A Pattern-Driven Approach to Code Analysis for Reuse [1]

R.B. France and V.R. Basili
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742, U.S.A
November 1991

Abstract

Reuse reengineering, as defined by Basili and Caldiera, is concerned with the identification and extraction of reusable portions of existing code, and the qualification of the extracted portions. The qualification process is concerned with the analysis, tailoring, and packaging of extracted components which are then used to seed a reusable software component repository. In this paper we focus on the analysis activity of qualification and describe an analysis method that utilizes expert programming and domain knowledge and formal techniques to generate formal, and intuitive specifications of functionality from software components. In a reuse environment, such specifications can be used to gain an understanding of a component's functionality, identify and adapt components, and serve to establish a high degree of confidence in the functionality of a component.

[1] Research for this study was supported in part by ONR (Grant NOOO4-87-k-0307)

- establishing confidence in the functionalities of components.

An analysis is said to be *successful* if it results in an understanding of the s-unit's functionality and a specification that reflects the understanding in an intuitive manner. An analysis is said to be *unsuccessful* if it does not result in an understanding of the s-unit's functionality. If an s-unit's functionality is not understood by a component user then its use in a reuse environment may create more problems than solve any. There is a good chance that the results of an unsuccessful s-unit analysis will not be understood by potential users, thus, an s-unit that has been unsuccessfully analyzed is rejected as being suitable for reuse.

The *extended* s-unit resulting from a successful analysis consists of a code or implementation part and a specification part. The extended s-unit is passed to the packaging activity which is concerned with the organization of information relevant to the reuse of the component in future development projects. Packaging may also involve generalizing the s-unit to make it applicable to a broader class of future application development efforts [5]. The packaging activity is also concerned with the generation of information pertaining to the certification of software components, for example, test cases, and to the classification of the component in the component repository. The classifications are made according to a predefined set of attributes and are used to aid the retrieval of the component in the repository and to distinguish it from other components. Problems encountered in classifying a component are used to modify the current set of predefined attributes as modeled by the feedback from the qualification phase to the taxonomy model.

This paper focuses on the analysis activity of the qualification phase. Results from empirical research on how programmers understand code indicate that programmers utilize expert programming and problem-domain knowledge in order to create and establish function hypotheses [8, 13]. We use the term *expert analysis* to refer to the use of expert programming and domain knowledge in analysis. There is evidence that some of the knowledge required to carry out an expert analysis can be captured and represented, facilitating their reuse [14, 18, 21]. Existing inverse engineering systems utilize such knowledge expressed, for example, as plans [18], cliches [21], or transformation rules [20], to produce a specification from code. For example, schema based systems (e.g. plans, cliches) relate patterns of code representations to abstract concepts. In this paper we show that similar expert analyses can be applied to formal specifications to gain intuitive insights into the functionalities they specify.

The use of formal specifications to express component functionality enhances the role of specifications in a reuse environment as follows:

- The unambiguous nature of formal specifications facilitates their use in identifying and ensuring the correct use of components.
- A formal specification can serve as the basis for a rigorous demonstration of consistency, which in turn serves to establish a high degree of confidence in the functionality of the s-unit it specifies [22].
- Adapting software components to a particular usage may also reap benefits from the use of formal specifications. Adaptation can be carried out at both the implementation

1 Introduction

The need to reduce the cost of developing software and to increase its reliability and quality has given rise to a software reuse technology. Reuse technology is mainly concerned with explicitly capturing software development experience in a form that is potentially (re)usable in future development efforts.

The ability to capture and package experience effectively is dependent on the availability of explicit expressions of experience. Program code, for example, provides explicit expressions of implementation experience that, when identified and packaged appropriately, can be reused in development efforts.

The setting up of a repository of reusable software components is, in general, an expensive activity. To alleviate such costs Caldiera and Basili [9] propose that existing software be analyzed to determine their suitability for reuse and then used to seed the repository. A model of their process for extracting reusable software components from existing software, called *reuse reengineering*, is shown in Figure 1 [9]. Reuse reengineering activities are divided into two phases: an *identification*, and a *qualification* phase.

In the *identification phase* a reusability measurement model is developed and used to identify and evaluate extracted portions of code. The definition activity models the current understanding of what constitutes a reusable component in terms of automatable measures of the characteristics identified. The extraction activity removes modular code units from existing code, completes them by including external references, and converts them to *s-units* (for software units), which are programming-language independent representations of the completed code. S-units are then evaluated against the reusability model to determine their suitability for reuse.

S-units deemed potentially suitable for reuse in the identification phase, then go through the qualification phase in which they are analyzed, packaged and stored in a component repository for reuse. Experience gained in the qualification phase can change our knowledge of what constitutes a reusable component, thus feedback from the qualification phase is used to change the reusability model.

Specifications can play a significant role in reducing the effort needed by potential component users to understand component functionalities, by providing the users with effective code abstractions. In the absence of such specifications the users must resort to code reading in order to gain an understanding of component functionality. For large and/or complex components this may be very time consuming, and difficult, thus reducing the chances that the component will actually be reused. The analysis activity in the qualification phase is concerned with gaining an understanding of an s-unit's functionality through the creation of a functional specification. The functional specifications derived from analysis are used as the basis for:

- understanding component functionality,
- identifying and ensuring the correct use of software components,
- adapting components for a particular usage, and

and specification level, using rigorous techniques to ensure consistency between the transformed implementation and specification. This approach is further enhanced if the formal specification is machine-processable, since this would allow some automated consistency checks to be made.

Within the context of reuse reengineering, it is important that the specifications derived from analysis be *understandable*, in the sense that they provide intuitive insight into the specified functionality of the implementation. In this respect formal specifications are notoriously inadequate. To enhance the understandability of formal specifications we propose that they be *abstracted* to a level understandable by an 'average' component user. An abstraction may take the form of another formal statement formulated in terms of more abstract concepts and notation taken from a widely known and understood mathematical model, or a natural language description of the functionality captured by the formal specification.

Based on these observations we have developed a method that integrates a formal analysis technique with an expert analysis of formal specifications to produce *understandable* formal specifications, in the form of detailed formal specifications and their abstractions, from an s-unit. The formal analysis technique is used to mechanically translate s-unit statements to predicative specifications. Given the mechanical nature of the formal analysis technique, the predicative specifications generated may not always be intuitive. In such cases, expert analysis of the specification is required. In our method, human expert analysis is supported by a knowledge base that relates specification patterns to descriptive narratives. The descriptive narratives provide clues to the functionality captured by the specification and are intended to aid a component specifier in comprehending an s-unit's functionality.

Section 2 gives an overview of our analysis method and the motivating concerns behind it. Section 3 describes the formal analysis technique of our analysis method, and section 4 describes the form of the knowledge representations supporting expert analysis and provides a demonstration of how the method is applied. Section 5 gives our conclusions and outlines related areas requiring further research.

2 Extracting formal specifications from code

The denotational semantics of programs provides rules for translating code structures to functions [19] and it has been shown how such rules can be used to derive a functional abstraction of a program (Sec. 5.1.1 Programs Are Functions in [3]). A practical formulation of denotational semantics developed by Mills [15, 4] can be used for the same purpose. In theory, the extraction of functional abstractions from code associated with a denotational-type semantics is entirely mechanizable. In practice, the resulting specifications are not always intuitive and in some cases, especially those involving recursive code structures such as loops, may be as complex as the code, if not more. Similar problems have have been observed in specification extraction methods based on other formal program semantic models (e.g. see [7, 20]).

The means to address this problem range from simply renaming syntactic elements of specifications (e.g. replacing a function symbol f by *sum* to reflect that the function carries out a sum) to reformulating specification elements in terms of more intuitive structures (e.g. replacing recursive function definitions by simple iterative definitions). Generally, one has to resort to expert programming and domain knowledge in order to make the specifications more intuitive.

Based on the above observation we have developed the simple conceptual model of analysis shown in Figure 2. Two types of analyses are identified in our conceptual model: *expert* and *mechanical* analyses. The *Mechanical Analyzer* utilizes knowledge that can be mechanically extracted from code to produce formal specifications. The *Expert Analyzer* utilizes domain and programming knowledge to obtain intuitive insights into the functionality of the code being analyzed. Such insights can be used to restate the formal specifications in a more intuitive manner, or can be formulated separately (e.g. in natural language) and appended to the formal specification. The *Component Specifier* interacts with the Mechanical and Expert Analyzers to produce understandable formal specifications from code. Activities of the Mechanical Analyzer are automatable, while those of the Expert Analyzer are not easily automated.

Our analysis method, depicted in Figure 3, is a particular organization of mechanical and expert analysis activities. The analysis activity takes an s-unit as input and analyzes it in a sequential manner, using both a formal analysis technique that mechanically translates s-unit statements to formal specifications, and human expert analysis, supported by a knowledge base called the *Functional Abstraction Schema Base* (FASB), to provide intuitive insights into the functionality captured by the formal specifications. A successful analysis results in an s-unit extended with a specification part that provides both a detailed, formal specification and an abstract, intuitive specification of functionality.

The following sub-sections give an overview of s-units and the manner in which they are analyzed in our method.

2.1 A programming language independent representation of code

An algorithm may be expressed in a variety of ways in a single programming language, for example, an algorithm that adds two numbers may be expressed in an imperative programming language as a function, a procedure, or as part of a larger piece of code. From an analysis perspective, there is a need to abstract over such notational variances to get at the functionally essential parts of the code. From a component reuse perspective, storing code representations that abstract over notational variances can bring about savings in repository space [12]. The most common solution to syntactic abstraction of code is to use a code representation that utilizes programming language independent constructs, together with mechanisms for translating between code structures and their representations. The representations used in our method, *s-units*, are variations of Joo's elementary *processes* [12]. S-units are generated in the identification phase of the reuse reengineering model.

An *s-unit* generated in the identification phase is an entity consisting of a set of input and output *ports*, a *variable declaration* part, and a *statement*. The statement part of an s-unit is a composition of basic statements, where a basic statement is either an assignment, if-then-else, non-deterministic choice, while-do, or an input/output statement.

An s-unit representation of an algorithm that sums a stream of integers and returns the result as output is given below:

```
S-unit ADDSTREAM($x$ : in stream(integer); $z$ : out integer)
var $u, v$ : integer
begin
 $u := 0$;
 while $\sim$ iseos($x$) do
  Receive($v, x$);
  $u := u + v$
 od;
 $z := u$;
 Send($z$)
end
```

The variable x is an input integer stream port and z is an output integer atomic port. Atomic ports are bound to single instances of their data type while stream ports are bound to streams of instances of their data type [12]. An s-unit associated with stream ports may carry out a series of communications via the stream ports in a single execution, as in the while loop of ADDSTREAM. In ADDSTREAM the input *Receive* statement assigns the head of a stream, x to the variable v, and the output *Send* statement makes the value of z available for communication. The predicate *iseos* tests whether the end of stream symbol has been encountered.

The result of an s-unit analysis is an *extended* s-unit, which is an s-unit extended with a *Specification Part.* A Specification Part consists of two parts: an *abstract definition*, and a *detailed definition.* The *abstract definition* part provides an intuitively appealing description of the functionality of an s-unit in terms of the relationship between its input port values and its output port values. The relationship may be expressed in natural-language and/or in terms of concepts and notation from a simple, well-understood mathematical model.

The *detailed definition* part provides a formal specification of an s-unit's functionality in terms of the relationship between the initial and final values of the s-unit's variables. The formal nature of the detailed definition makes it amenable to the type of rigorous analysis required to establish that a component possesses certain properties, or to determine the effect a change in the component's code may have on its functionality. The formal specification in the detailed definition may also be supplemented by natural-language descriptions which provide intuitive insights into the detailed functionality of the s-unit. Such intuitive insight is important to a human reader of the component who is interested in a detailed description of the input/output effect of s-unit variables for reasons related to adaptation and component use.

An extension of the s-unit ADDSTREAM is given in Figure 4. The following stream functions are used in its *Specification Part:*

nilstream : The empty stream.

```
S-component ADDSTREAM($x$ : $in\ stream(integer)$; $z$ : $out\ integer$)
var $u, v$ : $integer$
Specification Part
 abstract definition
  $z = SUM(x)$ where
   for all $h$ : $integer$; $t$ : $stream(integer)$
   1. $SUM(h|t) = h + SUM(t)$
   2. $SUM(nilstream) = 0$
 detailed definition
   $[< x_{out}, u_{out}, v_{out}, z_{out} >=< sum(x_{in}, 0, v_{in}), u_{out} >]$ where
      1. $iseos(x) \Rightarrow sum(x, u, v) =< x, u, v >$
      2. $\sim iseos(x) \Rightarrow sum(x, u, v) = sum(tail(x), u + next(x), next(x))$
  description of sum
  The function sum sums the integers in the stream
  $x_{in}$ and returns the result in $u_{out}$.
Implementation Part
begin
 $u := 0$;
 while $\sim iseos(x)$ do
  $Receive(v, x)$;
  $u := u + v$
 od;
 $z := u$;
 $Send(z)$
end
```

Figure 4: The extended s-unit ADDSTREAM

| : Stream concatenation, for example, $h|t$ is a stream with a head h and a tail t.

tail : The function *tail* returns the tail of a stream.

next : The function *next* returns the head of a stream.

The abstract definition is formally expressed in terms of a sum on streams of integers. It is not always possible to find such simple formal expressions of the input/output relationship of an s-unit. In cases where a simple formal definition is not available the abstract definition consists of natural-language text describing the relationship as it is understood by the component specifier. The detailed specification relates the initial values of s-unit variables, denoted var_{in}, with the final values of s-unit variables, denoted var_{out}.

In our analysis method the abstract definition is obtained as an abstraction of the detailed definition. In the following sub-section we give an overview of how the Specification Part of an extended s-unit is created.

2.2 Generating understandable formal specifications

At the abstract definition level an s-unit is a relation between its input and output port values. At the detailed level an s-unit is a relation between the initial and final values of its variables. When the s-unit is deterministic the semantic model at both levels is equivalent to Mills' functional abstraction model [4]. In fact, the semantic model we use can be viewed as an extension of Mills' functional semantics that caters to the specification of general relations.

In our analysis method, applications of expert and mechanical analysis techniques are interspersed to produce a meaningful *detailed definition* of an s-unit. The detailed definition is then abstracted to obtain the *abstract definition.* Figure 5 gives a more detailed view of the analysis method.

In deriving the detailed definition, mechanical analysis utilizes formal rules to translate s-unit statements to formal specifications, while human expert analysis, supported by the *Functional Abstraction Schema Base* (FASB), is used to to obtain intuitive insights into the functionality captured by the formal specifications. The FASB contains objects relating specification patterns to intuitive descriptions of their functionality,

Specifically, the following steps are taken in the sequential analysis:

- A functional abstraction of the current s-unit statement being analyzed is obtained mechanically.
- The resulting specification is then composed with the specification developed thus far.
- The specification formed by the composition is simplified by optimizing specification terms.
- If the simplified specification still contains elements that are complex (e.g. recursive function definitions) then the specification undergoes an expert analysis which takes advantage of previously recorded experience, in the form of FASB specification patterns, to aid the derivation of an intuitive description of the functionality defined by the specification.

Figure 5 depicts the above analysis as a module called *Generate Detailed Specification.* The asterisk in the module indicates that the activities represented in the modules below it are iterated in a left-to-right sequence. The current specification depicted in Figure 5 becomes the detailed specification if there are no more s-unit statements to translate.

As an example of how the above activities are related consider an analysis of the loop in the s-unit ADDSTREAM. A formal analysis is first carried out on the loop, starting with the statements in the body and then the loop structure. The mechanics of this analysis will be detailed later in this paper. Below, we give only the result of the formal analysis:
$[< x_{out}, u_{out}, v_{out} >= F(x_{in}, u_{in}, v_{in})]$
where
1. $iseos(x) \Rightarrow F(x,u,v) =< x,u,v >$
2. $\sim iseos(x) \Rightarrow F(x,u,v) = F(tail(x), u + next(x), next(x))$

The above is not intuitive, mainly because of the recursive nature of the definition. Subsequent human expert analysis, supported by the FASB, would reveal that the variable u accumulates a sum, which could result in a simple change of the function symbol F to say *sum*, and the addition of a natural language description of the function to the specification, resulting in the following, more descriptive formal specification:

$[< x_{out}, u_{out}, v_{out} >= sum(x_{in}, u_{in}, v_{in})]$
where
1. $iseos(x) \Rightarrow sum(x, u, v) =< x, u, v >$
2. $\sim iseos(x) \Rightarrow sum(x, u, v) = F(tail(x), u + next(x), next(x))$
description of sum
The function **sum** *sums the integers in the stream*
x_{in} *and adds the total to the value of* u_{in} *to give* u_{out}.

Loops (or any recursive code structures) present understandability related problems to the formal analysis process. As pointed out by Waters [21], to gain an understanding of a loop in a structured manner, the body is first analyzed and its meaning bootstrapped to a meaning for the loop, a process that can not be mechanized in general. The solutions to this problem in existing specification generation systems range from using human-supplied loop function hypotheses [2, 1, 17], to using transformation rules and/or heuristics to produce and simplify loop invariants [7, 20]. While it may seem easier to supply a loop hypotheses to the analysis process, our experiences indicate that it is difficult, in general, to guess a loop's functionality without resorting to expert programming and domain knowledge. On the other hand, the use of transformation rules to produce and simplify loop specifications is attractive, given the potential for automation, but we have observed that the resulting specifications are not always easy to comprehend.

In our method translation rules are used to transform loops to formal specifications which are then subjected to human expert analysis supported by the *Functional Abstraction Schema Base* (FASB). Patterns in the FASB may provide clues to the specification's functionality which a human can use in coming to an understanding of the specification's intent. Once an understanding is arrived at the human then attempts to test his/her understanding either by formally restating the specification in a more convenient manner and showing that it is correct with respect to the s-unit, or by checking that the formal specification satisfies properties that characterize the understanding. In the former approach one can use Mills' functional correctness technique [4] and its extensions [6, 16] provided the loops satisfy certain conditions. Such conditions apply to a large class of loops in practice. The latter approach can benefit from the use of an automated theorem prover. At this time our method does not stipulate a particular technique for checking an understanding, the onus is on the specifier to establish that the understanding is real and not apparent.

Once a specifier is convinced that an understanding has been arrived at, one or a combination of the following actions may be taken:

- Rename functions to reflect their intent.
- Append natural language descriptions of functionality to the specification.
- Reformulate the specification in terms of more intuitive structures.

The result should be a specification reflecting the specifier's understanding of the loop's functionality.

If expert analyses fails to come up with a meaningful specification the s-unit is no longer considered useful for reuse. The rationale behind the rejection can be stated as follows: if a specifier cannot come to an understanding of the s-unit via the method, then it is unlikely that a software user will be able to fully comprehend the functionality of the software component, in which case, the software component is less likely to (re)used.

After the documented detailed specification is produced as above, the component specifier attempts to use the insights gained and documented during its creation to produce an abstract specification of the s-unit (see Figure 5). Currently, this is a purely human activity, but it may be possible to provide automated expert support, in the form of a knowledge base, for translating some aspects of detailed specifications to related abstract concepts, for example, translating an array data structure to a more abstract linear data structure.

The remainder of this paper details our analysis method. The form of the objects in the Functional Abstraction Schema Base (FASB) and examples of their use in analysis are also provided. In this paper we are not concerned with how the FASB is created and maintained, the implicit assumption being that the method is embedded in an organization that provides such facilities. For this reason we do not detail the retrieval mechanisms associated with the FASB, but do describe the search processes involved.

3 Mechanical analysis: Transforming s-unit statements to formal specifications

In this section we describe the formal analysis technique which takes s-unit statements and mechanically transforms them to formal specifications.

The forms of s-unit statements are given below. In what follows x is a variable declared in an s-unit, e is an expression, S, $S1$ and $S2$ are statements and B is a boolean expression.

Null statement NULL

Assignment $x := e$ (e is assumed to be of the same type as x).

Composition $S1; S2$.

Non-deterministic choice $S1 \nabla S2$.

Deterministic Choice $if\ B\ then\ S1\ else\ S2\ fi$

Loop $while\ B\ do\ S\ od$

Input/output $Receive(x, streamport)$, $Receive(atomicport)$,
$Send(e, streamport)$, $Send(atomicport)$

An s-unit that contains no non-deterministic choice statements is said to be *deterministic*. The examples given in this paper pertain to deterministic s-units.

3.1 Relational specifications

Port names and variables declared in an s-unit are referred to as *s-unit variables.* A *data state* of an s-unit is a tuple of values, where each value is associated with an unique s-unit variable. A *relational specification* is an assertion about the initial and final data state of an s-unit execution and is of the form:

$[P(v_{in}, v_{out})]$

where $P(v_{in}, v_{out})$ is a predicative expression characterizing the relationship between the initial data state, v_{in}, and the final data state, v_{out}. The semantic model of a relational specification is a relation, R, determined as follows:

$\forall v_{in}, v_{out}\ P(v_{in}, v_{out}) \Rightarrow R(v_{in}, v_{out})$.

If $[P(v_{in}, v_{out})]$ is a relational specification characterizing a relation R, R is said to be *deterministic* if

$\forall v_{in} \exists 1 v_{out}\ P(v_{in}, v_{out})$

that is, for each input state, v_{in}, there is exactly one output state, v_{out}, that satisfies $P(v_{in}, v_{out})$. The mechanical transformation of deterministic s-units results in deterministic specifications in our method.

Relational specifications generated from deterministic s-units are given as a set of universally quantified equations of the form:

$[C1(v_{in}) \Rightarrow v_{out} = f1(v_{in}),$
$C2(v_{in}) \Rightarrow v_{out} = f2(v_{in}), \ldots,$
$Cn(v_{in}) \Rightarrow v_{out} = fn(v_{in})]$

The predicative expression $Cp(v_{in})$[1], $p = 1 \ldots n$, is called the *premise* or condition part, while the expression $v_{out} = fp(v_{in})$, where fp is a function symbol, is called the *conclusion* of the equation.

3.2 The translation rules

The rules presented here for translating s-unit statements to relational specifications are based on the predicative semantics for programs developed by Hehner [11]. The notation used in the rules is described in the Appendix.

The predicate Def_s on variables in a data state s will be used in the rules and is true if and only if the variable is defined in s, that is, it is not assigned the value $\perp$ in s (see Appendix A). The s subscript is dropped when the state can be implied from the context in which the predicate appears. When the predicate Def_s is given an expression as an argument then it is true if and only if all the variables in the expression have been assigned values, and the assigned values are in the domain of the expression. For example,
$Def(x/y) = Def(x) \wedge Def(y) \wedge y \neq 0$.

[1]the form of the expression states that v_{in} is free in the expression

Null rule

$NULL \mapsto [true]$

Assignment rule

$x := e \mapsto [Def(e[v \leftarrow v_{in}]) \Rightarrow v_{out} = v_{in}[x \leftarrow eval(e, v_{in})]]$

For example,
$x := x/y \mapsto [Def(x_{in}) \wedge Def(y_{in}) \wedge y_{in} \neq 0 \Rightarrow < x_{out}, y_{out} >=< x_{in}/y_{in}, y_{in} >]$
In the above predicate expression the input state is
$v_{in} =< x_{in}, y_{in} >$
and the output state is
$v_{out} =< x_{in}/y_{in}, y_{in} >$.

Composition rule

If $S1 \mapsto [P1]$, and $S2 \mapsto [P2]$ then
$S1; S2 \mapsto [\exists v : P1[v_{out} \leftarrow v] \wedge P2[v_{in} \leftarrow v]]$.

For example,
For $S1 = x := x/y$, where
$P1 = Def(x_{in}) \wedge Def(y_{in}) \wedge y_{in} \neq 0 \Rightarrow < x_{out}, y_{out} >=< x_{in}/y_{in}, x_{in} >$
and $S2 = y := x * y$, where
$P2 = Def(x_{in}) \wedge Def(y_{in}) \Rightarrow y_{out} = x_{in} * y_{in}$
$S1; S2 \mapsto [\exists x, y : (Def(x_{in}) \wedge Def(y_{in}) \wedge y_{in} \neq 0) \Rightarrow < x, y >=< x_{in}/y_{in}, y_{in} > \wedge$
$(Def(x) \wedge Def(y)) \Rightarrow < x_{out}, y_{out} >=< x, x * y >]$
which simplifies to
$[Def(x_{in}) \wedge Def(y_{in}) \wedge y_{in} \neq 0 \Rightarrow < x_{out}, y_{out} >=< x_{in}/y_{in}, (x_{in}/y_{in}) * y_{in} >]$.

Deterministic choice rule

If $S1 \mapsto [P1]$ and $S2 \mapsto [P2]$ then
$if\ B\ then\ S1\ else\ S2 \mapsto [Def(B[v \leftarrow v_{in}]) \Rightarrow (B[v \leftarrow v_{in}] \wedge P1) \vee (\sim B[v \leftarrow v_{in}] \wedge P2)]$

For example,
$if\ y > 0\ then\ x := x/y\ else\ y := y * x\ fi \mapsto$
$[Def(y_{in}) \wedge Def(x_{in}) \wedge y_{in} \neq 0 \wedge y_{in} > 0 \Rightarrow < x_{out}, y_{out} >=< x_{in}/y_{in}, y_{in} >,$
$Def(y_{in}) \wedge Def(x_{in}) \wedge y_{in} \not> 0 \Rightarrow < x_{out}, y_{out} >=< x_{in}, y_{in} * x_{in} >]$

Non-deterministic choice rule

If $S1 \mapsto [P1]$ and $S2 \mapsto [P2]$ then
$S1 \nabla S2 \mapsto P1 \vee P2$.

Input/output statements

Stream ports are associated with the following operations:

nilstream : The function (constant) *nilstream* $:\rightarrow$ *stream* creates the empty stream.

next : The function *next* : *stream* $\rightarrow$ *data* returns the head of a stream.

put : The function *put* : *data stream* $\rightarrow$ *stream* puts an element onto the end of a stream.

iseos : The boolean function *iseos* : *stream* $\rightarrow$ *boolean* returns true if and only if the end of a stream has been encountered.

tail : The function *tail* : *stream* $\rightarrow$ *stream* returns the tail of a stream.

S-unit input/output statements are interpreted as follows:

- $Open(portid, file)$: If *portid* is a stream port, this statement assigns a stream, *file* to the port. If the port is atomic then *file* is a data instance of the port type which is assigned to the port.
- $Receive(x)$: For atomic ports only - this statement makes the value on port x available to the s-unit.
- $Receive(x, portid)$: For stream ports only - this statement assigns the next element on the stream associated with the port *portid* to the s-unit variable x.
- $Send(e, portid)$: For stream ports only - this statement puts the evaluated value of the expression e onto the stream associated with the port *portid.*
- $Send(x)$: For atomic ports only - this statement makes the data assigned to port x available for communication.
- $Close(portid)$: This statement unbinds the current data stream, for stream ports, or the current data instance, for atomic ports, from the port, *portid.*

The translation rules for input/output statements are:

<u>Stream Ports</u>

$Open(portid, file) \mapsto [portid_{out} = file]$

$Close(portid) \mapsto [portid_{out} = \perp]$

$Receive(x, portid) \mapsto$
$[\sim (iseos(portid_{in})) \Rightarrow < x_{out}, portid_{out} > = < next(portid_{in}), tail(portid_{in}) >]$

$Send(e, portid) \mapsto [portid_{out} = put(eval(e, v_{in}), portid_{in})]$ [2]

<u>Atomic Ports</u>

$Open(portid, file) \mapsto [portid_{out} = file]$

[2] v_{in} is the data state just before execution of this statement

$Close(portid) \mapsto [portid_{out} = \perp]$

$Receive(x) \mapsto [x_{out} = x_{in}]$ that is, this statement has no effect on the data state (x must be a port name).

$Send(x) \mapsto [x_{out} = x_{in}]$ that is, this statement has no effect on the data state (x must be a port name).

Loop statements

A loop:

```
while B do
  $X = f(X')$
od
```

where X is the data state just after a loop iteration and X' is the data state just before a loop iteration, is translated to the specification:

$[X_{out} = F(X_{in})]$ where
$\sim B \Rightarrow F(X) = X,$
$B \Rightarrow F(X) = F(f(X))$

The first equation defining F is called the *base part*, while the second equation is called the *recursive* part of the definition of F.

Generally, loop specifications produced in this manner are difficult to understand. Expert analysis is usually needed to gain intuitive insight into the functionality being specified. Once an understanding of the loop function is arrived at the specifier may choose either to rename the function so that it reflects its meaning, and append a natural language description of the functionality, or reformulate the function definition in terms of more intuitive concepts and notation.

As stated earlier, Mills' functional correctness technique [4] and its extensions [6, 16] can be used in most practical cases to verify that a reformulated loop specification is correct with respect to the loop code. The condition under which such techniques can be applied is refered to as *closure* which, in general, implies that the intermediate values generated during a loop execution bear a constant relationship with the value when the loop execution terminates. It has been observed that a large class of non-closed loops created by programmers have other properties that allow them to be verified by the above techniques. Such properties have been characterized by Misra [16].

4 Expert analysis: Representing expert knowledge as patterns

In this section we describe the type of pattern schemas used to support expert analysis in our method, and demonstrate how they are used in an expert analysis of a relational specification.

A *pattern* is defined as an occurrence of a particular syntactic structure in a specification. The objects in the Functional Abstraction Schema Base (FASB) relate specification patterns to intuitive descriptions of their functionalities. The descriptions are not necessarily precise, for example, they may not describe the effect on variables in detail, but they provide the specifier with clues to the specified functionality. The developer can use the clues to gain a more precise understanding of the specification, using his/her own expert programming and/or domain knowledge. The FASB thus complements the human expert analysis required to obtain an understanding of a specification.

4.1 Defining specification patterns

In order to define specification patterns we need to associate types with syntactic structures of specifications. The formal specifications generated from s-units using the formal rules given in the previous chapter consist of two parts: a relational *specification* part (the predicative expression enclosed in [,]) and a *definition* part. The specification part expresses the relationship between the initial and final data states of an s-unit statement execution in terms of functions defined in the definition part. For example, the specification part of the while loop specification given in section 2 is:

$[< x_{out}, u_{out}, v_{out} >= F(x_{in}, u_{in}, v_{in})]$

while its definition part is:

1. $iseos(x) \Rightarrow F(x,u,v) =< x,u,v >$
2. $\sim iseos(x) \Rightarrow F(x,u,v) = F(tail(x), u + next(x), next(x))$

For deterministic s-units, the specification and definition parts consist of sets of *conditional equations*. In our patterns, conditional equations are elements of type $CondEqn$, that is, an element of $CondEqn$ is of the form $C \Rightarrow T$, where C is a predicative expression and T is an equation, that is, an expression of the form $f = t$, where f and t are terms [3]. The specification and definition parts of a specification generated from a deterministic s-unit are elements of type $Set(CondEqn)$, where Set is a pre-defined parameterized set type.

The terms of a specification (including variables), for example, $u + next(x)$, are elements of type $Term$. The type of a term is obtained by applying the function *type* to the term. Operators and data type names are elements of type Op and $Type$, respectively.

Specification patterns are defined in terms of functions applied to elements from the above syntactic types. The following syntax checking functions will be used in the pattern we define in this paper:

base - Takes a set of conditional equations and returns the set of base (non-recursive) equations in the set.

rec - Takes a set of conditional equations and returns the set of recursive equations in the set.

$\_ \xrightarrow{-} \_ : Term\ CondEqn\ Term \rightarrow Boolean$ - The term $x \overset{f(v)=g(v')}{\longrightarrow} p$, where x, and p are

[3]For unconditional equations, that is equations with $C = true$, C is omitted

specification terms, f and g are function symbols, and v and v' are tuples of terms (function arguments), evaluates to true if and only if x is a term in the tuple v, and p is a term in the tuple v' with the same absolute tuple position as x in v. For example, $a \xrightarrow{f(a,b)=g(b+a,0)} b+a$ evaluates to true since a and $b+a$ have the same absolute tuple positions

An object in the FASB is called a *Functional Abstraction Schema* (FAS). A FAS consists of five parts: a *Specification pattern variable*, a *Other pattern variables*, a *Pattern definition* a *Descriptive narrative*, and a *Related schemas* part. The *Specification pattern variable* declares the pattern variable that represents the specification to which the pattern is to be applied. The *Other pattern variables* declares the other pattern variables, representing specification syntactic elements, that will be used to define the specification pattern in the *Pattern definition* part. All pattern variables are preceded by the symbol @ indicating that the variable is to be unified with a syntactic element of a specification. The *Pattern definition* part consists of a formula characterizing the pattern to which the narrative in the *Descriptive narrative* part applies. The *Related schemas* part names other FASs related to the pattern defined in the FAS.

An example of a FAS defining a specification pattern in which values are accumulated into a variable from a stream via a single operator is given below. The loop specification for ADDSTREAM has the pattern defined in this FAS.

Accumulator Pattern Definition
Specification pattern variable
$@Eqns : Set(CondEqn)$ such that
if $(C \Rightarrow F1(v) = F2(v')) \in Eqns$ then $(F1 = F2 = @G)$
Other pattern variables
$@t : Type$
$@u', @x' : Term$ such that $type(u') = t$; $type(x') = stream(t)$
$@op : Op$ such that op is binary; prefix, postfix, or mixfix
Pattern definition
1. For all $(C \Rightarrow E) \in rec(Eqns) : x' \xrightarrow{E} tail(x') \wedge u' \xrightarrow{E} op(u', next(x'))$
2. For all $e = (C \Rightarrow E) \in base(Eqns) : u' \xrightarrow{E} u'$

Descriptive narrative
The variable @u' accumulates values in the stream @x'
via the operator @op.
Related Schemas
Summation

In the *Specification pattern variable* part the pattern variable $Eqns$ is declared as an element in $Set(CondEqn)$, that is, $Eqns$ must be instantiated with a set of conditional equations. The restriction, introduced by the statement 'such that', states that the function symbols of the right and left hand sides of the conclusions of every equation in $Eqns$ must be the same as the pattern variable G, that is, the equations define a single function [4].

[4]It is assumed that once the function symbols are the same then both the left and right hand sides have the same number of arguments.

In *Other pattern variables* two term variables, a type and an operator pattern variable are declared. The term pattern variable u' is restricted to be of type t (also a pattern variable, that is, it has to be unified with a specification syntactic element, in this case a data type name), and x' is restricted to be a stream of elements of type t. The operator pattern variable is restricted to binary operators, which can occur in postfix, prefix, or mixfix forms.

The definition in the *Pattern definition* part determines how the accumulator specification pattern is identified. Formula 1 states that for all recursive equations in $Eqns$ the term x' in the left hand side of the conclusion is changed to the term $tail(x')$ in the right hand side, and u' is changed to $op(u', next(x'))$[5]. Formula 2 states that the term u' in the left hand sides of all base equations in $Eqns$ is unchanged in the right hand sides.

Checking for the occurrence of a particular pattern in a specification involves:

- unifying specification syntactic forms with pattern variables and
- applying pattern definitions to the specification to determine whether, under the particular unification, the specification contains the defined pattern.

As an example of how specifications are checked for patterns, consider the definition part of the while loop specification given earlier:

1. $iseos(x) \Rightarrow F(x,u,v) =< x,u,v >$
2. $\sim iseos(x) \Rightarrow F(x,u,v) = F(tail(x), u + next(x), next(x))$

To check whether the specification has the pattern defined by the Accumulator FAS we try to unify the pattern variables with specification structures that satisfy the formulas in the *Pattern definition* of the FAS. The following is the required match between the specification and pattern variables:
$Eqns \mapsto \{iseos(x) \Rightarrow F(x,u,v) =< x,u,v >,$
$\sim iseos(x) \Rightarrow F(x,u,v) = F(tail(x), u + next(x), next(x))\}$
$G \mapsto F$
$t \mapsto integer$
$u' \mapsto u$
$x' \mapsto x$
$op \mapsto +$
The loop function can thus be characterized as an accumulator.

The *Descriptive narrative* is a natural language description of the functionality associated with the pattern. The description for the above FAS is not very precise. Another pattern that replaces the *op* variable by a particular operator, for example, +, can provide a more precise definition for a specification. This suggests that FASs can be organized as a hierarchy. The *Related Schemas* part of an FAS allows us to do this.

The entry in the *Related Schemas* part of the FAS is the name of an FAS defining a pattern in which elements are summed into a variable. The Summation FAS is similar to the Accumulator FAS and differs only in that the operator pattern variable is fixed to a +

[5]Despite the prefix notation used here the definition also applies to post and mixfix equivalents of the expression since the operator pattern variable declaration placed no restriction on where the operator is to be placed

symbol, that is, it is no longer a pattern variable. The narrative description associated with the Summation FAS is more precise than that of the Accumulator FAS as a result of fixing the operator.

A FAS hierarchy consists of FASs related via the *Related Schemas* part of FASs. As one goes down a hierarchy, the descriptions become more precise since more specific features of specification patterns are identified and related to functionality.

Given a specification, a search of the FASB starts at the top level of a FAS hierarchy and proceeds as far down as possible. A search only proceeds to a related schema if and only if the specification contains the pattern identified in the current schema. If the specification does not contain the pattern then the search proceeds from the last schema encountered whose pattern occurs in the specification. If the search reaches a point in the hierarchy where it cannot proceed downwards any further, then the information contained in the *Descriptive narrative* of the FAS at that point is passed to the specifier. The specifier can then request that the search continue for other patterns, in which case the search proceeds from the next upper level schema which is satisfied by the specification (if any), treating the results of any previous searches as unsuccessful to prevent revisiting schemas.

The results of searches on the FASB provides specifiers with clues to the functionality being defined by the specification. As described above, descriptions become more precise as one goes down a FASB hierarchy, thus a search which yields FASs deep in a hierarchy will be much more useful than one which yields no FASs or high-level FASs. In such situations the specifier has to do more work in interpreting the specifications.

A prototype of the above search mechanism has been built in Prolog. The prototype FASs are defined by Prolog predicates, and s-units are represented by a set of relations.

4.2 An example analysis

Here we detail the application of our method to the ADDSTREAM s-unit:

```
Process ADDSTREAM($x$ : $in\ stream(integer)$; $z$ : $out\ integer$)
var $u, v$ : $integer$
begin
  $u := 0$;
  while $\sim iseos(x)$ do
   $Receive(v, x)$;
   $u := u + v$
  od;
  $z := u$
  $Send(z)$
end
```

In the first step of our sequential analysis of ADDSTREAM the statement $u := 0$ is mechanically translated to the specification $u_{out} = 0$. No expert analysis is required for the specification developed thus far, thus we proceed to the while statement, which we analyze by first translating its body and then applying the while translation rule to obtain a set of recursive equations.

Mechanical analysis of the loop's body results in the following specification:

$[<x_{out}, v_{out}>=<tail(x_{in}), next(x_{in})>; u_{out}=u_{in}+v_{in}]$
which translates to
$[<x_{out}, u_{out}, v_{out}>=<tail(x_{in}), u_{in}+next(x_{in}), next(x_{in})>]$

Using the loop translation rule, the following specification is obtained for the loop structure:

$[<x_{out}, u_{out}, v_{out}>= F(x_{in}, u_{in}, v_{in})]$
where
1. $iseos(x) \Rightarrow F(x,u,v)=<x,u,v>$
2. $\sim iseos(x) \Rightarrow F(x,u,v)=F(tail(x), u+next(x), next(x))$

The specification at this stage is:

$[u_{out}=0; <x_{out}, u_{out}, v_{out}>= F(x_{in}, u_{in}, v_{in})]$
where
1. $iseos(x) \Rightarrow F(x,u,v)=<x,u,v>$
2. $\sim iseos(x) \Rightarrow F(x,u,v)=F(tail(x), u+next(x), next(x))$

which reduces to the following specification by the composition translation rule:

$[<x_{out}, u_{out}, v_{out}>= F(x_{in}, 0, v_{in})]$
where
1. $iseos(x) \Rightarrow F(x,u,v)=<x,u,v>$
2. $\sim iseos(x) \Rightarrow F(x,u,v)=F(tail(x), u+next(x), next(x))$

Expert analysis of the equations defining F utilizes the Summation FAS to obtain a description of the function as a summation into the variable u. The developer can use that information and the fact that u is initially 0 to determine that the function sums the integers in the input stream x_{in}. To reflect the understanding arrived at the developer can then choose to rename F to *sum* and append a natural language description to the specification resulting in the following specification:

$[<x_{out}, u_{out}, v_{out}>= sum(x_{in}, 0, v_{in})]$
where
1. $iseos(x) \Rightarrow sum(x,u,v)=<x,u,v>$
2. $\sim iseos(x) \Rightarrow sum(x,u,v)=sum(tail(x), u+next(x), next(x))$

<u>description of sum</u>
The function **sum** *sums the integers in the stream*
x_{in} *and returns the result in* u_{out}.

Since the final value of the variable v is not relevant to the s-unit there is no statement on this aspect of the specification's functionality.

In the last step the statement $z := u$ is translated to $[z_{out}=u_{in}]$, which is composed with the specification obtained above to yield the final specification:

$[< x_{out}, u_{out}, v_{out}, z_{out} >=< sum(x_{in}, 0, v_{in}), u_{out} >]$
where
1. $iseos(x) \Rightarrow sum(x, u, v) =< x, u, v >$
2. $\sim iseos(x) \Rightarrow sum(x, u, v) = sum(tail(x), u + next(x), next(x))$

<u>description of sum</u>
The function **sum** *sums the integers in the stream*
x_{in} *and returns the result in* u_{out}.

The above specification is a documented formal specification which is made into the *detailed definition* of the extended s-unit ADDSTREAM.

The component specifier armed with the intuitive insight obtained as a result of developing the detailed specification, can reformulate the specification abstractly as follows:

$z = SUM(x)$
where for all h : *integer*; t : *integer stream*
1. $SUM(nilstream) = 0$
2. $SUM(h|t) = h + SUM(t)$

This specification becomes the *abstract definition* part of the extended s-unit ADDSTREAM. One can easily demonstrate that the abstract specification is consistent with the detailed specification in this case. In general, the relationship may not be as obvious and may require the use of suitable formal verification techniques.

The above combination of natural language descriptions and formal detailed and abstract specifications should provide a potential component user with the means to comprehend an s-unit and to rigorously investigate its functional properties.

5 Conclusion and further work

The generation of understandable specifications from code is difficult to automate mainly because code does not directly reflect the abstractions used in their development. From our observations, the generation of understandable specifications from code requires the use of a significant amount of expert knowledge. Strictly formal analysis techniques are not always adequate, since they often do not allow for the development of intuitive expressions of functionality. This has led us to consider a pattern-driven approach to code analysis. The analysis method we propose in this paper supplements a formal analysis technique with facilities to assist in the creation of more intuitive descriptions of functionality.

The successful development of a system based on our model depends to a large extent on the ability to recognize, and represent expert knowledge required to extract abstract specifications from code and/or specifications. We have indicated in this paper how such knowledge can be represented and used. We are currently developing suitable knowledge representations and organizational structures for a knowledge-based system for code analysis. Such a system will provide organizations for identifying analysis related knowledge, and creating representations of such knowledge, as well as organizations for retrieving the representations when required. Expressing patterns will require some training in the formalism, but the use of syntax checking functions that relate directly to specification syntactic concepts should make learning easier with experience.

As a component developer gains experience in creating reusable components from code structures, a number of additional pattern/description relationships may become apparent. Also, more precise descriptions of existing FASB patterns may evolve from a developer's experience. Mechanisms are needed for incorporating the results of such experience in the FASB.

We are also investigating the possibility of automated expert support for the abstraction phase of our specification method, and the availability of suitable automated verification techniques to support consistency checks between abstract and detailed specifications.

Acknowledgements

The work described in this paper was spawned from the work of Salwa K. Abd-El-Hafiz on a code analysis prototype. Thanks to Marvin Zelkowitz, Gianluigi Caldiera, and Lionel Briand for their valuable comments on the research reported here. Thanks also to John Gannon, Gianluigi Caldiera, John Bailey, Lionel Briand, Salwa Abd-El-Hafiz, and Sandro Morasca for reading early drafts of this paper.

Appendix: Relational semantics concepts and notation

In what follows $x, x_1, x_2, \ldots,$ are *s-unit variables*, v is a *data state*, v_{in} is an *input data state*, v_{out} is an *output data state*, and $e, e_1, e_2, \ldots,$ are *function terms*. A function term is an expression built using only function and variable symbols, for example, $(x + y)/z$ is a function term built from the function symbols $+$ and $/$, and the variables x, y, z. Note that a value is also a function term since it can be represented by the term that reflects its generation. For example, the integer value 2 is represented by the term $succ(succ(0))$ where *succ* is the successor function, and 0 is the zero constant (a function with an empty domain) as defined in a specification characterizing integers.

A *data state* is a variable indexed vector written as follows:
$< x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n >$
where x_i is the variable index (an s-unit variable) and e_i is the term currently held by the variable. The value $\perp$ will be used to indicate that a variable has not been assigned a defined value. A variable is said to be *defined* in a state if it is assigned a defined value, that is, if it is not assigned the value $\perp$. A state in which all variables are not defined is said to be undefined, and is itself denoted by $\perp$. States of an s-unit P are characterized by an algebraic specification denoted $STATE_P$. The elements of type *state* provided by the semantic model of $STATE_P$ are data states of the form given above. Function terms are elements of type *fterm* in $STATE_P$, s-unit variables are elements of type *variable*, and the union of the data domains making up the state is denoted by the type *data*. The following functions on states are defined in $STATE_P$:

Data state access : The expression $v[x]$ is the value of the variable x in data state v. For convenience we denote the *value* of a variable, x, in the state v by x_v. In particular, the value of the variable x in the input (output) state v_{in} (v_{out}) is simply denoted by x_{in} (x_{out}).

Term evaluation : *eval* : *fterm state* $\rightarrow$ *data*
The expression $eval(e, s)$ is the result of evaluating the function term e in the data state s. Evaluation of a function term in a state s involves substituting for the variables in the function term their assigned values in the data state.

State assignment : $\_[\_ \leftarrow \_]$: *state variable fterm* $\rightarrow$ *state*
The expression $s[x \leftarrow e]$ represents the state resulting from assigning the value e to the variable x in state s. If there is no such variable in s then s is left unchanged. In what follows we use a more convenient form of state assignment, which allows us to state changes on more than one variable in an input or output state. For example a group of state assignments
$v_{out} = (v[x1 \leftarrow e1], \ldots, v[xn \leftarrow en])$
(where each $x1, \ldots, xn$ is unique) is conveniently expressed as:
$< x1_{out}, \ldots, xn_{out} >=< e1, \ldots, en >$
where it is understood that all variables not mentioned are unchanged.
Similar substitution operations are defined for function and boolean expression:
$E[x \leftarrow Y]$, where E and Y are expressions, denotes the expression formed by substituting Y for all (free) occurrences of the variable x in E. The following *syntactic* substitutions will be used:

$e[v \leftarrow v_{in}]$: subscript all the variables, v, in e, with *in*. A similar definition applies when v_{out} is substituted for v_{in}.
$e[v_{in} \leftarrow v]$: remove the subscripts of the variables of the form, v_{in}, in e. A similar definition applies when v_{out} is substituted for v_{in}.

Data types are specified algebraically. In our method, we express data type specifications in the executable language OBJ3 [10], thus allowing us to carry out some mechanical reductions on data type representations. An example of such a specification is given below.

```
obj ARRAY[INDEX VALUE] is sort Array .
  *** Constructors
  op nilar : - > Array .
  op put : Array Index Value - > Array .
  *** Auxiliary functions
  op _[_] : Array Index - > Value .
  *** Error function
  op undef : Index - > Value .
  *** Axioms
  var A : Array . var E : Value . var I I' : Index .
     eq put(A, I, E)[I'] = if I == I' then E else A[I'] fi.
     eq nilar[I] = undef(I) .
```

The specifications INDEX and VALUE in the above array specification characterize the index and value parameters of an array, respectively, for arrays. For example, ARRAY[NAT

INT] is a specification of integer arrays indexed by natural numbers. The constructors *nilar* and *put* create arrays, for example, $put(nilar, I, E)$ creates an array by assigning the value E to the location I of the uninitialized array *nilar*. The auxiliary function _[_] returns the value stored in the given location, for example, $A[I]$ returns the value stored in location I of the array A. The error function, *undef*, returns a value which indicates that the array location has not been assigned a value.

References

[1] S. K. Abd-El-Hafiz. A tool for understanding programs using functional specification abstraction. Master's thesis, University of Maryland at College Park, 1990.

[2] S. K. Abd-El-Hafiz, V. R. Basili, and G. Caldiera. Towards automated support for extraction of reusable components. *To be published in conference proceedings*, 1991.

[3] D.A. Achmidt. *Denotational Semantics: A methodology for language development.* Allyn and Bacon, Inc., 1986.

[4] V. R. Basili and H. D. Mills. Understanding and documenting programs. *TSE*, SE-8(3), 1982.

[5] V. R. Basili and H. D. Rombach. Support for comprehensive reuse. Technical Report UMIACS-TR-91-23, CS-TR-2606, Department of Computer Science, University of Maryland at College Park, 1991.

[6] S. K. Basu and J. Misra. Proving loop programs. *TSE*, SE-1(1), 1975.

[7] P.T. Breuer and K. Lano. REDO at Oxford. In *Utrecht Reuse workshop 89*, 1989.

[8] R. Brooks. Towards a theory of comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18, 1983.

[9] G. Caldiera and V. R. Basili. Reengineering existing software for reusability. Technical Report UMIACS-TR-90-30, UMIACS, 1990.

[10] J. A. Goguen and T. Winkler. *Introducing OBJ3.* SRI International, 1988.

[11] E. C. R. Hehner. Predicative programming:part 1. *CACM*, 27(2), 1984.

[12] B. G. Joo. *Adaptation and Composition of Program Components.* PhD thesis, University of Maryland at College Park, 1989.

[13] S. Letovsky. Cognitive process in program comprehension. In *Empirical Studies of programmers.* Ablex, 1986.

[14] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), 1986.

[15] H. D. Mills. The new math of computer programming. *CACM*, 18, 1975.

[16] J. Misra. Some aspects of the verification of loop computations. *TSE*, SE-4(6), 1978.

[17] S. S. Qian. A tool for understanding software components. Master's thesis, University of Maryland at College Park, 1989.

[18] E. Soloway and W. L. Johnson. Proust: Knowledge-based program understanding. *TSE*, SE-11(3), 1985.

[19] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[20] M. Ward. Transforming a program into a specification. (88/1), 1988.

[21] R. Waters. A method for analyzing loop programs. *TSE*, SE-5(3), 1979.

[22] M. Wirsing, R. Hennicker, and R. Breu. Reusable specification components. In *Mathematical Foundations of Computer Science, LNCS 324*. Springer-Verlag, 1988.

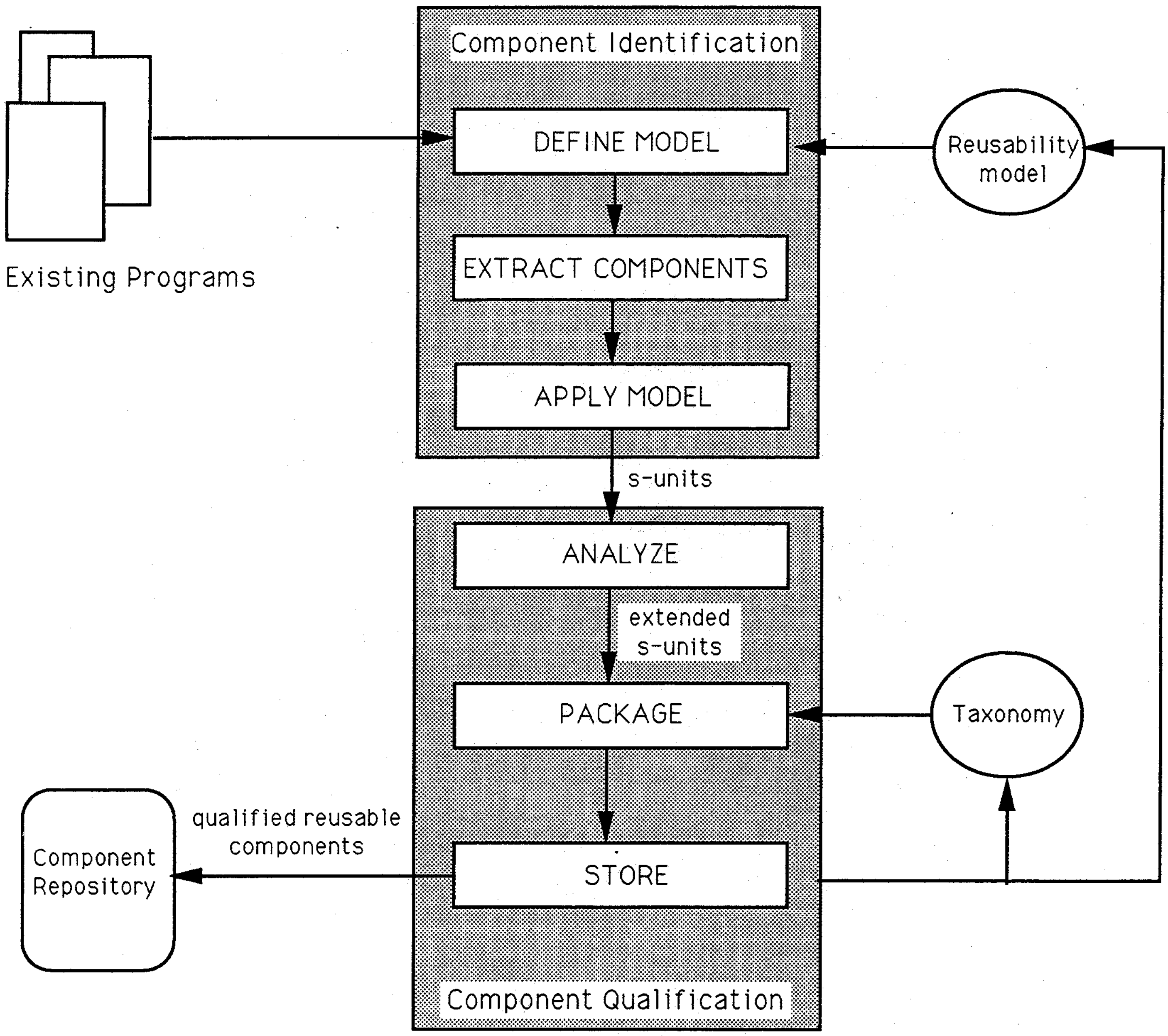

Figure 1: The reuse reengineering model

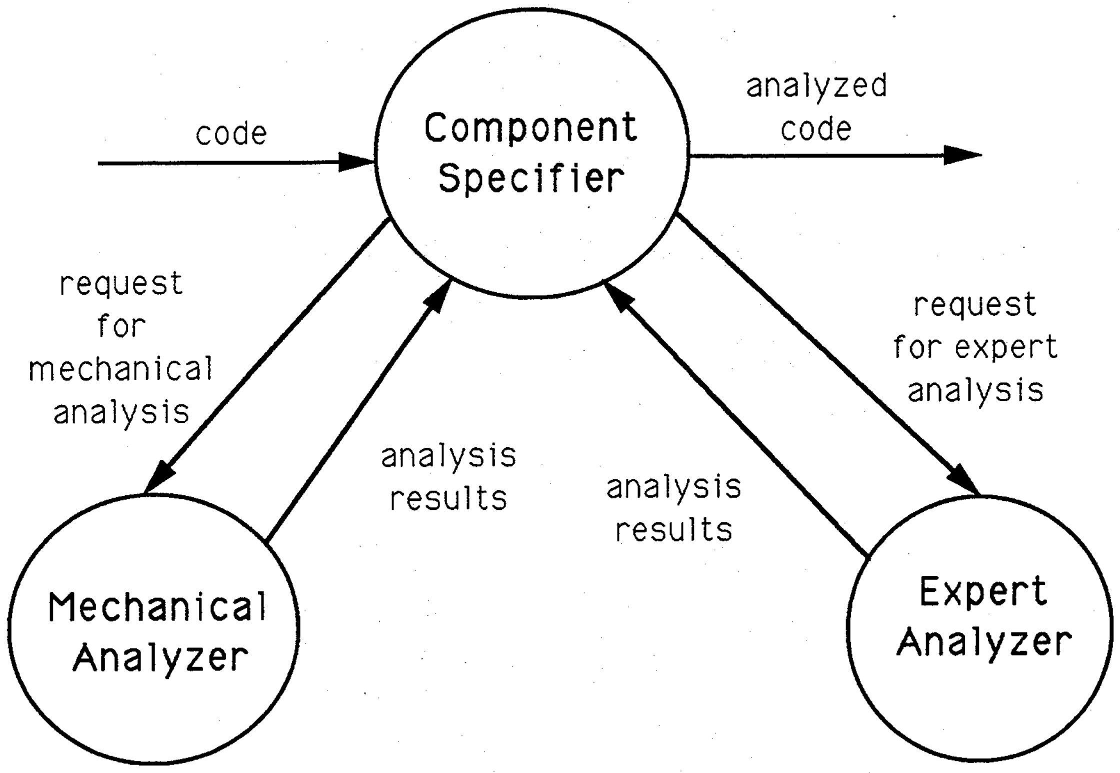

Figure 2: Conceptual model of Code Analysis

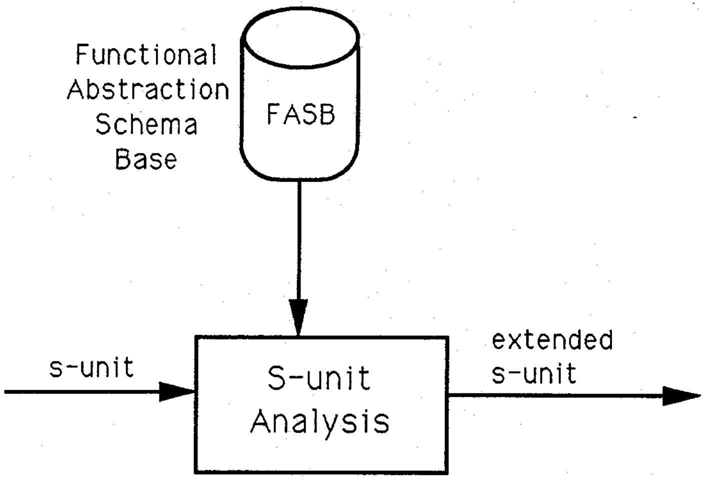

Figure 3: A pattern-driven code analysis method

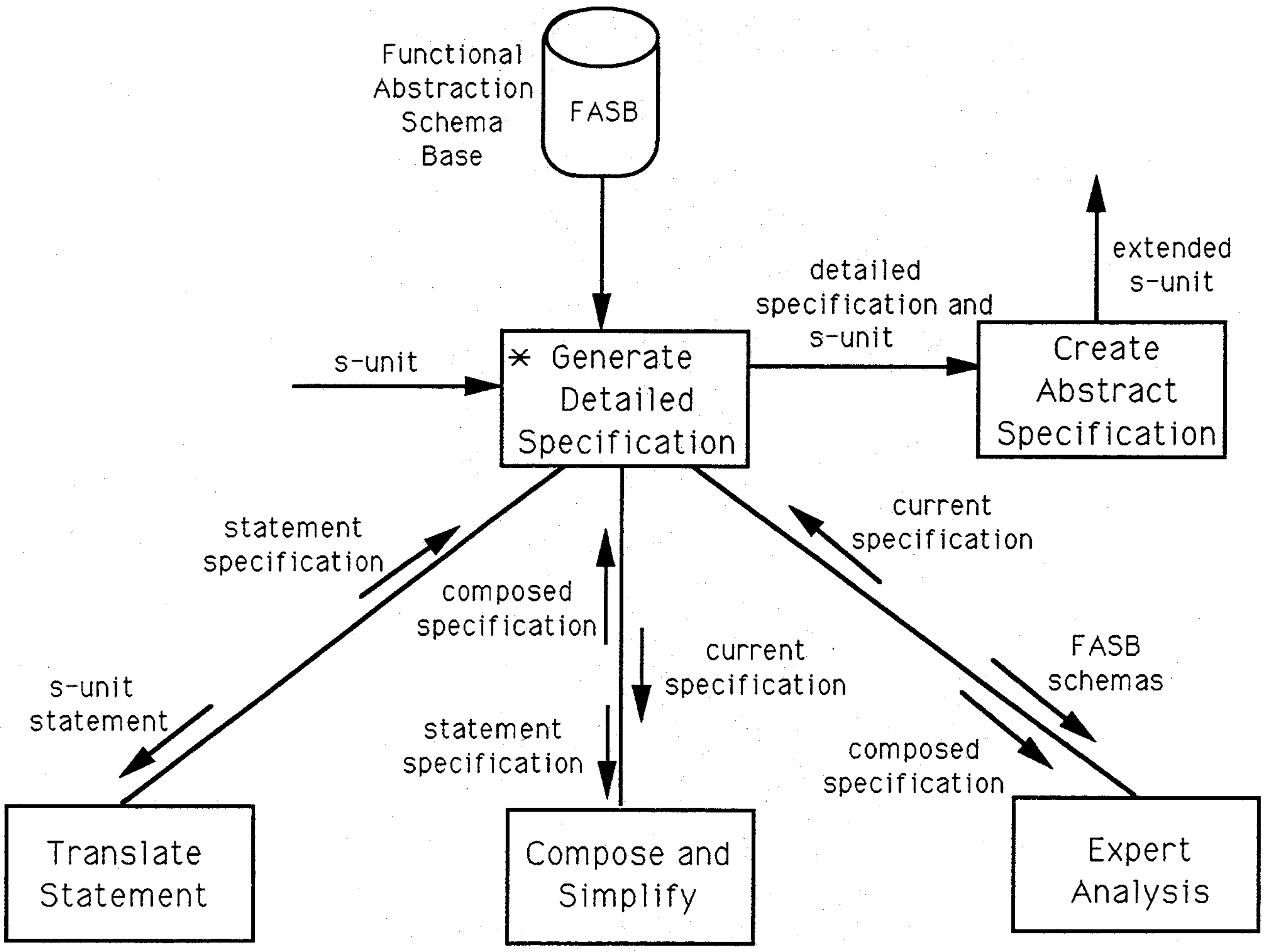

Figure 5: A ·detailed picture of s-unit analysis