biases selection of operation symbols to replace abstract (problem specification-oriented) operation symbols with concrete (solution implementation-oriented) operation symbols, thereby ensuring that the applicative term produced contains only operation symbols that represent components from the subroutine library.

2.4 Answer Extraction and Translation

The witness {tcOUT ← (et2utcD (utcC2et tcIN!))} represents a FORTRAN program but written using the signature of the Amphion/TIME domain theory. A syntactic transformation produces the following FORTRAN program from the witness.

```
      SUBROUTINE UTCCTO ( TCIN, TCOUT )

C           Code for Utc-C-to-Utc-D
C           Request-id: REQ-1996-09-16-15-01-40-274

C           Parameters
            INTEGER CLKPRC
            PARAMETER ( CLKPRC = 7 )

C           Input variables
            CHARACTER*(*) TCIN

C           Output variables
            CHARACTER*(*) TCOUT

C           Functions
            LOGICAL RETURN

C           Local variables
            DOUBLE PRECISION E


C           Error handling
            IF ( RETURN() ) THEN
              RETURN
            ELSE
              CALL CHKIN ( 'UTCCTO' )
            END IF

            CALL UTC2ET ( TCIN, E )
            CALL ET2UTC ( E, 'D', CLKPRC, TCOUT )

            CALL CHKOUT ( 'UTCCTO' )
            RETURN

            END
```

Amphion domain theories in use are of course more interesting than our example (Amphion/TIME can only solve 9 distinct I/O pair problems!). However the basic process of synthesizing a program from a specification is the same.

2.5 Structural Style for Amphion Domain Theories

The reader may have noticed that we use the terms "abstract," "concrete," and "abstraction" to describe kinds of sorts and operation symbols in the domain theory. Amphion domain theories are structured in a particular *style* which has been found to be useful for the class of synthesis problems targeted.

Amphion domain theories are structured into 4 parts. An abstract language (i.e., signature) ΣA provides most of the domain-specific specification language. ΣA is designed to provide Amphion end users (i.e., application program developers) with convenient access to software library component functionality.

A concrete language (i.e., signature) ΣC represents sorts and operations that directly correspond to software library components. The design of ΣC can be made more challenging when software library components are non-functional (i.e., causing side-effects), because ΣC operations are purely functional.

A set of abstraction functions ABS which map ΣC sorts to ΣA sorts must also be defined. Abstraction functions are often parameterized by either concrete or abstract sorts. For example, the coordinates-to-time abstraction function of the Amphion/TIME domain theory is parameterized by a value of sort TIME-SYSTEM. Parameters such as these are needed to correctly interpret the concrete values being mapped to abstract values.

The last part of an Amphion domain theory is the collection of axioms DT. Most axioms in an Amphion domain theory are associated with the abstraction functions in ABS. This is because the an Amphion system's main focus is program synthesis, the essence of which is mapping specifications to source code implementations.

Amphion domain theory axioms come in 2 basic styles. The first style are "triangle" axioms representing unit conversions between concrete values, such as the 4 axioms in Amphion/TIME's domain theory. The second style is the "rectangle" style, representing an equality of terms, one of which features an abstract operation and the other of which features a concrete operation (triangle axioms have no abstract operation). A generalized rectangle axiom is shown below in diagram form.

$$\begin{tikzcd} \cdot \arrow[r, "OPabstract"] & \cdot \\ x \arrow[u, "ABSsource"] \arrow[r, "OPconcrete"] & \cdot \arrow[u, "ABStarget"'] \end{tikzcd}$$

This diagram represents the following equation.

```
(fa (x)
    (equal
        (OPabstract (ABSsource x))
        (ABStarget (OPconcrete x)))).
```

In summary, Amphion domain theories are formulated in the style of Hoare [11].

We can further illustrate the structure of Amphion-style domain theories (and domain theories in general) by using a *diagram of specifications and specification morphisms*. Such a diagram is shown adjacent. In such a diagram, nodes represent algebraic specifications (which in turn represent domain theories) and arcs represent specification morphisms between theories.

$$\begin{tikzcd} \Sigma A \arrow[d, "i"'] \\ ABS \\ \Sigma C \arrow[u, "i"] \end{tikzcd}$$

A *specification morphism* is a total function that maps source specification sorts and operations to target specification sorts and operations (possibly renaming them) such that every axiom of the source specification is a theorem of the target specification. In a sense, the source specification is embedded in the target specification. The adjacent diagram shows how the theories ΣA, ΣC, and ABS are related. Most axioms in an Amphion-style domain theory are found in the theory ABS. ΣA and ΣC are mostly syntactic (i.e., have few axioms). In the adjacent diagram, specification morphisms from ΣA to ABS and from ΣC to ABS are decorated with an "i" symbol. This means that these arrows represent *inclusion morphisms*. Inclusion morphisms are a special kind of specification morphism that maps source specification symbols to copies of themselves in the target specification. Hence an inclusion morphism allows a theory to be simply extended. Although Amphion/TIME's domain theory is too small to make a good example of specification structuring, if we broke up Amphion/TIME's domain theory into this style of structure, we would have the following 3 specifications.

```
spec SIGMA-A is
  sorts TIME
end-spec

spec SIGMA-C is
  sorts TIME-COORDINATE, TIME-SYSTEM
  const Ephemeris-Time : TIME-SYSTEM
  const UTC-Calendar-Date : TIME-SYSTEM
  const UTC-Date-Day-Of-Year : TIME-SYSTEM
  op utcC2et : TIME-COORDINATE -> TIME-COORDINATE
  op et2utcC : TIME-COORDINATE -> TIME-COORDINATE
  op utcD2et : TIME-COORDINATE -> TIME-COORDINATE
  op et2utcD : TIME-COORDINATE -> TIME-COORDINATE
end-spec

spec ABS is
  import SIGMA-A,SIGMA-C
  op coordinates-to-time : TIME-SYSTEM,TIME-COORDINATE -> TIME
axiom UTCC-TO-ET is
  (equal
   (coordinates-to-time UTC-Calendar-Date tc)
   (coordinates-to-time Ephemeris-Time (utcC2et tc)))
axiom ET-TO-UTCC is
  (equal
   (coordinates-to-time Ephemeris-Time tc)
   (coordinates-to-time UTC-Calendar-Date (et2utcC tc)))
axiom UTCD-TO-ET is
  (equal
   (coordinates-to-time UTC-Date-Day-Of-Year tc)
   (coordinates-to-time Ephemeris-Time (utcD2et tc)))
axiom ET-TO-UTCD is
   (equal
    (coordinates-to-time Ephemeris-Time tc)
    (coordinates-to-time UTC-Date-Day-Of-Year (et2utcD tc)))
end-spec
```

We can use a diagram of specifications and specification morphisms to also represent the relationships between parts of the domain theory and those theories created by the end user during the problem specification process. In the adjacent diagram, user-spec represents the theory associated with the problem specification constructed by the user. user-prog represents (a specification containing only) the sentence formed by the term replacements produced during synthesis. A specification morphism from user-spec to ΣA indicates that the (single) axiom of user-spec is a theorem of ΣA. Likewise, the specification morphism from user-prog to ΣC indicates that the (single) axiom of user-prog is a theorem of ΣC.

$$\begin{tikzcd}
\text{user-spec} \arrow[r] & \Sigma A \arrow[d, "i"] \\
 & \text{ABS} \\
\text{user-prog} \arrow[r] & \Sigma C \arrow[u, "i"]
\end{tikzcd}$$

Note that the framework only considers the abstract syntax of languages, rather than concrete syntax. (Concrete syntax corresponds to program source code text in an editor or on paper, abstract syntax corresponds to the parse tree of the program's source code.)

2.6 Domain Theory Evolution in Amphion

Amphion-style domain theories must evolve with the application domain in which they are used. The adjacent diagram of specifications and specification morphisms represents domain theory evolution. $\sigma_{\Sigma A}$ is a specification morphism that maps the original ΣA theory to a new theory ΣA´, possibly changing the names of symbols in ΣA. Likewise σ_{ABS} and $\sigma_{\Sigma C}$ are specification morphisms that map the original ABS and ΣC theories to new theories ABS´ and ΣC´ respectively. Collectively $\sigma_{\Sigma A}$, σ_{ABS}, and $\sigma_{\Sigma C}$ *constitute an evolutionary step of the domain theory.*

$$\begin{tikzcd}
\text{user-spec} \arrow[r] & \Sigma A \arrow[r, "\sigma_{\Sigma A}"'] \arrow[d, "i"] & \Sigma A' \arrow[d, "i"] \\
 & \text{ABS} \arrow[r, "\sigma_{ABS}"'] & \text{ABS}' \\
\text{user-prog} \arrow[r] & \Sigma C \arrow[r, "\sigma_{\Sigma C}"'] \arrow[u, "i"] & \Sigma C' \arrow[u, "i"]
\end{tikzcd}$$

How can artifacts (e.g., test cases, end user specifications, end user programs, tools, etc.) created under the original domain theory be preserved during domain theory evolution? We believe a part of the solution is to use morphisms defining evolutionary steps of the domain theory to *translate* artifacts into the new domain theory. Thus the diagram below represents the situation in which $\sigma_{\Sigma A}$ is used to translate user-spec into user-spec´ and $\sigma_{\Sigma C}$ is used to translate user-prog into user-prog´.

Figure 2.4 Translating Domain Artifacts After Domain Theory Evolution

$$\begin{tikzcd}
\text{user-spec} \arrow[r] & \Sigma A \arrow[r, "\sigma_{\Sigma A}"'] \arrow[d, "i"'] & \Sigma A' \arrow[d, "i"'] & \text{user-spec}' \arrow[l] \\
 & ABS \arrow[r, "\sigma_{ABS}"'] & ABS' & \\
\text{user-prog} \arrow[r] & \Sigma C \arrow[r, "\sigma_{\Sigma C}"'] \arrow[u, "i"] & \Sigma C' \arrow[u, "i"] & \text{user-prog}' \arrow[l]
\end{tikzcd}$$

3 Framework for Domain Theory Structure and Evolution

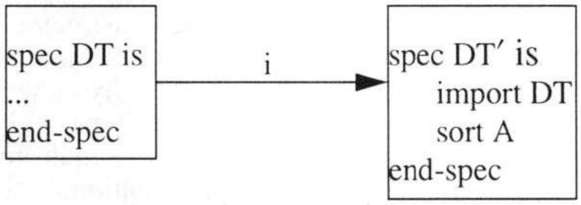
This section introduces a framework in which domain theory structure and evolution can be represented. This framework can be thought of as a language in which to describe the architectures of knowledge acquisition tools. Section 3.1 on page 12 describes atomic domain theory constructors, which form the vocabulary of this language. Section 3.2 on page 16 describes an initial reference architecture for knowledge acquisition tools, which forms a collection of important sentences in this language.

3.1 Atomic Domain Theory Constructors

This subsection describes different categories of atomic domain theory constructors. The framework used in this survey represents evolutionary steps a domain theory takes, such as defined by $\sigma_{\Sigma A}$, σ_{ABS}, and $\sigma_{\Sigma C}$ collectively in figure 2.4 above, by the application of one or more *atomic domain theory constructors*. The diagram of figure 2.4 represents a kind of combined "before and after" picture. The "before" picture is represented by user-spec, user-prog, ΣA, ΣC, ABS, and the specification morphisms between them. The "after" picture is represented by user-spec´, user-prog´, ΣA´, ΣC´, ABS´, and the morphisms between them. The "before" and "after" pictures are linked by $\sigma_{\Sigma A}$, σ_{ABS}, and $\sigma_{\Sigma C}$.

The mere presence of the "after" picture indicates that a collection of atomic domain theory constructors was applied to the specifications in the "before" picture. Application of an atomic domain theory constructor induces the creation of a "primed" specification and a morphism from the unprimed specification to the primed specification. Thus a diagram of specifications and specification morphisms showing domain theory evolution serves as a *visual design record.*

To better ground our discussion, the adjacent diagram illustrates the application of a basic sort constructor and how that application induces an inclusion morphism from the original domain theory (specification DT) to the extended domain theory (specification DT'). In the adjacent diagram, specifications, which usually appear only as labels in diagrams of specifications and specification morphisms, are shown in exploded form showing some of their textual struc-



ture.

In order to restrict the length of this survey, complete details of atomic domain theory constructors will be published under a different report.

3.1.1 Atomic Instance Constructors

An *instance constructor* produces a new instance of a sort, e.g., Cassini is a new instance of sort SPACECRAFT, and Sun-Earth-Photon is a new instance of sort PHOTON. The only symbol introduced is that of the new instance, sort symbols already exist. Currently, instances are either constants or functions, because SLANG (the algebraic specification language used to prototype this survey's framework) does not support free variables or variable with state. Application of an instance constructor to a domain theory causes an evolutionary step which is represented by a *definitional extension* of the domain theory. A definitional extension is an inclusion morphism that only introduces constants and operations that can be completely defined within the original specification. Creating a definitional extension of a programming language is like defining a function using the language and declaring that the function in now a primitive of the language. The semantics of a specification and a definitional extension of the specification are exactly the same. Thus if we consider the domain theory to define a language, then application of instance constructors defines a string in the language. Alternatively, if a domain model consists of a collection of software component classes, then the application of instance constructors defines a program built using those classes.

We index atomic instance constructors by whether they use implicitly- or explicitly-defined sorts. This can make a difference for the domain expert. Depending upon the type, familiar kinds of editors (e.g., tables) could be used to fill in the instance's attribute values. For example, if the new instance is of an explicitly-defined product sort, the domain expert would be able to use the table editor to compare the new instance's attribute values with those of instances of the same sort defined earlier. Alternatively, if the instance is a new function of an implicitly-defined sort, an empty table editor could be automatically generated on the fly in which mapplets of the function could be enumerated.

3.1.2 Atomic Sort Constructors

A *sort constructor* produces a new sort, e.g., CONE is a new sort representing the collection of all pairs selected from sorts RAY and ANGLE (i.e., CONE is the product sort of RAY and ANGLE). We believe the application of a sort constructor to a domain theory causes an evolutionary step which is represented by a *conservative extension* of the domain theory. A conservative extension is an inclusion morphism that preserves satisfiability of the specification. Thus, for example, if a domain model consists of a collection of software component classes, then the application of a sort constructor could produce a new domain model featuring a new, multiply-inherited class. Definitions of atomic sort constructors assume explicit sort construction only.

3.1.3 Atomic Sentence Constructors

A *sentence constructor* deduces or induces a new sentence from a collection of sentences. Logical deduction is a sentence constructor that establishes the validity of a new sentence given a collection of already-existing sentences (i.e., the hypothesis). Logical induction is a sentence constructor that establishes a hypothesis that covers a set of ground (i.e., contain no variables) sentences, declared to be either positive or negative training examples. Sentences can always be constructed manually by human invention, either by typing strings in a text editor or manipulating a graphical representation of sentences (e.g., an AND/OR graph).

3.1.4 Atomic Specification Constructors

A *specification constructor* produces a new domain theory from already-existing theories, e.g., the domain theory of supermarket checkout lines as a renaming of the theory of queues. Application of a specification constructor to a domain theory causes an evolutionary step which is represented by a (possibly a collection of) specification morphism(s). Thus, for example, if we have a domain theory of lists and a domain theory of natural numbers, we can construct a domain theory of lists of natural numbers by mapping the "element" sort of the list domain theory to the "number" sort of the natural number domain theory and computing the *pushout* or *colimit* of the domain theories and mappings between them. Alternatively, for example, if we have a library of software components (the current domain model) for producing graphical user interfaces (GUIs), application of a specification constructor can import another library of components (a different domain model), such as a library of components for calculating statistics. Our choice of specification constructors is taken from [27].

3.1.5 Atomic Institution Constructors

An *institution constructor* produces a new *logical system* in which domain theories can be constructed, e.g., translating a collection of domain theories in sorted, higher-order lambda calculus into a corresponding collection of domain theories in unsorted, first-order logic. We believe that application of an institution constructor to a domain theory causes an evolutionary step which is represented by an *institution morphism*. Our choice of institution constructors is taken from [8].

3.1.6 Summary of Atomic Domain Theory Constructors

Table 1, "Atomic Domain Theory Constructors," on page 14 below summarizes the current collection of atomic domain theory constructors.

Table 1: Atomic Domain Theory Constructors

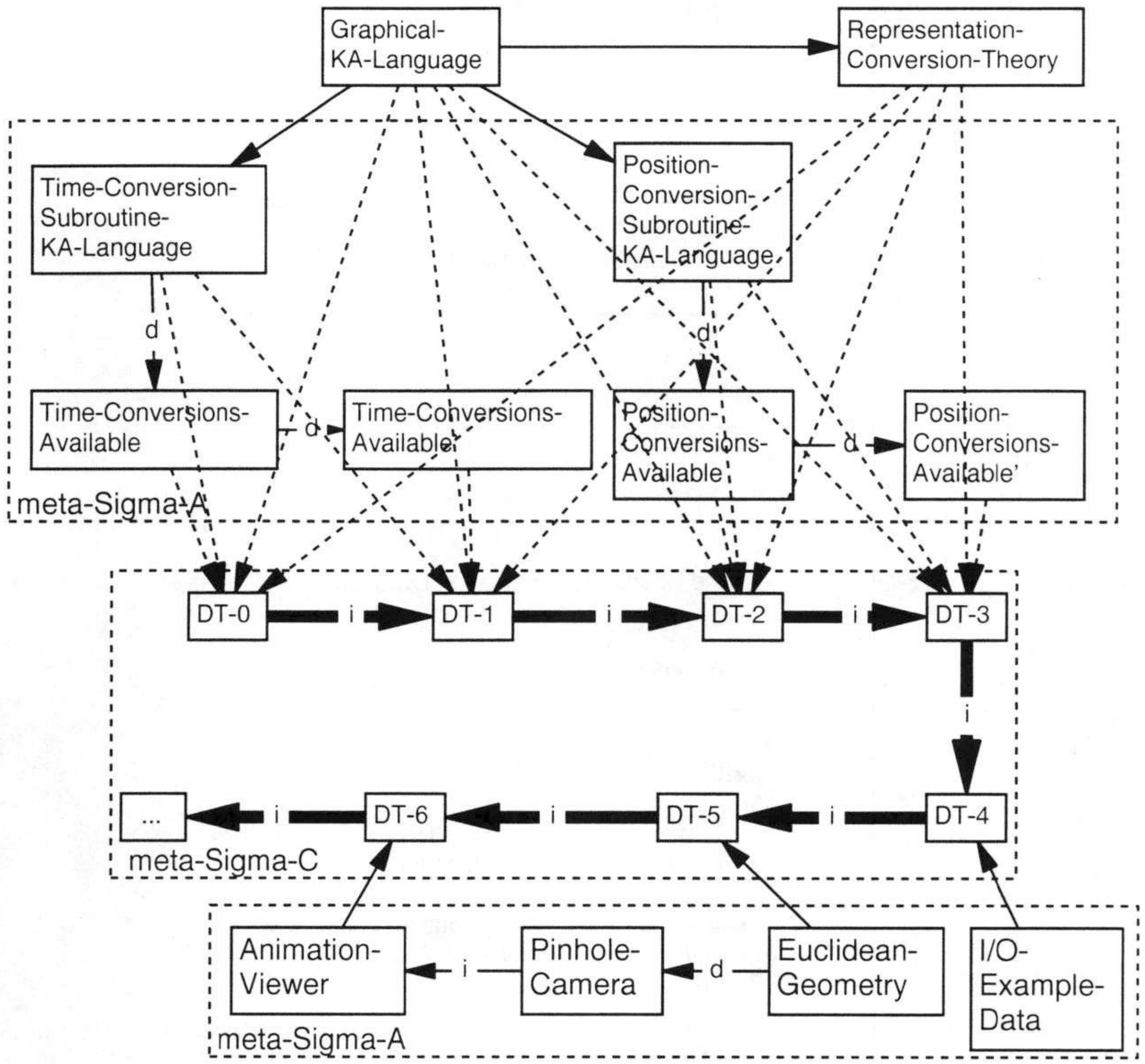
	Instances	Sorts	Sentences	Specifications (i.e., Domain Theories)	Institutions
Symbol Introduction (i.e., "basic" constructors)	i : S	sort S	Human Invention: axiom a is (p x)	spec SP is sorts S ops Ω axioms Φ end-spec	I = (**Sign**, Sen : **Sign** $\rightarrow$ **Set**, **Mod** : **Sign** $\rightarrow$ **Cat**$^{\mathbf{op}}$, $\models_\Sigma$)

Table 1: Atomic Domain Theory Constructors

	Instances	Sorts	Sentences	Specifications (i.e., Domain Theories)	Institutions
Symbol Constructors	(User-defined functions)	S1 , S2 (product)	Deduction	$\cup_{i \in I} SP_i$	$(\Phi,\alpha,\beta) : I1 \rightarrow I2$
		S1 → S2 (function)	Induction	translate SP by σ	Duplex ...
		S1 + S2 (coproduct)	Universal Instantiation	derive from SP′ by σ	Multiplex ...
		S I p (subsort) (where p : S → Boolean)		iso close SP	
				minimal SP wrt σ	
		S / e (quotient sort) (where e : S , S → Boolean)		abstract SP wrt Φ(X)	
				$\lambda X : \Sigma_{par} \bullet SP_{res}$	
		supersort of S_i		colimit of D	
Symbol Identification	axiom (equal i1 i2)	sort-axiom S1 = S2	(iff (p x) (q y))	spec A → spec B	?

Figure 3.1, "Richer Example of Domain Theory Evolution," on page 16 is shown below. It is included in order to provide a larger, visual example of how this framework is applied to represent both the structure and evolution of a domain theory. Evolution of the domain theory is indicated by the path along the bold inclusion morphisms which is framed within the region denoted by meta-ΣC. Structure of the knowledge acquisition tools which evolve the domain theory is indicated by the collections of specifications and morphisms in the region denoted by meta-ΣA. These labels will be described in Section 3.2 on page 16

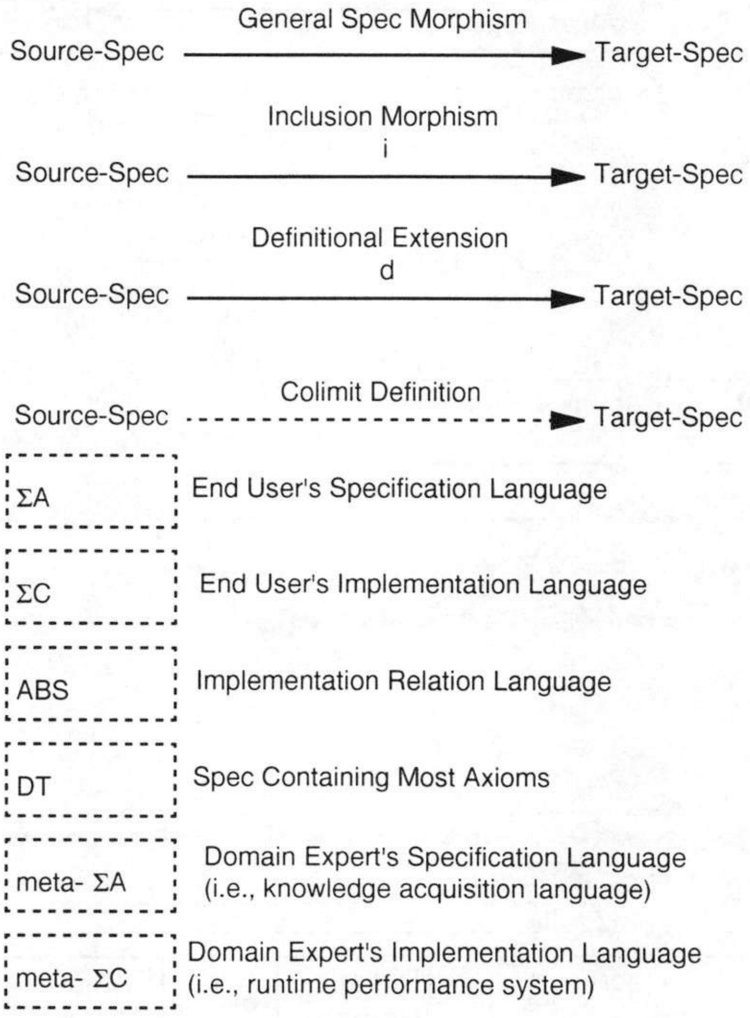
Figure 3.1 Richer Example of Domain Theory Evolution


Graphical-KA-Language
Representation-Conversion-Theory
Time-Conversion-Subroutine-KA-Language
Position-Conversion-Subroutine-KA-Language
d
Time-Conversions-Available
Time-Conversions-Available'
Position-Conversions-Available
Position-Conversions-Available'
meta-Sigma-A
DT-0
DT-1
DT-2
DT-3
DT-4
DT-5
DT-6
...
i
meta-Sigma-C
Animation-Viewer
Pinhole-Camera
Euclidean-Geometry
I/O-Example-Data
meta-Sigma-A

3.2 Draft Reference Architecture for Knowledge Acquisition Tools

In the architectural metalanguage of SAAM [15], a software architecture consists of a set of components and a set of connections between components. Components provide computational services (i.e., processors) or storage services (i.e., files and tables). Connectors enable the flow of data or control between components. The architectural metalanguage/framework for domain theory structuring and evolution used in this survey is somewhat different from SAAM's metalanguage. In our metalanguage, components are algebraic specifications and connectors are specification morphisms. Figure 3.2, "Architectural Metalanguage and Draft Reference Architecture Functional Elements," on page 17 below provides a legend for the architecture diagrams used in Section 4 on page 19.

Figure 3.2 Architectural Metalanguage and Draft Reference Architecture Functional Elements


General Spec Morphism
Source-Spec
Target-Spec
Inclusion Morphism
i
Source-Spec
Target-Spec
Definitional Extension
d
Source-Spec
Target-Spec
Colimit Definition
Source-Spec
Target-Spec
ΣA
End User's Specification Language
ΣC
End User's Implementation Language
ABS
Implementation Relation Language
DT
Spec Containing Most Axioms
meta- ΣA
Domain Expert's Specification Language (i.e., knowledge acquisition language)
meta- ΣC
Domain Expert's Implementation Language (i.e., runtime performance system)

The functional elements of a draft reference architecture for knowledge acquisition systems are shown in the lower half of figure 3.2. These elements group together collections of specifications and specification morphisms (i.e., architecture components and connectors) which provide the functionality required by the element. ΣA is the reference architecture functional element providing a problem specification language for end users (within the performance system). ΣC is the element providing a solution implementation language for end users (within the performance system). ABS is the element providing the implementation relation language between ΣC and ΣA (within the performance system). DT is the element containing the majority of the axioms used by the performance system. meta-ΣA is the element providing a knowledge acquisition language for the domain expert. meta-ΣC is the element providing an implementation language for the domain expert. meta-ΣC corresponds to the performance system as a whole.

Currently in our research the reference architecture functional elements do not form a partition, as is the case in SAAM. At this time the functional elements are related to each other in a way described by the figure below. meta-ΣA should be disjoint from meta-ΣC. We typically expect to find ΣA, ΣC, ABS, and DT within meta-ΣC. ΣA and ΣC should be disjoint, most likely separated by ABS and DT.

Figure 3.3 Preferred Relationship Between Functional Elements

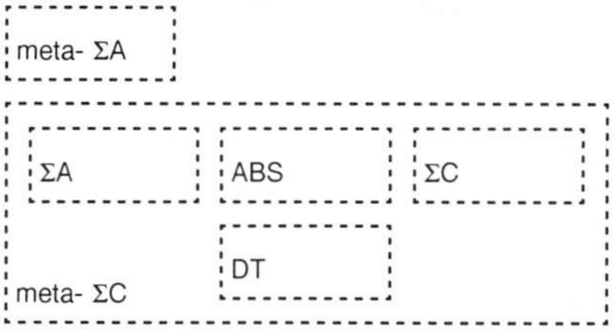
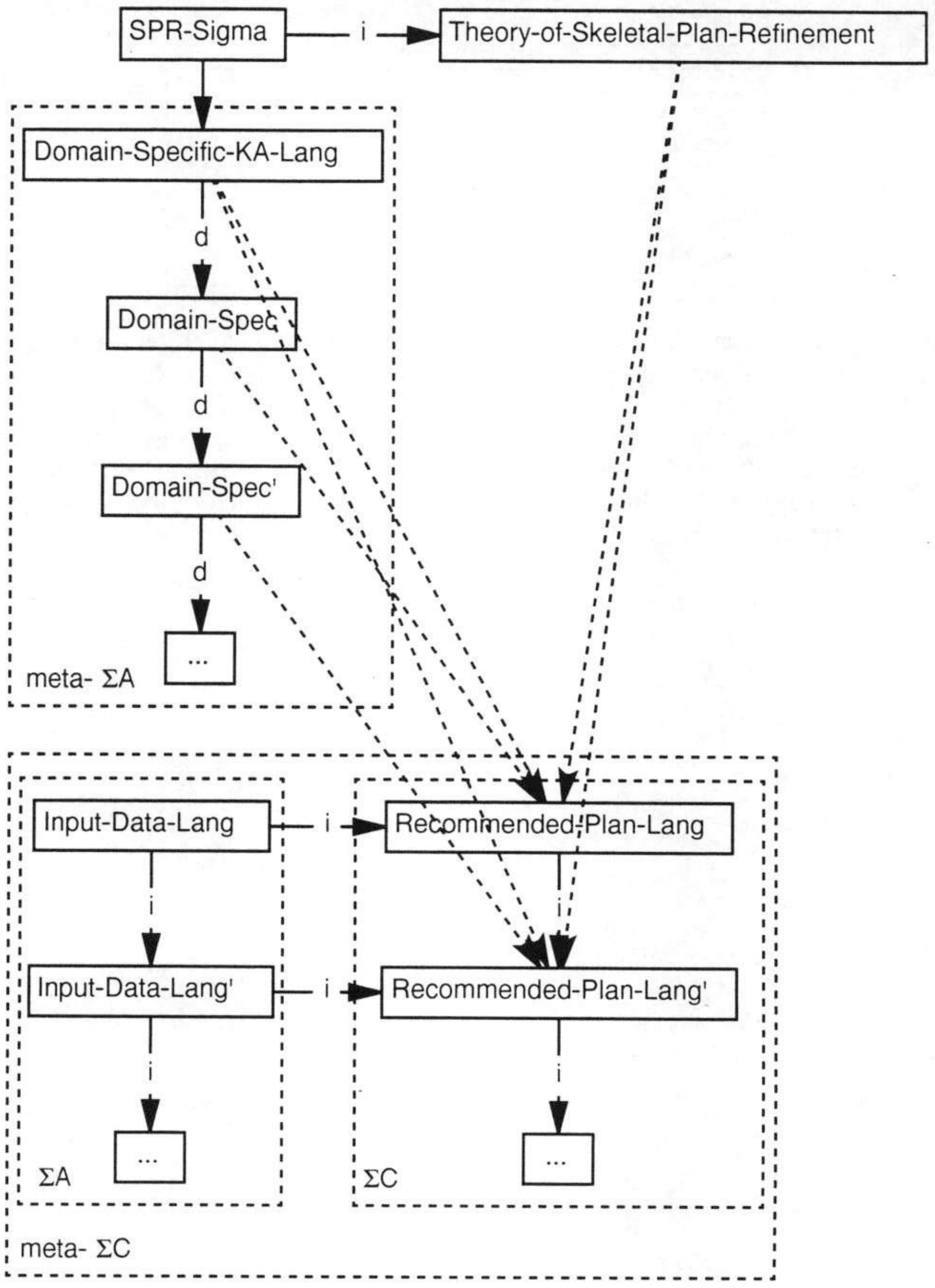


Figure 3.4, "Example Showing Protégé's Assignment of Architectural Components and Connectors to Reference Architecture Functional Elements," on page 19 is shown below as an example. It shows how PROTÉGÉ's architectural components and connectors are assigned to reference architecture functional elements. It is possible that a given tool may have no components and connectors to assign to a reference architecture functional element. For example, PROTÉGÉ does not assign any architectural components or connectors to the reference architecture functional elements of ABS and DT.

Figure 3.4 Example Showing PROTÉGÉ's Assignment of Architectural Components and Connectors to Reference Architecture Functional Elements



4 Reviews of Knowledge Acquisition Tools

This section reviews the selected knowledge acquisition tools by recasting them within our architectural metalanguage/framework and examining which atomic domain theory constructors they implement and how their architectural components and connectors are assigned to the draft reference architecture for knowledge acquisition tools.

When we recast a knowledge acquisition tool within the framework, we attempt to represent a typical domain model on which the actual tool has been applied. The framework always represents domain models as domain theories. This is because the framework is based upon formal, declarative theories, rather than semiformal or non-declarative domain models.

Recasting a knowledge acquisition tool within the framework requires a considerable amount of interpretation on the part of the evaluator. This is because each reviewed tool uses its own representation for domain models, which may or may not correspond to the structure imposed during the recasting process. Recasting knowledge acquisition tools within the framework is thus far an art form that must fill in many knowledge representation-related omissions in the original publications about the tools. The reader should view these architectural diagrams as the evaluator's best idea of how to reimplement the knowledge acquisition tool using algebraic specification and specification morphisms.

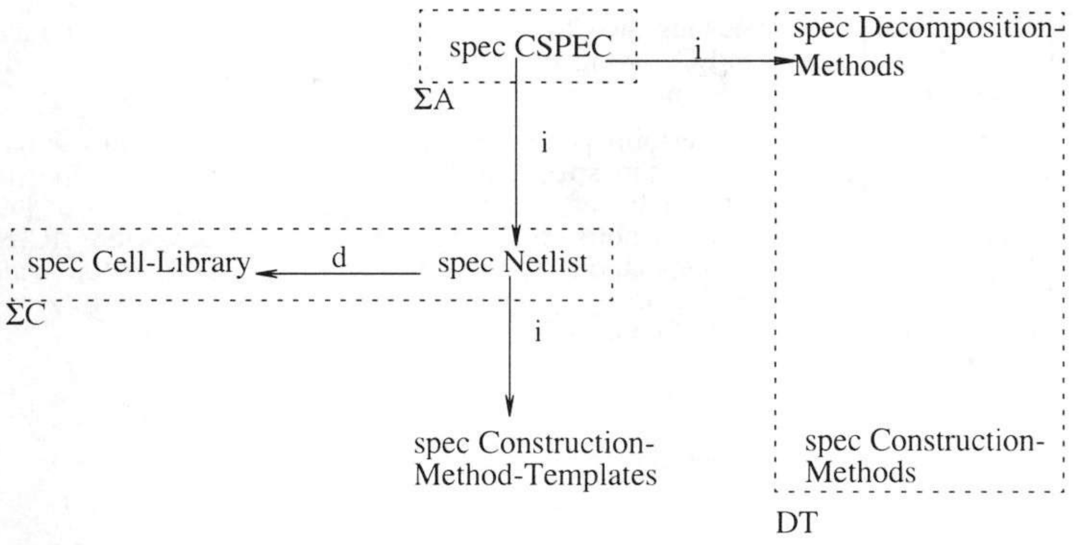
4.1 LOLA

LOLA (Logic Learning Assistant) is the knowledge acquisition tool developed for DTAS (Design Technology Adaptation System) [17]. The performance systems synthesized by DTAS are hardware register transfer level (RTL) design synthesis environments. LOLA enables a domain expert to evolve domain models in parallel with evolution of a library of reusable RTL cells. The performance system composes cells together to create designs which satisfy a high-level specification.

4.1.1 WHAT LOLA Is

When represented within the framework, a DTAS performance system consists of a collection of specifications and specification morphisms as shown in the diagram below.

Figure 4.1 LOLA's Languages


spec CSPEC
ΣA
i
spec Decomposition-Methods
i
spec Cell-Library
d
spec Netlist
ΣC
i
spec Construction-Method-Templates
spec Construction-Methods
DT

- **spec** CSPEC defines the language of component specifications (sort cspec) (i.e., the end user's problem specification language and abstract part of the domain theory, ΣA).
- **spec** Netlist defines the language of component implementations (sort cimpl).
- **spec** Cell-Library represents the contents of the chosen hardware cell library currently used by the performance system (**spec** Netlist and **spec** Cell-Library together make the end user's solution language and concrete part of the domain theory, ΣC).
- **spec** Decomposition-Methods contains axioms showing how a cspec can be decomposed into an assembled collection of other cspec's.

- **spec** Construction-Methods contains axioms showing how a cspec can be decomposed into an assembled collection of cspec's and component implementations (cimpl's), drawn from **spec** Cell-Library. **spec** Decomposition-Methods and **spec** Construction-Methods together form the collection of axioms used during design synthesis. In an Amphion-style domain theory, these specifications together would form DT.
- **spec** Construction-Method-Templates contains axioms used to *instantiate new axioms* in **spec** Construction-Methods, drawing from available cells (cimpl's) in **spec** Cell-Library. **spec** Construction-Method-Templates helps automate the knowledge acquisition process.

A DTAS performance system (i.e., design environment) takes as input from the end user a succinct specification of a *functional unit*, called a CSPEC (for component specification), such as an adder (ADD), multiplier (MULT), or arithmetic and logic unit (ALU). The language of CSPECs is shown below in **spec** CSPEC. Once defined, **spec** CSPEC need not change during evolution of a DTAS design environment.

```
spec CSPEC is

sorts cspec,type,iports,oports,attrs,port,name,value,width

sort-axiom cspec = type,iports,oports,attrs
sort-axiom type = string
sort-axiom iports = nat,port -> boolean
sort-axiom oports = nat,port -> boolean
sort-axiom attrs = nat,(name,value) -> boolean
sort-axiom port = name,width
sort-axiom name = string
sort-axiom value = nat + string
sort-axiom width = nat

end-spec
```

spec CSPEC was written by reviewing the CSPEC grammar definition in [17][3]. Additional axioms could be added to **spec** CSPEC, giving it semantics in addition to those provided by the sort axioms of **spec** CSPEC, but we omit them because of limited time resources.

The end user of a DTAS performance system constructs problem specifications using the language of **spec** CSPEC. Below in **spec** ALU-32-4-4-8 we show an example problem specification (using abstract syntax only) of an ALU, borrowed from Section 9 of [17]. The user specified that the ALU has input ports I0 (32 bits wide), I1 (32 bits wide), ICIN (1 bit wide), ISEL (4 bits wide), and output ports O0 (32 bits wide), OCOUT (1 bit wide), and OREL (1 bit wide); that the ALU is to be implemented in a "ripple" style with zero levels; and that the area and delay are not of concern.

3. Sorts correspond to *nonterminal symbols* in the grammar and sort-axioms, which define new sorts from other sorts by the application of various sort constructors, correspond to *production rules* of the grammar. When the right hand side of a grammar production rule features a list of elements (e.g., iports ::= [port*]), the list is represented by a relation on nat and the sort of the list element (e.g., sort-axiom iports = nat,port->boolean). Alternatives in grammar rules (e.g., value ::= nat | string) are represented by coproducts (i.e., disjoint unions) (e.g., sort-axiom value = nat + string).

```
spec ALU-32-4-4-8 is

import CSPEC

const ALU-32-4-4-8 : cspec
const ips : iports
const ops : oports
const ats : attrs

definition of ALU-32-4-4-8 is
axiom
(and (and (and (and (and (and
              (ips 0 <"I0" 32>)
              (ips 1 <"I1" 32>))
              (ips 2 <"ICIN" 1>))
        (ips 3 <"ISEL" 4>))
        (and (and
         (ops 0 <"O0" 32>)
         (ops 1 <"OCOUT" 1>))
         (ops 2 <"OREL" 1>)))
        (and
         (ats 0 <"STYLE" ((embed 2) "RIPPLE")>)
         (ats 1 <"LEVELS" ((embed 1) 0)>)))
 (equal
  ALU-32-4-4-8
  <"ALU" ips ops ats>))
end-definition

end-spec
```

A DTAS performance system takes a problem specification such as this and incrementally transforms it from a CSPEC to a netlist, by the application of decomposition methods and construction methods. Transformations act upon the problem representation until it ceases to contain CSPECs. CSPECs are replaced by CIMPLs, which are drawn from cell library components (i.e., **spec** Cell-Library). A DTAS performance system uses a depth-first search algorithm to explore the space of netlist representations and returns a set of representations with the best area and delay characteristics.

A netlist is a data structure representing hierarchical hardware designs. **spec** Netlist shown below defines the language (using abstract syntax) of these data structures. Once defined **spec** Netlist need not change during evolution of a DTAS performance system. Notice **spec** Netlist imports **spec** CSPEC (hence the inclusion morphism from **spec** CSPEC to **spec** Netlist). This means that all the sorts, operations, and axioms of **spec** CSPEC are included in **spec** Netlist. Additional axioms could be written for **spec** Netlist defining constraints on parts of the language, but they are omitted.

```
spec NETLIST is % The implementation language & KA language.

import CSPEC

sorts
  netlist,ipins,opins,wires,cells,cspecs,modules,
  pin,wire,module,cimpl,props,src,snk,list

sort-axiom netlist = ipins,opins,wires,cells,cspecs,modules
sort-axiom ipins = nat,pin -> boolean
sort-axiom opins = nat,pin -> boolean
sort-axiom wires = nat,wire -> boolean
sort-axiom cells = nat,(name,cspec) -> boolean
sort-axiom cspecs = nat,(name,cspec) -> boolean
sort-axiom modules = nat,(name,module) -> boolean
sort-axiom module = ipins,opins,cspec,cimpl
sort-axiom cimpl = name,cspec,props %+ cspec,netlist
sort-axiom props = nat,(name,value) -> boolean
sort-axiom pin = string
sort-axiom wire = name,src,snk
sort-axiom src = pin
sort-axiom snk = nat,pin -> boolean % at least one
sort-axiom list = nat,value -> boolean

end-spec
```

A cell library is a collection of reusable design units, each providing a behavior, which can be assembled and instantiated in silicon or other implementation medium. Cell libraries are made available by vendors in much the same way that mathematical and statistical software component libraries are provided by other software vendors. Cells are designed for single logic gates (e.g., AND) and complex, hierarchical units (e.g., MULT). An advantage of using cells is that their predefined implementations are often faster and smaller than custom-made, functionally-equivalent compositions of gates. Cell libraries accompany fabrication technology that reproduces composed cells in the implementation medium. We assume that cells behave according to their specification and need never be inspected, i.e., cells are "black boxes." As borrowed from Section 9 of [17], **spec** Cell-Library below is an example cell library, containing a single cell, ND2.

```
spec CELL-LIBRARY is

import NETLIST

const ND2 : cimpl
const ips : iports
const ops : oports
const ats : attrs
const prps : props

definition of ND2 is
axiom
(and (and (and (and
        (ips 0 <"I0" 2>)
        (ips 1 <"I1" 2>))
        (ops 0 <"O0" 2>))
        (and
          (ats 0 <"STYLE" ((embed 2) "RIPPLE")>)
          (ats 1 <"LEVELS" ((embed 1) 0)>)))
  (equal
   ND2
   <"ND2" <"NAND" ips ops ats> prps>))
end-definition

end-spec
```

ND2 is a component implementation (CIMPL), performing the NAND behavior with inputs I0 (2 bits wide) and I1 (2 bits wide), output O0 (2 bits wide), and with (omitted) area and delay values.

spec Decomposition-Methods contains dozens of axioms showing how a CSPEC can be replaced (or implemented) by a assembled collection of other CSPECs. Because the decomposition methods are cell library-independent, once **spec** Decomposition-Method is defined, it need not evolve during evolution of a DTAS performance system. We shall omit presentation of **spec** Decomposition-Methods.

spec Construction-Methods contains axioms showing how a CSPEC can be replaced (or implemented) by an assembled collection of CSPECs and CIMPLs. The only difference between a decomposition method and a construction method is that the right hand side of the construction method must contain at least one CIMPL from the cell library. Construction methods are always cell library-dependent and the set of construction methods changes every time the cell library changes. If the cell library is extended with a collection of new cells, new construction methods must be generated that use the new cells.

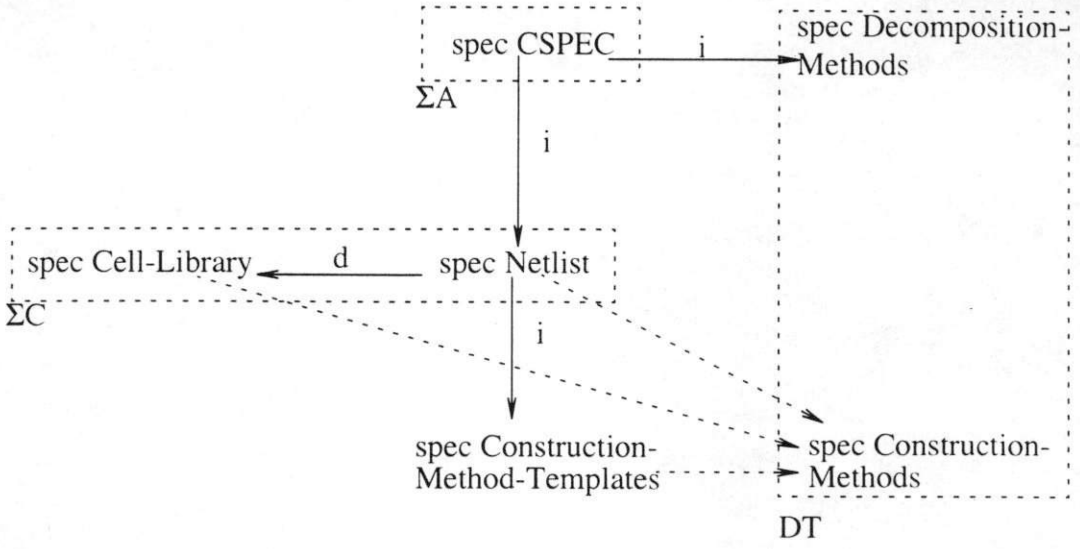
LOLA, the knowledge acquisition subsystem of DTAS, automatically generates construction methods from the current contents of the cell library. LOLA automatically generates construction methods by using a predefined set of *acquisition templates*. We represent an acquisition template as a template for instantiating a construction method axiom given that a certain kind of cell exists in the library and that another kind of cell does not exist in the library. The set of acquisition templates does not evolve. **spec** Construction-Method-Templates shown below demonstrates the concept.

```
spec CONSTRUCTION-METHOD-TEMPLATES is

import NETLIST

axiom
(fa (nand-cimpl-in : cimpl
     nand-cimpl-in-name : name)
 (iff
  (and (and
   (equal
    ((project 1) ((project 2) nand-cimpl-in))
    "NAND")
   (equal
    ((project 1) nand-cimpl-in)
    nand-cimpl-in-name))
   (not (ex (inv-cimpl-out : cimpl)
    (equal
     ((project 1) ((project 2) inv-cimpl-out))
     "INV"))))
  (fa (I0 : name
       WI0 : width
       O0 : name
       WO0 : width
       ni : iports
       no : oports
       ii : iports
       io : oports)
   (and (and (and (and (and
          (ni 0 <I0 WI0>)
          (no 0 <O0 WO0>))
          (ii 0 <I0 WI0>))
          (ii 1 <"pos-grnd" 1>))
    (io 0 <O0 WO0>))
    (implies
     (fa (inv-cspec : cspec)
      (equal
       inv-cspec
       <"INV" ni no ats1>))
     (equal
      nand-cimpl-in
      <nand-cimpl-in-name <"NAND" ii io ats2> prps>))))))

end-spec
```

The axiom states, if we find that the cell library contains a NAND cell, but the cell library does not contain an inverter (INV) cell, then and only then an axiom applies that takes a CSPEC of an inverter and replaces it by the library's NAND cell, but with one of the NAND cell's input ports ground to positive.

What does it mean to generate a specification, such as **spec** Construction-Methods? We represent the generation of **spec** Construction-Methods in part by taking the *colimit* of **spec** Netlist, **spec** Cell-Library, and **spec** Construction-Method-Templates, as shown in the diagram below.

Figure 4.2 LOLA's Architecture


spec CSPEC
i
spec Decomposition-Methods
ΣA
i
spec Cell-Library
d
spec Netlist
ΣC
i
spec Construction-Method-Templates
spec Construction-Methods
DT

The colimit produces the union of a collection of specifications while taking into account the sharing of sorts and operation symbols indicated by the specification morphisms. In the framework, which is thus far implemented using Specware™ [14], a colimit is defined by first defining the diagram from which the colimit will be computed. For the DTAS example, this diagram definition is shown below.

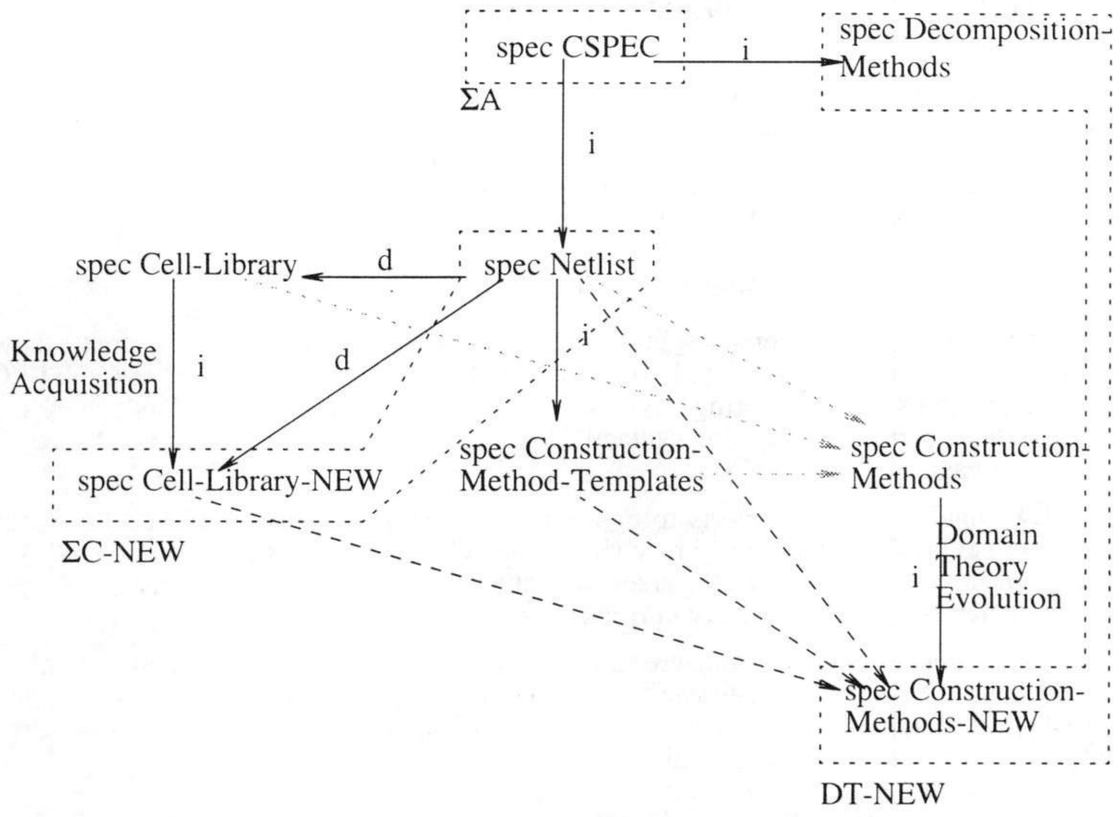
```
diagram DIAGRAM--INSTANTIATE-CONSTRUCTION-METHODS is

nodes
CSPEC,NETLIST,CELL-LIBRARY,
DECOMPOSITION-METHODS,CONSTRUCTION-METHOD-TEMPLATES

arcs
CSPEC -> NETLIST : {},
NETLIST -> CELL-LIBRARY : {},
CSPEC -> DECOMPOSITION-METHODS : {},
NETLIST -> CONSTRUCTION-METHOD-TEMPLATES : {}

end-diagram
```

The colimit is computed by interpreting the specification below.

```
spec CONSTRUCTION-METHODS is
colimit of DIAGRAM--INSTANTIATE-CONSTRUCTION-METHODS
```

Thus the domain theory of a DTAS performance system evolves in lockstep with evolution of the performance system's cell library. Knowledge acquisition occurs when the domain expert extends the cell library and domain theory evolution occurs automatically by recomputing **spec** Construction-Methods from a new colimit diagram. This approach to domain theory evolution is captured succinctly by the diagram below.

Figure 4.3 LOLA's Approach to Domain Theory Structuring and Evolution



This approach to domain theory evolution was formalized in [26] as part of research in automating domain theory evolution for Amphion domain theories. This approach to domain theory evolution appears to be especially applicable under the following conditions.

- The end user problem specification language does not evolve.
- The base language used to represent the component library does not evolve (definitional extensions such as from Netlist to Cell-Library are allowed).
- The direction of possible component library evolution is fully understood a priori.

4.1.2 HOW LOLA Does It

Until now, we only discussed the architecture of LOLA knowledge acquisition tools and synthesized DTAS performance systems. Now we discuss the *processes and UIs* LOLA uses to implement the atomic domain theory constructors and compositions thereof. Each subsection address whether LOLA/DTAS supports the atomic domain theory constructor mentioned and if so, how (i.e., by what processes and UIs) it is supported or implemented and also for which roles (e.g., domain expert, end user, or the system itself) the constructor is supported.

4.1.2.1 Instance of a Sort (generic example SLANG code shown adjacent)

```
spec DT is
sorts T,...
...
op t-constr : -> T
...
end-spec %DT

spec DT' is
import DT
const i : T
axiom def-of-i is
(equal i (t-constr))
end-spec %DT'
```

The simplest kind of evolutionary step a domain theory can undergo is the addition of a new symbol which is an instance of previously-defined sort(s). This kind of evolutionary step is exemplified by the adjacent (pseudo) SLANG code. The declaration of a new instance of a sort requires symbol introduction (i.e., const i : T) and an axiom assigning a value to the symbol (i.e., def-of-i).

4.1.2.1.1 Instance of Explicitly-defined Sort

A DTAS performance system enables end users (i.e., hardware designers) to create new instances of sort cspec and of associated sorts for problem specification (See "spec CSPEC is" on page 21.) presumably by entering text via a text editor. DTAS parses and checks these specifications to ensure that they are syntactically and semantically correct. See "spec ALU-32-4-4-8 is" on page 22.

LOLA enables domain experts to create new instances of sort cimpl and of associated sorts to define new cells in the cell library (See "spec CELL-LIBRARY is" on page 23.) again presumably via a text editor. LOLA parses and checks these cell descriptions to ensure that they are syntactically and semantically correct.

A DTAS performance system creates new instances of sorts cimpl and netlist during design synthesis. LOLA itself creates new instances of sort cspec during "technology compilation," in which it instantiates new construction methods based upon new instances of sort cimpl added to the cell library by the domain expert.

4.1.2.1.2 Instance of Operation, i.e., Instance of Implicitly-Defined Function Sort (generic example SLANG code shown adjacent)

```
spec DT is
sorts T1,T2,...
...
op g : T1,T2,T2 -> T2
...
end-spec %DT

spec DT' is
import DT
op f : T1 -> T2
axiom def-of-f is
(equal
   (f t)
   (g t u v))
end-spec %DT'
```

Neither DTAS nor LOLA supports the ability to construct a new operation[4]. This would require the ability to define new axioms defining the *behavior* of the new operation. Instances of sorts cspec and cimpl are purely syntactic and provide no explicit definition of behavior. This is why end users and domain experts can define only new examples of "functional units" from the (short) list of functional units supported by DTAS (i.e., ADD, MULT, MUX, ALU, AND, OR, etc.).

If this constructor was supported for the domain expert, the domain expert could represent components as functions. If the functions are first-order, arbitrary combinational (i.e., no internal state) logic devices could be added to the cell library. For example, if the domain expert wishes to add a component to the cell library that determines whether logical implication hold over the input signals, it could be represented as follows.

4. Recall that sorts iports, oports, and attrs are represented as function sorts (because our framework uses functions to represent predicates and lists). So, if an end user or domain expert defines an instance of iports, has a new function been defined? Technically yes, but sort iports is explicitly defined in DTAS. This section only asks whether functions of implicitly-defined sort can be constructed.

```
spec CELL-LIBRARY is
sort BIT
...
op bit-to-boolean : BIT -> boolean % abstraction function
axiom def-of-bit-to-boolean is
(and
    (equal true (bit-to-boolean 1))
    (equal false (bit-to-boolean 0)))
...
end-spec

spec CELL-LIBRARY' is
import CELL-LIBRARY
op IMPLIES-COMP-1 : BIT , BIT -> BIT % new component defined as a function
axiom
(iff
    (implies
        (bit-to-boolean b1)
        (bit-to-boolean b2))
    (bit-to-boolean (IMPLIES-COMP-1 b1 b2)))
end-spec
```

Alternatively, the new operation IMPLIES-COMP-1 could be represented as a predicate as follows. This representation is advantageous if circuits with cycles must be represented.

```
op IMPLIES-COMP-1 : BIT , BIT , BIT -> boolean
```

If second-order functions were allowed, arbitrary sequential (i.e., with internal state) devices could be added to the cell library. This would enable the cell library to contain much larger-grained components than are currently possible in DTAS. Sequential components could be represented as follows. Wires would be represented by signal functions from Natural numbers to BIT. Components are represented by predicates that constrain the behavior of signal functions.

```
spec CELL-LIBRARY is
import Nat
sort signal
sort-axiom signal = Nat -> BIT
...
end-spec

spec CELL-LIBRARY' is
import CELL-LIBRARY
op D-FLIP-FLOP-1 : signal , signal -> boolean
axiom def-of-D-FLIP-FLOP-1 is
...
end-spec
```

This representation of components would have explicit behavior (i.e., semantics) provided by axioms defining the new operations (i.e., component representations). Unfortunately, DTAS's representation for components is purely syntactic and hence adding the ability to define new operations would be a significant change, possibly effecting most aspects of the system.

4.1.2.2 Sorts

A more interesting kind of evolutionary step a domain theory can undergo is the addition of a new sort symbol which may or may not be structurally defined in terms of sorts defined earlier. Adding a new sort can effectively create a new non-terminal symbol in the languages of the domain models (i.e., change the syntax of problem specification and solution representation languages). Unfortunately, neither DTAS nor LOLA supports the ability to construct new sorts. All sorts are predefined and cannot be added to by domain experts, end users, or DTAS/ LOLA itself.

4.1.2.2.1 Product Sort (generic example SLANG code adjacent)

A product sort defines a new sort which represents the Cartesian product of the argument sorts. The Product Sort constructor is not supported in DTAS/LOLA. If new product sorts could be defined, it would allow the domain expert to extend the languages of the domain model with new juxtapositions of previously-defined nonterminal symbols.

```
spec DT is
sorts T1,T2,...
...
end-spec %DT

spec DT' is
import DT
sorts T
sort-axiom T = T1,T2
end-spec %DT'
```

4.1.2.2.2 Coproduct Sort (generic example SLANG code adjacent)

A coproduct sort defines a new sort which represents the disjoint union of the argument sorts. The Coproduct Sort constructor is not supported in DTAS/LOLA. If new coproduct sorts could be defined, it would allow the domain expert to extend the languages of the domain model with new nonterminals defined as alternatives of previously-defined nonterminal symbols.

```
spec DT is
sorts T1,T2,...
...
end-spec %DT

spec DT' is
import DT
sorts T
sort-axiom T = T1+T2
end-spec %DT'
```

4.1.2.2.3 Function Sort (generic example SLANG code adjacent)

A function sort defines a new sort which represents the set of all functions from the source sort to the target sort. The Function Sort constructor is not supported in DTAS/LOLA. If new function sorts could be defined, it would allow the domain expert to more easily define signal functions (Section 4.1.2.1.2 on page 27) and groups of components with identical input/output signature (e.g.,

```
sort-axiom binary-logic-gate = BIT,BIT -> BIT).
```

```
spec DT is
sorts T1,T2,...
...
end-spec %DT

spec DT' is
import DT
sorts T
sort-axiom T = T1->T
end-spec %DT'
```

4.1.2.2.4 Subsort (generic example SLANG code adjacent)

A subsort defines a new sort which represents a restricted set of values of the base sort, defined by some predicate on the base sort. The Subsort constructor is not supported in DTAS/LOLA. If new subsorts could be defined, it would allow the domain expert to organize components, wires, netlists, etc. into arbitrary subgroups. For example, the domain expert could group instances of cimpl into fast and non-fast groups as follows (using simplified syntax).

```
spec DT' is
import DT,Nat
op fast-cimpl? : cimpl -> boolean
sort-axiom cimpl-fast = cimpl | fast-cimpl?
axiom def-of-fast-cimpl? is
(iff
    (fast-cimpl? ci)
    (lte (delay-of ci) 10))
end-spec %DT'
```

```
spec DT is
sorts T,...
...
end-spec %DT

spec DT' is
import DT
sorts T1
op p? : T->boolean
sort-axiom T1 = T|p?
axiom def-of-p? is
(iff
    (p? t)
    ...)
end-spec %DT'
```

4.1.2.2.5 Quotient Sort (generic example SLANG code adjacent)

A quotient sort defines a new sort representing elements which are equivalence classes over the base sort. The equivalence classes are defined by an equivalence relation, which must also be defined. The Quotient Sort constructor is not supported in DTAS/LOLA. To apply quotient sorts, we need to ask ourselves "where in a DTAS domain theory can instances of a sort be gathered into equivalence classes?" An example equivalence class on cspec might be same-functional-unit-as? (e.g., ALU, AND, MUX, etc.). This would be a more elegant representation that the current DTAS functional unit representation as a string.

```
spec DT is
sorts T,...
...
end-spec %DT

spec DT' is
import DT
sorts T1
op e? : T,T->boolean
sort-axiom T1 = T/e?
axiom def-of-e? is
(iff
    (e? ta tb)
    ...)
end-spec %DT'
```

4.1.2.2.6 Supersort (generic example SLANG code adjacent)

```
spec DT is
sorts T,U,A,B,C...
sort-axiom T=A,B,C
sort-axiom U=B,C
...
end-spec %DT

spec DT' is
import DT
sorts T1
sort-axiom T1 = B,C
end-spec %DT'
```

A supersort is constructed from a collection of previously-defined product sorts and is the largest product sort common to all argument sorts. The supersort constructor is not a primitive in SLANG but we believe it can be implemented in SLANG. Supersorts are frequently constructed during reengineering of domain models. The Supersort constructor is not supported in DTAS/LOLA. If it was supported, it would enable the domain expert to define a new nonterminal in the languages of the domain model, being the maximal nonterminal of common nonterminal making up the argument nonterminals. The following concrete (but weak) example shows how a new sort, pins, is the supersort of sorts netlist and module.

```
spec NETLIST is
import CSPEC
...
sort-axiom netlist = ipins,opins,wires,cells,cspecs,modules
...
sort-axiom module = ipins,opins,cspec,cimpl
...
end-spec

spec NETLIST' is
import NETLIST
sort pins
sort-axiom pins = ipins,opins
end-spec
```

4.1.2.3 Sentences

Sentences are predicates written using variables, constants, and operations from a previously-defined signature. In our SLANG specifications, sentences always begin with the keyword "axiom." Like instances and sorts, sentences must be defined within a specification. In our framework, sentences cannot exist without being a part of a specification. This subsection discusses the processes by which DTAS/LOLA itself, domain experts, and end users create sentences to evolve the domain model.

4.1.2.3.1 Human Imagination

Human imagination is the sentence constructor which is the easiest to implement. Every time a DTAS end user types in a cspec in a text editor and compiles the definition, we represent this by a "sentence" being automatically constructed representing the cspec. For example, if the end user defines the following cspec (here written in DTAS's cspec surface syntax)

```
<ALU_32_4_4_8,<ALU,
[<I0,32>,<I1,32>,<ICIN,1>,<ISEL,4>],
[<O0,32>,<OCOUT,1>,<OREL,1>],
[<OP,ADD>,<OP,SUB>,<OP,INC>,<OP,DEC>,<OP,EQ>,<OP,LT>,<OP,GT>,<OP,ZEROP>,<OP,AND>,<OP,OR>,
<OP,NAND>,<OP,NOR>,<OP,XOR>,<OP,XNOR>,<OP,LNOT>,<OP,LIMPL>]>>
```

then, in an abstract sense, the specification given earlier is a theorem of spec CSPEC. See "spec ALU-32-4-4-8 is" on page 22. Likewise, component implementations (i.e., cell definitions) defined by the domain expert, when parsed, can be thought of as constructing a sentence such as the axiom assigning a value to ND2 given earlier. See "spec CELL-LIBRARY is" on page 23.

4.1.2.3.2 Deduction (e.g., find e such that DT |— e)

Deduction is the sentence constructor that given a collection of sentences (i.e, axioms) in a specification (call it DT) and a sentence e (a conjecture) written using the same signature as DT, determines whether e is deducible from the axioms of DT. The deduction constructor is not used for domain model evolution in DTAS/LOLA. DTAS uses *term rewriting* during design synthesis. The application of deduction to domain theory evolution (i.e., deductive synthesis of domain theories) is an area we hope to investigate in our Dissertation.

4.1.2.3.3 Induction (e.g., find H such that $B \cup H \vdash e$)

The goal of the induction constructor is to find a hypothesis (a collection of axioms) H such that

$$\forall \text{ pos-ex} \in E+ \bullet B \cup H \vdash \text{pos-ex (i.e., H is "complete") and}$$

$$\forall \text{ neg-ex} \in E- \bullet \neg(B \cup H \vdash \text{neg-ex) (i.e., H is "consistent").}$$

Where E+ is a collection of positive example (i.e., ground, having no variables) sentences, E- is a collection of negative example sentences, and B is a collection of background knowledge axioms (i.e., the existing domain theory).

The induction constructor is not supported by DTAS/LOLA. The application of the induction constructor to domain theory evolution for synthesis (rather than analysis) systems is an area we hope to investigate in our Dissertation.

4.1.2.4 Specifications

The framework for domain theory structure and evolution addresses the construction of ever-larger-grained parts of domain theories. The next larger domain theory part addressed are constructors that produce whole specifications from other whole specifications. Given our representation of domain theory evolution in DTAS, LOLA appears to support several specification constructors. This section discusses the processes LOLA uses to implement these specification constructors.

4.1.2.4.1 $\bigcup_{i \in I} SP_i$

The distributed union specification constructor takes an indexed collection of specifications and creates a new specification that is the disjoint union of all their signatures and axioms. Name conflicts are automatically disambiguated and therefore no sharing of symbols can occur. DTAS/LOLA does not support this constructor. The colimit constructor, described below, is used in our representation of DTAS/LOLA domain theory evolution.

4.1.2.4.2 translate SP by σ

The translate constructor takes a specification morphism σ and a source specification SP and produces a new specification which is a copy of SP, but with possibly renamed symbols via σ. We showed a typical DTAS/LOLA domain theory as a collection of specifications related by inclusion morphisms and definitional extension morphisms. The translate constructor is used to provide these inclusion morphisms and definitional extension morphisms.

Of course a typical DTAS performance system does not represent these morphisms explicitly, but rather implements them in several ways. We presume that **spec** CSPEC, **spec** Netlist, and the inclusion morphism from the former to the latter are implemented in the DTAS language compiler. **spec** Cell-Library is presumably represented as a text file and the definitional extension morphism is represented by the process of successfully compiling the cell library. **spec** Decomposition-Methods is presumably represented by a collection of rewrite rules that operate on compiled design problem specifications. The inclusion morphism from **spec** CSPEC to **spec** Decomposition-Methods is presumably represented by an object base manager that provides the compiled design problem specifications (in an internal form such as syntax trees) to the rewrite rules for manipulation. The inclusion morphisms from the old to the new versions of **spec** Cell-Library and **spec** Construction-Methods is represented only by the progression of time between additions to the **spec** Cell-Library and execution of LOLA's "technology compilation" algorithm.

4.1.2.4.3 derive from SP′ by σ

The derive constructor takes a target specification **SP′** and a specification morphism σ : SP $\rightarrow$ SP' and produces a source specification SP. SP is produced by following σ backwards. SP contains translations of only those symbols which map to **SP′** symbols in the image of σ. The semantics (i.e., "models" or "interpretations") of SP are defined by the *reduct* over σ. The reduct is a function that translates **SP′** models into SP models by forgetting the carrier sets and functions of **SP′** that are not in the image of σ. DTAS performance systems support the derive constructor by allowing the end user to create instances of CSPEC (problem specifications) and Netlist (solution implementation).

4.1.2.4.4 iso close SP

The isomorphic closure constructor takes a specification SP and produces a new specification **SP′** with the same signature as SP but whose models are isomorphic [27]. If a standard interpretation paradigm is chosen (e.g., "initial" or "final" semantics [27]), then **SP′** can have different axioms from SP. DTAS/LOLA does not support the isomorphic closure constructor.

4.1.2.4.5 minimal SP wrt σ

The minimal wrt constructor takes a specification SP (with signature Σ) and a *signature* morphisms $\sigma : \Gamma \rightarrow \Sigma$ and produces a specification with signature Σ but the models of which are minimal extensions of their σ-reducts. If a standard interpretation paradigm is chosen, then **SP′** can have different axioms from SP. DTAS/LOLA does not support the minimal wrt constructor.

4.1.2.4.6 abstract SP wrt Φ(X)

The abstract wrt constructor takes a specification SP and a collection of sentences with free variables in X and produces a specification **SP′** with the same signature as SP but the models of which are observationally equivalent [27] with respect to the sentences Φ(**X**). If a standard interpretation paradigm is chosen, then **SP′** can have different axioms from SP. DTAS/LOLA does not support the abstract wrt constructor.

4.1.2.4.7 λ X : $\Sigma_{par} \bullet SP_{res}$

The parameterize-by constructor takes a body specification $\mathbf{SP_{res}}$ (e.g., Queue) and an actual parameter specification SP_{par} (e.g., Nat) and instantiates a new specification (e.g., Queue-of-Nat). DTAS/LOLA does not support the parameterize-by constructor.

4.1.2.4.8 colimit of diagram D

The colimit constructor takes a diagram D of specifications and specification morphisms and produces a specification that is the minimal disjoint union of all the specifications in D while taking into account the sharing of symbols defined by the specification morphisms in D.

We represented LOLA's generation of the new collection of construction methods as the colimit of **spec** Netlist, **spec** Cell-Library, **spec** Construction-Method-Templates, and the morphisms between them. LOLA implements this colimit construction in its "technology compilation algorithm." The technology compilation algorithm scans the current cell library searching for cells that match the left hand side of "acquisition templates" and generates new construction methods. Acquisition templates are 2-tiered rewrite rules that when finding a matching pattern in the cell library, instantiate a new rewrite rule (i.e., a construction method) specific to what was found in the cell library. Acquisition templates take the form of "if the cell library has a functional unit X but no functional unit Y, then instantiate a construction method that implements Ys as Xs." For example, if the cell library has a NAND functional unit, but no INV functional unit, then a construction method will be instantiated that implements INVs as NANDs.

4.1.3 Conclusions on LOLA

LOLA enables a typical DTAS performance system to evolve in only 1 direction: the addition (and deletion) of cells to the cell library. This narrow focus enables the DTAS performance system's domain theory to be automatically regenerated. DTAS/LOLA uses component representations that are mostly syntactic. This decision results in languages for problem specification and cell library component representations that are difficult to evolve. These unchangeable languages prevent LOLA's adoption of more interesting knowledge acquisition and domain theory evolution processes. LOLA conflates the domain expert's knowledge acquisition language meta-ΣA and the end user's solution implementation language ΣC. These reference architecture functional elements should be disjoint.

For the reviews which follow, resource limitations prevent detailed descriptions of how each knowledge acquisition tool supports each atomic domain theory constructor. Please refer to Section 5 on page 57 for summaries of which constructors each tool supports.

4.2 Ozym

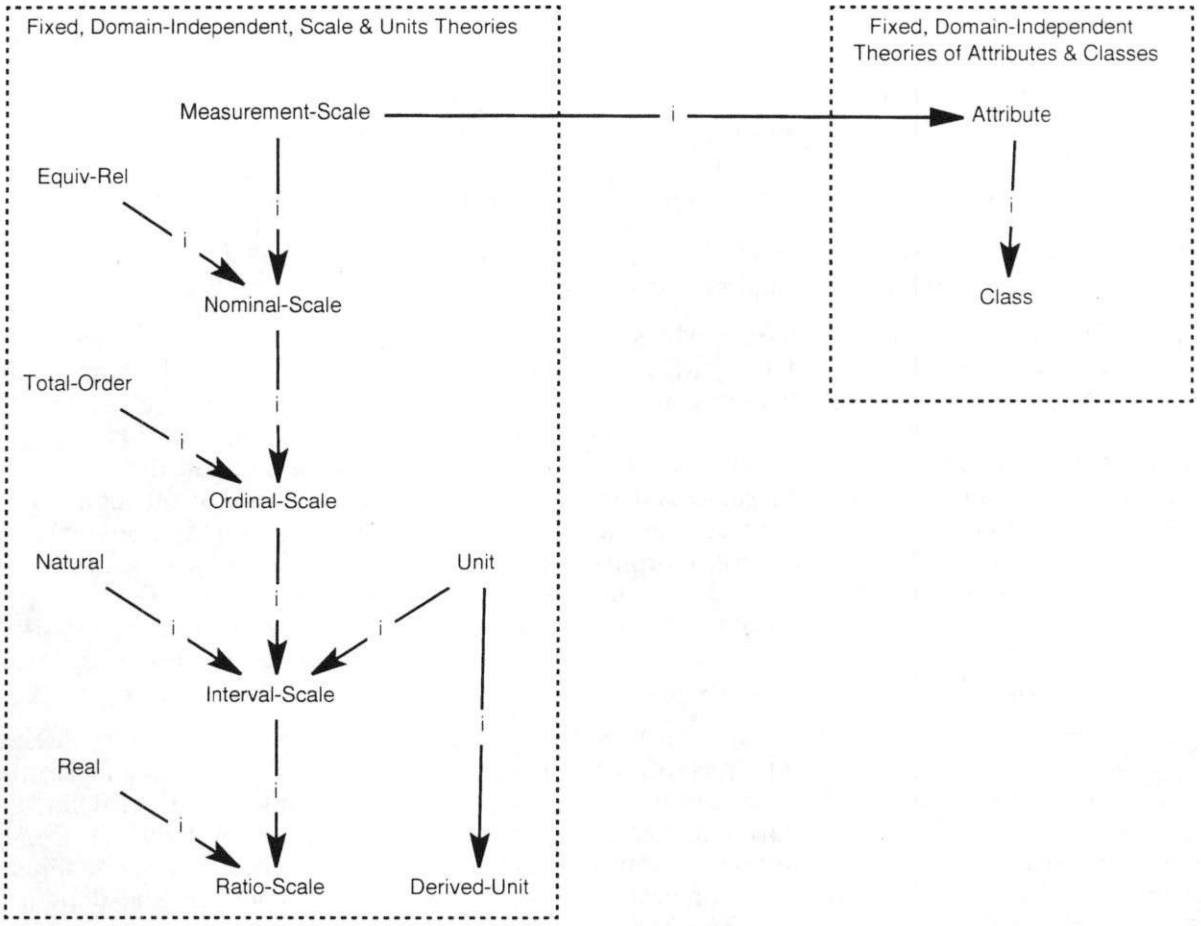
Ozym [13] is a research prototype knowledge acquisition tool which synthesizes domain-specific application generators (i.e., the performance system is an application generator). Application generators [4] are used mostly in the relational database development community to automate the construction of user interfaces, reports, and table definitions from high-level specifications. Application generators use simple translation techniques such as context-free macro expansion to generate implementation code from specifications. Ozym seeks to improve the quality of generated code by adding domain knowledge to the translation process. As a domain model evolves, the application generator must be regenerated to take the domain model's new knowledge into account. The new application generator generates code from specifications in the application domain.

Ozym domain models are classes in an object-oriented programming language. Ozym enables a domain expert to more easily extend a domain model by providing him with a high-level, structural specification language for defining new class compositions. The domain expert need not know all details of the composed classes in order to compose them together. In this way Ozym enables domain models to be evolved by composing elementary building-block classes into new classes. The new classes can serve as building blocks for new classes, and so on.

Using the framework for domain theory structure and evolution, we represent Ozym's architecture. This representation uses structured domain theories to represent Ozym's domain models. Many representational decisions have been made in the recasting process, but we believe we have captured the essence of Ozym's mechanisms for domain model evolution.

Every Ozym domain model features the structures shown in the diagram of specifications and specification morphisms below. Individual Ozym environments add domain-independent and domain-dependent domain theories to this fundamental structure. The structures shown below provide generic, non-evolvable, semantic definitions upon which Ozym environments build.

Figure 4.4 Ozym's Domain-Independent Architecture



Domain model construction in Ozym in based upon the following concepts.

- Definition of *attributes* based on well-defined *scales of measurement*
- Aggregation of attributes into *classes*
- *Composition* of classes into new classes

The diagram on the left represents generic theories of scales of measurement and of units of measurement. The diagram on the right represents generic theories of attributes and classes.

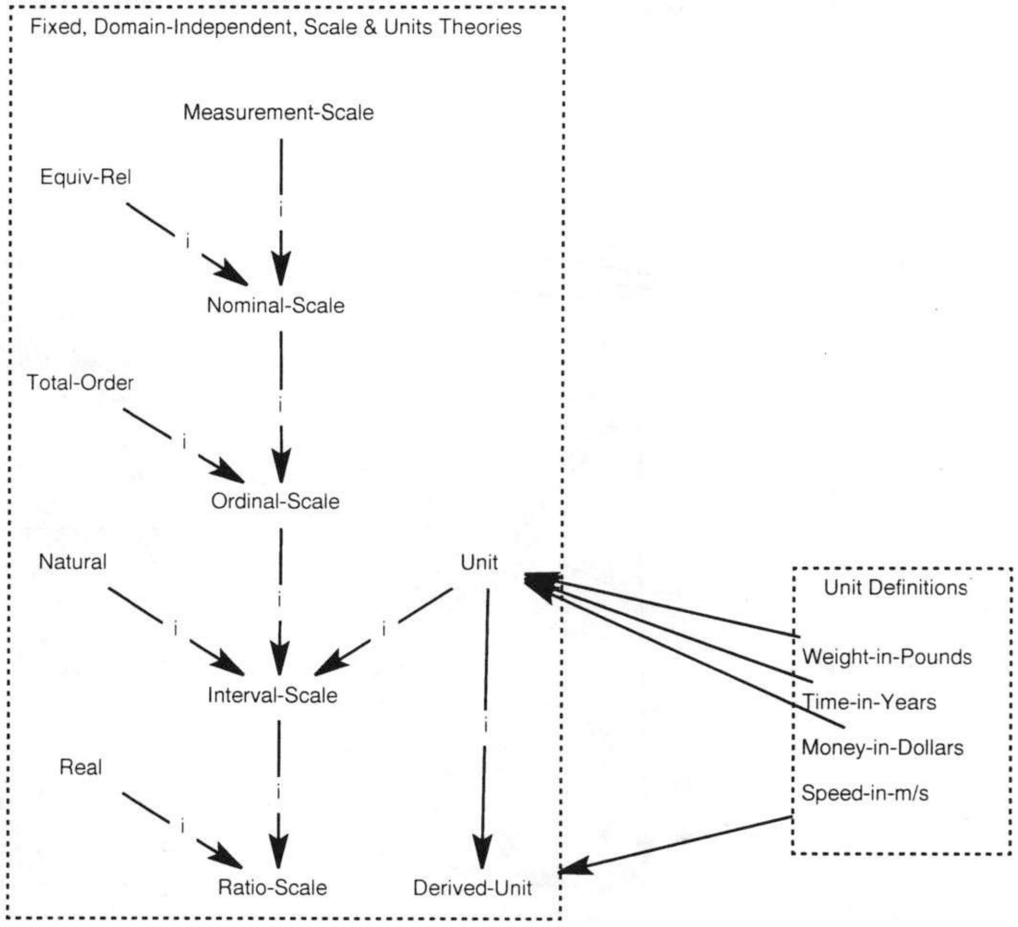
The subject of scales and units of measurement, which is familiar to statisticians, has been largely ignored in the creation of type systems for programming languages. Ozym is a notable exception. Scales of measurement are important to define valid transformations on data. For example, if we constructed a computer program that determines the average qualities of football players, an invalid transformation would be one that takes the jersey number of each player, calculates the average number, and assigns that number to a hypothetical "average" player. Without proper attention to scales of measurement, such illegal transformations produce subtle bugs which ultimately undermine the validity of other calculations.

The specification Measurement-Scale, show at the top of the diagram is syntactic and defines the language of values and scales shared by all measurement scales. The most basic measurement scale is the Nominal-Scale, which groups measurements into equivalence classes. Each equivalence class is a different "value." Examples of Nominal scales include gender (with values Male and Female), football player jersey numbers, etc. The Ordinal-Scale includes all the properties of Nominal-Scale (hence the inclusion morphism from Nominal-Scale to Ordinal-Scale) and adds to them the concept of a total order on values. Together the Nominal-Scale and Ordinal-Scale are known as the "qualitative" scales. The Interval-Scale extends the properties of an ordinal scale with the concept of *magnitude* represented by a numeric value of a unit of measurement. A unit of measurement provides a symbol, the unit name (such as meters, feet, seconds, etc.) and a granularity representing the smallest possible quantity of the unit considered in the application domain. To be meaningful, a unit must be associated either with the Interval-Scale or Ratio-Scale. The Ratio-Scale extends the concepts of an Interval-Scale by adding the concept of a non-arbitrary absolute zero, thereby enabling multiplication and division of measurements. Derived-Unit defines the concept of units such meters per second, kilogram-meters per second per second, etc.

The inclusion morphism from Measurement-Scale to Attribute indicates that every Attribute defined must be related to a Measurement-Scale. The specification Attribute defines the properties common to all attributes, namely that an attribute has exactly one measurement scale associated with it; possesses a set of *population parameters* defining the likelihood that the attribute will have a certain value; a set of admissible transformations on the value of the attribute, given the measurement scale and unit associated with it; and that attributes can be built as extensions to other attributes. The fact that attributes are aggregated together to form classes is represented by the inclusion morphism from Attribute to Class. Class defines properties common to all classes, namely the distinction of attributes as to whether they are primitive or derived; the distinction of attributes as to whether they are key or nonkey; the properties of functions (i.e., methods) that modify attribute values; and the properties of procedures to instantiate and delete class instances (i.e., objects).

The diagram of specifications and specification morphisms below represents the existence of 4 domain-expert-defined units of measurement. A specification morphism from the specification of each new unit of measurement back to the theories Unit or Derived-Unit represents the fact that each new unit is a theorem of the generic theories of Units or Derived-Units. We use undecorated morphisms such as these to denote instantiation relationships between specifications. The collection of new units of measurement created by the domain expert, Weight-in-Pounds, Time-in-Years, Money-in-Dollars, and Speed-in-m/s, can be reused in the construction of new Ozym application generators. Note that the non-derived units of measurement defined can later be associated with either the interval scale or ratio scale (or possibly both), depending upon the evolutionary path taken by the domain theory. Ozym thus decouples units from measurement scales in a well-defined manner and enables domain experts to take advantage of possible combinations of units and scales, as needs arise in the evolution of the domain model. Thus Ozym domain experts are provided with an elegant type definition facility.

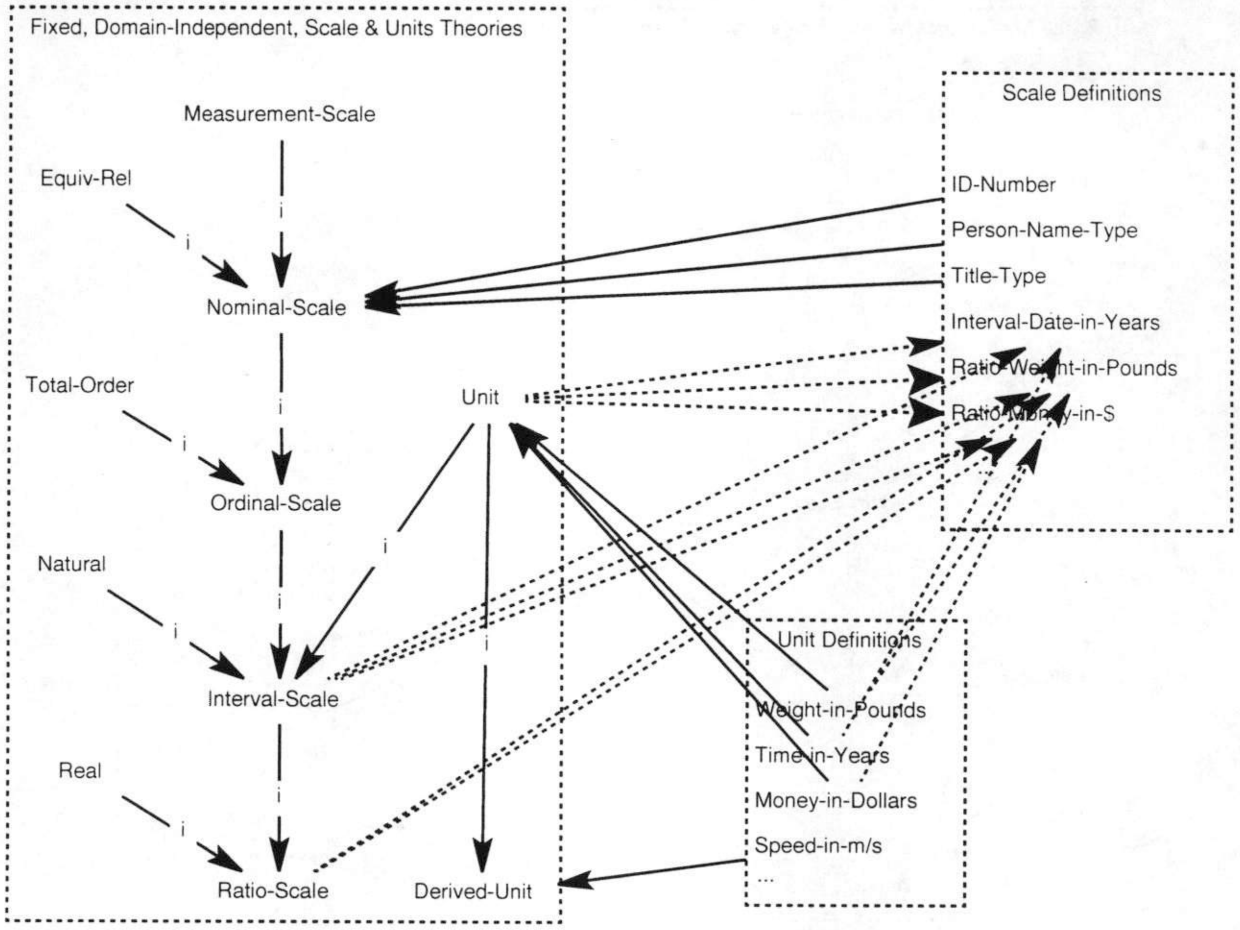
Figure 4.5 Ozym Domain Expert Instantiates New Unit Definitions



The diagram below represents the existence of domain-expert-defined theories defining new scales of measurement. The definition of nominal and ordinal scales of measurement, such as ID-Number, Person-Name-Type, and Title-Type is less complex than the definition of interval and ratio scales. This is because nominal and ordinal scales require no units of measurement. Interval-Date-in-Years, Ratio-Weight-in-Pounds, and Ratio-Cost-in-$ use the unit definitions represented in the previous diagram.

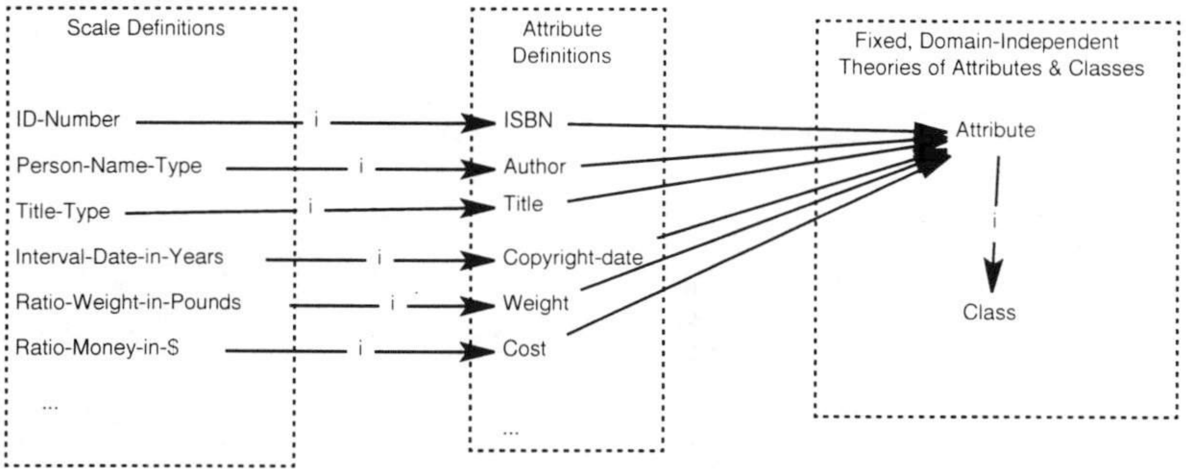
We represent the construction of interval and ratio scales of measurement via colimit diagrams. For example, Ratio-Cost-in-$ is defined as the colimit of the diagram formed by Money-in-Dollars, Unit, Interval-Scale, Ratio-Scale, and the morphisms shown between them. Thus Ratio-Cost-in-$ is the minimal theory containing all the properties of Money-in-Dollars, Unit, Interval-Scale, and Ratio-Scale, while taking into account the sharing of symbols defined by the morphisms. Ozym hides the complexity of the unit and scale combinations by providing GUI dialog boxes that offer lists of already-defined units and scales to assist in the definition of new units.

Figure 4.6 Ozym Domain Expert Defines New Scales



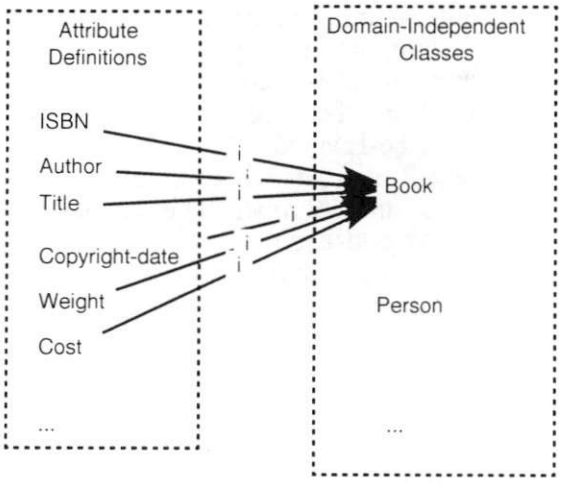
The diagram below represents the existence of a number of attributes, created by a domain expert. Each attribute is based upon exactly one scale of measurement, but it is possible for a scale of measurement to be used by more than one attribute. This association is partly represented by the inclusion morphisms from the domain-expert-created measurement scales discussed earlier to theories defining specific, new attributes. The attribute definitions add information about population parameters and admissible transformations associated with each attribute. As indicated in previous diagrams, the instantiation relation is shown as well.

Figure 4.7 Ozym Domain Expert Creates New Attributes



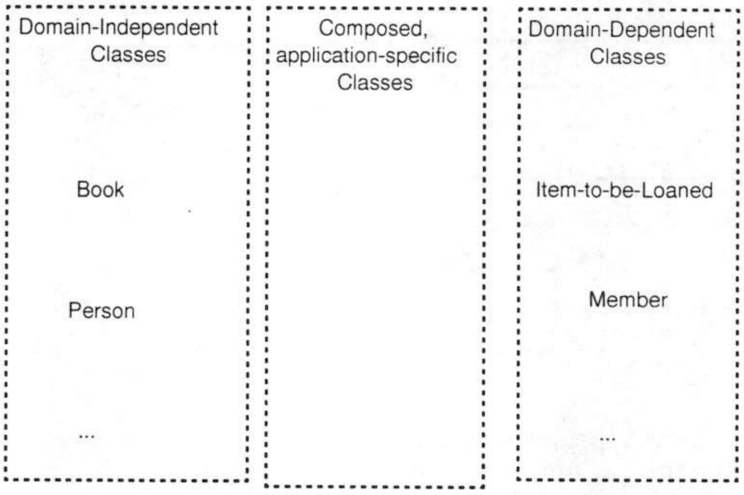
The diagram below represents how a collection of domain-expert-defined attributes are aggregated together in the definition of a new class (class Book in this case). In addition to containing all the attributes shown, the specification Book also defines operations to modify attribute values, such as the operation Age, which returns a value derived from the attribute Copyright-Date. Any operations defined for a particular class must be consistent with the general requirements for such operations defined in the specification Class.

Figure 4.8 Ozym Domain Expert Defines a New Class



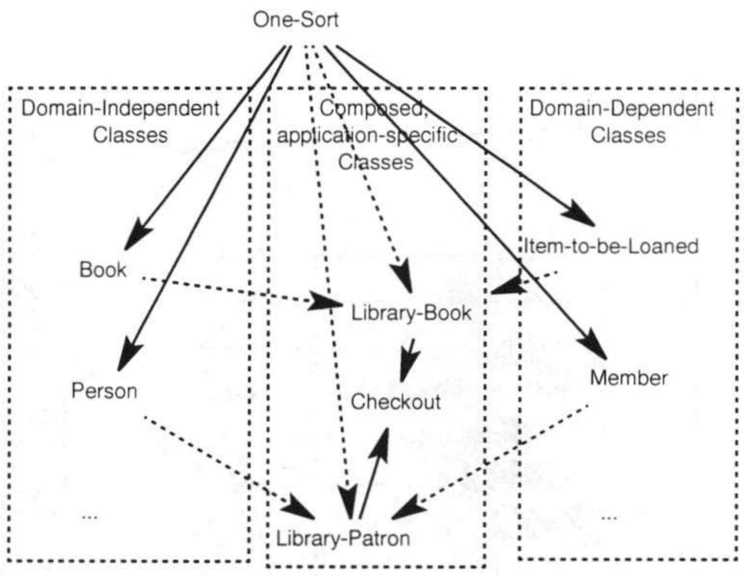
Classes in an Ozym performance system are distinguished as to whether they are domain-specific or domain-independent. The diagram below represents a collection of domain-specific and domain-independent classes. Domain-independent classes can be shared between different Ozym application domains. Examples of domain-independent classes include Book and Person, shown in the diagram in the left group of specifications. Domain-dependent classes are constructed by domain experts specifically for use in his application domain. Examples of domain-dependent classes include Item-to-be-Loaned and Member, shown in the diagram in the right group of specifications. We imagine that the application domain is relational databases for various lending & renting organizations e.g., libraries, sports equipment centers, car rental services, etc.).

Figure 4.9 Ozym's Segregation of Classes


Domain-Independent Classes
Book
Person
...
Composed, application-specific Classes
Domain-Dependent Classes
Item-to-be-Loaned
Member
...

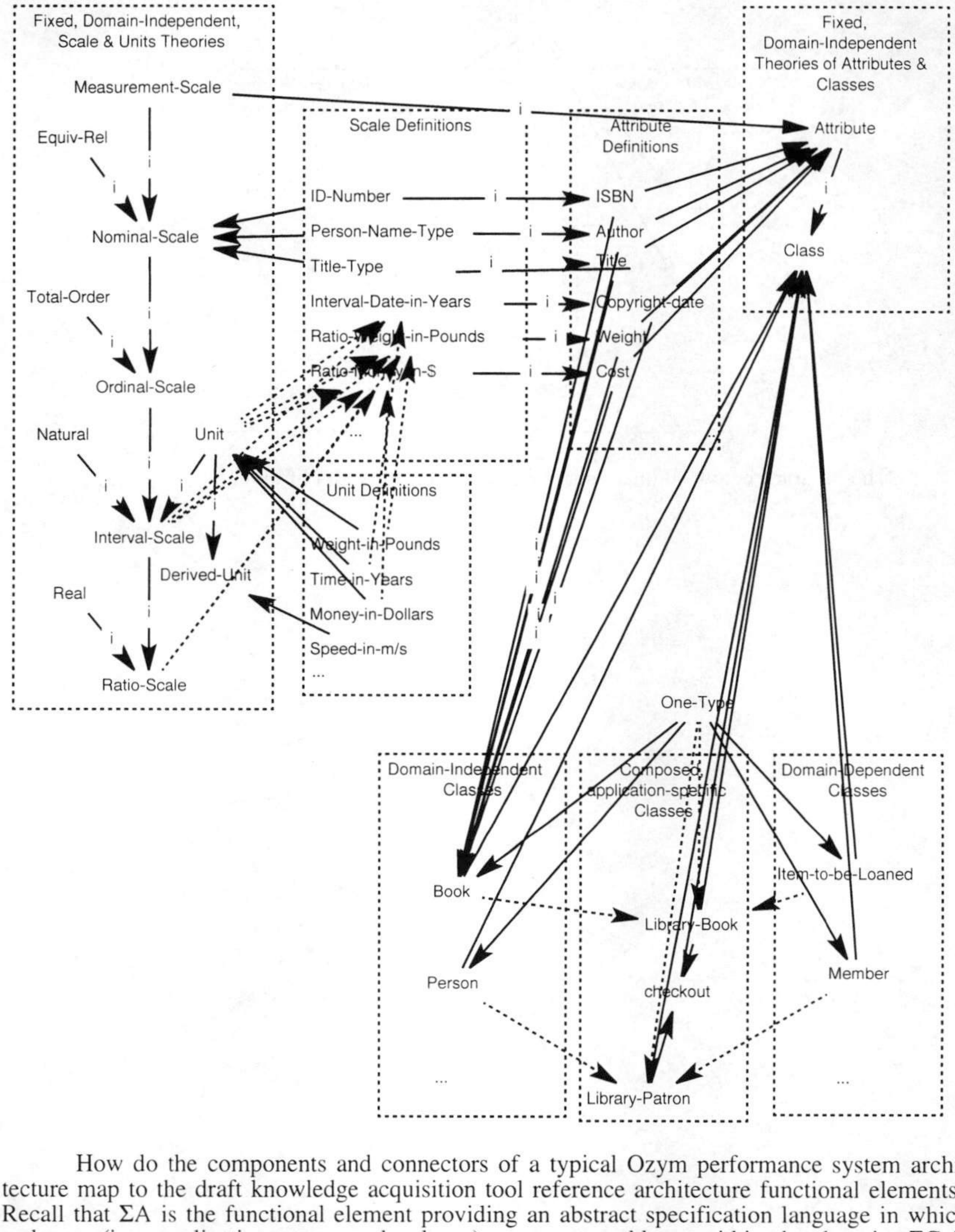
The diagram below represents the situation in which the *end user* defines two new, domain specific classes by combining the already-existing classes just mentioned. This is where the end user (i.e., application program developer) begins to use the classes already constructed by the domain expert. The end user wishes the attributes and properties of Book and Item-to-be-Loaned to be combined to form a new, domain-specific class, Library-Book. This combination operation is analogous to multiple inheritance in some object-oriented programming languages. We represent Ozym's class combination operation using a colimit, as shown in the diagram below. Specification Library-Book is shown below as the colimit of specifications Book, Item-to-be-Loaned, and One-Sort. The specification One-Sort is used only to indicate the shared parts of Book and Item-to-be-Loaned. One-Sort is not a class, but only an auxiliary specification used to compute the colimit. Likewise, One-Sort is used again to compute Library-Patron as the colimit of Person and Member. The specification Checkout is an association between the classes Library-Book and Library-Patron and is represented by an instance of the Class specification that imports both Library-Book and Library-Patron.

Figure 4.10 Ozym End User Composes Classes Together



The diagram below summarizes our discussion (thus far) of Ozym's architecture.

Figure 4.11 Ozym's Typical Domain-Specific Architecture



How do the components and connectors of a typical Ozym performance system architecture map to the draft knowledge acquisition tool reference architecture functional elements? Recall that ΣA is the functional element providing an abstract specification language in which end user (i.e., application program developer) represent problems within the domain, ΣC is functional element providing the concrete implementation language in which solutions are represented, and ABS is the functional element providing a collection of abstraction functions that relate ΣC objects to ΣA objects. Problem specifications constructed by Ozym end users take the form of terms in a language of class compositions. We imagine that the structure of this language can be represented by the specification below.

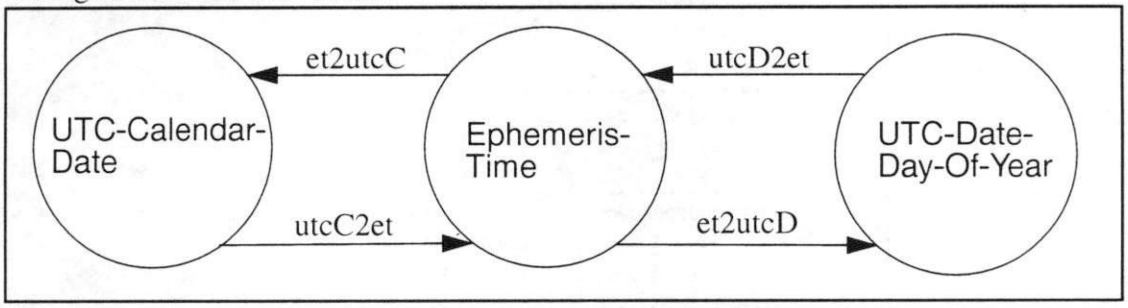
```
spec SIGMA-A is
sorts class
op ⊕ : class , class -> class
const Book : class
const Person : class
...
Item-to-be-Loaned : class
end-spec
```

A "string" in this specification language could be represented by the following specification.

```
spec Library-Book-Spec is
import SIGMA-A
axiom
(equal
    (⊕ Book Item-to-be-Loaned)
    Library-Book)
end-spec
```

This specification language is structural in nature and analogous to our diagrams of specifications and specification morphisms. It is not obvious how to represent a language like this in our framework, because it deals directly with program artifacts rather than domain-oriented concepts. In the specification language, a class is purley syntactic and is an instance of a sort (sort "class"), but we have represented classes in the domain theory as specifications. The framework does not (yet) have a formal mechanism to directly relate symbols (instances of a sort) to theories.

We encountered this kind of problem and documented it in [26]. There, we devised a language for adding knowledge to a domain theory about different kinds of representation conversions. We accomplished this by drawing a diagram such as the one we used to introduce **spec** TIME, which we provide again below. In [26], a meta-domain theory about representation conversions is used to extend a domain theory with specific new knowledge obtained from a diagram like the one below.



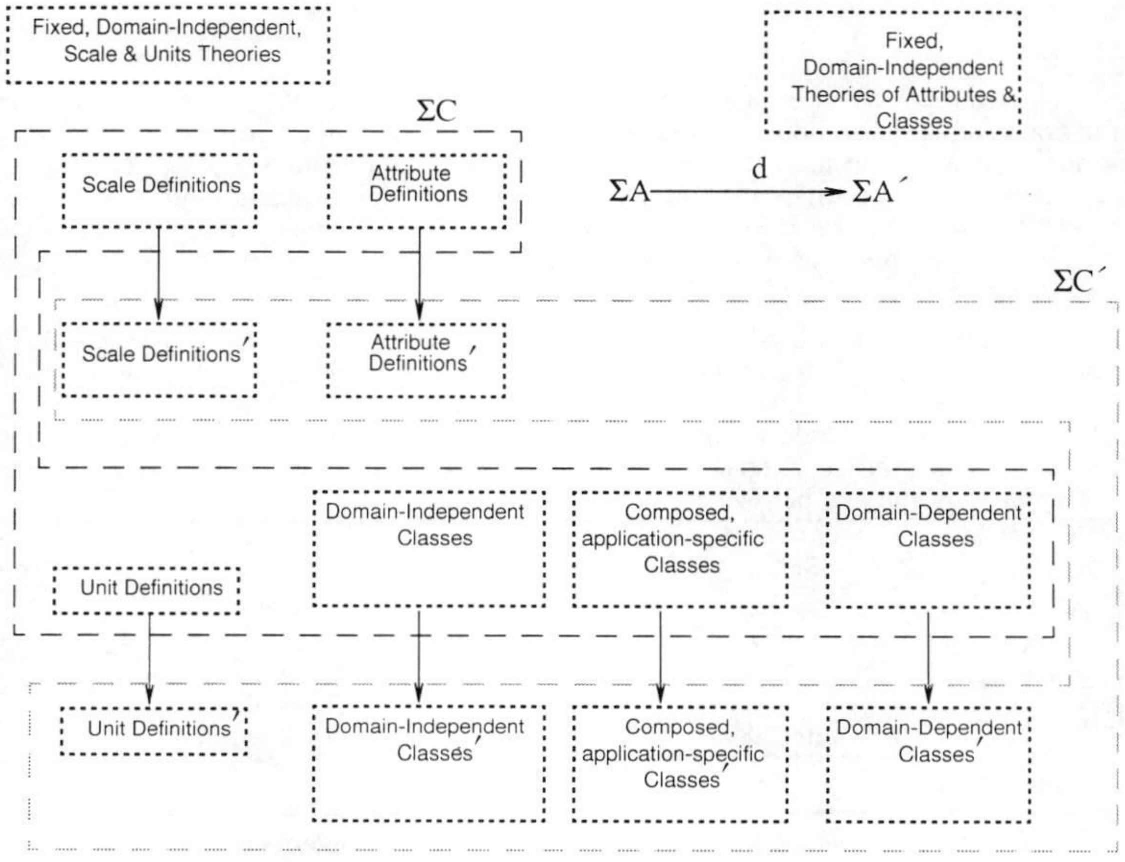
The Ozym specification language presents a similar problem, but at the level of writing problem specifications, rather than at the level of domain theory evolution. We envision a similar solution as used in [26], but cannot work out all the details of such a solution because of resource limitations. At this time, we do not know how to show the relationship between Ozym's specification language and other parts of its domain theory. Thus ΣA is not shown in the preceding Ozym domain theory diagrams.

Notice that the Ozym specification language evolves in a very restricted manner: Only terms of sort "class" can be added. Once the new terms are added, they can be immediately used to define new instances of sort "class." This kind of evolution can be succinctly represented by a definitional extension. Recall that a definitional extension is like extending the definition of a programming language by creating a new function and making the function a primitive of the language. The semantics (i.e., models) of the original and extended language are exactly the same. A definitional extension is a restricted inclusion morphism and is represented in a diagram by a "d" decoration next to the morphism's arrow, as shown below.

$$\Sigma A \xrightarrow{d} \Sigma A'$$

The quasi-architecture diagram below summarize evolution of Ozym domain-specific architectures. The specification language ΣA can only undergo definitional extension, as brought about by the end user creating new problem specifications (i.e., class compositions). Theories defining measurement scales, units, attributes in general and classes in general cannot evolve. The domain expert evolves theories defining specific units, measurement scales, attributes, domain-independent, and domain-dependent classes. We believe these evolutionary steps will usually be inclusion morphisms. Lastly, the end user evolves the collection of composed, application-specific classes as a result of writing a specification in ΣA.

Figure 4.12 Ozym Domain-Specific Architecture Evolves


Fixed, Domain-Independent, Scale & Units Theories
Fixed, Domain-Independent Theories of Attributes & Classes
ΣC
Scale Definitions
Attribute Definitions
ΣA
d
ΣA´
ΣC´
Scale Definitions´
Attribute Definitions´
Domain-Independent Classes
Composed, application-specific Classes
Domain-Dependent Classes
Unit Definitions
Unit Definitions´
Domain-Independent Classes´
Composed, application-specific Classes´
Domain-Dependent Classes´

4.3 SALT

SALT [20] generates domain-specific expert systems that solve a class of simple design problem using a method known as "propose and revise." SALT operates in design domains where designs can be represented by a list of *design parameters*. A design parameter is an global, imperative variable (i.e., it can remember its last value) of numeric or enumerated type that is calculated from conditional, equational formulas (i.e., axioms) of other design parameters or looked up in tables. The use of design parameters is common in many established engineering disciplines (e.g., mechanical, civil, and aeronautical engineering) where the principal means of organizing domain knowledge is via sets of equations that simultaneously hold.

Some design parameters are given as input or stated as constants by the engineer (i.e., the end user of a SALT-generated performance system). Input and constant design parameters are used to calculate the initial values of other design parameters. Thus a collection of design parameters forms a dependency graph. A dependency graph can be represented by a collection of axioms. Often cycles exist in the dependency graph. Several iterations around the cycle are needed for the collection of design parameters to converge on a design solution. When no conflicts exist among the design parameters, a design solution has been found. This kind of design makes no use of predefined units that encapsulate elementary functionality (e.g., reusable software or hardware components). Propose and revise design problems focus more on design parameter *optimization*.

While complex dependency graphs can be drawn for propose and revise domain theories, these graphs only show functional dependencies between design parameters. Specification structuring is poor. Thus propose and revise domain theories tend to be monolithic, every design parameter can conceivably view every other design parameter (i.e., all design parameters are global).

In this section we present our interpretation of SALT's architecture. The diagram of specifications and specification morphisms shown below is our interpretation of the part of the elevator design performance system's (i.e., the OPS5 expert system generated by SALT) domain model. The domain theory can be modularized into at most 2 parts, a specification language and an implementation language. Cycles in dependency graphs prevent the implementation language from being broken up into smaller specifications.

$$\begin{tikzcd} \text{Input-\&-Constant-Value-Procs} \arrow[r, hook, "i"] & \text{Other-Procs-Constraints-Fixes} \\ \Sigma A & \Sigma C \end{tikzcd}$$

The specification on the left, Input-&-Constant-Value-Procs ("Procs" is an abbreviation for "procedures."), contains declarations for design parameters which are given as input or constant by the end user. A collection of these design parameters with assigned values represents a problem specification. Hence Input-&-Constant-Value-Procs defines the domain theory's specification language, ΣA. In SALT domain models, every design parameter has a *procedure* to provide it a value. Procedures can be formulas (conditional, equationally-defined functions), table lookups, or provide constant, user-defined values.

The specification on the right, Other-Procs-Constraints-Fixes, contains declarations for all procedures that are not constants or inputs, procedures that compute constraints for other design parameters, and special procedures called "fixes" that change a design parameter's value when the parameters fails to satisfy a constraint. Fixes temporarily override the normal procedure associated with the design parameter and are what cause design calculations to iterate. Because the procedures in Other-Procs-Constraints-Fixes have visibility over all the input and constant design parameters, Input-&-Constant-Value-Procs is imported by Other-Procs-Constraints-Fixes, hence the inclusion morphism from the former to the latter. Because a design consists of the final values assigned to the design parameters in Other-Procs-Constraints-Fixes, we say that Other-Procs-Constraints-Fixes provides the implementation language, ΣC.

Recall that SALT generates performance systems that apply the propose-and-revise problem-solving method. In our interpretation of SALT's architecture, we show how propose-and-revise provides a domain-independent language in which to represent domain theories. Domain experts gain access to the language of propose-and-revise via a knowledge acquisition language for defining design parameters and procedures to calculate, constrain and fix design parameters. The relationship between SALT's knowledge acquisition language and a generic theory of propose-and-revise is shown in the diagram below. This approach is what Gruber

[9] calls "task-level architectures."

$$\begin{tikzcd}\text{KA-Language} \arrow[r, "i"] & \text{Theory-of-Propose-\&-Revise}\end{tikzcd}$$

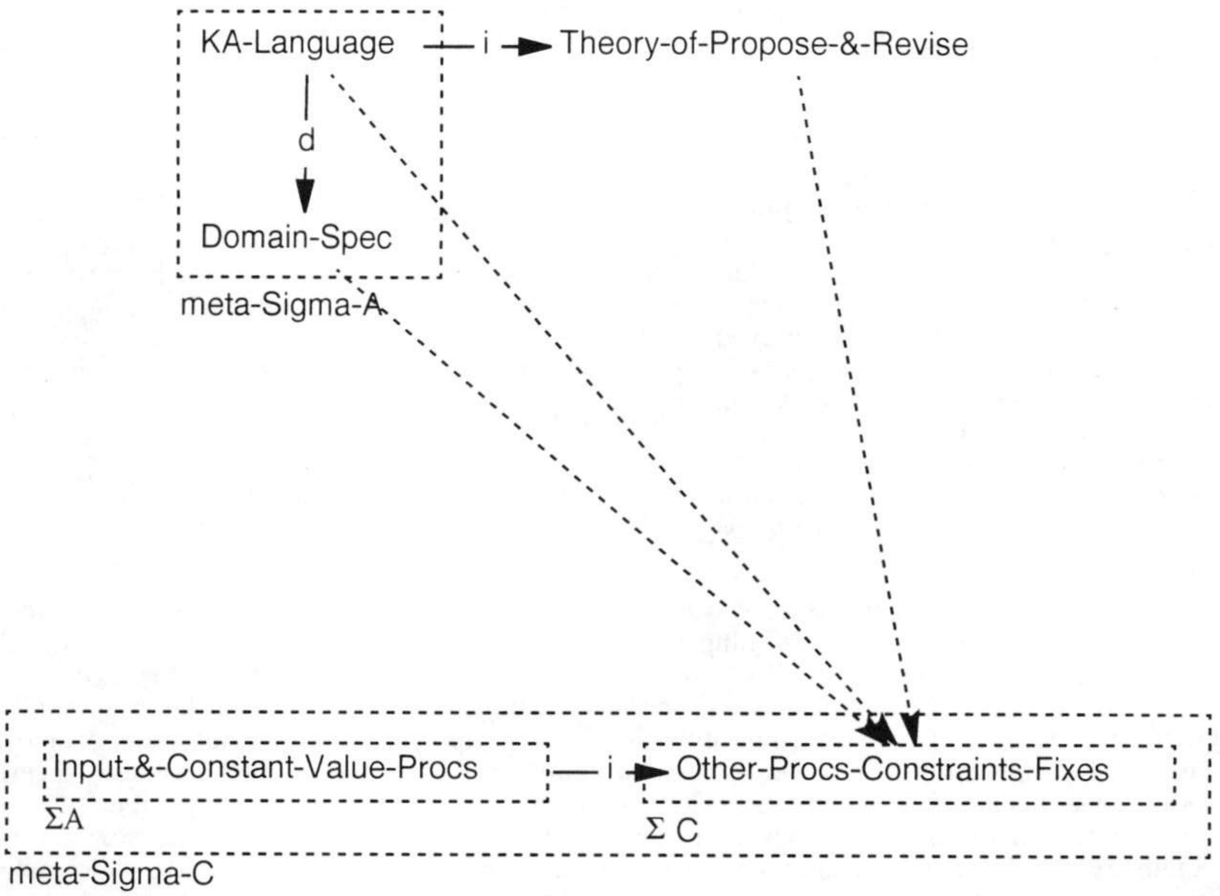
The specification on the left, KA-Language, defines the knowledge acquisition language. The vocabulary of this language consists of design parameters, procedures to compute design parameters, tables, constraints on design parameters, fixes to design parameters, terms, and sentences. The specification on the right, Theory-of-Propose-&-Revise, defines the propose-and-revise solution method. The specification Theory-of-Propose-&-Revise defines processes that apply to the vocabulary of KA-Language, hence the inclusion morphism from KA-Language to Theory-of-Propose-&-Revise. At the beginning of design synthesis, the "propose" process takes the input and constant design parameter values and propagates parameter design values as far as possible through the dependency graph. When a constraint on a design parameter is violated (e.g., the design parameter's value is too great), the "revise" process overrides the value of one of the design parameters on which the violated parameter depends. The "propose" process works again to propagate the new parameter value as until a constraint is violated or all constraints are satisfied and all parameters have been calculated.

A domain expert represents domain knowledge by definitionally extending KA-Language, as shown in the diagram adjacent. KA-Language provides a language and the specification Domain-Spec represents a "string" in that language. Thus Domain-Spec contains definitions of design parameters, constraints against design parameters, and fixes for design parameters when their constraints are violated. We the domain expert's process as a definitional extension because every design parameter he defines can be used as a primitive vocabulary element in a later knowledge acquisition interview.

$$\begin{tikzcd}\text{KA-Language} \arrow[d, "d"] \\ \text{Domain-Spec}\end{tikzcd}$$

Given KA-Language, Theory-of-Propose-&-Revise, and Domain-Spec, SALT generates a domain-specific performance system. This generation process is represented by the colimit diagram below. Recall that colimit generation constructs the minimal specification which is the union of all argument specifications while taking account of the sharing of symbols defined by specification morphisms. The environment generation process takes every design parameter defined by the domain expert and reformulates it into an equivalent, rule-based representation in the expert system program language used in SALT-generated design environments, OPS5. ΣA and ΣC represent the parts of the design environment represented in OPS5.

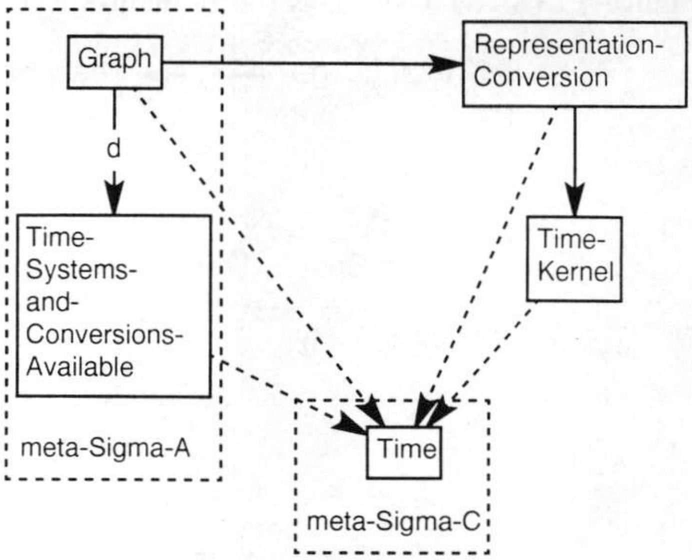
Figure 4.13 SALT Generates the Domain-Specific Performance System


KA-Language
i
Theory-of-Propose-&-Revise
d
Domain-Spec
meta-Sigma-A
Input-&-Constant-Value-Procs
i
Other-Procs-Constraints-Fixes
ΣA
Σ C
meta-Sigma-C

Our notation does not fully express the details of generating the design environment. Colimits alone are not enough to generate a representation of the design environment. Reformulation and instantiation are also required. Domain-Spec represents design parameters, procedures, constraints, and fixes by *instances* of sorts in the vocabulary of KA-Language. In order to provide real-world semantics and make the generated design environment perform inference, these instances (which are only symbols) need to be *reformulated into actual functions* and associated axioms. In other words, the representation of parameters, procedures, constraints, and fixes in ΣA and ΣC must closely map to their rule-based representation in the real-world OPS5 knowledge base (i.e., the domain theory of the design environment/performance system).

Recall that end users of a performance system represent problems using the domain-specific specification language ΣA. In an analogous manner, a domain expert uses a meta-design environment to construct specifications of domain knowledge. The domain expert uses a domain-specific specification language for representing domain knowledge (i.e., a domain-specific, knowledge acquisition language). In the diagram above this language is denoted by meta-Sigma-A. Also recall that end users of a design environment synthesize solutions to problems represented in ΣA. Solutions are represented in the language ΣC. In an analogous manner, a domain expert uses a knowledge acquisition tool to synthesize implementations of specifications of domain knowledge. A solution is the design environment/performance system. In the diagram above the language of design environments is denoted by meta-Sigma-C.

Figure 4.14 Approach to Knowledge Acquisition and Domain Theory Synthesis Employed in [26]

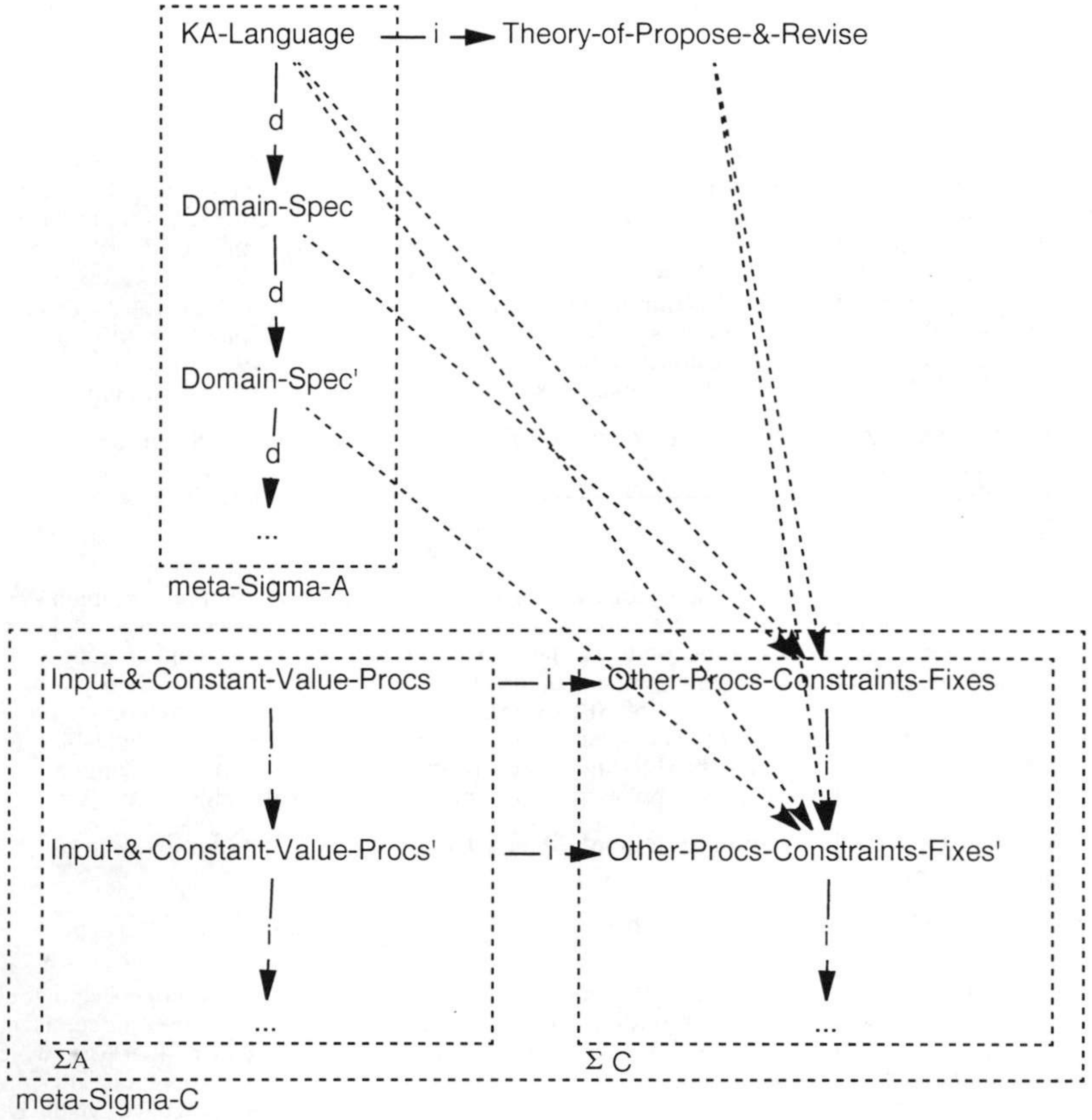

Graph
Representation-Conversion
d
Time-Systems-and-Conversions-Available
Time-Kernel
meta-Sigma-A
Time
meta-Sigma-C

SALT's approach to knowledge acquisition and design environment synthesis is conceptually similar to the approach we employed in [26]. The diagram in Figure 4.14, "Approach to Knowledge Acquisition and Domain Theory Synthesis Employed in [26]," on page 47 shows the organization of the tool used in that work. The problem we solved was to find a way to automatically generate significant portions of a domain theory automatically using an intuitive knowledge acquisition language.

The NAIF domain theory [19] was developed completely by hand. It contained a large number of very similar axioms defining representation conversions on time, position, and direction. Every time a new FORTRAN subroutine was added to the NAIF library (known as SPICELIB), a knowledge engineer would need to manually add 6 axioms to the domain theory defining the new conversion subroutine. This was error prone and seemed redundant. Because all the conversion subroutines possessed similar behavior, we defined a generic specification of that behavior, Representation-Conversion, and effectively instantiated copies of it for every conversion subroutine we wished to include in the domain theory. The idea was that the domain expert could draw a graph representing a family of similar conversion subroutines. The tool would take the graph and synthesize a fragment of the domain theory (i.e., the specification Time in figure 4.14) dealing with that family of conversion subroutines.

SALT's approach to domain theory evolution is represented in figure 4.15 on page 48. The idea is that every time the domain expert makes a definitional extension to KA-Language, i.e., defines new or revises the existing collection of design parameters, constraints, and fixes, the performance system is regenerated.

Figure 4.15 SALT's Approach to Domain Theory Evolution


KA-Language
i
Theory-of-Propose-&-Revise
d
Domain-Spec
d
Domain-Spec'
d
...
meta-Sigma-A
Input-&-Constant-Value-Procs
i
Other-Procs-Constraints-Fixes
i
Input-&-Constant-Value-Procs'
i
Other-Procs-Constraints-Fixes'
i
...
ΣA
Σ C
meta-Sigma-C

4.4 Protégé

PROTÉGÉ [23] is a knowledge acquisition tool with many similarities to SALT. The most important difference is that instead of using a theory of propose-and-revise to structure a domain theory (as is done in SALT), PROTÉGÉ uses a theory of "skeletal plan refinement." Skeletal plan refinement is a problem-solving (i.e., planning) method in which a rudimentary (or "skeletal") plan is instantiated and incrementally elaborated until all placeholders in the plan have been filled it by actual events. Skeletal plan refinement does not use backtracking: once a new skeletal plan has been instantiated, it cannot be replaced by a different skeletal plan. Once an event has been attached to a skeletal plan, the event cannot be replaced by a different event. PROTÉGÉ was applied to a domain in which this kind of planning makes sense, hypertension treatment planning. Hypertension treatments are executed in 1 or 2 week periods between patient visits. Once executed, treatments obviously cannot be "undone," hence there is no need for backtracking. Because the effects of treatments vary from patient to patient, it makes no sense to make long-term, detailed treatment plans.

This section describes our interpretation of a typical PROTÉGÉ architecture and how it supports domain theory structure and evolution. We begin by discussing the structure of a domain theory used by a typical PROTÉGÉ-generated performance system. Figure 4.16, "Protégé-generated Performance System Domain Theory Structure," on page 49 is shown below. The specification Input-Data-Lang on the left represents the end user's problem specification language ΣA. The specification Recommended-Plan-Lang on the right represents the end user's solution representation language ΣC. In the example performance system used in [9], the vocabulary of Input-Data-Lang deals with the patient's blood pressure, pulse rate, respiration, and weight. The vocabulary of Recommended-Plan-Lang deals with tablets, tests, and waiting periods. A patient visits the doctor's office, has his blood pressure, pulse, respiration, and weight measured, and a hypertension treatment for the next 1 or 2 weeks, consisting of a sequence of tablets, tests, and waiting periods, is synthesized by the performance system.

Figure 4.16 PROTÉGÉ**-generated Performance System Domain Theory Structure**

$$\begin{tikzcd}
\boxed{\text{Input-Data-Lang}} \arrow[r, "i"] & \boxed{\text{Recommended-Plan-Lang}} \\
\Sigma A & \Sigma C
\end{tikzcd}$$

PROTÉGÉ uses a general theory of skeletal plan refinement to structure a domain theory. The diagram below is our interpretation of how PROTÉGÉ breaks up the theory of skeletal plan refinement to make a portion of it available as a knowledge acquisition language. The specification on the left, SPR-Sigma, represents the signature or language of the theory of skeletal plan refinement. The vocabulary of SPR-Sigma deals with hierarchical planning entities, task-level actions, and input data. The specification on the right, Theory-of-Skeletal-Plan-Refinement, imports the language of SPR-Sigma and provides axioms giving that language semantics. Like SALT, PROTÉGÉ also applies the concept of a "task-level architecture" [9].

Figure 4.17 PROTÉGÉ**'s Domain-Independent KA Language and Problem-Solving Method Theory**

$$\begin{tikzcd}
\text{SPR-Sigma} \arrow[r, "i"] & \text{Theory-of-Skeletal-Plan-Refinement}
\end{tikzcd}$$

PROTÉGÉ's author argues that the one of the original goals of knowledge acquisition, to replace the knowledge engineer with a tool that a domain expert can use alone, is not achievable and is unnecessary. In PROTÉGÉ, a knowledge engineer extends PROTÉGÉ (which of itself is domain-independent) into a domain-specific knowledge acquisition tool. This process is represented by Figure 4.18, "Protégé's Domain-Independent KA Language Made Domain-Specific," on page 50. In collaboration with a domain expert, the knowledge engineer accomplishes this by extending SPR-Sigma into the specification Domain-Specific-KA-Lang. This extension renames the domain-independent vocabulary of SPR-Sigma into an domain-specific (i.e., mnemonic) vocabulary. Domain-Specific-KA-Lang's top-level planning entity is a *protocol*. Lower-level planning entities below protocol are tablet, test, and wait. Recall that tablet, test, and wait are elements of the vocabulary of the performance system's implementation language ΣC. Domain-Specific-KA-Lang's task-level actions are end-protocol, increase-dose, decrease-dose, add-tablet, stop-tablet, and order-test. Domain-Specific-KA-Lang's input data language vocabulary has elements for blood pressure, pulse rate, respiration, and weight.

Figure 4.18 PROTÉGÉ**'s Domain-Independent KA Language Made Domain-Specific**

$$\begin{tikzcd}\text{SPR-Sigma} \arrow[d] \\ \text{Domain-Specific-KA-Lang}\end{tikzcd}$$

After the knowledge engineer defines Domain-Specific-KA-Lang, a domain expert adds domain knowledge by creating a definitional extension, as shown in Figure 4.19, "Protégé Domain Expert Uses Domain-Specific KA Language," on page 50. The kind of knowledge added by the domain expert deals with skeletal hypertension treatment plans. These skeletal plans are simple process specifications. Within a given skeletal treatment plan, a domain expert can define actions to take in response to input data values. For example, under the SANDOZ 331 protocol, if the patient's sitting diastolic blood pressure exceeds 90 mm Hg, then the end-protocol action must be taken.

Figure 4.19 PROTÉGÉ **Domain Expert Uses Domain-Specific KA Language**

$$\begin{tikzcd}\text{SPR-Sigma} \arrow[d] \\ \text{Domain-Specific-KA-Lang} \arrow[d, "d" description] \\ \text{Domain-Spec}\end{tikzcd}$$

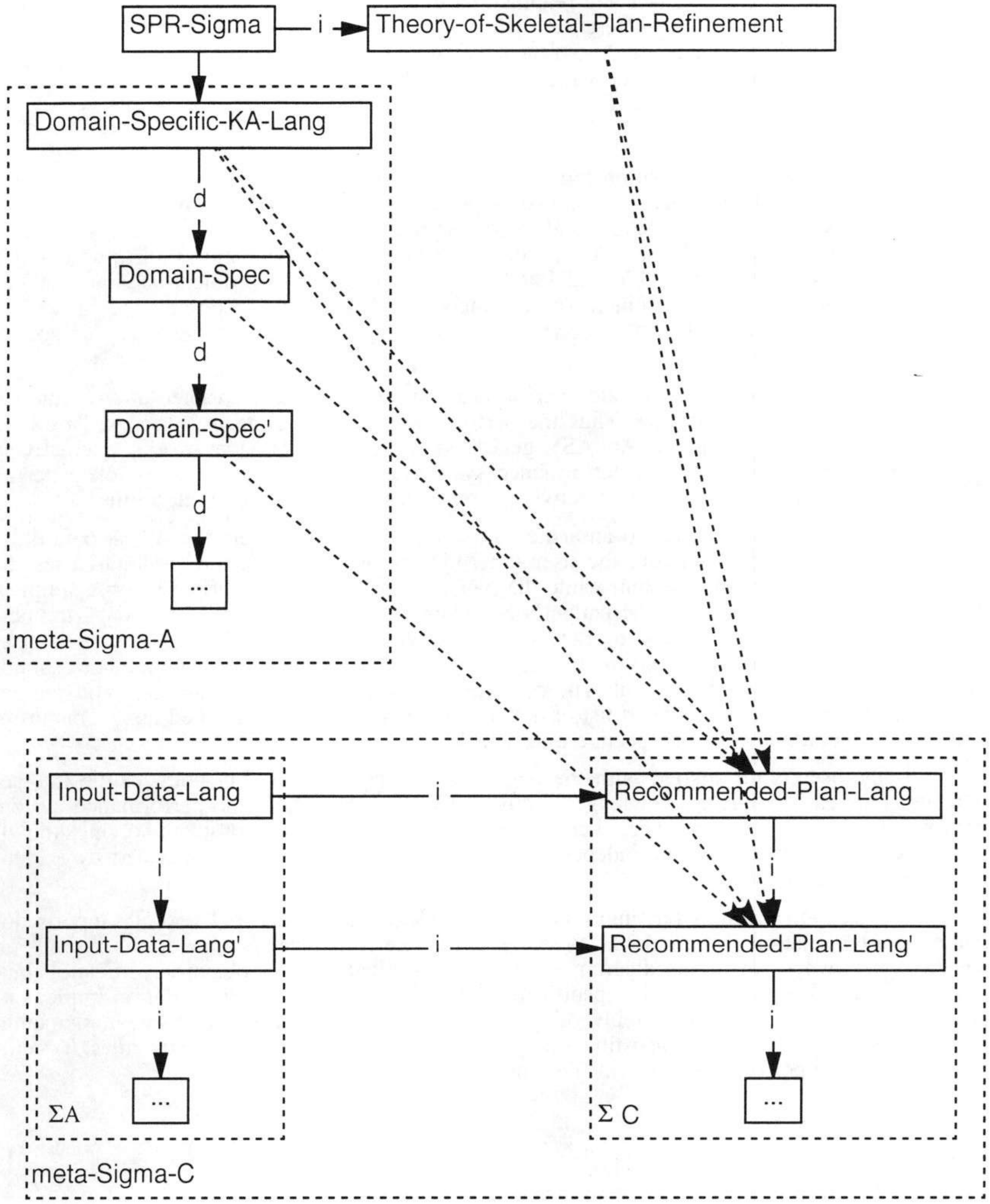
After the domain expert defines a collection of protocols (i.e., planning entities) and actions within each protocol (i.e., task-level actions), a new performance system can be generated. This process is represented in Figure 4.20, "Protégé Synthesizes the Domain-Specific Performance System," on page 51. We again represent this process using a colimit diagram. However, as was the case with SALT, we believe that colimits alone are not able to synthesize a complete representation of the performance system. Please refer to our discussion of these issues in page 46.

Figure 4.20 PROTÉGÉ **Synthesizes the Domain-Specific Performance System**

$$\begin{tikzcd}
\text{SPR-Sigma} \arrow[r, "i"] \arrow[d] & \text{Theory-of-Skeletal-Plan-Refinement} \arrow[ddd, dashed] \\
\text{Domain-Specific-KA-Lang} \arrow[d, "d"] \arrow[ddr, dashed] & \\
\text{Domain-Spec} \arrow[dr, dashed] & \\
\boxed{\text{Input-Data-Lang}} \arrow[r, "i"] & \boxed{\text{Recommended-Plan-Lang}} \\
\Sigma A & \Sigma C
\end{tikzcd}$$

Figure 4.21, "Protégé's Approach to Domain Theory Evolution," on page 52 concludes our interpretation of PROTÉGÉ'S architecture and how it supports domain theory evolution. Overall PROTÉGÉ and SALT have very similar structure (cf. Figure 4.15, "SALT's Approach to Domain Theory Evolution," on page 48). As revealed by recasting SALT and PROTÉGÉ within the framework, we see that there are only 2 important differences between the two tools.

Figure 4.21 PROTÉGÉ**'s Approach to Domain Theory Evolution**



The first key difference between SALT and PROTÉGÉ is the choice of domain-independent problem solving method. SALT uses the theory of propose-and-revise to structure domain theories, while PROTÉGÉ uses the theory of skeletal-plan-refinement to structure domain theories. The second difference is that PROTÉGÉ makes the knowledge acquisition language domain-specific by allowing the items of the vocabulary of the domain-independent knowledge acquisition language to be refined and renamed so that they are mnemonic within the application domain. Other than these differences, when viewed from within our framework for domain theory evolution, SALT and PROTÉGÉ have basically the same architecture and support domain theory evolution in the same ways.

4.5 ASK

ASK [9] is a knowledge acquisition tool which goes a step beyond the capabilities of SALT and PROTÉGÉ in the area of acquiring domain-specific problem-solving knowledge. The kinds of problem-solving knowledge SALT and PROTÉGÉ can apply to a specific problem are completely specified a priori with the selection of which knowledge acquisition tool to use, SALT or Protégé. This is because SALT and PROTÉGÉ are hardwired to problem-solving methods (i.e., the domain-independent problem-solving theories). Thus it is not possible for a domain expert to extend SALT- or PROTÉGÉ-generated performance systems with new domain-specific, problem-solving knowledge.

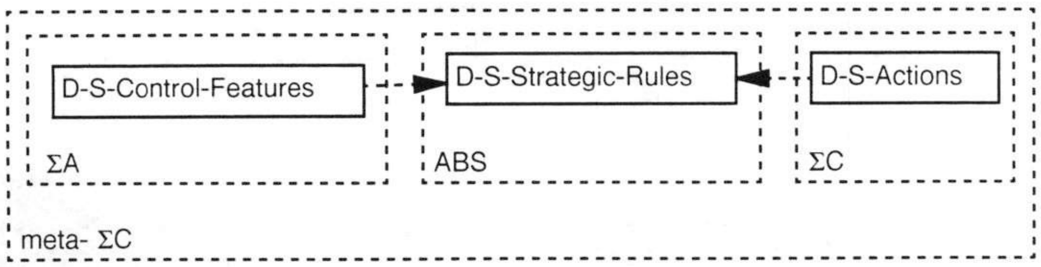
Execution of an ASK-generated performance system can be interrupted at any time for knowledge acquisition interviews. Thus the performance system is meant to be used by both end users and domain experts. An ASK-generated performance system works interactively with its user. An ASK-generated performance system does not synthesize a complete design or plan in batch mode, but rather interactively synthesizes a plan one action at a time.

In [9], ASK was applied to an acute medical care application domain, chest pain diagnosis. In this application domain, the user (an M.D. or other medical professional) uses the ASK-generated performance system while the patient is in the doctor's office being examined. The user inputs data describing the patient's complaint and condition. Given this data, the performance system provides a list of ranked actions which might be applied next. The user chooses one of these actions from the list, e.g., run an EKG test, and performs the corresponding real-world action on the patient. The performance system asks what the results of the action are and provides a new ranked list of actions one of which can be applied next. This problem-solving method is call "prospective diagnosis."

If the user is unsatisfied with the list of actions provided at any point in the interaction, the user can initiate a knowledge acquisition interview to help the performance system improve its choices. This is where the user can add domain-specific problem-solving knowledge to complement the domain-independent problem-solving knowledge provided by a theory of prospective-diagnosis.

This section presents our interpretation of ASK's architecture and how it supports domain theory evolution. Figure 4.22, "ASK-generated Performance System Architecture," on page 54 is shown below. This diagram shows the organization of problem specification language ΣA, consisting solely of the specification D-S-Control-Features; the solution implementation language ΣC, consisting solely of the specification D-S-Actions; and the implementation relation language ABS, consisting solely of the specification D-S-Strategic rules. Together these form the performance system (i.e., the language meta-ΣC).

Figure 4.22 ASK-generated Performance System Architecture



D-S-Control-Features contains the set of domain-specific control features. Gruber defines a control feature as a variable or function that gives the inference engine information about the current state of the problem (i.e., what is known about the problem) being executed by the performance system. Predicates can be written using control features in order to define "situations." A situation is a class of problem states, e.g., the patient is over 50 year old and is a smoker. D-S-Actions contains the set of domain-specific actions to perform, e.g., order an EKG test. D-S-Strategic-Rules contains the set of "strategic rules." Strategic rules map a situation (i.e., a class of problem states) to a ranked list of actions, one of which can be selected by the user and performed next. Thus D-S-Strategic-Rules includes both D-S-Control-Features and D-S-Actions.

Figure 4.23, "ASK Domain Language and Hardwired Problem-Solving Theory," on page 54 is shown below. D-S-Substantive-KB is the domain-specific "substantive knowledge base." Our interpretation is that the substantive knowledge base provides the domain-specific language upon which control features and actions are later defined. D-S-Substantive-KB is produced by the knowledge engineering in consultation with the domain expert when the domain-specific instantiation of ASK is being constructed. Note that ASK does not provide the domain expert with any means of evolving the basic domain vocabulary. D-S-Substantive-KB is mapped into the specification Theory-of-Prospective-Diagnosis via a specification morphism, thus embedding the domain-specific language into the domain-independent problem-solving method.

Figure 4.23 ASK Domain Language and Hardwired Problem-Solving Theory

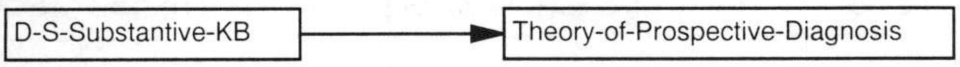


Figure 4.24, "ASK's Relationships Between Languages," on page 55 is shown below. The basic domain-specific vocabulary provided by D-S-Substantive-KB is imported by D-S-Control-Features and by D-S-Actions.

Figure 4.24 ASK's Relationships Between Languages

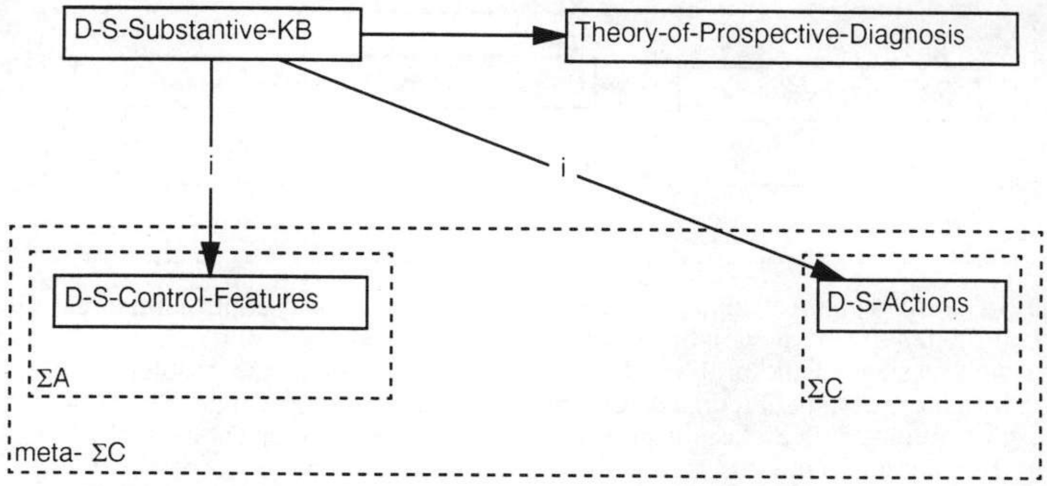


Figure 4.25, "ASK Synthesizes the Domain-Specific Performance System," on page 55 is shown below. Again we represent the synthesis of the domain-specific performance system using a colimit diagram. This diagram represents generation of only the default set of domain-specific strategic rules, i.e., the set of domain-independent strategic rules provided by Theory-of-Prospective-Diagnosis.

Figure 4.25 ASK Synthesizes the Domain-Specific Performance System

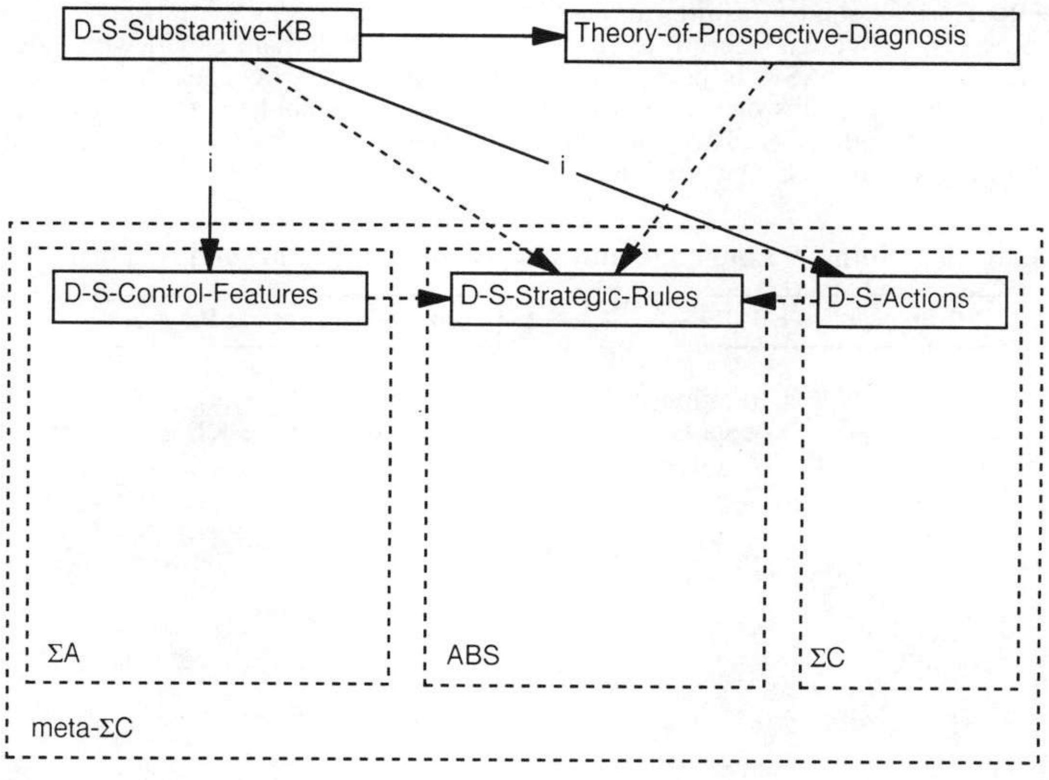
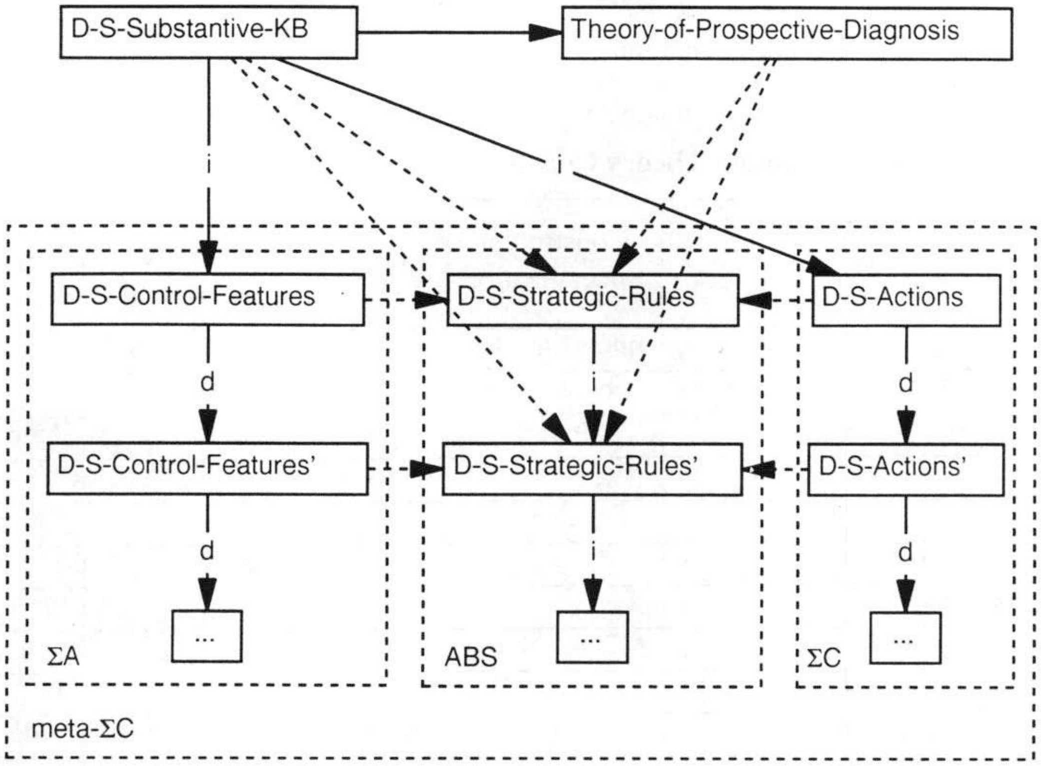


Figure 4.26, "ASK's Approach to Domain Theory Evolution," on page 56 is shown below. This diagram represents the process in which the domain expert initiates a knowledge acquisition interview. A knowledge acquisition interview can cause conservative extensions of D-S-Strategic-Rules, but only definitional extensions to D-S-Control-Features and D-S-Actions.

Figure 4.26 ASK's Approach to Domain Theory Evolution



ASK supports a similar group of atomic specification constructors as SALT and PROTÉGÉ as well as the induction sentence constructor. However, in achieving support for induction, ASK seems to have conflated meta-ΣC and meta-ΣA. These functional elements of the reference architecture should be disjoint. This conflation makes the architecture much more difficult to understand than well-structured tools such as SALT and PROTÉGÉ.

5 Summary

This section summarizes our evaluation of the 5 knowledge acquisition tools. The table below indicates which tool supports which atomic domain theory constructors and for which role ("eu" for end user and "de" for domain expert).

Matrix 2: Atomic Domain Theory Constructors Supported by KA Tools Surveyed

Part of Domain Theory	Constructor	Constructor Specialization	LOLA	Ozym	SALT	PROTÉGÉ	ASK
instances	`const i : S`	unconstructed	eu,de	de	de	de	eu,de
	`const i : S1,S2`	constructed					
	`op f : S1 -> S2`	function		de	de	de	
Sorts	`sort S`						
	`sort-axiom PS = S1,S2`						
	`sort-axiom CS = S1+S2`						
	`sort-axiom F = S1 -> S2`						
	`sort-axiom SS = S|p`						
	`sort-axiom QS = S/e`						
Sentences	human invention		eu,de	de	de	de	de
	deduction						
	induction						de
	universal instantiation		de		de	de	de
Specs	sorts S ops Ω axioms Φ						
	$\cup_{i \in I} SP_i$						
	translate SP by σ	inclusion	de	de	de	de	de
		definitional ext.	de		de	de	de
		conservative ext.	de	de	de	de	de
		renaming					
	derive from SP′ by σ	instantiation	eu	eu,de	eu	eu	eu
		analogy					
	iso close SP						
	minimal SP wrt σ						
	abstract SP wrt $\Phi(X)$						
	$\lambda X : \Sigma_{par} \bullet SP_{res}$			eu,de			
	colimit of D		de		de	de	de
Institutions	$I = (\mathrm{Sign},\mathrm{Sen},\mathrm{Mod},\models_{\Sigma})$						
	(Φ,α,β)						
	Duplex						
	Multiplex						

We clearly see that most atomic domain theory constructors are not supported. However, those constructors that are supported are supported by most of the tools surveyed. The large number of unsupported constructors may indicate a new research agenda for knowledge acquisition tools. Thus the framework indicates kinds of domain theory evolution that have not yet been explored in knowledge acquisition tools for synthesis.

Not shown in the summary table is the parsimony of each tool architecture with the draft reference architecture for knowledge acquisition tools defined in Section 3.2 on page 16. This aspect of architectural support is addressed next. We notice that SALT and PROTÉGÉ map the components and connectors of their architectures to the reference architecture in the most straightforward manner. There is no conflation of functional elements. LOLA, Ozym, and ASK do not map their architectural components and connectors to the reference architecture in a straightforward manner. LOLA conflates meta-ΣA and ΣC. Ozym does not have a clearly defined interface between ΣA and the remainder of its architecture. ASK conflates meta-ΣA and ABS.

6 Conclusions

This survey presented a framework for domain theory structure and evolution. This framework served as an architectural metalanguage in which to recast tools for evaluation. This framework/metalanguage laid out theoretical boundaries for the space of domain theory evolution. We used our understanding of Amphion [19] to help define a draft reference architecture for knowledge acquisition tools. Together the metalanguage and reference architecture were used to qualify knowledge acquisition tool support for domain theory structure and evolution in an embryonic, SAAM-style [15] evaluation. This survey discovered that most of the space of domain theory evolution remains unexploited by knowledge acquisition tools for synthesis.

By recasting knowledge acquisition tool architectures within the framework, this survey identified two important and interesting themes: "task-level architectures" and colimits. "Task-level architectures" (TLAs) are used in SALT an PROTÉGÉ and enable a domain-independent, problem-solving method to organize a narrowly-focused, domain-dependent knowledge acquisition tool. The two knowledge acquisition tools based on TLAs mapped neatly onto the draft reference architecture. The obvious drawback of TLAs is that hardwiring a knowledge acquisition tool to a single problem-solving method forever limits the applicability of the tool. Real-world application domains are complex and have aspects which cannot all be solved by a single problem-solving method. This indicates that perhaps our research should focus on using the framework to build a more general TLA-style knowledge acquisition tool, allowing multiple problem-solving methods to be applied to evolution of a single, complex domain theory.

This survey found colimits to be ubiquitous in structuring and evolving domain theories. Colimits provide a useful means of modularizing a collection of theories. However, in the area of synthesis of the performance system, colimits alone are not enough and must be complemented with representation reformulations and universal instantiation. This indicates that perhaps our research should focus also on using colimits to organize an elegant approach to domain theory synthesis.

Because so much of the space of domain theory evolution remains unexploited by current knowledge acquisition tools for synthesis, we believe we have established a research goal for the discipline of knowledge acquisition. This new research goal seeks to establish a synergy between the disciplines of knowledge acquisition and algebraic specification. Knowledge acquisition will benefit from the synergy by receiving a correctness-preserving, formal framework for domain theory structuring and evolution. Algebraic specification will benefit from the synergy by focussing on a new kind of theory, namely domain theories, rather than program specifications. We believe this synergy will help to solve long-range problems in Software Engineering's new era of domain-specificity.

7 References

1. Baffes, P.; Mooney, R. "Refinement-based student modeling and automated bug library construction." *Journal of Artificial Intelligence in Education*, 1996, vol.7, (no.1):75-116.
2. Booch, G., Rumbaugh, J., *Unified Method for Object-Oriented Development Documentation Set Version 0.8*, Rational Software Corporation, Santa Clara, CA, 1995.
3. *Readings in knowledge acquisition and learning : automating the construction and improvement of expert systems* / edited by Bruce G. Buchanan & David C. Wilkins. San Mateo, Calif. : M. Kaufmann Publishers, c1993.
4. C.J. Date, An introduction to database systems, 6th ed., Addison-Wesley Publishing Company, c1995.
5. DeLoach, S.; Bailor, P.; Hartrum, T., "Representing object models as theories." in: *Proceedings. The 10th Knowledge-Based Software Engineering Conference*, Boston, MA, USA, 12-15 Nov. 1995. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1995. p. 28-35.
6. Eriksson, H., "A survey of knowledge acquisition techniques and tools and their relationship to software engineering." *Journal of Systems and Software*, Sept. 1992, vol.19, (no.1):97-107.
7. G. Fischer, A. Girgensohn, K. Nakakoji, D. Redmiles, "Supporting software designers with integrated domain-oriented design environments," *IEEE Transactions on Software Engineering*, June 1992, vol.18, (no.6):511-22.
8. Goguen, J.A.; Burstall, R.M. "Institutions: abstract model theory for specification and programming." Journal of the Association for Computing Machinery, Jan. 1992, vol.39, (no.1):95-146.
9. Gruber, T.R., "Automated knowledge acquisition for strategic knowledge." Machine Learning, Dec. 1989, vol.4, (no.3-4):293-336.
10. Gruber, Thomas R., *The acquisition of strategic knowledge* / Thomas R. Gruber. Boston : Academic Press, c1989. Series title: Perspectives in artificial intelligence ; v. 4.
11. Hoare, C.A.R., "Proof of correctness of data representations." IN: *Programming methodology. A collection of articles by members of IFIP WG2.3*. Edited by: Gries, D. Berlin, West Germany: Springer-Verlag, 1978. p. 269-81.
12. Ipser, E.A., Jr.; Wile, D.S.; Jacobs, D. A multi-formalism specification environment. (Fourth ACM SIGSOFT Symposium on Software Development Environments, Irvine, CA, USA, 3-5 Dec. 1990). *SIGSOFT Software Engineering Notes*, Dec. 1990, vol.15, (no.6):94-106.
13. Iscoe, Neil Allen, Browne, James C., Werth, John, Generating Domain Models for Program Specification and Generation, University of Texas at Austin, Department of Computer Sciences, Technical Report CS-TR-89-13, May 1989.
14. R. Jullig and Y.V. Srinivas, "Diagrams for software synthesis," *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, Sept. 20-23, 1993, Chicago, IL, IEEE Computer Society Press, p10-19. Kestrel Institute Technical Report KES.U.93.2, 9pp.

15.Kazman, R.; Bass, L.; Abowd, G.; Webb, M., "SAAM: a method for analyzing the properties of software architectures," Proceedings of 16th International Conference on Software Engineering, Sorrento, Italy, 16-21 May 1994). Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994. p. 81-90.

16.Keller, Richard M., "Knowledge-intensive software design systems: can too much knowledge be a burden?" in Working Notes of the AAAI-92 Workshop on Automating Software Design, AAAI-92. Proceedings Tenth National Conference on Artificial Intelligence, San Jose, CA, USA, 12-16 July 1992). Menlo Park, CA, USA: AAAI Press, 1992.

17.Kipps, James Randall, An approach to component generation and technology adaptation, Department of Information and Computer Science, University of California, Irvine, California, 1993.

18.Lavrac, Nada. *Inductive logic programming : techniques and applications* / Nada Lavrac and Saso Dzeroski. New York : E. Horwood, c1994. Series title: Ellis Horwood series in artificial intelligence.

19.Lowry, M.; Philpot, A.; Pressburger, T.; Underwood, I. "A formal approach to domain-oriented software design environments." IN: *Proceedings KBSE '94. Ninth Knowledge-Based Software Engineering Conference*, Monterey, CA, USA, 20-23 Sept. 1994. Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1994. p. 48-57.

20.Marcus, S.; McDermott, J. "SALT:a knowledge acquisition language for propose-and-revise systems"Artificial Intelligence,May 1989,vol.39,(no.1):1-37.

21.Mettala, Erik LTC.; Graham, Marc H. (eds.). *The domain-specific software architectures program.* Software Engineering Institute report CMU/SEI-92-SR-9.

22.*Inductive logic programming* / edited by Stephen Muggleton. London ; San Diego : Academic Press in association with Turing Institute Press, c1992. Series title: A.P.I.C. studies in data processing ; no. 38.

23.Musen, M.A., "Automated support for building and extending expert models." Machine Learning, Dec. 1989, vol.4, (no.3-4):347-75.

24.Novak, G.S., Jr. "Conversion of units of measurement." *IEEE Transactions on Software Engineering*, Aug. 1995, vol.21, (no.8):651-61.

25.Prieto-Diaz, Ruben. *Domain analysis and software systems modeling* / Ruben Prieto-Diaz and Guillermo Arango. Los Alamitos, Calif.: IEEE Computer Society Press, c1991. Series title: IEEE Computer Society Press tutorial.

26.A. A. Reyes, "An Approach to Automatic Generation of Domain Theories from Intuitive, Semiformal Domain Models" *Proceedings of the California Software Symposium CSS'96*, W. Scacchi & R. Taylor, eds., 17 April 1996, University of California, Los Angeles, CA. USC Center for Software Engineering, UCI Irvine Research Unit in Software.

27.Srinivas, Yellamraju V. *Algebraic specification : syntax, semantics, structure* / Yellamraju V. Srinivas. [Irvine] : Information and Computer Science, University of California, Irvine, [1990]. Series title: Technical report (University of California, Irvine. Dept. of Information and Computer Science) 90-15.

28.Wirsing, M. Algebraic specification. In: *Handbook of Theoretical Computer Silence* (J. van Leeuwen, ed.). North-Holland (1990).