Structuring Languages as Algebraic Specifications: A Framework for Multilingual System Representation

Arthur Alexander Reyes
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425 U.S.A.
e-mail: artreyes@ics.uci.edu

Technical Report 95-23
23 June 1995

Abstract

With this paper we would like to introduce what we believe to be a unifying framework for the investigation of multilingual system representation phenomena. The framework applies the body of knowledge on algebraic specifications to the representation and manipulation of formal languages. In this framework, a formal language is represented by an algebraic specification; the language's syntax is represented by the algebraic specification's signature; and the language's semantics are represented by the algebraic specification's axioms. Strings in the language are represented by terms of the algebraic specification, or in the more general case by formulas of the algebraic specification or a definitional extension of the algebraic specification. Algebraic specification structuring operations, such as specification morphisms, parameterized specifications, and colimits are used to construct new languages from component languages.

We review a number of approaches to multilingual system representation that have been presented in the research literature. For each of these approaches, we show how the approach can be represented in our framework.

1 Introduction

In software development, the high cost of requirements errors has motivated research in specification languages. Specification languages possess mathematically precise syntax and semantics and are accompanied by proof rules that enable rigorous analysis. Representing a system in a specification language enables the power of mathematical logic to be applied to reduce requirements errors.

As evidenced by the plethora and long history of system development methodologies that require the use of different languages, the software development community has long recognized the need to use more than a single language to represent a complex system. A significant problem associated with these multilingual system development methodologies is that the developer has little control or insight into the ways in which the different languages interface.

With this paper we would like to introduce what we believe to be a unifying framework for the investigation of multilingual system representation phenomena. The framework applies the body of knowledge on algebraic specifications to the representation and manipulation of formal languages. In this framework, a formal language is represented by an algebraic specification; the language's syntax is represented by the algebraic specification's signature; and the language's semantics are represented by the algebraic specification's axioms. Strings in the language are represented by terms of the algebraic specification, or in the more general case by formulas of the algebraic specification or a definitional extension of the algebraic specification. Algebraic specification structuring operations, such as specification morphisms, parameterized specifications, and colimits are used to construct new languages from component languages.

In section 2, we review the fundamental issues of requirements errors and reinterpret requirements phenomena is a model-theoretic framework. This allows us to uniformly examine several approaches to reducing requirements errors. After demonstrating the limitations of each approach to reducing requirements errors, we examine how to enhance the most effective approach by the use of more than one formal language simultaneously. Section 3 presents the fundamentals of our framework, called Structuring Languages as Algebraic Specifications (SLAS) and how it can be used to represent the different ways in which languages can be used together. Section 4 reviews a number of approaches to multilingual system representation and demonstrates how they can be represented in the SLAS framework. By representing each approach within the framework, the most fundamental similarities and differences between the approaches are elucidated. Section 5 discusses a number of important directions in which this research may progress and the key risk area that threaten each direction. Section 6 presents our conclusions.

2 Requirements Errors

In which area of software engineering can formal methods produce the greatest gains? We believe that applying formal methods to the problems of system requirements definition and analysis [Davis] will result in the most significant early advances.

Eliminating an error during the requirements development phase of a system can be 1/100 the cost of eliminating the error during the maintenance phase [Davis, p. 23]. This is because undiscovered errors cumulatively propagate through system development. The cost of eliminating a error at a certain stage of development is dependent on the amount of subsequent work that was built on top of the error and which must be redone. Thus a requirement error which is not discovered until the maintenance phase requires that a long history of work be redone: all documentation and system representations related to the requirement itself along with design decisions which assumed the requirement was correct along with all implementations which assumed that the design was correct along with all tests performed which assumed the implementation was correct. Thus if we plan to use a system development methodology in which requirements specifications significantly influence later development decisions, it is very important that the requirements be correct.

Non-clerical requirements errors fall into 4 main groups with the following distributions (from [Davis]):

- incorrect facts 49%,
- omissions 31%,
- inconsistencies 13%, and
- ambiguities 5%.

In this section we will provide a simple mathematical framework within which the different groups of requirements errors can be represented. These representations will provide insight into the nature of the errors and will suggest solutions.

2.1 Requirements Phenomena in a Model-Theoretic Framework

We discuss requirements phenomena within the framework of *model theory*. Please see [Wirsing] for a full treatment of model theory within the discipline of *algebraic specifications*. An algebraic specification is a pair (Σ,Φ) consisting of a *signature* Σ and a set of *axioms* Φ. A signature is a set of *sort* symbols, S, and a set of *operation* symbols, Ω. Each operation symbol in Ω has a *rank* consisting of a finite string of sort symbols and a result sort. A *sentence* is a sequence of predicates or of equations between terms of the same sort (constructed using the signature) connected by the operators of the *logical system* used to write the axioms. An axiom is a sentence that appears in the definition of an algebraic specification. The signature provides syntax and the axioms provide semantics.

An *algebra* is a mathematical structure built on top of a signature. An algebra consists of a set of sets, each set of which (called a *carrier set*) corresponds to a sort symbol s in S; and a set of functions corresponding to the set of operation symbols f in Ω such that the rank and result sort of each operation symbol is preserved. An algebra *satisfies* a sentence if the sets and functions of the algebra preserve the meaning of the sentence. An algebra that satisfies a sentence is said to be a *model* of the sentence. If we write

$$A \models req$$

it means that A is a model of the sentence req. This is the *satisfaction relation*. Thus given a signature (for syntax), we can write a number of sentences, treat each sentence as a requirement, and attempt to identify a model of the set of requirements (i.e., a system satisfying the requirements).

|= is an overloaded symbol. We can also write

System-sentences |= new-fact

where System-sentences is a (possibly empty) set of sentences and new-fact is a single sentence (all sentences are generated from the same signature). This is the *logical consequence relation* and it means that every model of System-sentences is also a model of new-fact. The closure of a set of sentences under logical consequence is called a *theory*.

If we wish, we could take all of our requirement sentences and place them on the left hand side of the logical consequence relation and determine what new sentences are logical consequences of the set of requirements. Note that we could also place on the left side of the logical consequence relation a set of sentences that represents a system in such a way that System-sentences does not contain any requirement sentences (but which would be constructed using the same signature as the requirement sentences) and use the logical consequence relation to determine if each requirement sentence is a logical consequence. Thus we would be able to formally verify whether the system representation meets the system's requirements.

This framework allows us to identify a number of formal counterparts to the groups of requirements errors listed earlier:

- Incorrect facts: Incorrect facts are ostensibly the result of inadequate knowledge of the application domain. Consider the left hand side of the logical consequence relation. In any application of the logical consequence relation, the set of sentences on the left side is the *hypothesis* (i.e., the set of sentences that we assume are true). Incorrect facts correspond to hypothesis sentences that are not satisfied by the model that is the real world.
- Omissions: How does a system developer discover that a requirement was omitted? If we have a system that meets its requirements but displays a behavior which we did not intend and which may be undesirable, then we discover that a requirement to restrict the system's behavior was omitted. This can happen because a set of requirement sentences can have unforeseen logical consequences. This is a logical consequence view of the phenomenon. There is also a satisfaction relation view of the phenomenon: The set of models of a set of sentences can range over a broad spectrum. "Initial" models can be thought of as exhibiting only the behavior allowed by the axioms and nothing else. "Final" models can be though of exhibiting every behavior except the negation of the axioms. Thus models must be chosen carefully. Initial and final models of an algebraic specification will be discussed further in the section on Semantics.
- Inconsistencies: Inconsistent requirements mean that the set of requirement sentences is unsatisfiable, i.e., there does not exist a model for the set of requirement sentences. Another way to think about this is that there is a logical contradiction (the sentence true=false is a logical consequence) among the set of requirement sentences. Because any sentence and its negation is a logical consequence of a set of unsatisfiable sentences, systems based upon inconsistent requirements can exhibit unexpected and contradictory behavior. Thus it is desirable to determine early in the development cycle whether the set of requirement sentences is satisfiable.
- Ambiguities: Ambiguous requirements are often a result of the imprecision of natural language. Natural language requirements can often be parsed in more than one way and therefore can have more than one meaning. An ambiguous requirement can be met by more than one system (which are possibly inconsistent with each other). A formal requirement sentence is not ambiguous in the logical consequence view because its set of logical consequences is fixed. However, in the satisfaction view, a requirement

sentence can be satisfied by many algebras. The tasks is to choose only those algebras that are relevant to the application domain.

A number of solutions exist to minimize requirements errors. The solutions and their relative contribution to discovering requirements errors (from [Davis]) are given below.

- inspections 65%: These are manual and automated reviews and analyses of static system representations, such as documents, designs, and code. Kinds of languages used for these representations are
    - Informal languages, such as natural language;
    - semiformal languages, which represent system requirements and designs more or less directly and are supported by a number of simple analyses that can be performed automatically; and
    - formal languages and calculi, which represent requirements and systems in terms of algebraic specifications and models and which can directly determine whether the satisfaction and logical consequence relations hold.
- evaluations 10%: These are simulations to test as-required behavior of a system before it is implemented.
- unit testing 10%
- integration testing 5%
- other 10%

In particular, we will discuss limits to the effectiveness of simulations, semiformal languages, and formal languages and calculi in discovering requirements errors.

2.2 Simulations

System simulation has been an active area of research and application for over 30 years [Gould]. In system simulation, a number of "models" (i.e., simplified representations) of the system and its subsystems and constructed. These models are abstractions of the relevant parts of the system under examination and are usually constructed in executable languages. A model is subjected to a number of tests to determine if the simulated behavior of the system meets the system requirements and the expectations of the developers. Often a developer, playing the part of a prospective user of the system (or even an actual user) performs test activities in real-time with the model. Such testing provides fast feedback on the effects of requirements. Using simulations can narrow the number of major requirements and design decisions that must be made and focus the development effort on constructing a system that the user wants. Simulation is supported by a number of sophisticated languages, tools, and environments to develop and test models quickly. Examples of such languages and environments include MODSIM II [Belanger], and MATRIX_X [Winston].

How well do simulations reduce the number of requirements errors and how are they limited?

Incorrect facts: Because incorrect facts are mostly the result of incorrect or insufficient domain knowledge, it is important to obtain correct domain knowledge from people who "know best" i.e., the users. A user may be more likely than a system developer to test a simulation model in ways that reflect typical usage patterns. While this can illuminate incorrect fact requirement errors, a simulation model must be developed first. Simulation models available early in the system development process will be fairly crude abstractions of what the implemented system could be like, based upon the current requirements. As crude abstractions, the number of incorrect facts that can be discovered will be limited. As the system development progresses, more detailed simulation models will become available and

will enable other incorrect facts to be discovered. Hopefully, significant incorrect facts will be discovered early in the system development.

Omissions: Omitted requirements can be discovered by simulation when the model implements the requirements and yet manifests a behavior that was not anticipated and/or is undesired. Models are put through a series of tests that are designed to exercise its behavior in a number of normal and abnormal operating conditions. Unfortunately, as long as requirements are stated in informal language, it is difficult to know whether the model correctly implements the requirements, therefore possibly violating the assumption that must hold in order for a model to reveal an omission.

Inconsistencies: Inconsistent requirements may be discovered during development of the simulation model. But as with discovering an inconsistency during development of the actual system, a decision must be made about which requirement is changed, which is unchanged, or if both are changed. Any development of the model based upon the changed requirement will need to be reviewed. Testing the model can reveal inconsistent behaviors, if the test plan is sure to execute both points needed to reveal the inconsistent behavior.

Ambiguities: Ambiguous requirements must be discovered during model development, or else they will reveal themselves as inconsistent behavior during model testing. Because simulation models are abstractions of implementations, models often have model-specific requirements defined, must be designed, implemented and tested before they are used in any real-time tests or with prospective users. Also, models can become large and difficult to maintain.

2.3 Semiformal Languages

Semiformal languages allow developers to think about the characteristics of the system under development more abstractly than simulation languages permit. Many semiformal languages are used in software development today. Semiformal languages possess formal syntax and informal semantics. Examples of semiformal languages include Data Flow Diagrams (DFDs) [Gane & Sarson], Finite Automata [Hopcroft and Ullman], Class/Object Diagrams [Rumbaugh et al.], and Structure Charts [Jackson]. Many system development methodologies are distinguished by the different semiformal languages they use. Example system development methodologies are Object Modeling Technique (OMT) [Rumbaugh et al.], Object-Oriented Design (OOD) [Booch], and Jackson Structured Design (JSD) [Jackson].

Automated support for a semiformal language usually means a number of simple checks can be performed by a tool on strings written in the semiformal language. These checks might include parsing the string to verify syntactic correctness, verifying that inputs and outputs are consistent with upper and lower levels of the string's hierarchy, and other consistency checks. Automated support may also include the generation of source code or simulation models, provided that sufficient detail is given in the semiformal representations. These generated models and codes can provide a de facto semantics for the semiformal languages, but details of this generation are usually defined by the tool developer, not by the developer of the semiformal language (and especially not by the user of the semiformal language). This means that a typical user of a semiformal language has little control over the way in which a semiformal language interacts with the other languages the developer uses and that the developer has little insight into how a tool uses a semiformal language.

How do semiformal languages reduce requirements errors and how are they limited?

Incorrect facts: Semiformal languages provide a means of recording domain knowledge that is more abstract than via simulation models, and thus more likely to evolve and be validated. But because the semantics are informal, it is more difficult to know whether a portion of the domain knowledge has or has not been validated.

Omissions: The checking provided by some tools for semiformal languages may permit

certain types of omissions to be discovered, but these tools may only be able to reveal high-level omissions, such as failure to decompose a DFD process sufficiently. Because some tools and environments allow executable code or simulations to be produced automatically, testing of the executable objects may reveal omissions in the same manner as simulation models. Because executable objects can be generated automatically, there is greater hope that the objects will correctly meet the requirements.

Inconsistencies: Semiformal languages help reduce inconsistencies by virtue of the consistency checks that are built into the tools that support the languages. Unfortunately, the tools do not always give the developer insight into the details of the checks performed and do not allow new kinds of checks to be used with the tool. Thus inconsistencies which occur in low-level details of a semiformal representation may not be discovered by the tools. This can allow the inconsistency to remain undiscovered until later.

Ambiguities: Semiformal languages help reduce ambiguity in requirements by forcing the developer to represent requirements in a concrete, but abstract way. Because semiformal languages do not have formally defined semantics, the set of models that satisfy a sentence is not limited (except by generation tools that define de facto semantics).

2.4 Formal Languages and Calculi

A *formal language* is characterized by both a formally defined syntax and formally defined semantics. A *calculus* is a formal language along with a *deductive apparatus*. The purpose of the deductive apparatus is to determine whether a sentence in the formal language is a logical consequence of a set of sentences in the formal language or whether an algebra satisfies a set of sentences. The deductive apparatus usually takes the form of a set of sound proof rules used to determine whether the logical consequence or satisfaction relations hold in a certain case. See [Wordsworth] for the deductive system for the calculus Z. A *Specification language* is a kind of calculus which is nonprocedural and more abstract than programming languages.

Specification languages can be used over a wide spectrum of rigor. At one end of the spectrum, simply constructing a formal representation of a system (i.e., a *formal specification* of the system) in a specification language causes the developer to think about the system more abstractly and to focus on the problem to be solved by the system. Many specification languages are supported by tools that allow syntax and type checking, editing, and proof checking. Some tools allow formal specifications to be executed and therefore tested. Testing formal specifications performs a similar task to system simulation. Further rigor can be achieved by proving properties (determining logical consequences) of formal specifications. The greatest rigor is achieved by refining implementations of a formal specification using the deductive apparatus.

How do formal languages and calculi reduce requirements errors and how are they limited? Note that calculi attempt to reduce requirements errors by directly addressing the issues of determining requirements satisfiability, desired and undesired logical consequents, and selection of models.

Incorrect facts: Formal languages allow application domain knowledge to be represented precisely, evolved and validated. Application domain-oriented approaches to software development [Prieto-Diaz & Arango] seek to formalize domain knowledge so that it can be incrementally constructed and validated against the real world. The resulting domain specification becomes an important asset to the development organization and is a centerpiece of organizational training and documentation. The resulting domain specification can be used to automatically derive computer programs [Lowry et al.] that are guaranteed to meet a formal specification.

Omissions: Calculi discover requirements omissions in the course of proving properties of the formal specification. During the course of determining if a formal specification meets a

requirement sentence by proof, it may be discovered that the proof cannot be accomplished. The attempted proof explicitly shows what sentences were needed in order to make the proof succeed. Thus omissions are made explicit. There is a similarity with simulation in this regard: Just as if a test process never exercises simulation model behavior to reveal an omission, a proof process may not require a proof that if executed would reveal a requirement omission.

Inconsistencies: Calculi address requirement inconsistencies by executing a proof of satisfiability. The inconsistencies are made explicit by the attempted proof.

Ambiguities: A formal specification has a fixed set of logical consequents, and is therefore unambiguous from a syntactic point of view. From a semantic point of view, a formal specification constrains the set of algebras satisfying it. The task then becomes picking the most appropriate model of the specification.

2.5 More than One Formal Language Needed

Having been sufficiently impressed with the ability of formal languages and calculi to minimize requirements errors, we now seek to discover ways in which formal languages and calculi can be augmented to reduce requirements errors further.

It is unlikely that a single calculus will be sufficient to represent all relevant aspects of a system with equal perspicuity. Consider a large, complex system implemented using a single programming language. The chosen programming language will represent some aspects of the system with greater clarity than others.

A clear shortcoming with the application of a single calculus to the representation of a complex system is answering the question: How can we deal with requirements that cannot be described using the single calculus? This problem is of central concern to us. Solving this problem can be approached in two principal ways:

- Requirements that cannot be represented by the chosen calculus must be represented and analyzed via simulation, or
- write the unrepresentable requirements in a language that can represent them.

Analysis via simulation leaves us with the requirements error discovery problems discussed in the earlier section.

Writing the requirements in a language that can represent them means that the other language could be

- an informal or semiformal language,
- an extension of the chosen formal language so that the requirement can be represented, or
- a different formal language better suited to representing the requirements.

Representing the requirement in an informal or semiformal language leads to the associated problems discussed in the earlier section.

Extending the chosen formal language to represent the requirement is a common solution for researchers in formal languages. Accomplishing this while keeping the language's semantics clean can be challenging. An interim informal semantics may need to be adopted for the extended language until a clean semantics can be worked out. More commonly used calculi, such as Z, are frequently the subject of such extension efforts, e.g., [Carrington et al.]. Note that If all desired extensions for a given language could be accommodated simultaneously, they would diminish the language's uniqueness and produce a language that attempts to do all things for all developers (and perhaps fails to do any particular thing very well).

We believe that system developers should be given freedom to choose the most suitable formal language to represent each system requirement and aspects of the system under development. Thus we seek a situation in which a number of simple, special-purpose languages are correctly used together to represent a complete system. Thus we seek a *multilingual* approach to system representation.

3 Framework

We now present a mathematical framework within which to describe multilingual system representation phenomena. The framework applies the body of knowledge on algebraic specifications to the definition and manipulation of formal languages. We refer the reader to [Wirsing] for a comprehensive and concise treatment of the discipline of algebraic specifications.

Our framework represents each formal language of interest by an algebraic specification, hence the framework's name, Structuring Languages as Algebraic Specifications (SLAS). "Strings" in each language are represented by terms or formulas of the algebraic specification of each language. Languages are constructed from other, simpler languages by the application of well-defined algebraic specification structuring operations, such as specification morphism, parameterization, and colimit. Because an algebraic specification uses a certain logical system (e.g., equational logic, conditional equational logic, Horne clause logic, first order predicate logic, etc.) in which to write its axioms, and because each logical system has a deductive apparatus associated with it, algebraic specifications of languages are automatically calculi.

3.1 Algebraic Specifications of Languages

Algebraic specification languages have been used to represent formal languages in two principle ways:

- Write an algebraic specification to define the syntax and semantics of a language directly. This is the approach taken by [Broy et al.] and us.
- Write an algebraic specification for *individual strings* in a language. This is the approach taken by [Feijs et al.]. For example, a particular DFD (i.e., a string in the language of DFDs) can be represented by an algebraic specification in which the operations define the DFD processes directly.

In general, constructing algebraic specifications of individual strings of a language is simple, but the ability to reuse these algebraic specification is limited. Constructing an algebraic specification of a language is challenging, but the resulting algebraic specification also defines any string in the language, thus reuse of the specification can be much greater. Henceforth, we will use the terms "language" and "algebraic specification" interchangeably.

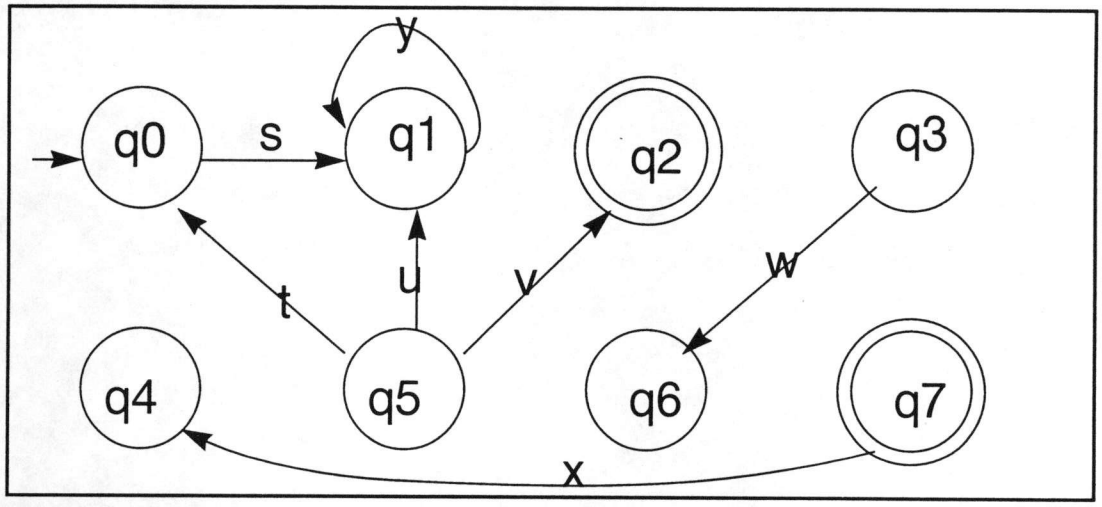
With a few simple examples, let us illustrate the use of algebraic specifications to represent and manipulate languages. We present (currently incomplete) algebraic specifications for two languages used in software engineering; the language of Finite Automata (which we will refer to as FA) and Dijkstra's language of Guarded Commands (which we will refer to as GUARDED) [Dijkstra]. We will then show a specification morphism to translate from FA to GUARDED.

The (incomplete) algebraic specification of FA follows:

spec FA
imports ID
sorts FA, Q, ARROW, Q-LABEL, ARROW-LABEL
operations
()Q: q0
(Q)FA: new-fa
(FA,ARROW)FA: add-arrow
(Q-LABEL)Q: make-q
()ARROW-LABEL: noop
(Q)Q: final
(Q,ARROW-LABEL,Q)ARROW: make-arrow
()Q-LABEL: a,b,c,...
axioms
/* Syntactic axioms are needed to make the set of finite automaton pictures a carrier of sort FA of an initial model of this specification.*/
...
/* Semantic axioms are needed to define the static qualities of finite automata (e.g., acceptance of strings). */
end spec.

An individual finite automaton (i.e., a term in **spec** FA; a string in the language FA; a program written in FA; an *instance* of the *abstract data type* FA; etc.) can be represented by a term of sort FA of this specification. The figure below is an example finite automaton.

Figure 3.1 A Finite Automaton



A term in **spec** FA representing this finite automaton is

```
add-arrow(
        add-arrow(
                add-arrow(
                        add-arrow(
                                add-arrow(
                                        add-arrow(
                                                add-arrow(
                                                        new-fa(q0),
                                                        make-arrow(q0,s,make-q(q1))),
                                                make-arrow(make-q(q1),y,make-q(q1))),
                                        make-arrow(make-q(q5),t,q0))),
                                make-arrow(make-q(q5),u,make-q(q1))),
                        make-arrow(make-q(q5),v,final(make-q(q2)))),
                make-arrow(make-q(q3),w,make-q(q6))),
        make-arrow(final(make-q(q7)),x,make-q(q4)))
```

Please note that this term is not a unique representation for the automaton in the figure. A term records the *sequence of operations* that were applied to construct the term. The automaton in the figure could have been constructed in a number of different ways. Another way to think of this is that the automaton in the figure could be parsed in different ways. Because we want these different terms to represent the same automaton, **spec** FA requires *syntactic axioms* [Broy et al.] in order to place these terms into the same equivalence class. Note that common subterms of this term could be replaced by constants and the constants defined using equations.

An (incomplete) algebraic specification of GUARDED follows:

```
spec GUARDED
import EXPR,STMNT
sorts
        GSC /* guarded command set */
        GC /* guarded command */
        G /* guard */
        GL /* guarded list */
        STMNT /* statement */
        AC /* alternate command */
        RC /* repetitive command */
operations
        ()STMNT: skip,abort
        (GC)GCS: makeGCS
        (GC,GC)GC: _[]_
        (G,GL)GC: _->_
        (BE)G: makeG
        (STMNT)GL: makeGL
        (STMNT,STMNT)STMNT: _;_
        (AC)STMNT: makeACtoSTMNT
        (RC)STMNT: makeRCtoSTMNT
        (GCS)RC: do_od
        (GCS)AC: if_fi
axioms
/* syntactic axioms */
        (gc1[]gc2)[]gc3 = gc1[](gc2[]gc3)
        (s1 ; s2) ; s3 = s1 ; (s2 ; s3)
...
/* semantic axioms */
...
end spec.
```

The syntax of GUARDED does not feature a "start symbol." However the sort STMNT is used when writing complete GUARDED programs.

The following GUARDED program

```
k := 0 ;
j := 1 ;
do j/=n ->
        if f(j)<f(k) -> j := j+1
        [] f(j)>=f(k) -> k := j ; j := j+1
        fi
od.
```

can be represented by the following term of sort STMNT:

```
(k := 0 ;
        (j := 1 ;
                (makeRCtoSTMNT(
                        do(
                                makeG(
                                        (j/=n)
                                        ->
                                        (makeGL(
                                                makeACtoSTMNT(
                                                if
                                                makeG(f(j)<f(k)) -> makeGL(j := j+1)
                                                []
                                                makeG(f(j)>=f(k)) -> makeGL(k := j ; j :=
j+1)
                                                fi)))))
                        od))))
```

3.2 Specification Morphisms

A specification morphism between two algebraic specifications enables one algebraic specification to be mapped into another algebraic specification in such a way that axioms of the source specification are theorems of the target specification. Specification morphisms play an important role in our framework for the construction of languages from other languages, translation of languages, and viewing of a language via another language. Our specification morphisms are equivalent to "interpretations" in [Turski & Maibaum]. Specification morphisms are total functions by definition.

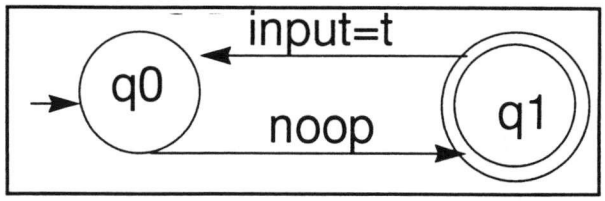
For our example, we construct a (unverified) specification morphism from **spec** FA to **spec** GUARDED. We refer to **spec** FA as the *source* language and **spec** GUARDED as the *target* language.

```
FA              |-> AC
Q               |-> GL
ARROW           |-> GC
Q-LABEL         |-> ID
ARROW-LABEL     |-> BE

q0              |-> makeGL(q := q0)
noop            |-> true
new-fa          |-> λ(gl).if(makeGCS(true->gl))fi
add-arrow       |-> λ(ac,gc).if(makeGCS(acbody[]gc))fi
                        IF ac = if(makeGCS(acbody))fi
make-q          |-> λ(x).makesGL(q := x)
final           |-> λ(gl).makeGL(s ; abort) IF gl=makeGL(s)
make-arrow      |-> λ(gl1,be,gl2).(id1=expr1&be)->makeGL(id2 := expr2)
                IF ((gl1=makeGL(id1 := expr1) or (gl1=makeGL(id1 := expr1 ;
s1)))
                & ((gl2=makeGL(id2 := expr2) or (gl2=makeGL(id2 := expr2 ;
s2)))
```

A specification morphism translates terms of **spec** FA into terms of **spec** GUARDED. Given an FA we can convert it into an equivalent term of **spec** FA:

Figure 3.2 The FA to be translated



A corresponding term of **spec** FA is

```
add-arrow(
        add-arrow(
                new-fa(q0),
                make-arrow(
                        final(make-q(q1)),
                        input=t,
                        q0))
        make-arrow(
                q0,
                noop,
                final(make-q(q1)))).
```

Which via the specification morphism translates into

```
if(
        makeGCS(
                true->makeGL(q := q0)
                []
                q=q1 & input=t -> makeGL(q := q0)
                []
                q=q0 & true -> makeGL(q := q1 ; abort)))
fi.
```

The GUARDED program corresponding to this term is

```
if      true -> q := q0
[]      q = q1 & input = t -> q := q0
[]      q = q0 & true -> q := q1 ; abort
fi.
```

Using specification morphisms in this way should follow the following process.

- Convert the concrete syntax of the source language LANG-A string to the abstract syntax of a corresponding term in **spec** LANG-A.
- Translate the term via the specification morphism.
- Convert the translated term (now a term of **spec** LANG-B) from its abstract syntax to the concrete syntax of a string of target language LANG-B.

In general, if given two languages, **spec** LANG-A and **spec** LANG-B, between which we wish to establish a relationship, we should allow ourselves to consider any possible specification morphism between them (without regard for the direction we may want the specification morphism to go). In this way we hope to always construct the most natural relationship possible between the two languages.

Note that a term translated via a specification morphism may not result in a complete term in the target language. An everyday example of this is the generation of code *fragments* from a DFD by a CASE tool.

In order to make a specification morphism that maps a language to itself in such a way that terms are restructured, we believe it is also possible to define a kind of specification morphism that maps sorts to sorts, operations to operations (and possibly to terms of compatible sort), and terms to terms of compatible sort. We will make use of this idea in this paper for specify optimizing compilers and code reformatters. We expect to pursue the details of this concept in future research.

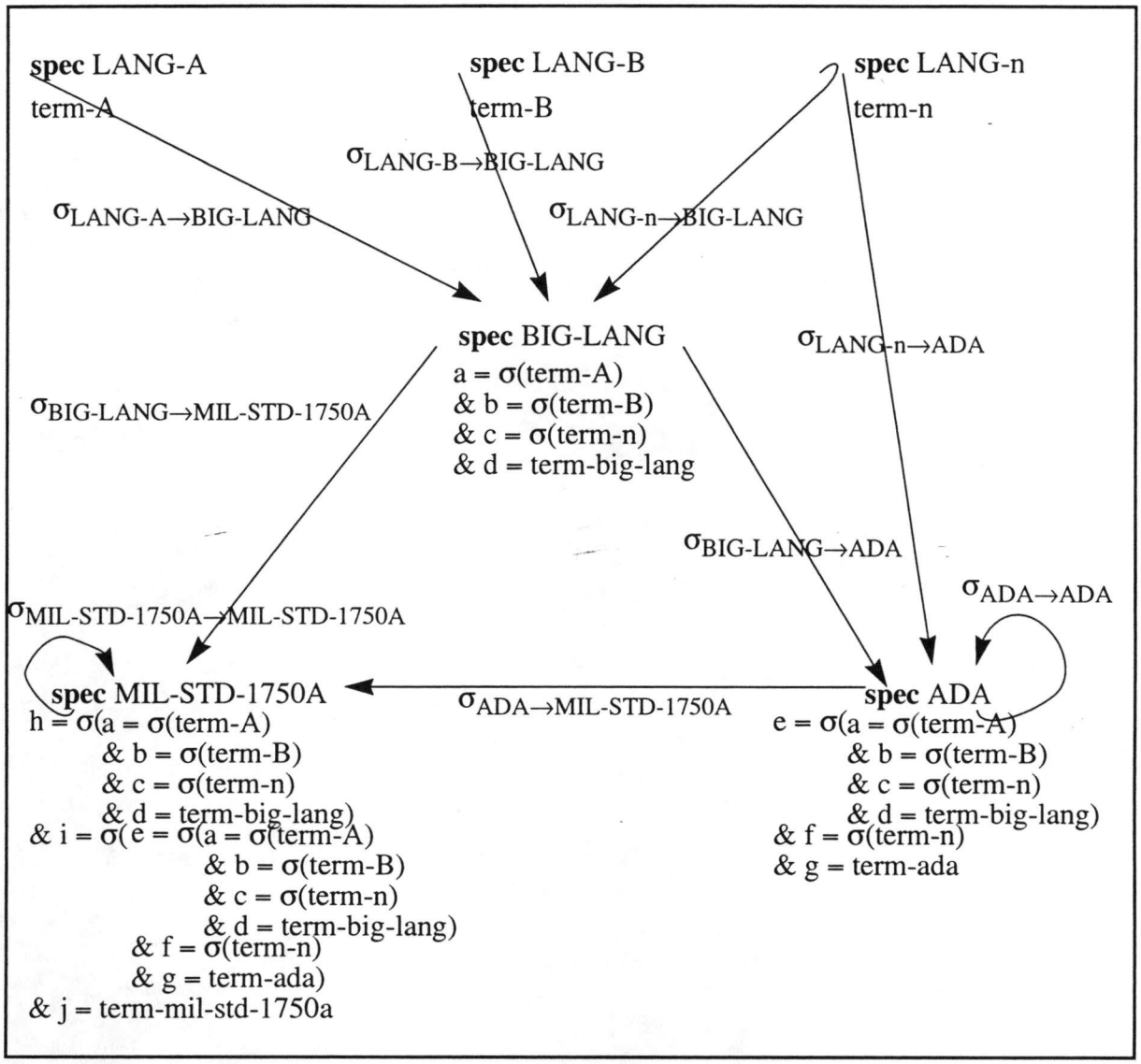
3.3 Diagrams

Assuming that all languages needed for a desired multilingual representation of a system exist in the same institution [Goguen & Burstall] (i.e., the same logical system), we can graphically depict relationships between languages by drawing a diagram in the *category* [Barr & Wells] $\mathbf{Spec}_{\mathbf{I}}$ of specifications and specification morphisms associated with institution I.[1]

The figure below shows an enhanced diagram in $\mathbf{Spec}_{\mathbf{I}}$. A true diagram would have only specifications and specification morphisms. The diagram shows five languages, **spec** LANG-A, **spec** LANG-B, **spec** LANG-n, **spec** BIG-LANG, **spec** ADA, and **spec** MIL-STD-1750A; nine specification morphisms between the languages; and a term or formula in each language. Terms of each of the source languages are translated into terms of the target languages via the specification morphisms. Notice that $\sigma_{\text{LANG-n}\rightarrow\text{BIG-LANG}}$ is simply an inclusion specification morphism that maps sorts and operations in **spec** LANG-n to themselves in **spec** BIG-LANG.

1. In general, we doubt that the set of all languages that will be needed to represent a complex system will belong to the same institution. Therefore more powerful means of relating languages will need to be used, such as institution morphisms [Goguen & Burstall].

Figure 3.3 An Enhanced Diagram in the Category $Spec_I$



The diagram shows translated terms conjuncted together with new terms of the target language via variables and equations. We believe that translated terms could also be assembled into a *single new term* in the target language, but we have not discovered the general mechanism by which this could occur. An operation to accomplish this would work like an algorithm that takes a number of program fragments and assembles them into a complete program.

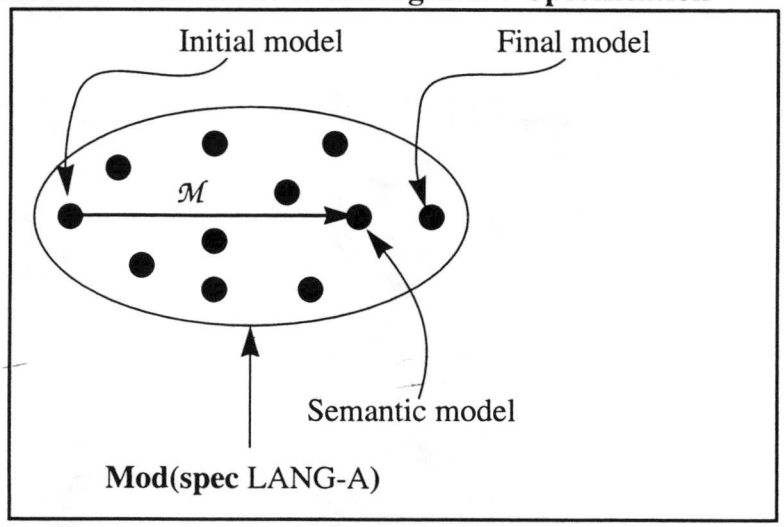
3.4 Semantics

A model of an algebraic specification is used to denote its meaning. Algebraic specifications give us great freedom in dealing with the models. Every specification **spec** LANG-A generates a category of models and homomorphisms between models, denoted **Mod(spec** LANG-A). In this category, an *initial* model is a model from which a unique homomorphism to every other model can be constructed. A *final* model is a model to which a unique homomorphism can be constructed from every other model.

The denotational semantics of an algebraic specification is defined by an initial model which

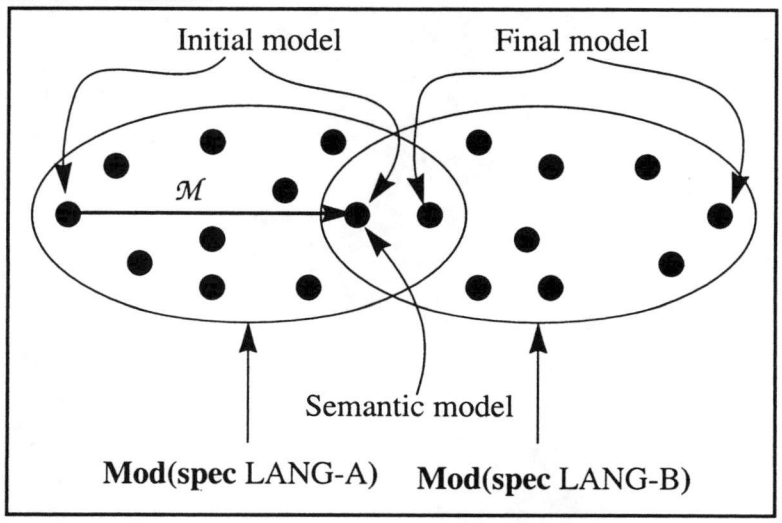
defines the sets of equivalent terms (i.e., the syntax), a model of semantics and the unique homomorphism $\mathcal{M}$ (the "meaning" function) from the initial model to the semantics model. These relationships are summarized in the figure below. Black dots represent models. An arrow between black dots represents a homomorphism. Note that non-initial models can have additional sets that are not carriers of any sort and additional functions that don't correspond to an operation symbol in the algebraic specification.

Figure 3.4 Denotational Semantics of an Algebraic Specification



Consider a **spec** LANG-B which has some models in common with **spec** LANG-A. Imagine that the semantics model of **spec** LANG-A happens to be an initial model of **spec** LANG-B as shown in the figure below.

Figure 3.5 Models in Common



The meaning function $\mathcal{M}$ may induce a specification morphism from **spec** LANG-A to **spec** LANG-B. Thus given a language **spec** LANG-A and a desired semantic model, from the semantic model and $\mathcal{M}$ we may be able to reverse engineer a language **spec** LANG-B that has the semantic model as an initial model. This process enables us to deal with semantics at the level of specifications, rather than the level of models.

Given our use of algebraic specifications to define languages, we can now think of semiformal languages as algebraic specifications which have no semantic axioms. Because of this, it will be easy to define the semantics of a semiformal language by constructing a

specification morphism from the semiformal language to the formal language used to represent the semantics of the (previously) semiformal language. We will look at this again when we consider [Semmens & Allen].

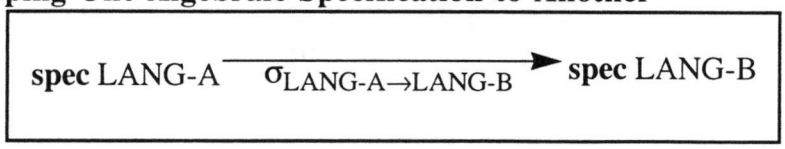
3.5 Using Languages Together

To prepare for our survey of approaches to multilingual system representation, we will examine the various ways in which just two languages can be used together to describe a system. In section 4, we will generalize these concepts to handle more than two languages.

3.5.1 Translation

Consider the diagram below in which $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$ is a specification morphism.

Figure 3.6 Mapping One Algebraic Specification to Another



This diagram can represent several approaches to multilingual system representation. We will see that within the SLAS framework, multilingual system representation is the norm.

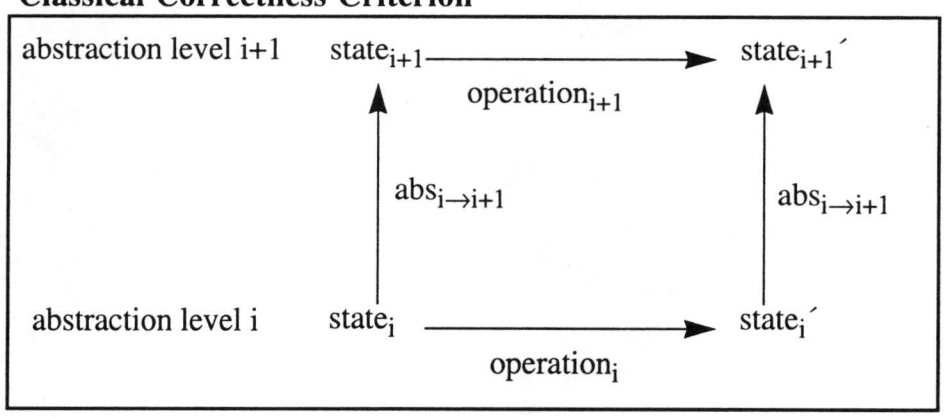
- **spec** LANG-A could be a programming language and **spec** LANG-B could be a machine language. In this case, $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$ represents the specification of a compiler.
- If **spec** LANG-B is another programming language, then $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$ would represent the specification of a translator. Recall that in section 3.2 we demonstrated the use of a specification morphism to translate a term in **spec** FA to a term in **spec** GUARDED.

3.5.1.1 Classical Formal Refinement/Verification

The SLAS framework can represent the *monolingual* approach to system representation. The monolingual approach is the one typically used in small-scale applications of formal methods. In the monolingual approach, at each abstraction "level" of the system, exactly one formal language is used to represent the system.

In the monolingual approach, the essence of relating levels of abstraction is by constructing a homomorphism from the representation algebra at a given abstraction level to the representation algebra at the next abstraction level above. This is summarized by the diagram below.

Figure 3.7 Classical Correctness Criterion



It must be proven between every pair of adjacent abstraction levels that

$$abs_{i\rightarrow i+1}(operation_i(state_i)) = operation_{i+1}(abs_{i\rightarrow i+1}(state_i))$$

for all types that represent state information at level i and for all operations on these types at level i. $abs_{i\rightarrow i+1}$ maps an equivalence class of states at level i to a state at level i+1. $abs_{i\rightarrow i+1}$ is called an *abstraction function* for the type of $state_i$.

The SLAS framework represents classical formal verification and formal refinement by constructing a specification morphism between the language used to represent abstraction level i+1 and the language used to represent abstraction level i. In which direction the specification morphism goes depends on the languages.

Because **spec** LANG-i+1 can abstract **spec** LANG-i, this implies that **spec** LANG-i is a "bigger" language than **spec** LANG-i+1 (in the sense of the number of sorts in each specification). If we can construct $\sigma_{LANG\text{-}i\rightarrow LANG\text{-}i+1}$, this induces an abstraction function in the ordinary sense. If we can construct an injective $\sigma_{LANG\text{-}i+1\rightarrow LANG\text{-}i}$, then we believe that its inverse, $\sigma_{LANG\text{-}i+1\rightarrow LANG\text{-}i}^{-1}$, can be used to induce a different kind of abstraction function, namely one that may not abstract every object in **spec** LANG-i. Using $\sigma_{LANG\text{-}i+1\rightarrow LANG\text{-}i}^{-1}$ in this way may produce only term *fragments* in **spec** LANG-i+1. This is an area requiring further research.

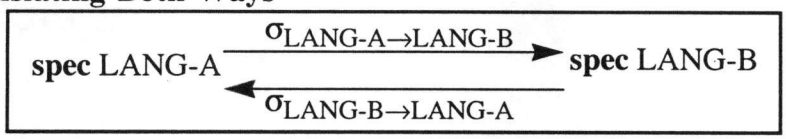
By representing refinement/verification in this way, the SLAS framework enforces that all decisions about relating representations at different abstraction levels are made at the time the specification morphism is defined. For example, this could mean that *sets* in **spec** LANG-i+1 are always implemented by *sequences* in **spec** LANG-i (i.e., the abstraction function between the two languages is fixed). SLAS does not permit the developer to construct abstraction functions in an ad hoc manner. If a different abstraction function is desired, the specification morphism must be modified, which may not be practical or possible.

Monolingual development considers the details of representations at each abstraction level on a case-by-case basis, but SLAS refinement is a syntactic translation of text. Refinements in SLAS are correct by definition. Some may argue that one of the advantages of classical refinement is that it provides the developer with complete freedom to choose the most appropriate representation at each abstraction level. While this is true, by the same argument we would conclude that *programs should be compiled by hand* so that the most appropriate machine language representation can be constructed for each algorithm in the program. Thus we must be willing to trade representational freedom for automation.

3.5.1.2 Other Ways to Translate

Imagine that we can construct specification morphisms in both directions, as shown in the diagram below.

Figure 3.8 Translating Both Ways



This may suggest that **spec** LANG-A and **spec** LANG-B are isomorphic. Translations between isomorphic languages are useful when we need both graphical and textual representations of a language. This is the case with Specification and Description Language (SDL) [Olsen et al.].

3.5.2 Views

We can think of "views" is two ways; as translations or as abstractions. As translations, we think of viewing **spec** LANG-A terms and formulas as **spec** LANG-B terms and formulas by

translating the **spec** LANG-A terms and formulas into **spec** LANG-B terms and formulas via a specification morphism $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$. If $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$ is injective, then we might be able to use its inverse $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}^{-1}$ to untranslate (another way of viewing) **spec** LANG-B terms and formulas into **spec** LANG-A terms and formulas. As abstractions, we think of viewing a "concrete" algebra in an "abstract" algebra via an abstraction function.

3.5.3 Combination

When someone states that a language has been "extended" to include new features, what does this mean? It means that the original language and the language of the "extensions" have been combined to produce a new language. Within the SLAS framework, we can precisely represent the ways in which two languages can be combined.

3.5.3.1 Definitional Extension

The simplest way to extend a language is via a definitional (or procedural [Horspool & Levy] extension. A definitional extension is a new language that includes the original language via an inclusion specification morphism and which defines new sorts and operations in a way that preserves all the original sorts and operations. Definitional extensions enable a language to be "bootstrapped" into existence from a small set of primitive sorts and operations.

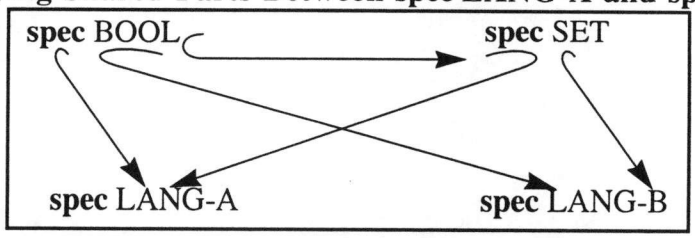
3.5.3.2 Assume No Sharing

The simplest way to combine two languages is to assume that they share no parts (i.e., have nothing in common) and obtain their disjoint union. If we wish to obtain the disjoint union of **spec** LANG-A and **spec** LANG-B, we simply write a new **spec** LANG-C and copy the texts of **spec** LANG-A and **spec** LANG-B into **spec** LANG-C. If a name of a sort or operation is common to both **spec** LANG-A and **spec** LANG-B, then one of the copied sorts or operations must be renamed in **spec** LANG-C. We do this because *we assume that common names are coincidental and unrelated* (we deal with shared sorts and operations later). The resulting language, **spec** LANG-C, has only the sorts, operations, and axioms that are in either **spec** LANG-A or **spec** LANG-B. Thus there is no way to relate the two component languages within **spec** LANG-C. This is not a commonly used method of extending a language.

3.5.3.3 Sharing Considered

A more useful variation of this approach is to identify the shared parts of the two original languages (e.g., sorts BOOL, SET, etc.). Think of this as a way of identifying common concepts or common features of the two languages. Identifying shared components is represented by the diagram below.

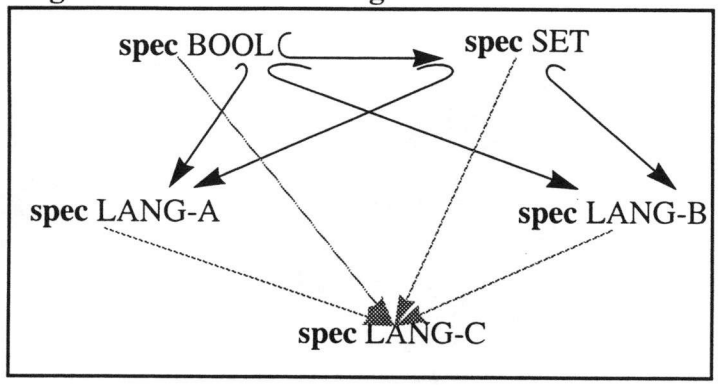
Figure 3.9 Identifying Shared Parts Between spec LANG-A and spec LANG-B



spec LANG-C can be constructed from this diagram in such a way that the parts common to **spec** LANG-A and **spec** LANG-B will be identified as the same in **spec** LANG-C and that **spec** LANG-C contains exactly the sorts, operations, and axioms needed for the component languages (**spec** LANG-A and **spec** LANG-B) to interface correctly (via their shared components). We construct **spec** LANG-C by taking the *colimit* of the diagram above. The colimit produces **spec** LANG-C and a specification morphism to **spec** LANG-C from every

other language in the diagram. The language produced by taking the colimit of a diagram will be referred to as the *colimit language* (CL). **spec** LANG-C is the colimit language for the diagram.

Figure 3.10 Computing the Colimit of a Diagram

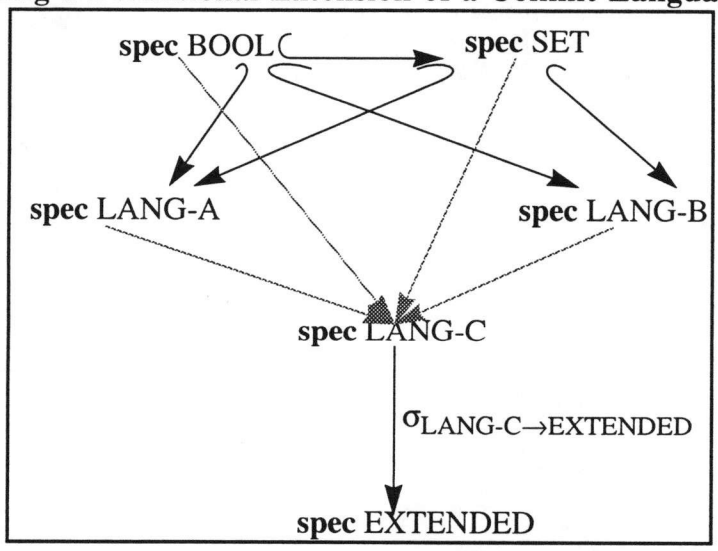


Notice that the only way for the component languages to interface is via their shared parts. Thus in the example above, we would be able to construct terms using subterms from both languages **spec** LANG-A and **spec** LANG-B by building up terms of sort BOOL or sort SET.

3.5.3.4 Sharing with Additional Interfaces

We can add to the colimit language additional sorts, operations, and axioms to make the component languages interface in ways other than just via their shared sorts and operations. The additional sorts, operations, and axioms added to **spec** LANG-C will create a definitional extension to it (**spec** EXTENDED) as shown in the diagram below.

Figure 3.11 Obtaining a Definitional Extension of a Colimit Language



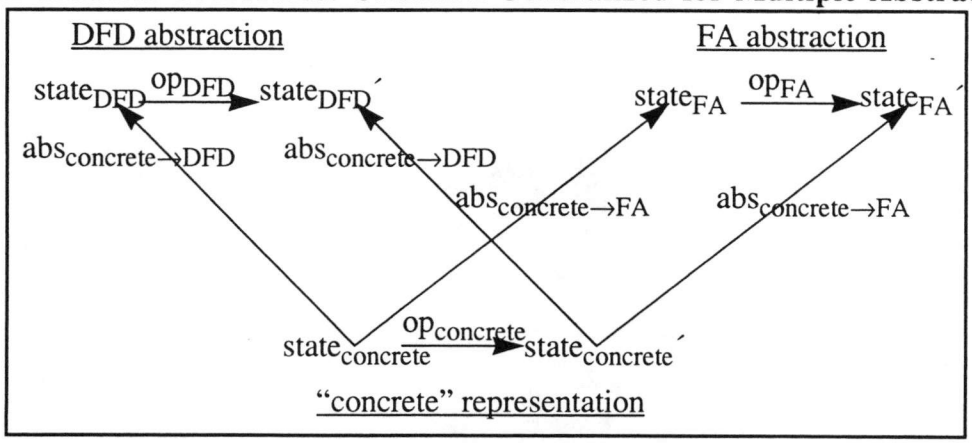
When adding the new sorts, operations, and axioms to make **spec** EXTENDED, we must be careful to ensure that the morphism from **spec** LANG-C to **spec** EXTENDED ($\sigma_{\text{LANG-C}\rightarrow\text{EXTENDED}}$) remains a specification morphism. This approach represents the most general way that a language can be extended by another language.

3.5.4 Combinations Revisited

Another way to think about sharing is to consider two abstraction functions that map equivalence classes of states in a "concrete" algebra to states in two "abstract" algebras. This

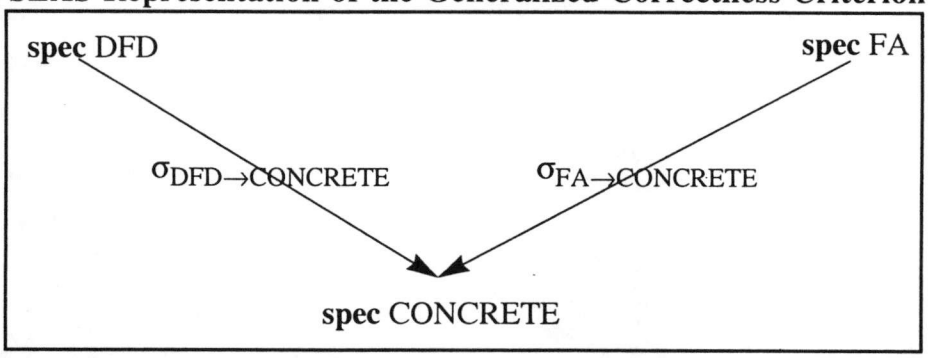
is simply a generalization of the classical correctness criterion mentioned earlier. The diagram below summarizes this concept. A DFD algebra and a FA algebra are the abstract algebras.

Figure 3.12 Classical Correctness Criterion Generalized for Multiple Abstractions



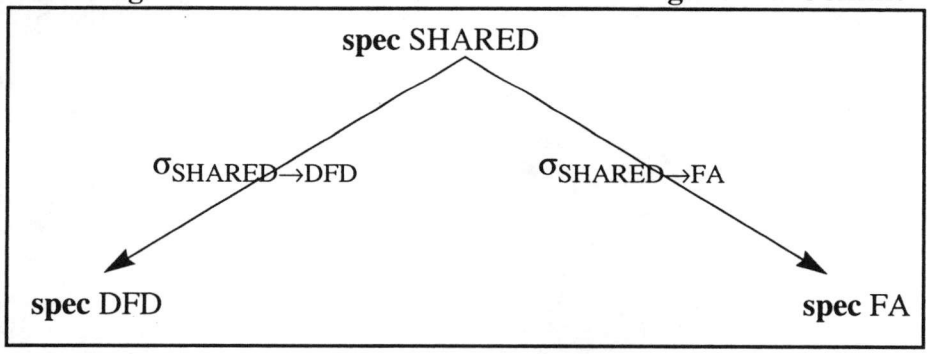
In the SLAS framework, this might be represented by the following diagram.

Figure 3.13 SLAS Representation of the Generalized Correctness Criterion



Assuming that the concrete algebra contains exactly the information for it to be abstracted as both DFDs and FAs, then we could obtain the diagram above by computing the colimit of the diagram below. **spec** CONCRETE would be the colimit language.

Figure 3.14 The Diagram Needed to Obtain the Above Figure via Colimit

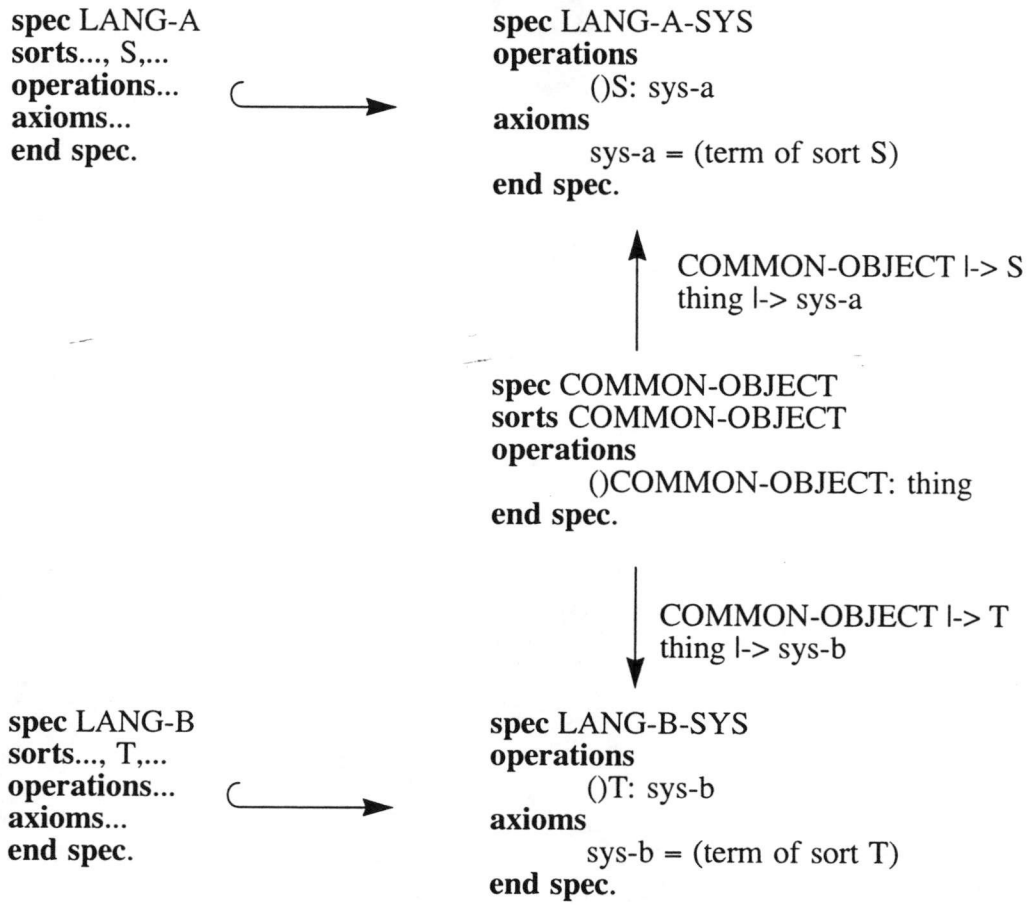


Assuming that the concrete algebra contains more information than what is necessary for it to be abstracted only as DFDs and FAs, then we could obtain **spec** CONCRETE as a definitional extension to the colimit language obtained from the diagram above.

There are several details to abstracting a system in more than one way that we will now discuss. Until now, the SLAS framework has treated terms and formulas as atomic objects. Now we describe how SLAS obtains access to the contents of terms and formulas.

Consider two languages with which we wish to represent a system., **spec** LANG-A and **spec** LANG-B. Construct a definitional extension to **spec** LANG-A (**spec** LANG-A-SYS) that introduces a new constant of the sort that will represent the system in **spec** LANG-A (sort S) and that introduces an axiom that makes this constant equal to a term of sort S. Do likewise for **spec** LANG-B. We will use another algebraic specification to identify the constants in each definitional extension. This situation is summarized by the diagram below.

Figure 3.15 Identifying Shared Constructed Objects



Computing the colimit of this diagram identifies sys-a with sys-b. This is how we expect to identify that the abstractions are about the same object (the system). Note that name of the common object need not be global, because we can use renaming via the specification morphisms from **spec** COMMON-OBJECT to the definitional extensions.

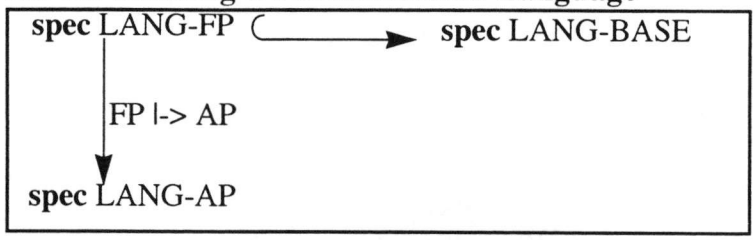
Notice that now a string in a language can be represented by a term, formula, sentence, or a definitional extension to an algebraic specification. We believe that definitional extensions can also be used to assign a name to a term (for when more than one term in an algebraic specification are being considered at the same time) and to replace common subexpressions with constants (define the constant to equal the common subexpression). We may also use definitional extensions to determine if a number of strings in a language are collectively satisfiable.

3.5.5 Parameterization

The last approach to using two languages together that we will discuss is to use one language as an actual parameter to another language. In the diagram below, **spec** LANG-FP represents

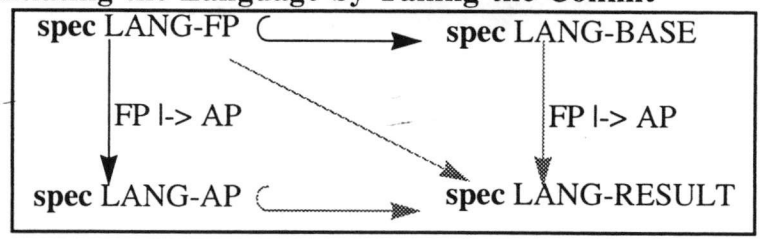
the formal parameter language of **spec** LANG-BASE, **spec** LANG-AP is the actual parameter language, and **spec** LANG-RESULT is the instantiated language. The parameterized language can be defined by a diagram such as the one below.

Figure 3.16 Diagram for Defining a Parameterized Language



Taking the colimit of the diagram above produces the instantiated language as the colimit language. This is shown in the diagram below.

Figure 3.17 Instantiating the Language by Taking the Colimit



Parameterized languages have not been investigated in a general framework such as this before (but parameterized algebraic specifications have been thoroughly investigated in this framework, of course [Wirsing]). The work coming closest to parameterized languages has been performed under the rubric of "heterogeneous refinement" or "Multimodeling" [Fishwick & Ziegler].

Parameterized languages offer a number of interesting possibilities. For example, we can consider more than one of the sorts of **spec** LANG-BASE to be a formal parameter sort (we do not follow the convention of identifying specific sorts as formal parameter sorts, as is done in some research on parameterized algebraic specifications). Recall **spec** FA of finite automata. Consider sort Q and sort ARROW-LABEL to be formal parameter sorts. We could instantiate sort Q with

- a simple procedural programming language for when we want states to represent loci of computation,
- a language of block diagrams containing derivatives and integrals to solve simultaneous equations in dynamics (this is the language presented in [Fishwick & Ziegler]), or
- a DFD language, etc.

Simultaneously, we could instantiate sort ARROW-LABEL with

- a language based on first-order predicate logic for when we want state transitions to represent global conditions, or
- the sort GC (guarded command) of **spec** GUARDED for when we want transitions to represent a logical condition followed by a process.

Parameterized languages allow specifiers to customize base languages to best suit the needs of the system being represented and the developer's work organization. Parameterized languages also allow designers of development methodologies, e.g., OMT, to define the methodology down to the level of the methodology's base languages, and invite users of the methodology to instantiate the base languages with the actual parameter languages that the developers find

the most suitable.

3.6 Paradigms

We all agree that several examples of programming paradigms are

- Imperative; exemplified by FORTRAN, Pascal, ALGOL60, and C,
- Functional; exemplified by LISP and ML,
- Relational, exemplified by SQL, and
- Object-Oriented, exemplified by Smalltalk and C++.

What common characteristics do languages of the same paradigm possess? Before we answer this, we should examine different notions of what a paradigm is.

3.6.1 A Paradigm as a Model of Computation

We might like to think of a paradigm as a model of computation. Assume that a "model of computation" is any computable algebra. Because every computable algebra can be specified using axioms in conditional logic with hidden functions [Wirsing], any language defined using this logical system (or a subsumed logical system) can serve as a model of computation. Given this, we can conclude that all models of computation can be represented in the SLAS framework by algebraic specifications using a logical system no more expressive than conditional logic with hidden functions. Thus in this notion, a paradigm is defined by a set of logical systems.

3.6.2 A Paradigm as any Language

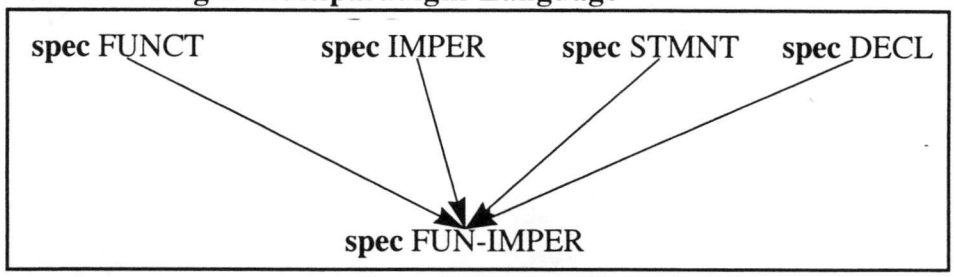
However, we may not want to limit the notion of a paradigm to a model of computation. We hypothesize that a paradigm can be represented by any language (an algebraic specification). This "paradigm language" possesses only those sorts, operations, and axioms needed to define the paradigm in its most primitive form. A paradigm language need not represent a complete programming language definition.

Because paradigm languages define only the paradigm and none of the associated constructs needed for a programming language, we expect that the algebraic specification of a programming language would need to include at least one paradigm language along with simple languages defining input/output, statements, and declarations. For this paper, we will use the notion of a paradigm as any language.

3.6.3 Multiparadigm (a.k.a. Wide-Spectrum) Languages

A *multiparadigm* language imports more than one paradigm language. The diagram below provides intuition for these concepts. **spec** FUNC and **spec** IMPER are the functional and imperative paradigm languages, respectively, **spec** STMNT and **spec** DECL are languages defining statements and declarations, respectively. These four languages are mapped via specification morphisms to **spec** FUN-IMPER, a multiparadigm programming language.

Figure 3.18 Constructing a Multiparadigm Language



After **spec** FUN-IMPER has been constructed, the developer deals only with (i.e., writes terms only for) **spec** FUN-IMPER when writing a system representation. Examples of multiparadigm languages include REFINE [Kotik & Markosian], RAISE [RAISE], and COLD [Feijs et al.].

3.7 Common Base Languages

We define a *Common Base Language* as a language to which a unique specification morphism can be drawn from every other language in the diagram. The main reason for developing a common base language is to construct a semantically broad language with a number of useful properties (theorems). These properties can enforce a certain style in which representations are constructed. common base languages are often too complicated to be used directly by the developer (i.e., we do not expect a developer to construct terms of the common base language directly).

Think of a common base language as defining a semantic model able to represent the semantics of languages with different paradigms. A language developer might use the classical denotational semantics model of states and environments to define the semantics of programming languages. The semantic model of states and environments is especially useful for defining the semantics of imperative languages. In the same way, we can use a common base language to define the semantics of logical, functional, and imperative languages simultaneously.

To use a common base language, a language developer must redefine the semantics of each language of interest in the common base language. Obviously this can be a significant undertaking (in the SLAS framework it means that a specification morphism must be constructed from every language of interest to the common base language).

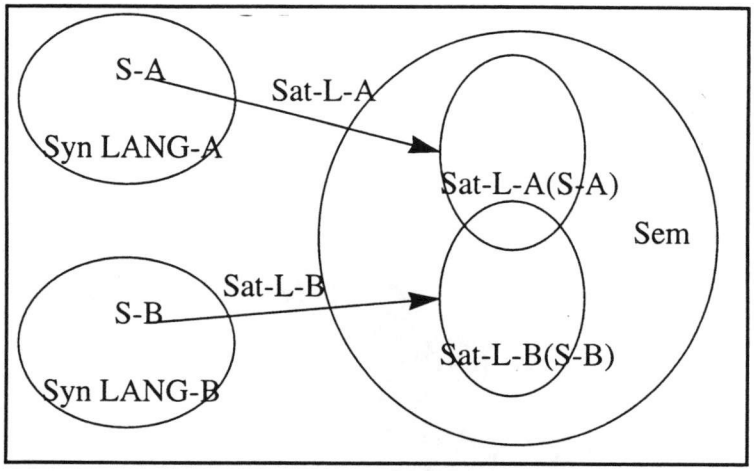
3.8 Other Frameworks

Other frameworks for describing multilingual system representation phenomena exist. This section reviews two of them and shows how the SLAS framework represents each.

3.8.1 Wing's Framework

In [Wing] a framework is presented within with to describe several concepts in multilingual system representation. In Wing's framework a specification language is characterized by a triple (Syn, Sem, Sat), where Syn is the language's syntactic domain (i.e., the set of all strings in the language), Sem is the language's semantic domain (i.e., the set of all behaviors that can be given to a string in the language), and Sat is the satisfaction relation between individual strings in the language and a set of behaviors for an individual string. The figure below summarizes the relationships. Syn LANG-A and Syn LANG-B are the syntactic domains of LANG-A and LANG-B respectively; S-A and S-B are specifications in Syn LANG-A and Syn LANG-B respectively; Sem is the common semantic domain; Sat-L-A and Sat-L-B are the satisfaction relations for LANG-A and LANG-B respectively; and Sat-L-A(S-A) and Sat-L-B(S-B) are the sets of behaviors of specifications S-A and S-B respectively.

Figure 3.19 Wing's Framework



Wing's framework assumes that all specification languages of interest share a common semantic domain. From this framework we infer a criterion for using specifications written in two different languages. It is that the intersection of the sets of behaviors for the two specifications be nonempty. We can write

$$\text{S-A comp S-B} = \text{Sat-L-A(S-A)} \cap \text{Sat-L-B(S-B)}$$

where "comp" is *specification composition* in [Zave & Jackson] and also *parallel specification composition* in [Abadi & Lamport].

Within the SLAS framework, we can represent Wing's framework by considering the set of models of a language. The models of the union of a set of languages is equal to the intersection of models of each language individually (from [Srinivas]):

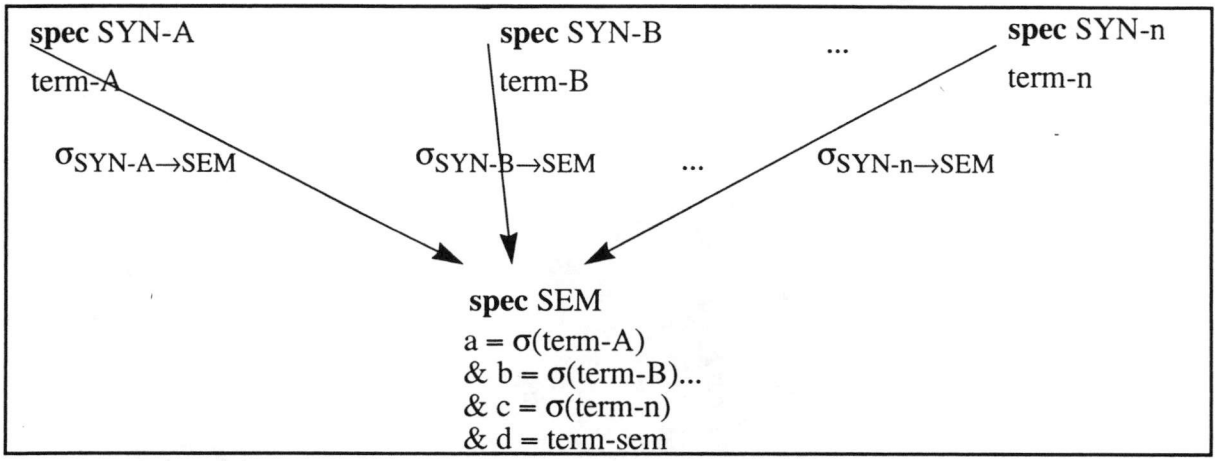
$$|\textbf{Mod}(\bigcup_{i \in \text{LANG-SET}} \textbf{spec LANG-i})| = \bigcap_{i \in \text{LANG-SET}} |\textbf{Mod}(\textbf{spec LANG-i})|$$

where LANG-SET is the collection of all languages of interest to us, **Mod**(_) is a function that returns the category of models and model homomorphism of a specification, and |_| returns the objects of a category.

This equation requires that the collection of *languages* be satisfiable, while the inferred criterion in Wing's framework requires that a collection of *specifications* be satisfiable. We can restrict our equation to be equivalent with Wing's framework's criterion by considering a term (to Wing a specification) in each of the languages. The collection of languages may have no models in common but there might exist *common submodels picked out by each of the terms* (because each term need not use all the sorts and operations of its algebraic specification).

If we want to represent Wing's framework at the level of languages, we can construct a diagram in the category $\textbf{Spec}_{\textbf{I}}$ as follows. Wing's framework's criterion requires that the formula of **spec** SEM be satisfiable.

Figure 3.20 Wing's Framework as a Diagram in Category $Spec_I$



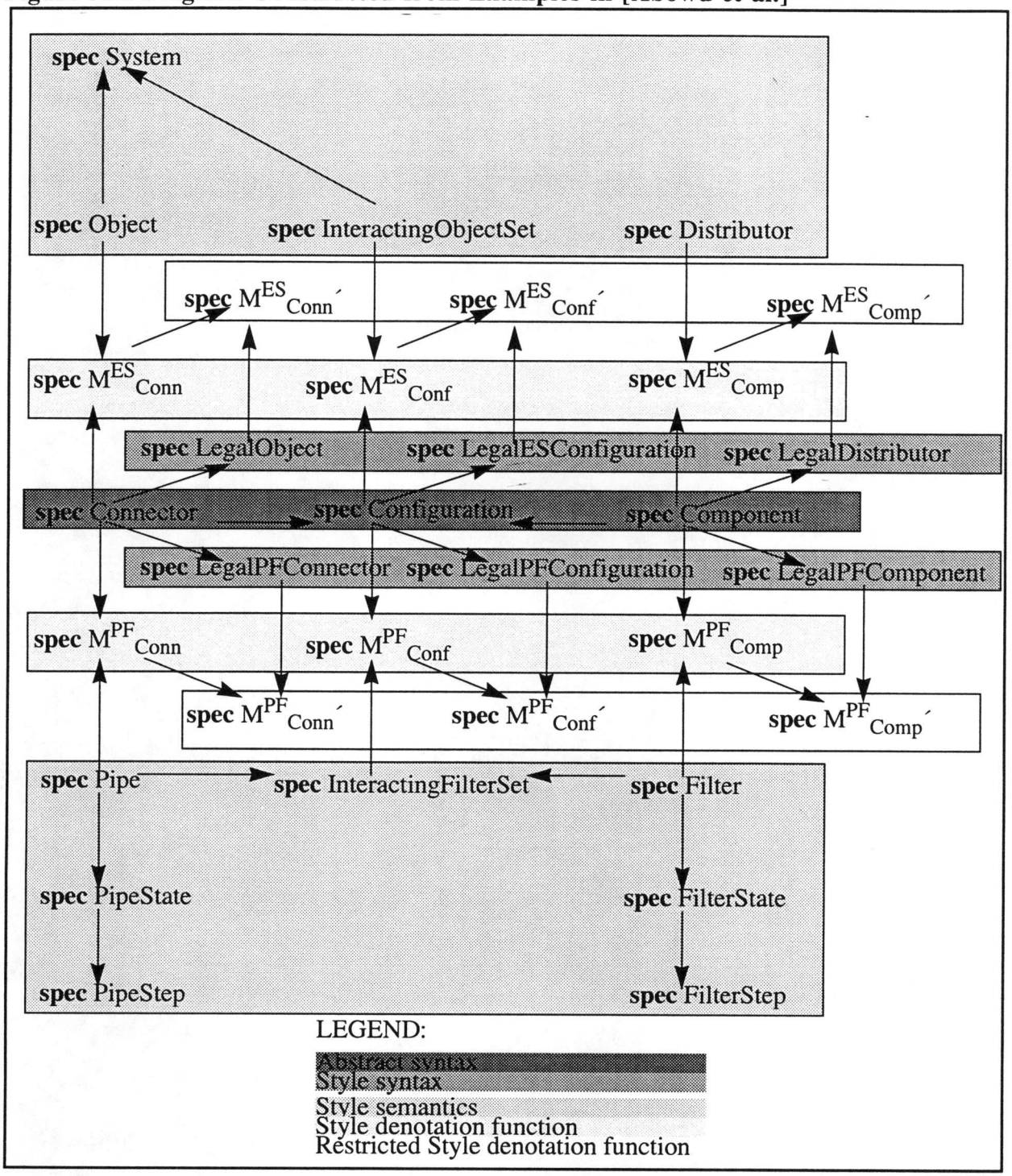
3.8.2 Formalized Software Architectures

Formalized software architectures (FSA) [Abowd et al.] applies the Z specification language [Wordsworth] to the definition of languages for software architectures. These languages are called "architectural styles". Z schemas are constructed which define an architectural language's syntax, semantics, and denotation function. In other words, Z is a *metalanguage* used to define architectural languages. We have not yet chosen a specific algebraic specification language to be the metalanguage of the SLAS framework.

Formalized software architectures is the most similar framework to the SLAS framework that we have discovered. In effect, Z schemas define languages and these languages are put together using the Z schema structuring mechanisms.

In the next figure we have drawn a diagram that we believe represents the structuring of Z schemas and global axioms used in the two main examples of [Abowd et al.]. Nodes represent schemas (or in the case of the denotation functions, global axioms), arrows represent inclusion by a definitional extension of the source specification or else inclusion of the source specification in the target specification by virtue of the fact the source specification is used in the definition of the target specification's operations.

Figure 3.21 Diagram Constructed from Examples in [Abowd et al.]



Representing the main examples of [Abowd et al.] in the SLAS framework elucidates a number of details of the FSA framework. Much of the detailed structure of FSAs is hidden by the fact that Z makes use of global names. Because nothing is global in the SLAS framework, all information must be conveyed by specification morphisms.

The FSA's strict adherence to the denotational style of semantics tends to complicate matters. Because FSA does not use powerful structuring mechanisms such as specification morphisms, FSA relies on denotation functions (in effect a third specification) to relate a specification of

syntax to a specification of semantics. The SLAS framework does not require denotation function specifications because denotation functions are encapsulated in specification morphisms.

4 Approaches to Multilingual System Representation

In this section we review a number of approaches to multilingual system representation that have appeared in the research literature. In section 3.5, we examined the different ways in which two languages can be used together. In this section we will generalize these ways to use more than two languages together.

4.1 Key Characteristics of Approaches

Within the SLAS framework, the different approaches to multilingual system representation can be distinguished by answering the following questions (and sub questions):

What is the diagram in the category $\mathbf{Spec}_{\mathbf{I}}$?

> What are the languages?
>
> What languages are related to what languages (i.e., where are the specification morphisms)?
>
> What is done with terms, formulas and sentences of the languages?

What are the details of the specification morphisms (inclusion, injective, surjective, etc.)?

4.2 Translation-Based Approaches

4.2.1 Horspool & Levy

In [Horspool & Levy], a *translation-based* approach to multilingual representation is described. In this approach, two languages, LANG-A and LANG-B, and a specification morphism from LANG-A to LANG-B is used. The translator must be able to convert LANG-A code into "functionally equivalent" LANG-B code, thus the translator must be a specification morphism.

[Horspool & Levy] list a number of observations regarding this approach. Our representation of these observations in the SLAS framework follows.

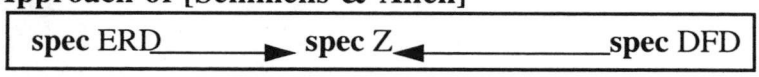
- "User-supplied LANG-B code can directly access objects and operations of translated LANG-A code." This means that $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$(term-a) is combined with terms of **spec** LANG-B in such a way that the internal details of $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$(term-a) are accessible by terms of **spec** LANG-B. This is enabled because names used in term-a are compatibly translated to recognizable names in **spec** LANG-B.
- "If LANG-B supports abstract data types (ADTs) and type encapsulation, translated LANG-A objects and operations can be safely mixed with user-supplied LANG-B code." We believe that this means that the sort of $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$(term-a) must be the sort of **spec** LANG-B ADTs or encapsulated types.
- "If LANG-B supports operator overloading and/or syntactic extensibility, it may be possible to provide access to facilities of the LANG-A language from a LANG-B program using syntax that is similar to the syntax of the LANG-A language." We believe this can be achieved by an interpretation specification morphism that maps a **spec** LANG-A operation (to which access from within LANG-B is desired) to the same operation name in **spec** LANG-B and which defines the new **spec** LANG-B operation in terms of **spec** LANG-B sorts and operations as well as translated **spec** LANG-A sorts and operations.

- "Access to facilities of the LANG-B language from within LANG-A language code can be provided by external interface mechanisms built into the LANG-A-to-LANG-B translator." We believe this means that LANG-A can be extended by an operation corresponding to a LANG-B operation, such as extending PROLOG with (C++) stream operations _<<_ and _>>_. Notice that the new operations in **spec** LANG-A need not have any axioms defining their semantics, because the new operations may only have meaning in **spec** LANG-B.
- "If LANG-B is a systems implementation language (such as C), there is a reasonable expectation that the translated LANG-A code will execute efficiently." Other than by restricting a language's logical system to be no more expressive than conditional logic with hidden functions, we do not yet know how to represent execution efficiency within the SLAS framework.
- "An application programmed as a mix of LANG-A modules and LANG-B modules is as portable as an application programmed entirely in LANG-B." We believe this assumes all terms of LANG-A module sort are translated into terms of LANG-B module sort and that the specification morphism from **spec** LANG-B to **spec** HARDWARE can be composed with $\sigma_{\text{LANG-A}\rightarrow\text{LANG-B}}$.

4.2.2 Semmens & Allen

In [Semmens & Allen], a technique for translating Entity-Relation Diagrams (ERDs) and DFDs into Z is presented. This effectively provides formal semantics for the languages of ERDs and DFDs and gives development organizations familiar with ERDs and DFDs a gentler transition into applying formal methods. This approach is represented by a diagram featuring **spec** ERD, the algebraic specification of the language of ERDs; **spec** DFD, the algebraic specification of the language of DFDs; and **spec** Z, the algebraic specification of the Z language. Specification morphisms are constructed from **spec** ERD to **spec** Z and from **spec** DFD to **spec** Z.

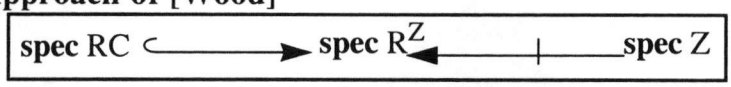
Figure 4.1 The Approach of [Semmens & Allen]



4.3 Combinational Approaches

An example of combining languages together to produce a new language can be found in [Wood]. There Z and the Refinement Calculus [Morgan] are combined into a new language R^Z. The idea behind the combination of languages is to produce a new language with the strengths of both. Within the SLAS framework, we can represent this particular approach by constructing a definitional extension to the algebraic specification of the Refinement Calculus (**spec** RC) and constructing a *partial* specification morphism from the algebraic specification of Z (**spec** Z) to the extended language, as shown in the diagram below.

Figure 4.2 The Approach of [Wood]



Representing this approach has raised the issue of whether partial specification morphisms should be allowed. Because the body of work on algebraic specifications always defines specification morphisms as total functions, partial specification morphisms will need to be closely examined in future research. From a utility standpoint partial specification morphism are attractive because they permit only a portion of a language to be translated. This could be especially useful if we wish to combine a language with only the most relevant portions of

another language.

4.4 Common Base Language Approaches

In this section we discuss two important common base languages. A more comprehensive review of the common base languages can be found in [Meyers].

4.4.1 Abadi & Lamport

[Abadi & Lamport] defines an algebra in which to investigate the composition of specifications and the preservation of safety and liveliness properties in composition. Although the algebra is developed for the purpose of providing the paper's main theorem, the authors suggest that the algebra should be used to define the semantics of a variety of languages. They argue that placing languages on a common semantic representation makes it possible to analyze and compose different languages together. Theorems of the algebra then become proof rules for a language whose semantics is defined in the algebra. The algebra treats a specification as a behavior set (BS), which is akin to a set of paths through a state space.

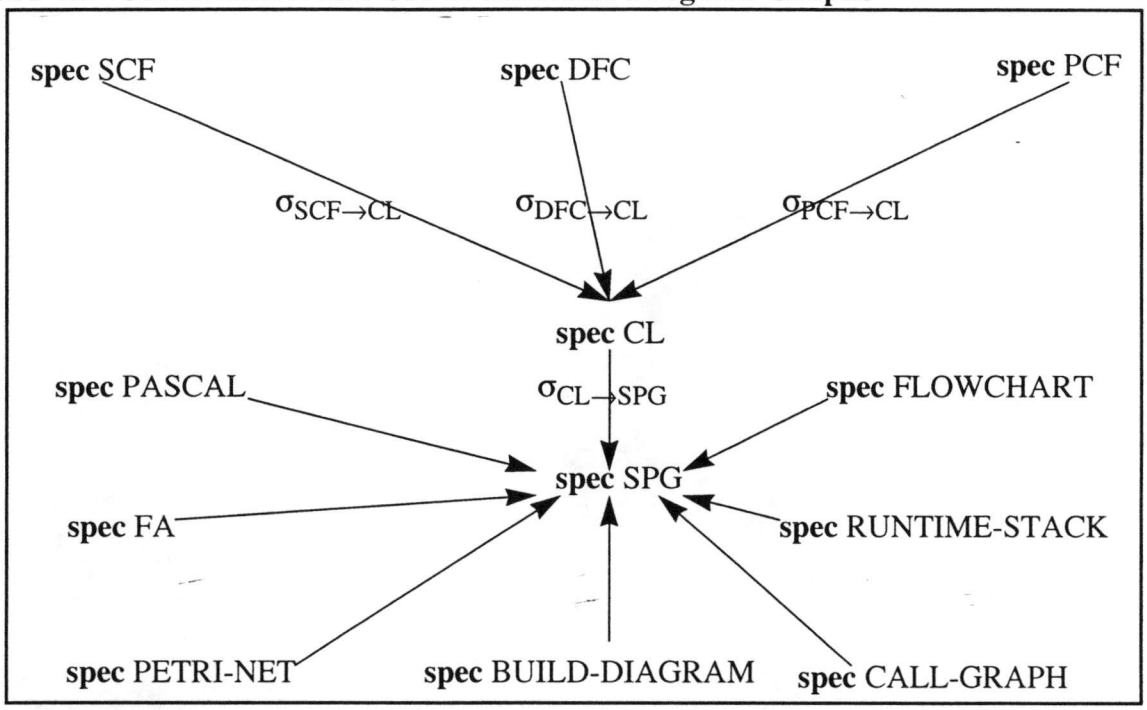
To use this common base language, we first need to construct **spec** BS for the language of behavior sets. Then, for every source language of interest we must construct a (hopefully injective) specification morphism from the algebraic specification of the source language to **spec** BS.

4.4.2 Meyers

In [Meyers], the language of Semantic Program Graphs (SPG) is proposed as a common base language that can represent the paradigms of Sequential Control Flow (SCF), Dataflow-based computation (DFC), and Parallel Control Flow (PCF).

Although the definition of SPG in [Meyers] is informal, we believe it can be represented by the diagram below. In the diagram, **spec** CL is a multiparadigm language constructed from **spec** SCF, **spec** DFC, and **spec** PCF. **spec** SPG is a definitional extension of **spec** CL. Smaller languages, **spec** PASCAL, **spec** FA, **spec** PETRI-NET, **spec** RUNTIME-STACK, **spec** FLOWCHART, **spec** CALL-GRAPH, and **spec** BUILD-DIAGRAM are shown with specification morphisms going to **spec** SPG.

Figure 4.3 Construction and Use of Semantic Program Graphs



Because the small languages can both produce terms of **spec** SPG and abstract a **spec** SPG term in their own language, we assume that the specification morphisms to **spec** SPG are injective. Please note that in [Meyers], all translations to and from an SPG are presented in pseudocode.

Within the SLAS framework, we do not yet know how to represent dynamism at the specification level, or if this is even desirable (we currently address dynamism at the level of models of a specification). Thus we do not know how to represent **spec** RUNTIME-STACK.

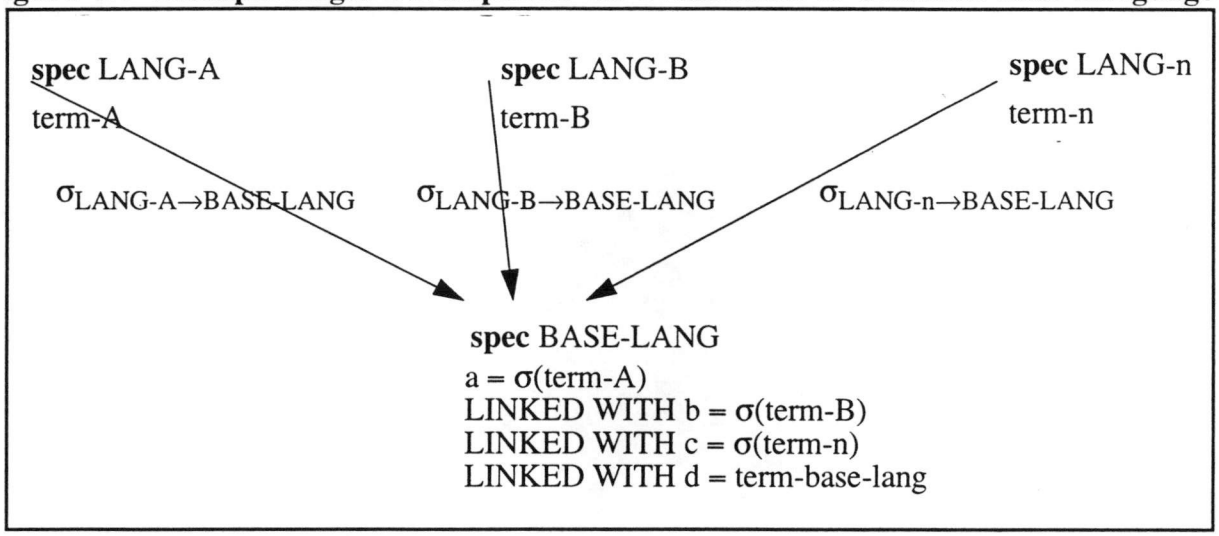
4.5 Multiparadigm Development

In [Meyers] a number of environment integration mechanisms for Multiple View Development Environments (MVDEs) are described. An MVDE uses several interacting viewers (tools) to describe a system. Viewers provide a specific abstraction (i.e., a language) of a system and update each other via different means. Meyers differentiates "multiparadigm development" from "multiple-view development" by saying that multiparadigm development approaches have no requirement for their view languages to interact, i.e., information can flow only from view languages to target languages. In the SLAS framework, we believe that source (view) languages interact by untranslating target language terms via inverse specification morphisms. Simply put, multiparadigm development does not require that the specification morphisms to be injective.

4.5.1 Multiparadigm Development via Translation to Common Language

In this approach, view languages are source languages and terms of each source languages are compiled into a common base language (a common target language). The base language is usually an executable language and the translated terms are modules that are assembled together in a manner appropriate for their level of abstraction (i.e., withing for Ada modules, linking for executable modules). Inconsistencies between terms of the source languages are revealed by runtime errors of the assembled module. This approach is illustrated in the diagram below.

Figure 4.4 Multiparadigm Development via Translation to Common Base Language



4.5.1.1 Zave & Jackson

The approach described in [Zave & Jackson] is an example of this approach. The common base language used is first-order predicate logic with marked sequences to quantify temporal phenomena. Time is represented as a alternating (i.e., marked) sequence of events and intervals. Every predicate in the common base language has either an event or an interval as an argument.

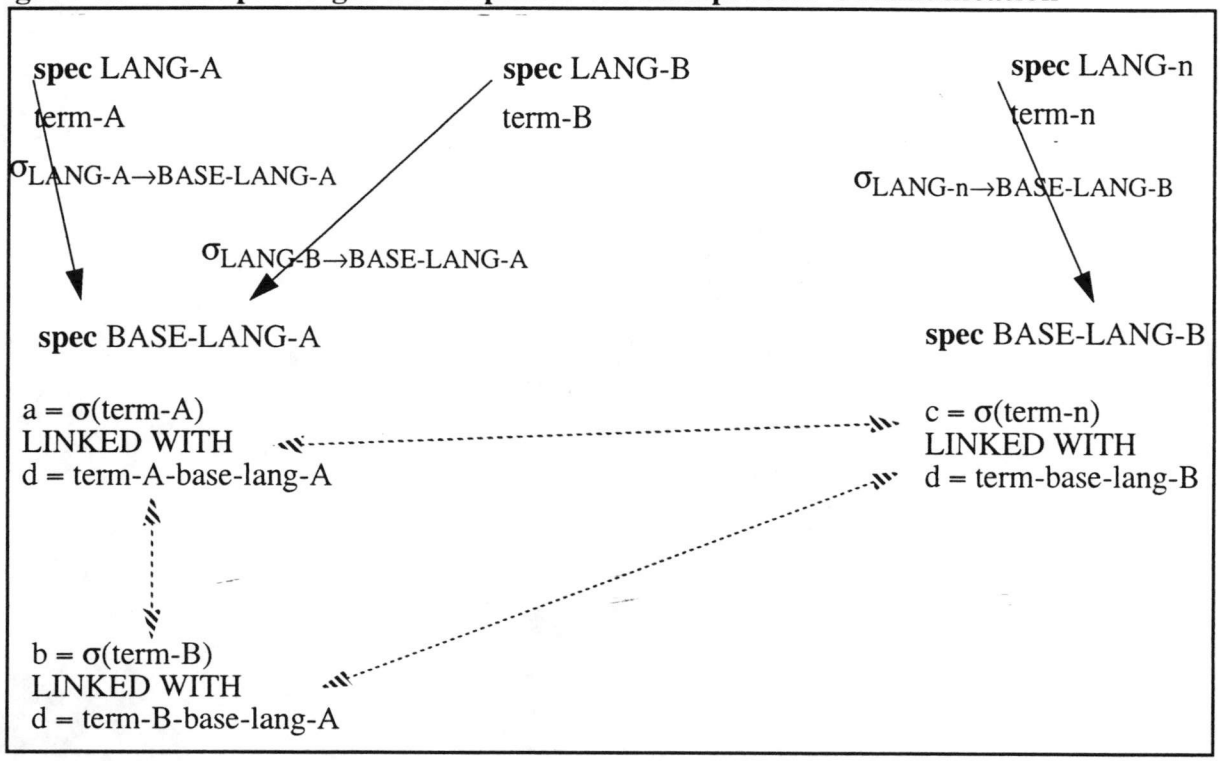
Source languages, such as Z, FA, and Petri-Nets are mapped to the common base language (via specification morphisms). Because almost every sort in a source language is mapped to sort BOOL in the common base language, and because almost every operation in a source language is mapped to a predicate in the common base language, the specification morphisms are unlikely to be injective. Because of this, it is unlikely that terms in the common base language can be abstracted back to a source language, thereby preventing interacting views.

Examples of other systems that can be represented this way are Draco [Neighbors] and Ipser and Wile's environment [Ipser & Wile].

4.5.2 Multiparadigm Development via Interprocess Communication

Another approach to multiparadigm development is to compile each source language term into an separate process and let the processes communicate via message passing, remote procedure calls, sockets, and other interprocess communication mechanisms at run time. This approach can be generalized to distributed computing, thus allowing for more than one executable language. Inconsistencies between terms in the source languages are revealed by runtime failures. This approach is summarized by the diagram below. The dashed lines represent message exchanges between the processes. Please note that it is the models of the base languages (the processes) that pass messages, not formulas. An example system that can be represented this way is Zave's methodology [Zave].

Figure 4.5 Multiparadigm Development via Interprocess Communication



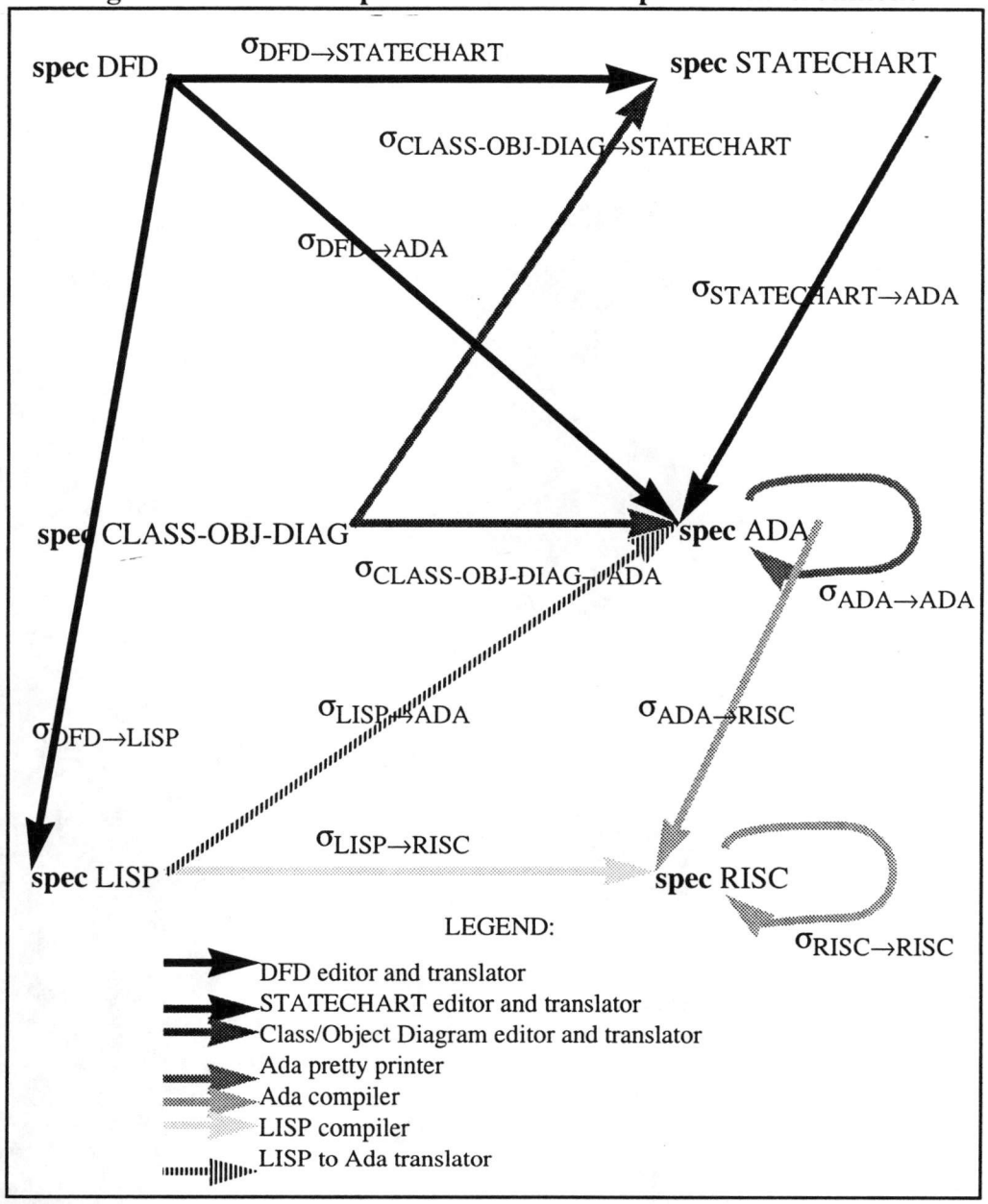
4.6 Multiple-View Development Environment Integration Mechanisms

In the SLAS framework, we represent a MVDE as

- a diagram in the category $\mathbf{Spec}_{\mathbf{I}}$,
- an assignment of tools to sets of arrows in the diagram,
- locations to store and access terms and formulas,
- a mechanism for processing terms and formulas. This includes providing terms and formulas to specification morphisms, assembling new terms and formulas from a set of terms or formulas, moving terms and formulas from one location to another.

The diagram below represents an example MVDE in which DFDs, STATECHARTs, Class/ Object Diagrams, Ada and LISP source code are used to produce RISC assembly implementations.

Figure 4.6 Diagram of the Example Software Development Environment



Tools in the SDE are identified by the shade of their arrow as shown in the legend and are described below.

A DFD editor and translator:

- Enables the creation of a DFD from scratch.
- Via $\sigma_{DFD \rightarrow STATECHART}$, translates a DFD into a STATECHART fragment or abstracts a STATECHART into a DFD.
- Via $\sigma_{DFD \rightarrow ADA}$, translates a DFD into Ada fragments or abstracts Ada code into a DFD.
- Via $\sigma_{DFD \rightarrow LISP}$, translates a DFD into LISP fragments or abstracts LISP code into a DFD.

A STATECHART editor and translator:

- Enables the creation of a STATECHART from scratch.
- Via $\sigma_{STATECHART \rightarrow ADA}$, translates a STATECHART into Ada fragments or abstracts Ada code into a STATECHART.

A Class/Object Diagram editor and translator:

- Enables the creation of a diagram from scratch.
- Via $\sigma_{CLASS\text{-}OBJ\text{-}DIAG \rightarrow STATECHART}$, translates a diagram into a STATECHART fragment or abstracts a STATECHART into a diagram.
- Via $\sigma_{CLASS\text{-}OBJ\text{-}DIAG \rightarrow ADA}$, translates a diagram into an Ada fragment or abstracts Ada code into a diagram.

An Ada pretty printer: Via $\sigma_{ADA \rightarrow ADA}$, prettifies Ada code.

An Ada compiler for RISC:

- Via $\sigma_{ADA \rightarrow RISC}$, compiles Ada into RISC assembler.
- Via $\sigma_{RISC \rightarrow RISC}$, optimizes the RISC assembler.

A LISP compiler for RISC:

- Via $\sigma_{LISP \rightarrow RISC}$, compiles LISP into RISC assembler.
- Via $\sigma_{RISC \rightarrow RISC}$, optimizes the RISC assembler.

A LISP to Ada translator (via $\sigma_{LISP \rightarrow ADA}$).

Where and how terms and formulas are stored and accessed depends upon the specific MVDE integration mechanism used for the environment.

4.6.1 Environment Integration via Shared File System

Under this approach to environment integration, "each tool contains its own internal representation (format) of the data it uses, and the only way that tools communicate is through the file system," [Meyers].

Assume that terms and formulas are stored as files in a file system. The system as a whole is represented by the union of all the terms and formulas in the environment. Imagine that each tool represents terms and formulas in its own tool-specific format. Thus if format(_) is a function that returns the format type of a term or formula, then for example, we can say

$$format(\sigma_{DFD \rightarrow STATECHART}(term\text{-}dfd)) \neq format(term\text{-}statechart)$$

for any term term-dfd generated using the DFD tool and for any term term-statechart generated using the STATECHART tool. This is the case even though $\sigma_{DFD \rightarrow STATECHART}(term\text{-}dfd)$ is a term in **spec** STATECHART. In order for tools to work together, it may be necessary to explicitly convert the format of $\sigma_{DFD \rightarrow STATECHART}(term\text{-}dfd)$ into the format used by the STATECHART tool via some dfd2statechart conversion program.

The Unix operating system and associated tools [Unix] is the classic example of this approach to MVDE integration.

4.6.2 Environment Integration via Message Passing

Under this approach, we start by assuming the same environment structure as in the shared file system approach. Except now, every time a tool performs an action, the tool sends a message to a central message server describing the action. The message server examines the message and determines which other tools need to be informed of the action. When a tool

receives a message from the message server, the receiving tool may use the information in the message to update its representation of the system, which will mean modifying the terms and formulas which it controls.

For example, after a user causes the DFD tool to translate term-dfd into a STATECHART via $\sigma_{DFD \rightarrow STATECHART}$, this tool event would generate a message for the message server. Assuming the message server is aware that the STATECHART tool is interested in such changes, the message server would forward the message to the STATECHART tool. Upon receipt, the STATECHART tool calls the dfd2statechart conversion program on the file containing $\sigma_{DFD \rightarrow STATECHART}$(term-dfd) and integrates this term into its existing set of terms.

A MVDE that uses this approach is PRISMA [Niskier et al.].

4.6.3 Environment Integration via Message Passing: Pairwise Mappings

In this approach, mutual function calls between tools are used instead of message passing. The name given to this approach is something of a misnomer, because every tool need not ever make a function call to every other tool in the environment. This is easy to see when we consider the translation-oriented representation of tools within the SLAS framework.

4.6.4 Environment Integration via Simple Database

Using a simple database as the integration mechanism, primitive versions of terms and formulas for each language are kept in a database, which enforces consistency among the primitive versions of terms and formulas. When a tool needs to work with a term or formula, it retrieves the primitive version from the database and rebuilds it into the full representation format used by the tool. When a tool is finished working on a term or formula, its tool-specific representation details are removed and a primitive version is placed back into the database. There is the possibility for information loss in this process. The amount of semantic detail used in the primitive terms and formulas is dependant on the degree to which the database can enforce consistency between elements of the database.

4.6.5 Environment Integration via a Canonical Representation

The canonical representation approach to MVDE integration attempts to exploit a single representation that all tools in the environment can use. The representation, by being modified, accomplishes the task of informing other tools that a tool has modified a term or formula. Because the representation is canonical, maintaining consistency is no longer an issue. Implementing this approach requires a common base language.

Consider our example diagram for a MVDE. Is that a language in the diagram that could be used as a canonical representation? Ada is almost a suitable choice for a canonical representation in the given environment, because a unique specification morphism can be drawn to **spec** ADA from every other language in the diagram, except **spec** RISC, and because the DFD, STATECHART and Class/Object Diagram tools can abstract Ada code. The reason Ada is not a suitable choice for a canonical representation is because the LISP to Ada translator cannot abstract Ada code. Using a canonical representation assumes that the tool-specific format of terms and formulas are automatically converted when using the specification morphisms.

Examples of MVDEs using a canonical representation are Meyers's environment [Meyers], and Harrold and Malloy's maintenance environment [Harrold & Malloy].

5 Future Directions

Very few of the concepts in the SLAS framework have been validated. Therefore research is needed in many areas. What is called for now is to validate the key concepts of SLAS on a number of examples of increasing complexity. In order to pursue the research in a manner that attacks the highest risks areas first, we propose the following research plan:

1. Construct algebraic specifications of several simple, complete languages. Discover under what circumstances strings should be represented by terms, formulas and sentences, or by definitional extensions. Learn how to work with more than one string at a time. Write more than one string to describe an object and determine the logical consequences. Finish defining **spec** FA and **spec** GUARDED. Dijkstra's language is an especially attractive candidate language because its formal semantics has been defined, and it is frequently used in formal methods, usually as a design language into which formal specifications are refined [Wordsworth] and from which programming language implementations are refined [Morgan] [Gries].

2. With a small set of languages (assumed to use the same logical system), investigate the details of specification morphisms between the languages. Determine the general circumstances under which it is possible to construct specification morphisms, what kinds of specification morphisms are better than others, details of how to use specification morphisms for refinement and abstraction, etc.

3. Represent a popular multilingual development methodology in the framework. We would like to represent Object Modeling Technique (OMT) [Rumbaugh et al.] within SLAS. OMT uses three semiformal languages to represent complex system: Statecharts, Data Flow Diagrams, and Class/Object Diagrams. In order to represent OMT in SLAS, it will be necessary to construct an algebraic specification of each of these languages and use specification morphisms to represent how the methodology makes the languages interface. It may be possible to construct different diagrams with the OMT languages. Each diagram could represent how OMT is applied in a certain application domain. The colimit of the diagram could be the domain specification.

6 Conclusions

We presented a framework within with to describe multilingual system representation phenomena. Because this framework can represent both other frameworks and multilingual approaches, we consider this to be a *unifying* framework. Unifying frameworks are desirable at this point in the discipline of formal methods, because literally hundreds of concepts are being researched to determine which of them are profitable.

7 References

[Abadi & Lamport]

Abadi, M.; Lamport, L. Composing specifications. *ACM Transactions on Programming Languages and Systems*, Jan. 1993, vol.15, (no.1):73-132.

[Abowd et al.]

Abowd, G.; Allen, R.; Garlan, D. Using style to understand descriptions of software architecture. (SIGSOFT'93. First ACM SIGSOFT Symposium on the Foundations of Software Engineering, Los Angeles, CA, USA, 7-10 Dec. 1993). *SIGSOFT Software Engineering Notes*, Dec. 1993, vol.18, (no.5):9-20.

[Barr & Wells]

Barr, Michael. *Category theory for computing science* / Michael Barr, Charles Wells. New York: Prentice Hall, 1990. Series title: Prentice-Hall international series in computer science.

[Belanger]

Belanger, R. MODSIM II-a modular, object-oriented language. IN: *1990 Winter Simulation Conference Proceedings* (Cat. No.90CH2926-4). (1990 Winter Simulation Conference Proceedings (Cat. No.90CH2926-4), New Orleans, LA, USA, 9-12 Dec. 1990). Edited by: Balci, O.; Sadowski, R.P.; Nance, R.E. New York, NY, USA: IEEE, 1990. p. 118-22.

[Booch]

Booch, Grady. *Object-oriented analysis and design with applications* / Grady Booch. 2nd ed. Redwood City, Calif.: Benjamin/Cummings Pub. Co., c1994. Series title: The Benjamin/Cummings series in object-oriented software engineering.

[Broy et al.]

Broy, M.; Wirsing, M.; Pepper, P. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems*, Jan. 1987, vol.9, (no.1):54-99.

[Carrington et al.]

Carrington, D.; Duke, D.; Duke, R.; King, P.; and others. Object-Z: an object-oriented extension to Z. IN: *Formal Description Techniques, II. Proceedings of the IFIP TC/WG 6.1 Second International Conference on Formal Descriptive Techniques for Distributed Systems and Communications Protocols, FORTE'89.* (Formal Description Techniques, II. Proceedings of the IFIP TC/WG 6.1 Second International Conference on Formal Descriptive Techniques for Distributed Systems and Communications Protocols, FORTE'89, Vancouver, BC, Canada, 5-8 Dec. 1989). Edited by: Son T Vuong. Amsterdam, Netherlands: North-Holland, 1990. p. 281-96.

[Davis]

Davis, Alan M. (Alan Michael), 1949- *Software requirements: analysis and specification* / Alan M. Davis. Englewood Cliffs, N.J.: Prentice Hall, c1990.

[Dijkstra]

Dijkstra, Edsger Wybe. *A discipline of programming* / Edsger W. Dijkstra. Englewood Cliffs, N.J.: Prentice-Hall, c1976.

[Feijs et al.]

Feijs, L. M. G. (Loe M. G.) *Notations for software design* / Loe M.G. Feijs, Hans B.M. Jonkers, and Cornelis A. Middelburg. London; New York: Springer-Verlag, c1994. Series title: Formal approaches to computing and information technology.

[Fishwick & Zeigler]

Fishwick, P.A.; Zeigler, B.P. A multimodel methodology for qualitative model engineering. *ACM Transactions on Modeling and Computer Simulation*, Jan. 1992, vol.2, (no.1):52-81.

[Gane & Sarson]

Gane, Chris, 1938- *Structured systems analysis: tools and techniques* / Chris Gane and Trish Sarson. Englewood Cliffs, N.J.: Prentice-Hall, c1979. Series title: Prentice-Hall software series.

[Goguen & Burstall]

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.

[Gould]

Gould, Harvey, 1938- *An introduction to computer simulation methods: applications to physical systems* / Harvey Gould and Jan Tobochnik. Reading, Mass.: Addison-Wesley, c1988.

[Gries]

Gries, David, 1939- *The science of programming* / David Gries. New York: Springer-Verlag, c1981.

[Harrold & Malloy]

Harrold, M.J.; Malloy, B. A unified interprocedural program representation for a maintenance environment. *IEEE Transactions on Software Engineering*, June 1993, vol.19, (no.6):584-93.

[Hatley & Pirbhai]

Hatley, Derek J., 1934- *Strategies for real-time system specification* / by Derek J. Hatley, Imtiaz A. Pirbhai. New York, NY: Dorset House Pub., c1987.

[Horspool & Levy]

Horspool, R.N.; Levy, M.R. Translator-based multiparadigm programming. *Journal of Systems and Software*, Oct. 1993, vol.23, (no.1):39-49.

[Hopcroft & Ullman]

Hopcroft, John E., 1939- *Introduction to automata theory, languages, and computation* / John E. Hopcroft, Jeffrey D. Ullman. Reading, Mass.: Addison-Wesley, c1979. Series title: Addison-Wesley series in computer science.

[Ipser & Wile]

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.

[Jackson]

Jackson, M. A. *System development* / M.A. Jackson. Englewood Cliffs, N.J.: Prentice/ Hall, 1983. Series title: Prentice-Hall international series in computer science.

[Kotik & Markosian]

Kotik, G.; Markosian, L. Application of REFINE Language Tools to software quality assurance. IN: *Proceedings. The Ninth Knowledge-Based Software Engineering Conference* (Cat. No.94TH0664-3). (Proceedings. The Ninth Knowledge-Based Software Engineering Conference (Cat. No.94TH0664-3)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. 4.

[Lowry et al]

Lowry, M.; Philpot, A.; Pressburger, T.; Underwood, I. A formal approach to domain-oriented software design environments. IN: *Proceedings. The Ninth Knowledge-Based Software Engineering Conference* (Cat. No.94TH0664-3). (Proceedings. The Ninth Knowledge-Based Software Engineering Conference (Cat. No.94TH0664-3)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.

[Meyers]

Meyers, S. D. *Representing Software Systems in Multiple-View Development Environments*. Department of Computer Science, Brown University, Report CS-93-18, May 1993.

[Morgan]

Morgan, Carroll, 1952- *Programming from specifications* / Carroll Morgan. New York: Prentice Hall, 1990. Series title: Prentice-Hall international series in computer science.

[Neighbors]

Neighbors, J.M. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, Sept. 1984, vol.SE-10, (no.5):564-74.

[Niskier et al.]

Niskier, C.; Maibaum, T.; Schwabe, D. A look through PRISMA: towards pluralistic knowledge-based environments for software specification acquisition. IN: *Proceedings of Fifth International Workshop on Software Specification and Design* (Cat. No.89CH2717-7). (Proceedings of Fifth International Workshop on Software Specification and Design (Cat. No.89CH2717-7), Pittsburgh, PA, USA, 19-20 May 1989). Washington, DC, USA: IEEE Comput. Soc. Press, 1989. p. 128-36.

[Olsen et al.]

Systems engineering using SDL-92 / Anders Olsen... [et al.]. Amsterdam; New York: North-Holland, 1994.

[Prieto-Diaz & Arango]

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.

[RAISE]

The RAISE specification language / the RAISE Language Group. New York: Prentice Hall, 1992. Series title: BCS practitioner series.

[Rumbaugh et al.]

Object-oriented modeling and design / James Rumbaugh... [et al.]. Englewood Cliffs, N.J.: Prentice Hall, c1991.

[Semmens & Allen]

Semmens, L.; Allen, P. Using Yourdon and Z: an approach to formal specification. IN: *Z User Workshop*, Oxford 1990. Proceedings of the Fifth Annual Z User Meeting. (Z User Workshop, Oxford 1990. Proceedings of the Fifth Annual Z User Meeting, Oxford, UK, 17-18 Dec. 1990). Edited by: Nicholls, J.E. Berlin, Germany: Springer-Verlag, 1991. p. 228-53.

[Srinivas]

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.

[Turski & Maibaum]

Turski, Wladyslaw. *The specification of computer programs* / Wladyslaw M. Turski, Thomas S.E. Maibaum. Wokingham, England; Reading, Mass.: Addison-Wesley, c1987. Series title: International computer science series.

[Unix]

UNIX software development tools: UNIX SVR4.2. Englewood Cliffs, N.J.: UNIX Press, c1992.

[Wing]

Wing, J.M. A specifier's introduction to formal methods. *Computer*, Sept. 1990, vol.23, (no.9):8, 10-22, 24.

[Winston]

Winston, A. The use of $Matrix_X$ in teaching and understanding analog/digital systems. *CoED*, April-June 1991, vol.1, (no.2):28-32.

[Wirsing]

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

[Wood]

Wood, K.R. A practical approach to software engineering using Z and the refinement calculus. (SIGSOFT'93. First ACM SIGSOFT Symposium on the Foundations of Software Engineering, Los Angeles, CA, USA, 7-10 Dec. 1993). *SIGSOFT Software Engineering Notes*, Dec. 1993, vol.18, (no.5):79-88.

[Wordsworth]

Wordsworth, J. B. *Software development with Z: a practical approach to formal methods in software engineering* / J.B. Wordsworth. Wokingham, England; Reading, Mass.: Addison-Wesley Pub. Co., 1992. Series title: International computer science series.

[Zave]

Zave, P. A compositional approach to multiparadigm programming. *IEEE Software*, Sept. 1989, vol.6, (no.5):15-25.

[Zave & Jackson]

Zave, P.; Jackson, M. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, Oct. 1993, vol.2, (no.4):379-411.