The Composition Model of Specification: A Comparative Study

John Anderson

CIS-TR-91-03
February, 1991

Department of Computer and Information Science
University of Oregon

The Composition Model of Specification: A Comparative Study

John S. Anderson
Computer Science Department
University of Oregon
Eugene, OR 97403
john@cs.uoregon.edu

Abstract

Our interest is in formal models of the specification process. In recent years, there have been a number of small-scale studies investigating the feasibility of automating some or all of the specification process. Unfortunately, because the field is quite young, each project team has been forced to invent their own vocabulary to describe their results. This makes it difficult to compare two projects and identify their similarities and differences.

In this paper we propose a single model of specification, the *composition model.* We claim that three recent projects exploring automated specification are all based on the same model. The three projects are the Requirements Apprentice [Reubenstein 91], the IDeA / ROSE project [Lubars 89], and the ASAP / OPIE project [Anderson and Fickas 89]. We first describe the general model, then discuss the individual differences between the three projects.

1 Introduction

Our interest is in formal models of the specification process. In this paper we propose a single model of specification, the *composition model.*

Compositional specification: an example

Figure 1 presents a graphical description (i.e., a specification) of a simple check-out operation. A specification is made up of three kinds of elements: *object* symbols, *relation* symbols, and *action* symbols. These symbols are associated with each other via *association links*. There are several kinds of links. Action symbols may be linked to relation symbols via *produces*, *maintains* and *consumes* links. Action symbols may be linked to object symbols via *agent* and *patient* links. Action symbols may be linked to relation symbols via *subject* and *object* links.

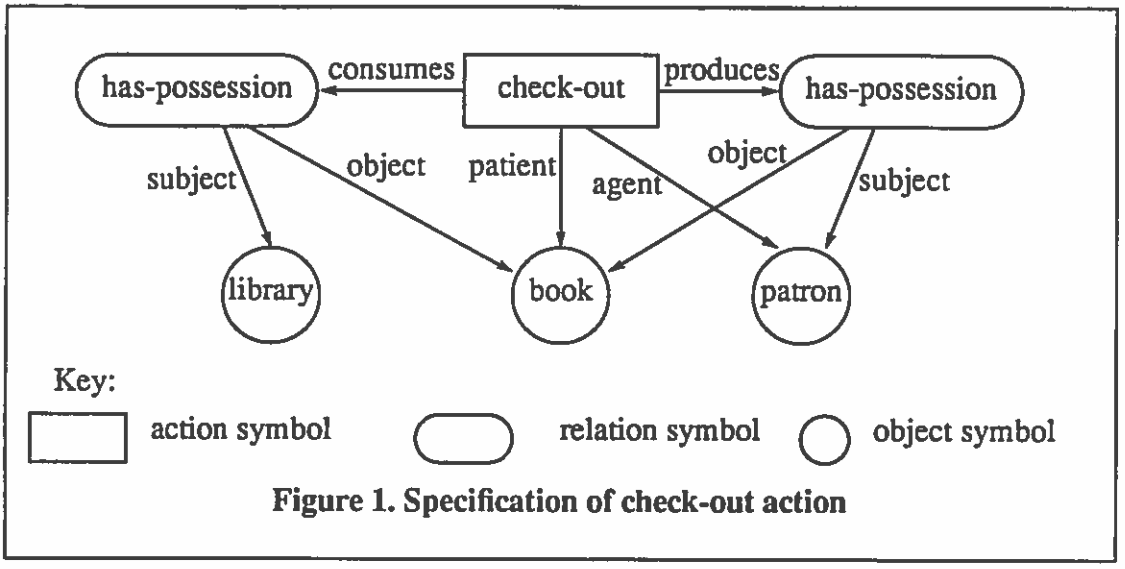

Figure 1. Specification of check-out action

In Figure 1 an action symbol, check-out, is linked by a consumes link to a relation symbol, has-possession, which in turn is associated with object symbols for library and book. The check-out symbol is also linked by a produces link to another has-possession relation, this one with patron as its subject. The check-out action symbol is linked to patron as its agent and book as its patient.

Figure 2 shows a condensed description of the same specification, this time using the standard shorthand of listing objects as arguments of action and relation symbols. While this notation is easier to read, it is somewhat misleading in that it hides the fact that object, relation and action symbols are all equal 'first-class citizens' in this representation.

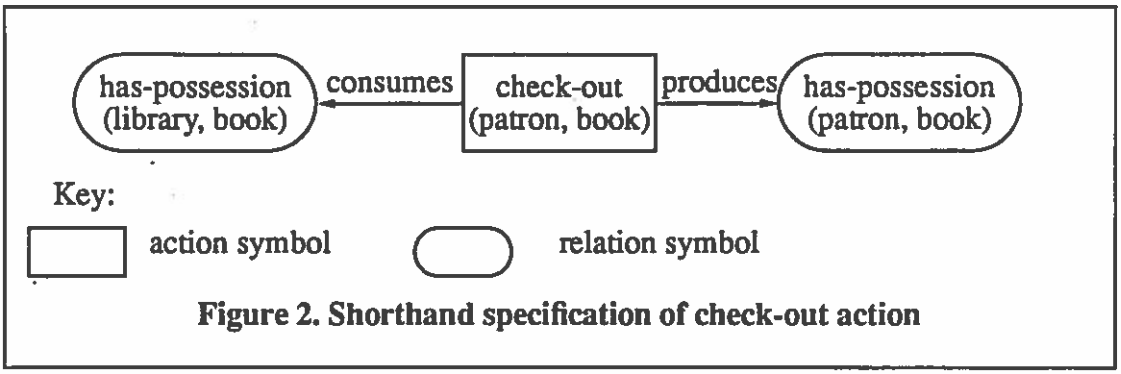

Figure 2. Shorthand specification of check-out action

In the composition model, a requirement is basically a specification with holes in it. For example, Figure 3 shows the requirement that a patron be able to obtain a book from the library. The holes in a requirement are called *roles*.

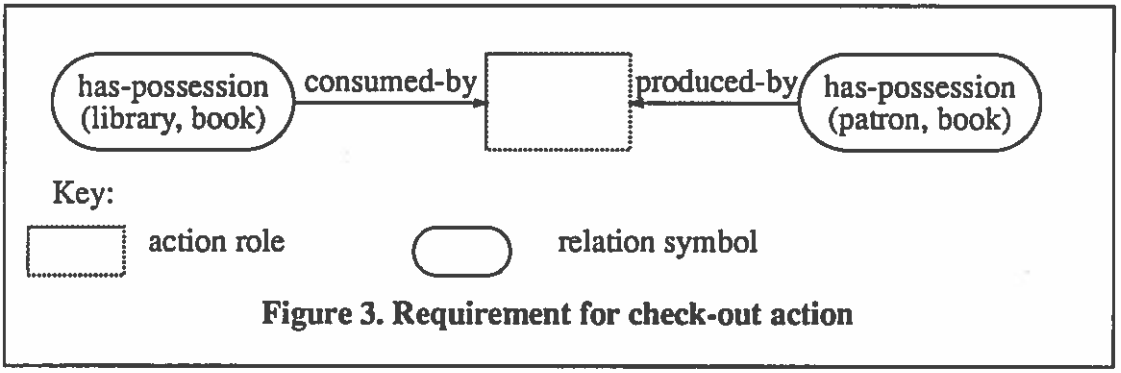

Figure 3. Requirement for check-out action

In order to get from a requirement to a specification, we must have a way of determining possible *fillers* for roles. This requires storing information about the possible associations among objects, relations and actions in a knowledge base. For example, suppose we have stored the knowledge that check-out is an action that produces has-possession(patron, book). We use pattern-matching on has-possession(patron, book) to access the corresponding relation in the knowledge base. We then follow the produced-by link to find possible fillers for the role.

Note that check-out is only one possible filler for the role of producer for has-possession(patron, book). Other fillers include buy and steal. Therefore, at some point we must make a choice among the alternatives. However, by using *incremental selection*, a decision need not be made immediately. Instead, a single action symbol that represents all of the alternatives is used to fill the role at first. In our example, the action obtain(patron, book) would be inserted as an abstract representation of all actions resulting in has-possession(patron, book).

After the abstract action symbol is inserted, it can be specialized with the help of constraints from other design decisions. Constraints are used to prune unsatisfactory alternatives from the set of possible fillers for a role. For example, suppose that at some point the requirement is added that the relation owns(library, book) must be maintained. This requirement acts as a constraint to eliminate buy and steal as possible specializations of obtain(patron, book). Assuming check-out is the only remaining alternative, it is selected to fill the role.

Frequently, after a filler is specialized the next step is to decompose the specialization into parts. For example, the check-out operator might be subdivided into the steps of locating the book, bringing the book to the counter, and recording the transaction.

Introducing or specializing an element may introduce a new set of requirements. For example, locating the book may require that the patron have access to a catalog of books in the library. These new requirements become additional roles to be filled. Also, a new or specialized element may add constraints to existing roles. For example, some other decision might add the constraint that reference-books be excluded from the check-out operation.

Finally, the specification must be evaluated to ensure that none of the requirements are violated. Recall that during specialization any filler that directly violated a constraint was pruned. However, the possibility remains that some combination of elements causes a constraint violation even though no one element alone is responsible. For example, placing an unchecked book in one's backpack is not stealing, nor is carrying one's backpack out of a library. But the combination of those two actions results in a stolen book. Therefore, we require an evaluation step to check the overall consistency of the specification. If an inconsistency is found, some previous commitment has to be changed.

2 The composition model

In this section we discuss the representations used in compositional specification, followed by the operations used to compose the specification.

2.1 Representations

Knowledge base

Compositional problem solving requires a knowledge base of stored descriptions of potential solution components and the possible associations among them. The knowledge base contains descriptions of *types* of objects, relations and actions. A description consists of a symbol plus all of the symbols directly linked to it. Such a description is called a *schema.*[1]

Each of the three kinds of schemas (object, relation, action) are organized into a

taxonomic hierarchy. For example, 'obtain' might be the parent of make, buy, steal and check-out. Furthermore, the schemas are also organized into a partonomic hierarchy. For example, one form of check-out may be composed of locate-book, bring-book-to-counter, and record-transaction. The combined taxonomic and partonomic hierarchies form an AND / OR graph. In addition to the links mentioned previously, the knowledge base contains specialization and decomposition links to form these hierarchies.

Specification

While descriptions in the knowledge base refer to generic *types* and *possible* associations, a specification describes a particular combination of objects, relations and actions and the associations among them. Every specification element is linked to its corresponding type in the knowledge base. An element linked to an abstract type is considered to be an abstract element.

Requirements

A requirement is a partial specification in which some *roles* are left unfilled. A role is a place-holder for an element just as a variable is a place-holder for a value. Roles can be action, relation or object roles. A requirement consists of sub-requirements which determine how the roles are to be filled. Composition is essentially a process of filling the unfilled roles while conforming to the requirements. A standard representation for unfilled roles is a *frame* [Minsky 77]. In a predicate calculus representation, unfilled roles correspond to unbound variables.

A software requirement generally contains action roles described in terms of relations. These requirements fall into four classes: achievability, retractability, maintenance, and avoidance requirements. For example, the requirement for a library might state that the relation 'has-possession (patron, circulating-book)' should be both achievable and retractable, 'in-library (reference-book)' be maintained, and 'stolen (book)' be avoided.

Achievability and retractability requirements define action roles to be filled; maintenance and avoidance requirements serve as constraints on the fillers for those roles. It is important for an artifact to not only enable certain goals of potential users, but also to disable certain actions of potential mis-users.

Action roles are not the only kinds of roles. Preconditions of actions define relation roles; both actions and relations can define object roles.

2.2 Operations

Figure 4 presents a high-level view of the operations in compositional specification. The process begins with a requirement containing unfilled roles and/or abstract ele-

1. See [Sowa 84] for an extensive review of structured representations.

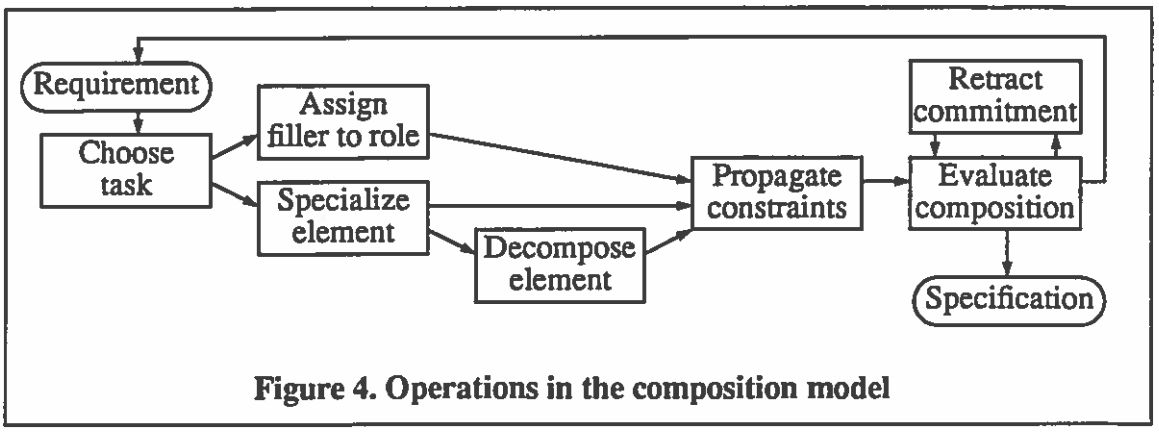

Figure 4. Operations in the composition model

ments. The first step is to select a task: either a role to fill or an element to specialize. If the task selected is an empty role, the next step is to fill it; if the task selected is an abstract element, the next step is to specialize it. If the newly specialized element is composed of sub-parts, it is decomposed. The next step is to introduce and propagate constraints based on the roles associated with the new filler or specialization. These constraints may prune non-solution components without search.

Once the constraints have been introduced, the entire specification is evaluated for unforeseen constraint violations. If found, the specification must be revised by retracting a previous commitment and making an alternative choice.

In order to be considered finished, a specification must be complete (all roles are filled), unambiguous (all elements are completely specialized), and consistent (no constraints are violated).

Choose a composition task

A task is either a slot to fill or an abstract element to specialize. Methods for choosing a task include LIFO [Fikes et al 71], criticality [Sacerdoti 74, Knoblock 90], deterministic tasks first [Stefik 81], tasks with fewest remaining options first [Anderson and Farley 90], and random selection. Another approach is to treat task selection as a separate problem to be solved [Laird 86].

Assign an element to a role

Achievability and retractability requirements drive the initial assignment process. For every achievability requirement, an action which achieves the relation must be included in the specification. The action is found by using the relation as a retrieval cue into the knowledge base. The producer of the corresponding schema in the knowledge base is used as the filler's type. This is essentially the process used in means-ends analysis.

Relation and object roles are filled using the same process. An element linked to the

role is used as a retrieval cue, the corresponding association link is followed to a schema which is used as the filler's type.

A new element of the correct type may be created, or an existing element in the specification may be assigned as the filler. For example, the same door used by a patron to enter the library may be used to leave it.

Constraint propagation

Typically, a new element will be linked to an abstract type whose descendants represent alternative candidates for filling the role [Friedland 85]. In the compositional model, selecting a single alternative is a process of selection by elimination. Constraint propagation [Stefik 81] is used to prune alternatives which are inconsistent with the overall requirements. There are two primary forms of constraints: *type constraints* and *link constraints*.

A type constraint ensures that two linked elements are of compatible types. For example, suppose we start with a generic form of check-out that has two specializations: one for checking out library books and another for checking out rental videos. Since our requirement states that the patient of the action must be a book, the video check-out can be eliminated.

A link constraint prohibits certain links between certain elements. A maintenance requirement is a link constraint that prohibits a consumes relation between certain relations and actions. An avoidance requirement is a link constraint that prohibits a produces relation between certain relations and actions. For example, the requirement that owns(library, book) be maintained is a link constraint that eliminates buy and steal as producers of has-possession(patron, book).

Specialize an element

If all but one of the candidates for a role is eliminated by constraint propagation, the choice becomes deterministic. At this point it is an easy matter for the remaining alternative to be selected as the specialization.

If more than one alternative remains and additional constraints are not forthcoming, it may be that all options are equally acceptable. In this case the best of the remaining alternatives can be selected, or one may be selected at random. However, it might turn out that the selected alternative conflicts with a constraint that is introduced later. Therefore, whenever an arbitrary choice is made it is marked as a potential backtracking point.

The method of first filling a role with an abstract element, then using constraints to eliminate alternatives, and finally selecting a specialization is referred to as *incremental selection*. Incremental selection allows interactions among solution elements to be detected before one option or another has been selected arbitrarily. By avoiding arbi-

trary selections and using constraints to prune alternatives, much of the work associated with search can be avoided. Rather than trying each of the options in turn, constraints are used to prune some of the options without search.

Decomposition

Decomposition occurs whenever an element is introduced that is composed of multiple parts. There is a separate parent node for each possible decomposition, so that once the parent is selected the decomposition is deterministic.

Evaluation

The next phase of the design process is to critique the specification. Many, but not all constraint violations can be avoided using constraint propagation. Additional evaluation is required to ensure that the interaction of several actions do not violate a constraint. For example, if an avoidance requirement is a conjunction of several relations, and no one action adds all of the relations, it is still possible that several actions can combine to produce the prohibited condition.

Revision

If a constraint violation is discovered during evaluation, the solution is to go back to a decision made earlier in the specification process that contributed to the conflict. That decision is retracted, which means that every subsequent choice that depended on that decision may need to be reconsidered. By saving the complete dependency history, it is possible to retract only the suspect decisions while preserving other, unrelated decisions made after the initial error.

3 Comparison of compositional specification systems

In this section we look more closely at alternative views of each aspect of the general composition model. We base our discussion primarily on three projects: the Requirements Apprentice [Reubenstein 91], the IDeA / ROSE project [Lubars 89], and the ASAP / OPIE project [Anderson and Fickas 89].

We begin by noting that the three projects had very different research goals, and therefore differences are to be expected. In fact the degree of agreement among the projects about the basic operations required is rather remarkable.

The Requirements Apprentice (henceforth RA) was proposed as an automated tool to be used by an expert requirements analyst. IDeA / ROSE (henceforth ROSE) was proposed as an expert system to be used by a novice. ASAP / OPIE (henceforth OPIE) was proposed as a model rather than as a tool.

According to the authors' descriptions, the three systems perform three different tasks. RA takes informal requirements as input and returns a formal requirement. OPIE takes a requirement and returns a specification. ROSE takes a specification and

returns a design. It would appear that it takes all three of these systems to get from informal requirements to a design. Actually, despite the differences in terminology, all three systems solve the same problem.

3.1 Representation

Knowledge Base

All of the projects use schemas (called cliches in the RA) to represent information stored in the knowledge base. The representations in the knowledge base define types while the representations in the specification stand for instances of those types. However, the schemas themselves are still somewhat ad hoc structures that are used in slightly different ways in each of the systems. It is likely that the contents of the knowledge base will be the last aspect of the composition model to become standardized.

In this paper we have proposed a uniform treatment of objects, relations and actions as schemas linked by association links. All three projects explicitly include schemas for objects and actions. However, it is difficult to determine exactly how relations are handled in RA and ROSE.

ROSE uses at least three kinds of components: "design schemas" expressed as data flow transformations correspond to action schemas. "Data object definitions" correspond to object schemas and "data object features" seem to correspond to relation schemas. However, the discussion of features is a sub-section under the discussion of data objects, which suggests that features are not treated as 'first-class citizens'.

Discussion of the RA also indicates that relations may not be given full schema status. "The roles of a cliche are represented by the slots of the associated cliche frame type. The constraints on the cliche are represented by a predicate on these slots." [p11]. That is, at least some relations are contained within another schema rather than being a separate schema.

All of the projects organize schemas into hierarchies. In OPIE, this is accomplished automatically by taking the intersection of two schema descriptions and making an abstract schema with the shared association links. The generalization process is described in detail in [Anderson and Farley 88]. In ROSE and RA the hierarchies are constructed by the programmer.

Requirements

All of the systems use basically the same information as input. The input describes the objects and relations that are to be manipulated by as yet unspecified actions. However, this information is expressed in different ways.

OPIE assumes inputs are relations which should be achievable, retractable, maintained, or avoided. The input also includes any objects associated with the relations.

RA assumes inputs are a series of definitions of terms. The terms are primarily objects and needs. Although expressed in a different form, the needs appear to correspond to OPIE's achievability requirements. "The needs section gives a high-level description of the desires of the end-users (e.g., the need to keep track of which books are in the library)." [p10]

In the data flow representation, actions produce and consume objects rather than relations. Therefore, in ROSE inputs are descriptions of data objects and not relations. However, the data objects serve to define the required actions in the same sense that relations are used to define required actions in OPIE and RA.

Specification

All of the projects appear to output a network of action and object descriptions.

The RA provides two forms of output: a human-readable requirements document and a machine-manipulable "Requirements Knowledge-Base" (RKB). One of the key features of the RKB is that it is "coherent," whereas the input is "disorganized imprecise statements." Since the RKB is composed of instantiated cliches, we can assume that it resembles the specification described at the beginning of this paper.

ROSE uses data flow diagrams as the output representation. Again, it is not too much of a stretch to map data flow diagrams onto the representation used in this paper. Also, the next version of ROSE will be based on Petri net representations [Lubars 90]. The diagram in Figure 2 can also be viewed as a type of Petri net representation.

3.2 Operations

Choose a task

All three projects treat unfilled roles and abstract fillers as tasks to be completed. The RA also treats terms which have been introduced but not defined as tasks.

OPIE first looks for tasks which can be completed without making an arbitrary decision. If an abstract element is left with only a single specialization then specializing that element is the next task. If all of the tasks have more than one possible completion, then the task with the fewest alternatives is selected. If there is still a tie, one is selected at random.

ROSE allows the user to select a task or turn control over to the program. If the program is allowed to select the task, it first looks for deterministic tasks. If no tasks are deterministic it makes a random selection.

The RA, in keeping with its role as a tool used by an expert, maintains a list of tasks but leaves it to the expert to select the task to address next. However, like the others, if a specialization has only one remaining alternative that alternative is immediately selected without consulting the user.

Assign filler to role

ROSE and OPIE use pattern matching between requirements and schemas in the knowledge base to identify potential fillers for roles. In OPIE, means-ends analysis is viewed as the primary method of filling roles. ROSE also employs MEA under some circumstances, but uses object descriptions as the primary retrieval cue. In both cases the program makes the assignment of filler to role. In OPIE there is always one schema which represents all of the alternative fillers for a role [Anderson and Farley 88]. A choice need not be made immediately; all of the alternatives remain valid candidates until pruned by constraints.

In ROSE, not all levels of abstraction are represented in the knowledge base. Therefore, it is sometimes necessary to make a choice among alternatives immediately. This is done by counting the number of matches between the requirements and the schemas. If this heuristic fails to produce a clear winner, the choice is turned over to the user.

In RA the user generally makes the assignment. However, "some roles have default values associated with them that are used when no other value is specified." [p14]

Propagate constraints

All three of the systems use some form of constraint propagation to help select among alternative specializations of abstract elements. Unfortunately, the descriptions of how constraints are used in making choices are somewhat vague.

In the RA, specialization is referred to as "disambiguation." An ambiguous term is one which has multiple meanings, i.e., multiple descendants in the cliche hierarchy. Disambiguation is a process of using clues from the context (i.e., constraints) to select the most appropriate interpretation. We can find references to both type constraints and link constraints in RA.

In contrast to OPIE and RA, ROSE has specialization rules which direct it to the correct alternative. However, the specialization rules accomplish exactly the same purpose as constraint propagation in OPIE and RA. The rules take the form "If design family is X and Y is a constraint then subfamily is X'". If it is not known whether Y is a constraint then ROSE asks the user. We can find explicit reference to type constraints ("constraint variables") in ROSE, but nothing resembling link constraints.

Specialize element

In all three systems, whenever only a single option remains, that option is immediately selected. If more than one alternative remains, the systems take different approaches. ROSE uses specialization rules to select the best choice. OPIE resorts to search to explore one option and, in case of failure, backtrack and try another option. In RA the choice is left to the user.

Decompose element

OPIE uses the partonomic hierarchy to determine whether an element should be decomposed and, if so, into what parts. ROSE uses refinement rules to accomplish the same task. RA does not appear to do decomposition.

Evaluate specification

OPIE uses a planner as a devil's advocate, trying to poke holes in the specification by finding plans which violate maintenance and avoidance requirements. RA uses a bounded amount of deductive inference to detect contradictions. No mention is made of evaluation in ROSE. This may be based on the assumption that the rules are correct and need no evaluation.

Retract commitment

If the RA detects a contradiction, it can propose alternative premises that might avoid the contradiction. Furthermore, by recording the dependency history of every design decision and inference, the RA can quickly re-compute the ramifications of a change to the requirements.

OPIE uses less sophisticated backtracking methods to explore alternative options.

ROSE does not appear to have an explicit retraction mechanism. One might presume that an expert system is designed to make the right choice the first time, so that if it fails there is no alternative that might have succeeded.

4 Summary

We have proposed a general model of compositional specification. We have argued that several recent projects studying the automation of the specification process can be described in terms of this model. Finally, we have discussed the differences between the projects.

The contribution of our work is two-fold. First, we have described the model underlying current efforts towards automating the specification process. More importantly, we have tied this model to current work in AI planning. This means that many of the operations that are currently performed manually in the RA and ROSE can be automated using existing AI planning techniques.

References

[Anderson 90] Anderson, J.S. Partial commitment and requirements engineering, in Proceedings of AAAI Workshop on Planning in Complex Domains, Boston, 1990.

[Anderson and Farley 88] J. S. Anderson and A. Farley, Plan abstraction based on operator generalization, In Proceedings of AAAI-88, St. Paul, 1988, pages 100-104.

[Anderson and Farley 90] J.S. Anderson and A. Farley, Incremental selection in plan composition, CIS-TR-90-11, University of Oregon, 1990.

[Anderson and Fickas 89] J.S. Anderson and S. Fickas, Viewing specification design as a planning problem: a proposed perspective shift, in Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh, 1989. Also to appear in Artificial Intelligence and Software Engineering, D. Partridge (ed), Ablex, 1990.

[Anderson and Fickas 90] J.S. Anderson and S. Fickas, Using approximations and abstraction in software engineering,

[Friedland and Iwasaki 85] Friedland, Peter E. and Yumi Iwasaki. 1985. "The concept and implementation of skeletal plans," Technical Report KSL 85-6, Stanford University.

[Lubars 89] Lubars, M.D. and M.T. Harandi, Addressing software reuse through knowledge-based design, in Biggerstaff and Perlis, eds, Software Reusability Vol II, ACM Press, 1989.

[Robinson 1989] Robinson, W., Integrating multiple specifications using domain goals, In Fifth International Workshop on Software Specification and Design, Pittsburgh, 1989.

[Robinson 1990] Robinson, W., Negotiation behavior during requirement specification, In 12th International Conference on Software Engineering, Nice, 1990

[Reubenstein 91] Reubenstein, H.B. and R.C. Waters, The Requirements Apprentice: automated assistance for requirements acquisition, to appear in IEEE Transactions on Software Engineering, 1991.

[Sacerdoti 1974] Sacerdoti, E. 1974. "Planning in a hierarchy of abstraction spaces," Artificial Intelligence 5, 115-135.

[Sowa 84] J. Sowa, Conceptual Structures, MIT Press, 1984.

[Stefik 81] Stefik, Mark J. 1981. Planning with Constraints (MOLGEN: Part 1), in Artificial Intelligence.