Software Reuse using Formal Specification of Requirements

by

Jeyashree Chidambaram

A thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfilment of the Requirements
for the degree of

MASTER OF SCIENCE

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

SOFTWARE REUSE USING FORMAL SPECIFICATION OF REQUIREMENTS

by

JEYASHREE CHIDAMBARAM

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree

MASTER of SCIENCE

Abstract

Reuse can be attempted at any stage in a software life cycle. However, reuse will be more effective at a higher level of abstraction such as requirements specification. The reason is that, one can easily understand the functionalities of a reusable component when it is abstractly specified and can also justify that the component is indeed reusable. Software can be reused if and only if its structure and behavior are compatible with those of another software that is being developed. In this thesis, a methodology is given to ensure structural compatibility in software reuse using the formal requirements specification of the software to be developed and that of the software to be reused. Algorithms to implement the methodology are given and a partial implementation of the methodology is illustrated through a case study. The formal notation Z is used in this thesis.

Acknowledgements

First, I would like to thank my supervisor, Dr. Kasilingam Periyasamy for showing me the right directions in research; needless to mention his pleasing smile and motivating ideas. My thanks also goes to all my friends who helped me in times of need. Finally, my thanks goes to my parents and family for their emotional support and encouragement all through. This work was partially supported by a grant from Natural Sciences and Engineering Research Council, Canada.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software reuse is the process of using existing software artifacts with little or no modification to build new software. The artifacts include code, design, specification and documentation. Software reuse promises advantages such as reduction in development time of new software, reduction in cost, and increase in the productivity of a software organization significantly.

Reuse can be attempted at any stage in the life cycle of a software. However, reuse will be more effective at a higher level of abstraction in the software life cycle, the reason being as follows. One of the important requirements for software reuse is that a developer must understand the functionality of a reusable component before actually reusing the component. This situation warrants that the reusable component must be accompanied by precise and unambiguous description of its functionality or must itself be self-descriptive. Even though code reuse has been in practice for several decades, understanding the functionality of a piece of code is a challenge for programmers. One can easily understand the functionalities of a reusable component when it is abstractly specified; in addition, one can also justify the reusability of the component.

Previous research on software reuse mainly concentrated on reusing code [2, 18]. Design reuse is also practiced by some industries, particularly when a new version of a software product is being developed while its previous version is currently in use in the software community. In [2], code reuse has been illustrated through libraries. With code reuse, it is practically impossible to identify a reusable component based only on the code. Often reusable code is accompanied by an informal documentation; however, this documentation in general, is inadequate to understand the intended

functionality of the accompanying code [1, 3]. In [1, 2], Basili and his colleagues have introduced a comprehensive framework of models, schemes and support mechanisms for code reuse. They concentrated more on selecting appropriate data structures to store, maintain and retrieve reusable components. However, the work does not reflect any methodology for reuse and seems to have evolved out of experience.

Maiden [18] proposed an alternative paradigm to support software reuse through requirements analysis. The paradigm is founded on a descriptive model of analogy in software engineering problems, which suggests critical determinants of software engineering analogies. He claimed that analogy permits the use of entire specifications as a basis for the development of new specifications and supports reuse across different domains. Analogical reasoning has the potential to exploit specifications representing a wide variety of applications such as those held in CASE repositories. In [18], two applications - an air traffic control system and a flexible manufacturing system, were taken as examples; though the two domains appear to be quite different, the analogy was constructed to maximize potential reuse between specifications of the two examples. In this approach, software reuse is achieved using expert systems but no formal justification of reuse was given.

Although Maiden's approach seems to be quite promising, there were some constraints in the approach because reuse using specifications should describe a more determined theory of software engineering problems than that proposed by existing models of analogy. A specification should assist an analyst by providing domain and method knowledge in a form that is easily comprehensible and exploitable.

Kedar-Cabelli [16] proposed *purpose* as a major constraint on analogical mappings. Her purpose-directed model of analogy only mapped features of domains which justified the analogous goals of those domains. The aim of software engineering analogies is to support reuse through the transfer of concepts (processes, sources, data stores) from the reusable specification to a target specification. These concepts are modeled in such a way that they support the *goals* and *purposes* of the system. Hence, *purpose* would appear to be a major constraint on analogical mappings.

Reusing specifications during the requirements phase has been discussed in [9, 11]. A few concrete conclusions have been arrived at about the nature of the process of specification reuse. Existing paradigms of software reuse include keyword-based retrieval, faceted classification schemes, formal approaches, instantiation of generic objects and domain analysis, but none appear to be suitable for specification-level reuse. Some of the problems associated with the above-mentioned approaches are as follows:

- The use of simple keywords that retrieve and select software-components based on their functionality is a much vaunted approach to software reuse. However, software engineering problems are too complex to be described comprehensively using keywords.

- Software reuse in well understood domains has been achieved through domain analysis, during which an experienced domain analyst constructs a model of the problem domain. The domain is modeled as a transformation of rules, which are applied to refine a specification or design. Domain analysis is intended to support many instances of reuse in a single domain. However, requirements analysis often takes place in poorly understood domains. Keywords describing the functionality or the surface features of systems cannot support reuse across different applications.

- Faceted classification schemes overcome several problems of simple keyword retrieval by describing non-functional features of a software component and providing a complex lexicon to support individual differences in software reuse terminology. However, development of a comprehensive classification scheme still requires difficult and time-consuming domain analysis of each application across which reuse is intended to occur.

- Reuse through generic objects is another much vaunted approach to software reuse. In this approach, the requirements analyst uses a template solution or specification of the type of system under analysis. However, research has shown that abstract specifications and solutions are difficult to understand [29]. In addition, constructing template solutions that are both beneficial to the analyst and sufficiently, abstract to be of benefit during reuse was found to be a difficult, if not fruitless, task.

- In [5], a formal approach was used for software reuse which uses pre-conditions and post-conditions, and assertions about component properties. The resulting proof developed a list of candidate components. However, determination of pre-conditions, postconditions and assertions about reusable specifications requires additional knowledge about each application, and specifications represented using structured analysis techniques are inherently formal.

Jeng and Cheng [14, 15] have also used analogy for specification reuse but they have included automated reasoning by which reusable components are selected from a hierarchy of software components. Following the concept of analogy, an approach has been suggested to modify a reusable component at the specification level instead of at the code level. Specification reuse involves transfer of a network of domain and method knowledge. This approach suggests that reuse of specifications during

requirements analysis is one form of analogous problem solving. In this case, the dissimilarities between the specification of existing software and the specification of the new software are determined at an early stage. The changes in specification drive the corresponding modifications at the code level.

Reuse using specifications involves two problems: retrieving the correct specification and developing and customizing that specification to fit the new domain. Most research into software reuse has investigated the first problem. Information retrieval methods based on the analyses of natural language documentation have been proposed for constructing software libraries. However software components represented by natural language may hinder the retrieval process due to the problems of ambiguity, incompleteness, and inconsistency inherent to natural language. Correct customization of a specification, however, is also critical for successful reuse.

All of the above mentioned problems can be minimized by using formal specifications to represent software components. Using formal specifications, reusability of a software component can be justified because formal specifications characterize the functionality of the software more precisely and unambiguously. In addition, the well-defined syntax of a formal notation also makes processing amenable to automation. The work reported in [14, 15, 31] as well as the work presented in this thesis all fall under this category. While the work in [14, 15, 31] use algebraic and Larch specification languages, the work reported in this thesis uses the model-based specification notation Z. Jeng and Cheng [14, 15] talk about specification reuse. This is dependent on a subsumption algorithm and a cluster algorithm developed by Jeng and Cheng. Wing and Zaremski [31] uses formal specifications to describe the behavior of software components, and hence to determine whether two components match. The formal specifications of components are written in terms of pre- and post- condition predicates. Theorem proving is used to determine match and mismatch.

We have used the formal requirements specification of a software product that is to be developed (hereafter referred to as *target software*) and that of a reusable component (hereafter referred to as *library software*) as candidates in this thesis. It is assumed that both specifications are written using the same formal notation so that reasoning becomes simpler. The candidate components that are analogous to the target specification are retrieved from the hierarchy of reusable software components. A retrieved component is compared to the target specification to determine what changes needed to be applied to the corresponding program component in order to make it satisfy the target specification. We have used the Z formal notation in this thesis . The notations of Z are concrete and are currently being standardized and hence the proposed methodology is amenable to automation.

A specification in Z consists of a structure and behavior. Structure of a specification in Z can be identified by the set of declarations while behavior is described by a set of operations and functions defined on the structure. Complete details of the Z notation are not given in this paper; refer to [27] for full details. For ease of reference, a list of Z notations used in this thesis is given in Appendix A.

Our work is confined to the specification level. This enhances maintenance and error-free reusability. The use of formal methods enables automation and, the discrepancies that creep in when expert systems are used for matching, are thus avoided. The thesis claims that if correct specifications for library and target software are given, then the library software can be reused in place of the target software if there exists a compatibility between them. Only structural compatibility is described in the thesis. The notion of behavioral compatibility is briefly described but has not been implemented. The work includes exact and partial matches of structural compatibility.

The rest of the thesis is organized as follows. Chapter 2 explains the basic concepts of formal methods and the Z notation. Chapter 3 describes our new methodology for software reuse with algorithms. In Chapter 4, we illustrate the algorithms through a case study and its implementation. Chapter 5 gives a critical analysis of the methodology based on the case study. Some limitations of the current work are described in Chapter 6. Finally, Chapter 7 concludes our study and discusses future research directions.

Chapter 2

The Z Notation

The formal notation Z is based on set theory and first order predicate calculus and was developed at Oxford University U.K. Currently, the Z notation is undergoing the process of standardization. The Z notation has proved to be an useful tool for developing correct and reliable designs. A specification in Z can be viewed as consisting of two components: structure and behavior. Structure of a software component in Z can be identified by schemas and global declarations while its behavior is described by a set of functions and operations defined on the structure. In addition to data type declarations, the structure also includes invariants which are specified as constraints on the data types. Structural compatibility for software reuse can be ensured by comparing the structures of library and target software and finding exact and partial matches. Behavioral compatibility is briefly mentioned, but not further elaborated in this thesis.

2.1 Basic Z Definitions

There are five types of declarations in Z that we use to ensure structural compatibility: *basic type definition, global declaration, schema declaration, abbreviation* and *free type definition.* Below we give the definitions and informal semantics of these declarations. These definitions have been directly extracted from [27].

type A type in Z is an expression of a restricted kind. The value of a type is a set called *carrier set* [1] of the type. Any object said to be belonging to a type is in fact a

[1] In Z, types are treated as sets. The carrier set of a type τ is the maximal set of values associated

member of the carrier set for that type. The three built-in types in Z are $\mathbb{N}$(natural numbers), $\mathbb{N}_1$(positive numbers) and $\mathbb{Z}$(integers).

basic type A specification may contain several basic types which are assumed to be elaborated further during refinement of the specification. For example, a specification for a course registration system may have *COURSE* as a basic type and it is given by the syntax

$$[COURSE]$$

schema type A schema type in Z is denoted by

$$(\!| p_1 : t_1;\ \ldots;\ p_n : t_n |\!)$$

where $p_1, \ldots, p_n$ is a set of identifiers and $t_1, \ldots t_n$ denote a set of types. An instance of this schema type is a binding

$$z = (\!| p_1 \Rightarrow x_1, \ldots, p_n \Rightarrow x_n |\!)$$

between the names $p_1, \ldots, p_n$ and the objects $x_1, \ldots x_n$. These objects are drawn from the types $t_1, \ldots t_n$ respectively. Under the binding z, the component $z.p_i$ is equal to the object x_i for all i from 1 to n.

signature A signature is a collection of variables, each associated with a type. Thus,

$$x : \mathbb{N};\ y : \mathbb{Z}$$

is a signature with variables x and y associated with types $\mathbb{N}$ and $\mathbb{Z}$ respectively. Every signature can also be thought of as equivalent to a schema type. Thus the above signature is equivalent to the schema type

$$(\!| x : N;\ y : \mathbb{Z} |\!)$$

The advantage of thinking of a signature as a schema type is to express some properties of the signature using the bindings of the schema.

property A property of a signature is expressed by a predicate and is characterized by the set of bindings under which the predicate is true. Thus, the predicate $x < y$ is a property of the above signature. Notice that it is possible to write the predicate in more than one way expressing the same property. For example, the predicate $y > x$ expresses the same property of the signature. Consequently, two predicates over a

---

with τ. The values in the carrier set of a given type are regarded as atomic objects. See [32] for more details.

signature are said to be *logically equivalent* if both express the same property of the signature.

global declaration A global declaration is a signature along with a property over the signature. The scope of a global declaration extends from the point of declaration till the end of the current specification.

schema A schema is a signature together with a property over the signature. The scope of each variable in the signature of the schema is the schema itself. The property of a schema may use global declarations defined earlier to the schema and in this case, the property of the schema is a conjunction of the property expressed within the schema and the property of the global declaration. A variable within the schema having the same name as that of a global variable, takes precedence over the latter. In effect, this variable hides the global variable of the same name and hence any reference to this name within the property of the schema always refers to the schema component and not the global variable.

free type definition The free type definition in Z is used to define recursive data types such as lists and trees and enumerated data types. Following are two examples which illustrate this:

$$List ::= nil \mid cons\langle\!\langle \mathbb{N} \times List \rangle\!\rangle$$
$$Status ::= Ugrad \mid Grad \mid Staff$$

abbreviation An abbreviation introduces a new name for an expression. The idea is to use the name as a short hand for the expression and hence abbreviation is generally used to rename complex expressions. A type abbreviation is used to define user defined types by composing previously defined types in a specification. The scope of the name introduced through abbreviation definition extends from the point of declaration to the end of the current specification. See below for an example of type abbreviation:

$$Student == Name \times IDNumber \times \mathbb{P}\ Course$$

Among the five categories of declarations mentioned earlier, only schemas and global declarations constitute the structure of a specification.

2.2 Structure of software in Z

The structure of a software in Z can be identified by state space declarations and global variable declarations. A state space is described by one or more schemas. A schema contains a unique name in the specification, a signature and optionally a property over the signature. A signature is represented by a set of <name, type> pairs and the property of the signature is represented by a predicate. This predicate is expressed as a logical conjunction of well-formed formulas. As an example, a schema representing the structure of a course in a course registration system is given below:

Course

$courseno : CourseNumber$
$start, end : Time$
$registered : \mathbf{P}\ StudentNumber$
$allocated : \mathbf{N}_1$

$start < end \land$
$\#registered \leq allocated$

In this case, the signature consists of the declarations
$courseno : CourseNumber$,
$start, end : Time$,
$registered : \mathbf{P}\ StudentNumber$,
$allocated : \mathbf{N}_1$

The notation $\mathbf{P}$ denotes a power set. The property of this schema is represented by the conjunction of predicates

$start < end$,
$\#registered \leq allocated$

which asserts that (i) the start-time of the course must be less than the end-time of the course and (ii) the number of students registered in a course must be less than or equal to the total number of seats allocated for that course. According to the convention of a Z specification, there exists an implicit conjunction between predicates representing the property, when these predicates are written on several lines. Thus the property of this schema is logically equivalent to

$start < end \land \#registered \leq allocated.$

A global variable declaration in Z contains a signature representing one or more global variables and their respective types. The declaration may optionally contain a predicate representing some invariant properties of the signature. For example, the maximum number of courses offered can be defined by a global constant *maxcourses* as follows:

$$maxcourses : \mathbb{N}_1$$

We can define the following abbreviations also - Hour (0 - 23), Minute (0 - 59), Time (a cross product of Hour and Minute), CourseNumber (100 - 700), StudentNumber (100000 -999999).

$$Hour == 0..23$$
$$Minute == 0..59$$
$$Time == Hour \times Minute$$
$$CourseNumber == 100..700$$
$$StudentNumber == 100000..999999$$

Chapter 3

The methodology

In this chapter, we discuss the methodology to ensure structural compatibility. We use the term "library software" to represent the reusable component and "target software" to represent the component being developed. Software reuse can be ensured only if the structure and behavior of the library software (to be reused) match with those of the target software (to be developed). In this process, structural compatibility precedes behavioral compatibility because the latter uses the former. We consider only the details of structural compatibility in this thesis. We have introduced algorithms to ensure structural compatibility and have also implemented these algorithms towards automation of the whole process. We briefly describe behavioral compatibility but do not elaborate on it further. This is because automating the process of ensuring behavioral compatibility requires the use of term rewriting rules and human interaction, which are beyond the scope of the current thesis.

Given the requirements specifications of the library and target software in Z, we extract the structure of each of them along with the type information to determine structural compatibility. This chapter discusses the methodology with its theoretical foundation, algorithmic details and its partial implementation.

The structure of a software component is generally given by the collection of schemas more commonly known as the state space, in the specification. An important part of a signature is the set of properties that are defined over the signature. These properties are expressed as invariant conditions which assert certain relationships among the data type declarations of the signature. Later sections in this chapter provide examples to illustrate these concepts. Structural compatibility is ensured by the extent of signature matching in two specifications; signature matching includes

both declaration match and property match. Often, finding an exact match of declarations and/or properties is not feasible. Therefore, we need to consider partial matches as well. We have described both exact and various categories of partial matches in this thesis.

For purposes of structural compatibility, we consider the process of matching two state space declarations as the same process of matching two global variable declarations. The difference between the two types of declarations is that the former is associated with a unique name and is generally defined by a schema type. Our notion indicates that the name of a schema does not play any role in ensuring structural compatibility except for identifying the declaration; we use internal identifiers for global variable declarations within the algorithms that implement our methodology. For simplicity,we use the term *declaration match* to refer to the process of matching data type declarations in a pair of signatures and matching their respective properties. Thus, *declaration match* can be used for matching state space declarations as well as global variable declarations. Below, we discuss the various types of matching in detail and the notations used in the definitions that follow:

3.1 Notations used

The following notations are used in the definitions that follow.

$< \sigma, P >$	-	σ refers to a signature and P refers to the property over σ .
n	-	set of identifiers
t	-	set of type names
τ	-	type name
$f(x)$	-	bijection from the domain of a signature σ_1 to the domain of another signature σ_2
$p_1, p_2, \ldots, p_m$	-	well formed formulas
$F(p_i)$	-	Set of free variables of p_i
Op_l	-	Operation in the library component
Op_t	-	Operation in the target component
S_l	-	State Space of the library component
S_t	-	State Space of the target component

3.2 Definitions

3.2.1 Declaration Match

Definition 1[declaration]: *A declaration d is defined to be an ordered pair* $d : < \sigma, P >$ *where* σ *represents the signature of d and P refers to the property over* σ.

Since the property P is always associated with the signature σ, we use the notation $P(\sigma)$ to denote the P component of d in this thesis.

Definition 2[signature]: *A signature* σ *is defined as a partial function* $\sigma : n \nrightarrow t$, *where n represents a set of identifiers and t represents a set of type names.*

An example of a signature is as follows:
$id : \mathbb{N}_1$
$name$: *STRING*
$age : \mathbb{N}_1$

While matching declarations, we have to consider two cases: *exact match* and *partial match.* This classification applies individually to signatures and properties.

Definition 3[declaration exact match]: *A declaration* d_1 *exactly matches with another declaration* d_2 *if and only if the signature* σ_1 *of* d_1 *exactly matches with the signature* σ_2 *of* d_2 *and the property* $P(\sigma_1)$ *exactly matches with the property* $P(\sigma_2)$.

Definition 4[declaration partial match]: *A declaration* d_1 *partially matches with a declaration* d_2 *if and only if one or both of the following conditions is true:*

- *The signature* σ_1 *of* d_1 *partially matches with the signature* σ_2 *of* d_2.
- *The property* $P(\sigma_1)$ *partially matches with the property* $P(\sigma_2)$.

3.2.2 Signature Match

Signature matching is the process of determining which library components "match" a target signature. As is the case with any information retrieval method, it becomes difficult to find an exact match always. Under such situations we go for a partial

match based on type compatibility. The expectation is that relaxed matching returns components that are close enough to be useful to the software developer.

Definition 5[signature exact match]: *A signature σ_1 exactly matches with another signature σ_2 if and only if for every name n_1 in the domain of σ_1, there is exactly one name n_2 in the domain of σ_2 such that $\sigma_1(n_1) = \sigma_2(n_2)$. The ordering of names in either signature is immaterial.*

A necessary condition for an exact match of two signatures is that the number of names in both signatures must be the same (i.e., cardinality of domain of σ_1 = cardinality of domain of σ_2).

A consequence of definition 5 for exact match is that there may be more than one possibility for exact match between two signatures. See the two signatures given below for an example:

User
id : $\mathbb{N}_1$
name : *STRING*
age : $\mathbb{N}_1$

Book
callno : $\mathbb{N}_1$
title : *STRING*
year : $\mathbb{N}_1$

In this case, *id* of *User* can match with either *callno* or *year* of *Book*, giving two possible exact matches.

We next consider partial match between two signatures. In defining partial matching, we use the notion of type compatibility; this is defined first.

Definition 6[type compatibility]: *A type τ_1 is compatible to a type τ_2 if the result of performing some operation on a member of the carrier set of τ_1 is the same as the result of performing one or more operations on a unique member of the carrier set of τ_2.*

Informally, type compatibility between two types indicates that the functional behavior of both types as determined by their carrier sets and operations on carrier sets

$\mathbb{N}$	$\mathbb{Z}$
$\mathbb{N}_1$	$\mathbb{N}$
$\mathbb{P}\, X$	iseq X
iseq X	seq X
$\text{seq}_1\, X$	seq X
$\leftrightarrow$	$\mathbb{P}(\_ \times \_)$
$\rightarrow$	$\leftrightarrow$
$\twoheadrightarrow$	$\leftrightarrow$
$\rightarrowtail$	$\leftrightarrow$
$\rightarrowtail\!\!\!\!\twoheadrightarrow$	$\leftrightarrow$
$\rightarrow\!\!\!\!\!\!\bullet$	$\leftrightarrow$
$\twoheadrightarrow\!\!\!\!\!\!\bullet$	$\leftrightarrow$
$\rightarrowtail\!\!\!\!\!\!\bullet$	$\leftrightarrow$
bag X	$X \twoheadrightarrow \mathbb{N}$

Table 3.1: Compatible Types in Z

must be equivalent. A necessary condition for ensuring type compatibility is that the carrier set of τ_1 must be a subset of the carrier set of τ_2, both sets being derived from the same semantic universe. Thus, a type τ_1 can be ensured to be compatible to a type τ_2 by taking an element x from the carrier set of τ_1, performing one of the operations associated with τ_1 on x and showing that the result y is the same as performing one or more operations associated with τ_2 on x.

To illustrate, consider the types P X and *iseq* X(injective sequence). Since *iseq* is injective, the sequence will not contain duplicates. The operator # will return the length of an injective sequence in the same way as it returns the cardinality of a power set. The result of union operation on two sets is behaviorally equivalent to performing concatenation operation on two injective sequences. In a similar way, behavioral equivalence for other operations associated with the two types can be shown. Hence, P X is compatible with *iseq* X. Table 3.1 shows a list of compatible types in Z; the type in column 1 is compatible with the type in column 2. This table can be extended dynamically for other types defined in a specification.

The following properties hold regarding type compatibility:

- Every type is compatible with itself.
- Type compatibility is transitive.
- Type compatibility is antisymmetric.

A close observation of these properties reveals that the type compatibility relation between two types is a partial order. We caution the readers that *type compatibility* as described in this paper is not the same as *type abbreviation* as defined in [27]. The purpose of *type abbreviation* is to derive an equivalent type t_2 for a given type t_1. For example, the relational type (denoted by A $\leftrightarrow$ B) is equivalent to $\mathbf{P}$(A x B). Our notion of type compatibility is targeted towards reuse.

Subtypes Consider the notion of *subtype* as given in [5]. A requirement for a type τ_1 to be a subtype of another type τ_2 is that every property ϕ provable for objects belonging to τ_2 should also be provable for objects belonging to τ_1. To summarize, objects of τ_1 ought to behave at least the same manner as those of τ_2.

¿From the notion of subtype, every property ϕ for objects of τ_2 also holds for objects of τ_1. This is possible only if both objects belong to the same domain (or drawn from the same set). Thus, the first condition for type compatibility is satisfied. We can prove that a property ϕ holds for an object x of τ_2 only by invoking some operation ι on x and observing the result. Therefore, for the same property to hold on x when considered to be a member of τ_1, a set of operations from τ_1 has to be invoked and the result has to be compared. Thus, the second condition for type compatibility is also satisfied. Hence, a subtype is compatible to its super type.

Using type compatibility, we can now define a partial match between two signatures.

Definition 7[signature partial match]: *A signature σ_1 partially matches with another signature σ_2 if there exists a bijection f from the domain of σ_1 to the domain of σ_2 such that for every x in the domain of σ_1, $\sigma_1(x)$ is type-compatible with $\sigma_2(f(x))$.*

A necessary condition for partial match is that the number of components in both the signatures must be the same. As seen from the definition of partial match, if a signature σ_1 exactly matches with another signature σ_2, then σ_1 partially matches with σ_2 as well. This is a consequence of the definition for type compatibility, and is also consistent with similar definitions for signature match given in [31].

3.2.3 Property Match

Definition 8[property]: *The property of a signature σ, is defined by a predicate. This predicate is a conjunction of well formed formulas $p_1, p_2, .., p_m$.*
Thus,

$$P(\sigma) = p_1 \wedge p_2 \wedge \ldots \wedge p_m.$$

We use the notation $\wedge_{i=1}^{m}(p_i)$ to denote the logical conjunction of the well formed formulas $p_1, p_2, \ldots, p_m$. Note that a predicate expressed in some other form can always be normalized to the conjunctive normal form and hence the above representation is reasonable for our discussion.

Each predicate contains free variables and bound variables. The free variables come from the declaration part of the signature. We use the notation $F(p_i)$ to represent the set of free variables of p_i and $B(p_i)$ to represent the set of bound variables of p_i.

$$\bigcup_{i}^{n} B(p_i) \subseteq dom(\sigma),\ p_i \in P(\sigma)$$

We abuse the membership notation $\in$ in the above definition to denote that p_i is one of the conjuncts of $P(\sigma)$.

Substitution for free variables Let $\theta(t_1 \leftarrow x_1, t_2 \leftarrow x_2, \ldots, t_n \leftarrow x_n)(p)$ denote the substitution function for a well-formed formula p in which the free variables $x_1, x_2, \ldots, x_n$ are substituted by the variables $t_1, t_2, \ldots, t_n$ respectively. Then,

$$x_1, x_2, \ldots, x_n \subseteq F(p) \wedge t_1, t_2, \ldots, t_n \cap B(p) = \varnothing$$

We use the notation $\theta_{i=1}^{n}(t_i \leftarrow x_i)(p)$ to denote the expression

$$\theta(t_1 \leftarrow x_1, t_2 \leftarrow x_2, \ldots, t_n \leftarrow x_n)(p)$$

in the rest of the thesis.

Now we formally define property matching between two signatures σ_1 and σ_2. Exact match of properties holds when the types of respective variables in σ_1 are compatible with those in σ_2.

Exact Match of properties

Definition 9[property exact match]: *A property $P(\sigma_1)$ associated with the signature σ_1 exactly matches with a property $P(\sigma_2)$ associated with the signature σ_2 if and only if $P(\sigma_1)$ is equivalent to $P(\sigma_2)$ after renaming the names in the domain of σ_1 to match with the names in the domain of σ_2.*

A necessary precondition for exact match of two properties $P(\sigma_1)$ and $P(\sigma_2)$ is that σ_1 and σ_2 must exactly match with each other. As an example, consider the two following schemas whose signatures and properties match exactly with each other.

User

$id : \mathbb{N}_1$
$name : STRING$
$age : \mathbb{N}_1$

$\#name \leq 20$

Book

$callno : \mathbb{N}_1$
$title : STRING$
$year : \mathbb{N}_1$

$\#title \leq 20$

In this case, the properties of the two signatures exactly match with each other after renaming the names in *User* as follows: $(id \leftarrow callno, name \leftarrow title, age \leftarrow year)$. The property of the signature of *User* exactly matches with the property of signature of *Book* after renaming.

Partial match of properties

Definition 10[property partial match]: *Given two signatures σ_1 and σ_2, the property $P(\sigma_1)$ of σ_1 partially matches with the property $P(\sigma_2)$ of σ_2, if $P(\sigma_1)$ implies $P(\sigma_2)$ after renaming the names in the domain of σ_1 to match with the names in the domain of σ_2.*

This use of implication in partial match can be justified since the property of the library component is weaker than that of the target component to enable reuse. Only then the library component can match several target components. This is in principle agreed by the two groups [14] and [31]. As seen from the above definition, partial match between two signatures is not symmetric. This is expected; otherwise, partial match will converge to exact match. Use of implication in the definition for partial match can be justified from the following fact: $P(\sigma_1)$ must be weaker than $P(\sigma_2)$, when σ_1 represents a signature in a library specification and σ_2 represents a signature in a target specification, thus facilitating reuse.

In the previous example, if the predicate part of the schema *User* is:

$$\#name = 20$$

then, the property of *User* will imply the property of *Book* after renaming the variables and hence *User* will partially match with *Book*.

3.2.4 Schema Matching

A schema is a triple (s, σ, P) where s is the unique name of the schema, σ is its signature and P is the property of the schema defined over σ. The name of the schema does not play any role in the process of ensuring structural compatibility. Therefore, schema matching is the same as signature matching.

Definition 11[schema match]: *A schema S will match with a schema T if their respective signatures and properties match with each other.*

3.2.5 Global Declaration Matching

A global declaration is a signature along with a property. Therefore, signature and property matching are equally applicable to global declarations as well. The difference between global declaration matching and schema matching comes only during the automation of the entire process when the list of matched schemas are stored along with the schema names while the matched global declarations may be stored with some internal identifiers such as line numbers in the specification.

3.3 Algorithms

In this section, we describe a set of the algorithms to implement the methodology discussed earlier. Only important algorithms have been described in this thesis; trivial ones have been left out for brevity. The implementation includes all the algorithms.

Algorithm 1 describes the process to ensure partial match between two signatures LSIG and TSIG, corresponding to library and target specifications respectively. It returns a name table which consists of the matching variables from LSIG and TSIG

whose respective types are compatible.

Algorithm 1 *signature_match*(LSIG,TSIG)
Purpose: This algorithm describes the partial match of signatures LSIG and TSIG. As required by the specification, the two signatures must have the same number of components. The type of LSIG(i) must be the same or compatible with the type of TSIG(i), $1 \leq i \leq \#LSIG$.
Input: LSIG, TSIG: Signature
/* LSIG represents library signature and TSIG represents target signature. The type 'Signature' is a sequence of <name,type>. */
Output: table: Name_Table
/* Name_Table contains the pairs of names <l,t>, l ∈ LSIG and t ∈ TSIG. If any name in LSIG is not matched, the table will be empty.*/
Local variable:table: Name_Table; flag: Boolean
/* The type Name_Table is defined as a sequence of ordered pairs of names */
Procedure:

```
    table ← ⟨⟩;
    /* Initialize table to an empty sequence */
    flag ← true;
        if #LSIG ≠ #TSIG return table
            while (LSIG ≠ ∅) do
                if (type_compatible(type(head(TSIG)), type(head(LSIG))))
                    table ← table ⌢ ⟨ name(head(LSIG)) ↦ name(head(TSIG))⟩;
                else
                table ← ⟨⟩
                flag ← false;
                endif
                LSIG ← tail(LSIG);
                TSIG ← tail(TSIG);
            endwhile
        endif
        return table;
endproc
```

Given two signatures, there might be more than one possible way of matching them. This is because the ordering among the entries (pairs of names and types) in a signature is immaterial. Therefore, we store all possible combinations of matches for a given pair of signatures in a table, called T. Each entry in T itself is a table that contains one set of matching names. Later, during declaration matching, those com-

binations that result in unmatched properties will be eliminated from T. Algorithm 2 describes the construction of all name tables from a pair of signatures.

Algorithm 2 *construct-name-tables*(LSIG,TSIG)
Purpose: To construct all possible name tables from the two signatures LSIG and TSIG.
Input : LSIG, TSIG: Signature
/* LSIG represents library signature and TSIG represents target signature. */
Output: result: seq Name_Table;
/* A set of name tables of type Name_Table. This type is a set of ordered pairs; the first element of each such pair represents a name from LSIG and the second from TSIG. We use the notation **P** to declare a set.*/
Local variables: permut: **P** Signature; current: Signature
Procedure:

```
result ← ⟨⟩;
if #LSIG = #TSIG
    permut = permut-signatures(TSIG);
    /* compute all possible permutations of names in TSIG */
    while permut ≠ ∅ do
        current ← head(permut);
        if (signature-match(LSIG, current) ≠ ⟨⟩)
            result ← result ⌢ signature − match(LSIG, current);
        endif
        permut ← tail(permut);
    endwhile
endif
return result;
endproc
```

Algorithm 3 compares two predicates LP and TP representing the properties of the library signature and the target signature respectively. The predicates are input in conjunctive normal form which includes only the three operators $\vee, \wedge, \neg$. At first, the two predicates are rewritten into prefix form. If they both have the same operators, then their respective operands are compared for logical equivalence. This recursively invokes Algorithm 3. If this case fails, one of the predicates is commutated and then the algorithm is exercised once again for the operands. However Algorithm 3 does not ensure completeness; i.e., there may be a match existing between the two predicates but the algorithm may not find it. On the other hand, if they are simple predicates without logical operators (the termination condition for recursion), then they are directly compared for logical equivalence. In successful cases, the output

is the target expression TP rewritten after substituting the variables in the target expression with the names of the variables in the library expression. The name table is assumed to be available for this algorithm. In case of failure of Algorithm 3, the target predicate is returned.

Algorithm 3 *property-match*(LP, TP, name-table)
Purpose: This algorithm describes partial match of properties represented by the predicates LP and TP. The properties match as long as the predicates representing the properties are logically-equivalent .
Input: LP, TP: Predicate;
name-table: Name_Table;
/* Library predicate(LP) and target predicate(TP), both in conjunctive normal form. The name table contains the matched names from LSIG and TSIG which are used by LP and TP. */
Output: result: boolean
Local variable: L_{new}, T_{new}, new_1, new_2:Predicate;
Procedure:

```
declare Op == {∧ | ∨};
result ← false;
/*Obtain the prefix expressions for both the predicates */
L_new ← prefix_expression(LP);
T_new ← prefix_expression(TP);
/* If the first element in each expression is not an operator, then they
are simple predicates which can be directly compared for logical equiv-
alence. */
if (head(L_new) ∉ Op ∧ head(T_new) ∉ Op)
    result ← (logically-equivalent(L_new, T_new, name-table));
else
/* If both the expressions have the same operator at the front, then
check for logical equivalences of their respective operands by recursively
calling this function. If they don't match, try for logical equivalence
once again after commuting the target expression. */
    if (head(L_new) = head(T_new))
        result ← (property-match(first-operand(LP),
                    first-operand(TP), name-table) and
                  property-match(second-operand(LP),
                    second-operand(TP),name-table)) or
                  (property-match(first-operand(LP),
                    second-operand(TP), name-table) and
                  property-match(second-operand(LP),
```

```
                    first-operand(TP), name-table));
        endif
    endif
    return result;
endproc
```

Deciding logical equivalence of two predicates is not a simple task since the same property can be expressed in more than one way. For example, the two predicates $x > y$ and $y < x$ denote the same property and hence are logically equivalent. Automating this process requires term re-writing rules which considers the expression involving $<$, for example, and rewrites it into an expression involving $>$. Thus, we can conclude that the above two expressions are logically equivalent. On the other hand, there are a number of term rewriting rules such as

$P \vee (P \wedge Q) \equiv P$

which makes the algorithm for logical equivalence exhaustive. Since there are a number of tools available for checking logical equivalence of predicates, we do not discuss the algorithm in this thesis. Even the tools which check the logical equivalence of the two predicates are not sufficient for our case. For example, the following two predicates are logically equivalent in our case when the two declarations corresponding to these two predicates are of type P S and iseq S respectively:

$x \in S; (\exists i : 1 .. \#S \bullet S(i) = x)$

Therefore, we will not elaborate further the discussion on logical equivalence at this stage.

In our current implementation, we manually check the predicates for logical equivalence. This is done by (i) substituting the names in TP by the matching names in LP using the name-table (that is why the name-table is passed as a parameter to Algorithm 3) , and (ii) displaying the modified TP and unmodified LP to the user to check for logical equivalence manually.

Algorithm 4 describes the signature match between two signatures.

Algorithm 4 declaration-match(LD, TD)
Purpose: This algorithm matches the two declarations LD and TD. They match if their respective signatures and properties match. The algorithm outputs all possible declaration matches based on the number of possible permutations among the signatures.
Input: LD, TD: Declaration;
/* LD represents library declaration and TD represents target declaration. */
Output: result: seq Name_Table

```
/* A sequence of name tables containing matching names from LD and TD for which
both signature and property match. */
Local variables:
    LSIG,TSIG: Signature; LP,TP: Predicate; tables: seq Name_Table;
Procedure:
    LSIG ← signature-of(LD);
    TSIG ← signature-of(TD);
    LP ← property-of(LD);
    TP ← property-of(TD);
    tables ← construct-name-tables(LSIG,TSIG);
    result ← ⟨⟩;
    while (tables ≠ ⟨⟩) do
        if property-match(LP,TP,head(tables))
            result ← result ⌢ ⟨ head(tables) ⟩;
        endif
        tables ← tail(tables);
    endwhile
    return result;
endproc
```

Behavioral Compatibility

For completeness, we define behavioral compatibility in this section. However, the thesis does not deal with behavioral compatibility any further.

Definition 10[behavioral compatibility]: *A software component Lib is behaviorally compatible with another software component Tar if and only if every operation Op_l in Lib is behaviorally compatible with some operation Op_t in Tar.*

As in the case of declaration match, we can define exact and partial matches for behavioral compatibility. We now define behavioral compatibility of an operation in *Lib* with an operation in *Tar*.

Definition 11[operation compatibility]: *An operation Op_l is behaviorally compatible with an operation Op_t if and only if the state space S_l on which Op_l operates is structurally compatible with the state space S_t on which Op_t operates, and the predicate of Op_l matches with the predicate of Op_t.*

Depending on the structural compatibility between S_l and S_t and the predicate match between Op_l and Op_t, we can define exact and partial matches of behavioral compatibility between operations.

At this stage, we stress the importance of behavioral compatibility to be used in conjunction with structural compatibility. For example, in Z, a schema can represent a state or an operation. The syntax of an operation schema includes decorations for variables such as x', where x' indicates the value of the variable x after the operation successfully terminates. Similarly, the decorations ? and ! are used for input and output respectively. Since the signature matching algorithms are based on types, rather than on the names of the declarations, it is possible that an operation schema in the target matches with a state schema in the library or vice versa. This is conceptually not agreeable but it still satisfies the conditions for signature match. Such faults can be resolved only after checking behavioral compatibility.

Automating the process to ensure behavioral compatibility is difficult. This is because the same behavior can be expressed in more than one way. In that case, pattern matching does not help in checking for compatibility. It requires manual intervention to decide if two specifications are behaviorally compatible. For example, consider the following schemas:

Flight

$$
\begin{array}{l}
flightnumber : FlightNumber \\
departure, arrival : Time \\
reserved : iseq\ PassengerNumber \\
capacity : \mathbb{N}_1 \\
\hline
arrival > departure \land \#reserved \leq capacity
\end{array}
$$

Course

$$
\begin{array}{l}
courseno : CourseNumber \\
start, end : Time \\
registered : \mathbb{P}\ StudentNumber \\
allocated : \mathbb{N}_1 \\
\hline
start < end \land \#registered \leq allocated
\end{array}
$$

After substituting the variable names in the target schema, we have

$arrival > departure$

$start < end.$

In this case we see that though the two schemas are behaviorally compatible, but it is difficult to automate the process as the relational operators used are different in both cases.

Chapter 4

Case Study

In this chapter, we illustrate our methodology with a case study. We consider two specifications - a course registration system and a flight reservation system. We give only the state space schemas representing the two applications.

4.1 Course Registration System

Information for a course contains a unique course number, starting time, ending time, a set of students attending the course and the maximum seats allocated for the course.

$$Hour == 0..23$$
$$Minute == 0..59$$
$$Time == Hour \times Minute$$
$$CourseNumber == 100..700$$
$$StudentNumber == 100000..999999$$

Course

$$courseno : CourseNumber$$
$$start, end : Time$$
$$registered : \mathbb{P}\ StudentNumber$$
$$allocated : \mathbb{N}_1$$

$$start < end \land \#registered \leq allocated$$

Every student has a unique ID number. The set of courses taken by each student is also modeled as part of the student information. Other irrelevant details such as name, address and phone number are omitted for brevity. The property asserts that the student cannot attend two different courses simultaneously.

Student

$$
\begin{array}{l}
id : StudentNumber \\
taken : \mathbb{P}\ Course \\
\hline
\forall t_1, t_2 : taken \bullet \\
\quad (t_1.start \neq t_2.start \land \\
\quad\ t_1.start < t_2.start \Rightarrow t_1.end < t_2.start)
\end{array}
$$

A course registration system contains a set of courses and a set of students. The property asserts that (i) all courses have unique course numbers, (ii) all students have unique ID numbers, and (iii) information regarding courses and students with respect to this system is consistent.

RegistrationSystem

$$
\begin{array}{l}
courses : \mathbb{P}\ Course \\
students : \mathbb{P}\ Student \\
\hline
(\forall c_1, c_2 : courses \bullet c_1.courseno = c_2.courseno \Rightarrow c_1 = c_2) \\
(\forall s_1, s_2 : students \bullet s_1.id = s_2.id \Rightarrow s_1 = s_2) \\
(\forall c : courses \bullet c.registered = \{s : students \mid c \in s.taken \bullet s.id\}) \\
(\forall s : students \bullet s.taken \subseteq courses)
\end{array}
$$

4.2 Flight Reservation System

The information about a flight includes a unique flight number, its arrival and departure times, the passengers who have reserved a seat and the capacity of the flight. For simplicity, each passenger is identified by a unique number. Note that *arrival* in the following structure represents the arrival of the flight at its destination and *departure* refers to the flight at its origin.

$$FlightNumber == 100..999$$
$$PassengerNumber == 1..1000$$

Flight

$$
\begin{array}{|l}
\hline
flightnumber : FlightNumber \\
departure, arrival : Time \\
reserved : \mathrm{iseq}\ PassengerNumber \\
capacity : \mathbb{N}_1 \\
\hline
arrival > departure \land \#reserved \leq capacity \\
\hline
\end{array}
$$

Every passenger has a unique number (used internally by the system; for the current version, other details such as name, address and nationality are ignored as irrelevant). The set of flights reserved by a passenger is also stored as part of the information for the passenger. It is asserted that no two flight schedules for the same passenger should overlap.

Passenger

$$
\begin{array}{|l}
\hline
token : PassengerNumber \\
booked : \mathbb{P}\ Flight \\
\hline
\forall f_1, f_2 : booked \bullet \\
\quad (f_1.departure \neq f_2.departure \land \\
\quad f_1.departure < f_2.departure \Rightarrow f_1.arrival < f_2.departure) \\
\hline
\end{array}
$$

A reservation system contains a set of passengers and a set of flights. Each reserved flight by a passenger should have been entered in the database. Consequently, passengers of each flight must also be known to the system. The property of the schema *Reservation* asserts that (i) all flights have unique flight numbers, (ii) all passengers have unique identification numbers, and (iii) information regarding flights and passengers within this reservation is consistent.

ReservationSystem

$$
\begin{array}{|l}
\hline
flights : \mathbb{P}\ Flight \\
passengers : \mathbb{P}\ Passenger \\
\hline
(\forall f_1, f_2 : flights \bullet f_1.flightnumber = f_2.flightnumber \Rightarrow f_1 = f_2) \\
(\forall p_1, p_2 : passengers \bullet p_1.token = p_2.token \Rightarrow p_1 = p_2) \\
(\forall f : flights \bullet ran\, f.reserved = \{p : passengers \mid f \in p.booked \bullet p.token\}) \\
(\forall p : passengers \bullet p.booked \subseteq flights) \\
\hline
\end{array}
$$

Registration	Reservation	Compatible Types
courseno	flightnumber	CourseNumber $\xrightarrow{t}$ FlightNumber
start	departure	
end	arrival	
registered	reserved	**P** StudentNumber $\xrightarrow{t}$ **iseq** PassengerNumber
id	token	StudentNumber $\xrightarrow{t}$ PassengerNumber
taken	booked	Course $\xrightarrow{t}$ Flight
courses	flights	Course $\xrightarrow{t}$ Flight
students	passenger	StudentNumber $\xrightarrow{t}$ PassengerNumber

Table 4.1: Declaration Match in the Case Study

4.3 Structural Compatibility

The case study illustrates a partial match between the course registration system and the flight reservation system. Table 4.1 shows the declaration match between the entries in the two specifications. The last column in the table indicate compatible types in the case study. The notation X $\xrightarrow{t}$ Y indicates that X is compatible with Y. Notice that the type *StudentNumber* is also type compatible with the type *FlightNumber* since both are subranges of natural numbers. However, when considering property match, such illegal matches are eliminated. Table 4.1 summarizes only those entries of a successful match.

Property match is trivial except in two cases: the predicate
$start < end$
in *Course* matches with the predicate
$arrival > departure$
in *Flight*, even though the logical operators are different. The predicate
$c.registered = \{s : students \mid c \in s.taken \bullet s.id\})$
in *RegistrationSystem* partially matches with the predicate
$ran f.reserved = \{p : passengers \mid f \in p.booked \bullet p.token\})$
Hence the registration system can be partially reused for the reservation system.

The state space declaration for the flight reservation system consists of three schemas *Flight*, *Passenger* and *Reservation* and the state space of the course registration system is defined by the three schemas *Course, Student and System.* The two state spaces match partially with each other because *Flight* matches partially with *Course, Passenger* matches partially with *Student* and *Reservation* matches partially with *System.* All three matches include both signature match and property match.

We notice that the invariant of *Flight* and that of *Course* use different logical operators but still they match exactly. This ensures that there is only one possibility of name matching between *Flight* and *Course*, namely

$number \mapsto courseno$, $departure \mapsto start$,
$arrival \mapsto end$, $occupied \mapsto registered$.

A slightly different version of the flight structure is given below which leads to partial match.

*Flight*1

$number : \mathbb{N}_1$
$departure, arrival : \mathbb{N}_1$
$seats : \mathrm{seq}\,\mathbb{N}$
$capacity : \mathbb{N}_1$

$arrival > departure$
$\forall i, j : 1..\#seats \bullet$
$i \neq j \land seats(i) > 0 \land seats(j) > 0 \Rightarrow seats(i) \neq seats(j)$
$\#seats \leq capacity$

The variable *seats* indicates a sequence of passenger numbers. Thus, $seats(i)$ refers to the passenger number occupying the i^{th} seat. It is assumed here that seats with passenger number zero are empty seats. The second invariant in *Flight*1 ensures that no passenger occupies more that one seat.

One can see that *Course* partially matches with *Flight*1 under the following mapping:

($number \mapsto courseno$, $departure \mapsto start$,
$arrival \mapsto end$, $seats \mapsto registered$,
$capacity \mapsto allocated$)

Partial match occurs because the type of *registered* is compatible to the type of *occupied*. The properties of *Flight*1 and *Course* match partially because the signatures match partially only. Even though *Flight*1 has an additional predicate, it still enriches the type compatibility which would otherwise be imposed anyway.

Chapter 5

Implementation

This chapter explains the implementation of algorithms to ensure structural compatibility and some comments on the implementation, particularly the additional components introduced during the implementation of the process. The algorithms have been implemented in C, under the Unix operating system. The specifications used as case studies for the implementation have been type checked using *fuzz* type-checker[27]. The output of *fuzz* (containing type information) is used as input for the program.

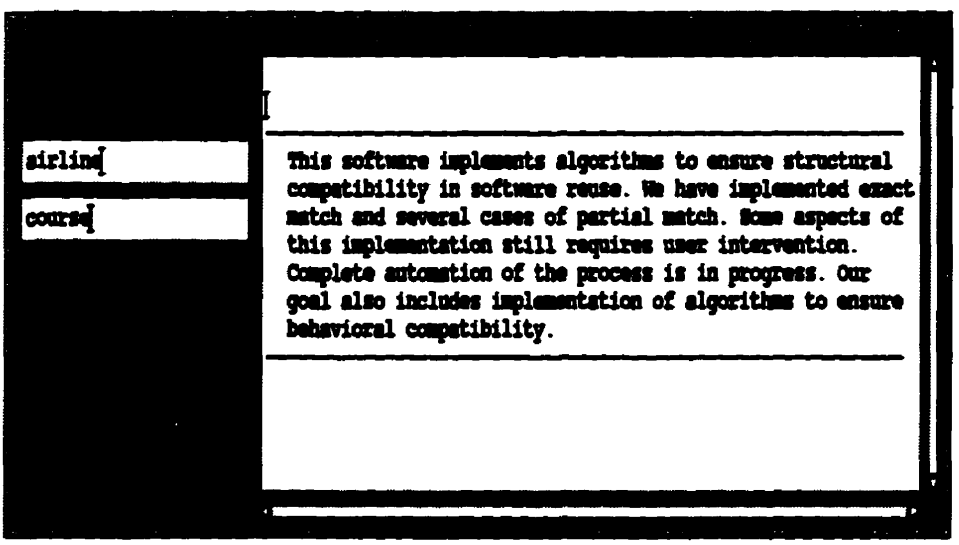


Figure 5.1: Input Manager

An example of *fuzz* output for the schema *Flight* discussed in the previous chapter is given below:
Schema Flight
number: NN
departure: NN
arrival: NN
occupied: P NN
End

5.1 Exact and Partial match

Implementation of exact matching of signatures and properties is trivial. For exact matching of signatures, every schema in the library specification is matched with every schema in the target specification. All possible combinations of exact matches are stored. Later during property matching, name substitutions from those matching schemas from the table are tried. By this process, those schemas whose signature match exactly, but whose properties do not match exactly, are deleted from the table. At the end, we will be left with those schemas whose signature and properties both exactly match.

During partial match, first we store all possible combinations of partial signature match. This process also includes those schemas which exactly match. Later, the entries in the table are selected one at a time and tried for partial matching of properties. One may see that this process may lead to tremendous increase in storage requirements as the specifications become larger. However, we do not address complexity issues in this thesis.

While considering partial match of declarations, we have included another option in our implementation: if only a portion of the declaration d_1 in the library software matches with the declaration d_2 in target software, then we consider this as a partial match. The reason for considering this special case is to provide as much flexibility to the reuse process as possible. In the implementation, a matching index α has been set $(0 < \alpha < 1)$ for partial match of signatures. We then declare that a partial match can occur when (i) there are n entries in the signature of d_1; (ii) m out of n entries in d_1 partially match with all entries in d_2; and (iii) $m/n \geq \alpha$. The predicates in $P(\sigma_1)$ of d_1 which contains the $n - m$ variables are ignored while considering the partial match of properties, since they will not become part of the reusable component. As a result, the portion of d_1 containing the m variables is tailored for reuse. We however, have not explored this option further.

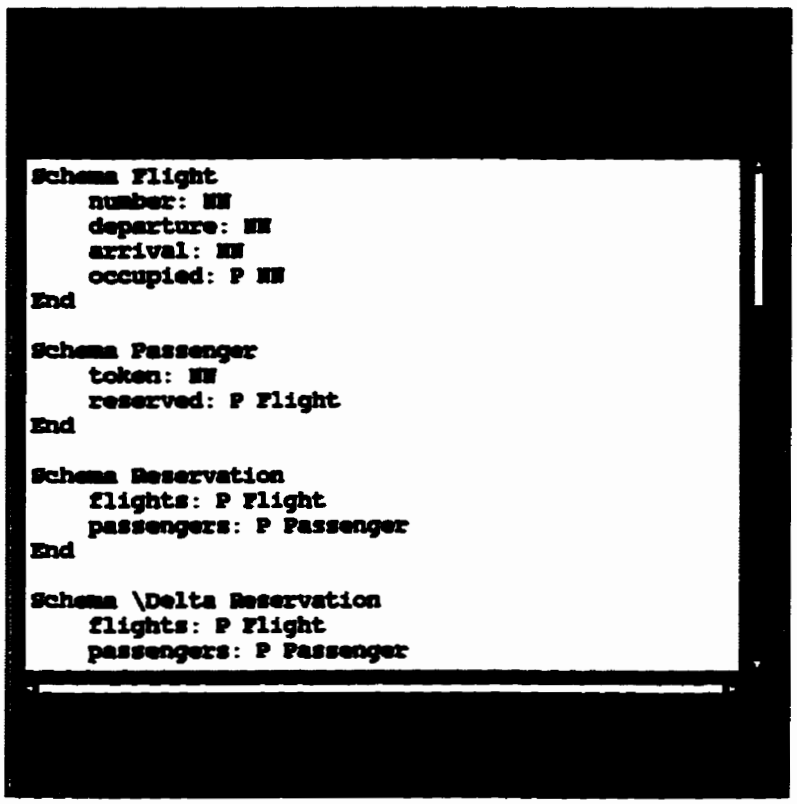


Figure 5.2: View Manager-Library Specification

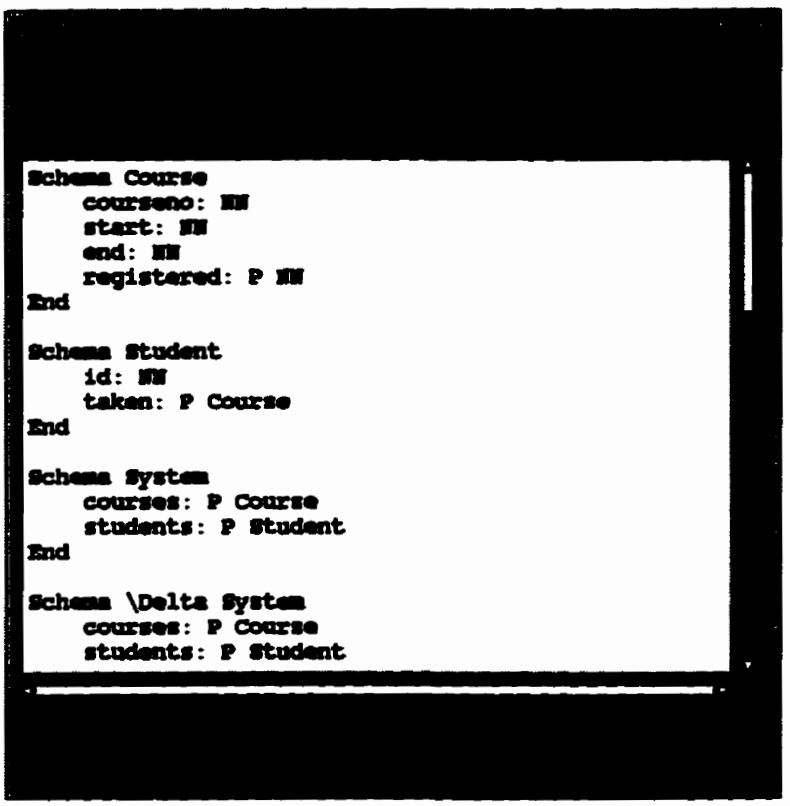


Figure 5.3: View Manager-Target Specification

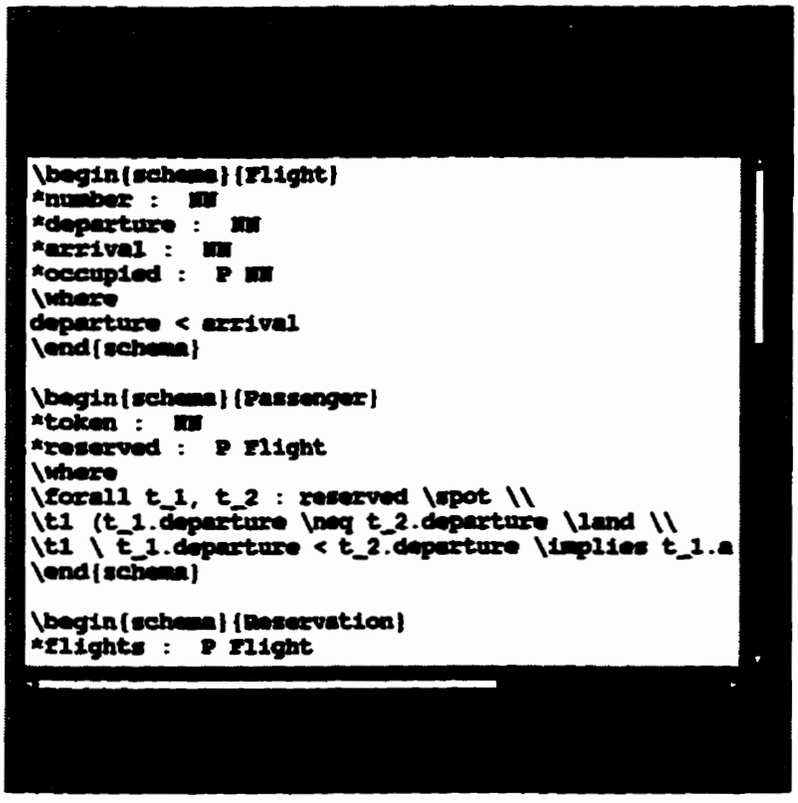


Figure 5.4: View Manager-Modified Specification

In case of partial matches using α factor, the library specification can be split into two pieces such that one of the pieces matches either exactly or partially with the target specification. The other piece is considered as irrelevant to the target application. It can be easily justified that the conjunction of the two pieces is equivalent to the library specification using schema calculus. We do not discuss the splitting of library specifications any further in this thesis.

As an example, consider the schema *Course* in the course registration system contains additional information such as

Course

$$
\begin{array}{l}
courseno : CourseNumber \\
start, end : Time \\
registered : \mathbb{P}\, StudentNumber \\
allocated : \mathbb{N}_1 \\
roomno : RoomNumber \\
teacher : String \\
\hline
start < end \land \#registered \leq allocated
\end{array}
$$

In this case, n = 7; m = 5. If $\alpha = 0.6$, then $m/n > 0.6$. Hence, partial reuse can still occur with *roomno* and *teacher* excluded from the declaration.

The program concludes that two schemas match if more than 70% (arbitrarily chosen for this thesis) of the number of variables in the schema match - either exactly or partially. Table 3.1 gives the partial compatibility between types. The matched variables in the schema are highlighted for the user. In figure 5.4, the variables preceded by an asterisk are an exact match of the variables in the target schema and the rest do not match. The corresponding variables are substituted in the predicates and shown. The user is given the option to try out different possible permutations and combinations of the variables in his quest for a match between the schemas.

For each permutation of the variables, a new specification file is produced with the variable names in the library specification being replaced by the variable names in the target specification. The user can view these specification files to make the final decision as to which library specification can be reused in place of the target specification. The interface offers the user a text viewer to view the files. In case of less/no match between the library specification and the target specification, the user can abandon the library specification and start all over from scratch with a new library specification. This process is repeated until a match is found from the library of specifications.

As has been already stated, the behavioral compatibility has not been taken care of, because of the complexity and interaction involved. However, the user can extend the algorithms for property match to implement behavioral compatibility.

5.2 User interface

The implementation also has a friendly user interface. The interface screens were implemented using X-Motif; portions of this interface are shown in Figures 5.1 and 5.2. The first window prompts the user to enter the file names containing the library and target specifications as input. The View screen enables the user to visualize the library specification and the target specification independently. The results of signature and property matches are stored in various files. These files can also be viewed from the View Manager. The View Manager shown in Figure 5.2 shows the library specification, Figure 5.3 shows the target specification before modification. Figure 5.4 shows the modified target specification. This modified specification indicates with asterisk marks those variables that match. Thus the user can visually identify structural matching. In addition the same window can be used to see the other tables generated during the matching process. This enables the user to interactively assert compatibility and partial match between the library and target specifications.

Chapter 6

Conclusions and future directions

Reuse of existing software is an evolving technology. However, the methods for software reuse as reported in literature are targeted towards specific application or do not justify the reusability of an existing software product. Formal methods, when employed in a software reuse process, promise to justify reusability and also to automate the reuse process. The latter claim is due to the well defined syntax and semantics of the formal notations used. Since formal methods have been incorporated into software development process only recently, and the formal specification languages are continuously evolving, use of formal methods is still not popular in industrial environments.

This thesis proposes a methodology by which software reuse can be achieved effectively using formal requirements specifications. Due to evolving nature of both formal notations and reuse technology, we claim that this thesis is a novel contribution towards automating the software reuse process. We assert that ensuring software reuse requires that both structural compatibility and behavioral compatibility must be ensured. We have presented the methodology to ensure structural compatibility, have devised algorithms to implement the methodology and have also implemented the algorithms. The implementation has been tested for a couple of case studies; the results are promising. An important contribution in this thesis is the notion of type compatibility; we have used this notion to define various cases of partial matches for software reuse.

6.1 Limitations

Some of the limitations of the present work are discussed below.

- The suggested methodology is based on the Z formal notation. So, if specifications are written in another specification language then the methodology requires appropriate modifications. We still claim that the proposed methodology can be applied to several model-based specification languages with minor modifications to the algorithms. The pattern matching techniques in our implementation are based on the format of the *fuzz* type-checker output. Hence, if this methodology is used for another specification language, then appropriate modifications are required in the implementation as well.
- Though both structural compatibility and behavioral compatibility are essential to ensure reuse, only structural compatibility is discussed in the paper. Automation of behavioral compatibility is complex and requires user interaction and term rewriting rules.
- It has been assumed that the library and target specifications are error-free and consistent. The methodology does not check for validity of the specifications. As these specifications are type-checked by *fuzz*, they are ensured to be correct with respect to syntax and types. But semantic consistency of the specifications must still be ensured before considering reuse.
- The methodology also relies a lot on the way the specifications are written. Specifications written in different styles give different results. Although the specifications may be targeted towards the same application, they may be syntactically different and so the methodology suggested may not find a match in this case. For example, one can write a property-oriented specification in Z using only the axiomatic declarations while another person can write a specification using a state space model for the same problem. Obviously, ensuring structural compatibility in this case is very difficult.
- Since the reuse process involves a lot of user-interaction and decision-making, automation of the whole process is still beyond the scope of the current methodology.

We propose the following work items as possible continuations of our current work.

- Notion of behavioral compatibility can be elaborated further and implemented.

- The methodology can be exercised using other model-based specification languages and verified.
- Partial reuse can be considered in depth and the algorithms can be fine-tuned. As a consequence, the process will not only justify partial reusability but also will propose hints and guidelines as to how to tailor the library specification towards replacing portions or whole of the target specification.

Bibliography

[1] S.K.Abd-El-Hafiz, V.R.Basili and G.Caldiera, "Towards Automated Support for Extraction of Reusable Components", *IEEE Conference on Software Maintenance*, May 1991, pp. 212-219.

[2] V.R.Basili and H.D.Rombach, "Support for Comprehensive Reuse", *Software Engineering Journal*, Sep 1991, Vol 6, No 9, pp.303-316.

[3] D.Batory and S.O'Malley, "The Design and Implementation of Hierarchal Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodology*, Vol 1, No 4, Oct 1992, pp.355-398.

[4] M.Barr and C.Wells, *Category Theory for Computing Science,* Prentice Hall International Series in Computer Science, Hertfordshire(UK), 1990.

[5] T.Biggerstaff and R.Ritcher, "Reusability framework, assessment and directions", *IEEE Software*, 1987, vol.2, pp. 41-49.

[6] Gianluigi Caldiera and Victor R.Basili, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Sep 1991, pp.61-69.

[7] J.J.Jeng and B.H.C.Cheng, "A Formal Approach to Reusing More General Components", *Proceedings of the IEEE 9th Knowledge-Based Software Engineering Conference*, Sep 1994.

[8] J.J.Jeng and B.H.C.Cheng, "Specification Matching for Software Reuse: A Foundation", *Proceedings of ACM SIGSOFT Symposium on Software Reusability*, Seattle, Washington, Apr 1995, pp. 97-105.

[9] A. Finkelstein, "Reuse of formatted requirements specifications", *Software Engineering Journal*, 1988, vol.3, no.3), pp.186-197.

[10] D.Gentner, "Structure Mapping: a theoretical framework for analogy", *Cognitive Science*, 198, vol.5, pp.121-152.

[11] V.Karakostas, "Requirements for CASE tools in early software reuse", *ACM SIGSOFT Software Engineering Notes*, 1989, vol.14, no.20, pp.39-41.

[12] G.Gratzer, *Universal Algebra* (Second Edition), Springer-Verlag, NY, 1979.

[13] M.L.Griss, "Software Reuse at Hewlett Packard", Technical Report HPL-91-38, Software and Systems Laboratory, Hewlett Packard, CA, Mar 1991.

[14] J.J.Jeng and B.H.C.Cheng, "A Formal Approach to Reusing More General Components", *Proceedings of the IEEE 9th Knowledge-Based Software Engineering Conference*, Sep 1994.

[15] J.J.Jeng and B.H.C.Cheng, "Specification Matching for Software Reuse: A Foundation", *Proceedings of ACM SIGSOFT Symposium on Software Reusability*, Seattle, Washington, April 1995, pp.97-105.

[16] S. Kedar-Cabelli, " Analogy - from a unified perspective" in *Analogical Reasoning*, Kluwer Academic Publishers, 1988.

[17] B.H.Liskov and J.M.Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, vol.16, no.6, Nov 1994, pp.1811-1841.

[18] N.A.Maiden, "Analogy as a Paradigm for Specification Reuse", *Software Engineering Journal*, vol.6, no.1, Jan 1991, pp.3-15.

[19] N.A.Maiden and A.C.Sutcliffe, "Exploiting Reusable Specifications Through Analogy", *Communications of the ACM*, vol.35, no.4, Apr 1992, pp.55-64.

[20] B.Meyer and Jean-Marc Nerson, *Object-Oriented Applications*, Prentice Hall Object-Oriented Series, 1993.

[21] K.Periyasamy, "A Formal Approach to Software Reusability", *Workshop on Incompleteness and Uncertainty in Information Systems*, Montreal, Canada, Oct 1993, Workshops in Computing Series, Springer-Verlag, 1994.

[22] B.Potter, J.Sinclair and D.Till, *An Introduction to Formal Specification and Z*, Prentice Hall International Series in Computer Science, 1991.

[23] R.Prieto-Diaz, "Software Reuse", *Communications of the ACM*, vol.34, no.5, May 1991. pp.89-97.

[24] M.B.Ratcliffe and R.J.Gautier, "System Development Through the Reuse of Existing Components", *Software Engineering Journal*, vol.6, no.11, Nov 1991, pp.406-412.

[25] H.A.Schmidt, K.Schutte and H.J.Thiele, *Contributions to Mathematical Logic*, Proceedings of the Logic Colloquim, Hannover 1996, North Holland Publishing Company, Amsterdam, 1968.

[26] J.M.Spivey, *The Z Notation: A Reference Manual* (Second Edition), Prentice Hall International Series in Computer Science, 1992.

[27] J.M.Spivey, The *fuzz* Manual (Second Edition), J.M.Spivey Computing Science Consultancy, Oxford, 1992.

[28] *Proceedings of the ACM Symposium on Software Reusability*, Seattle, Washington, Apr 1995.

[29] A.G.Sutcliffe and N.A.M.Maiden, "Specification Reusability: Why Tutorial Support is Necessary", Proc. SE '90, Brighton, UK, July 1990, pp. 489-509.

[30] W.Wechler, *Universal Algebra for Computer Scientists*, EATCS Monographs on Theoretical Computer Science, W.Brauer, G.Rozenberg and A.Salomaa(Eds.), Springer Verlag, 1992.

[31] A.M.Zaremski and J.M.Wing, "Signature Matching: A Key to Reuse", *ACM Transactions on Software Engineering and Methodology*, 1995.

[32] *The Z Notation - Version 1.2*, ISO/JTC1/SC22 document submitted for standardization, Oxford University, September 1995.

Appendix A

Z Notations used in this paper

This appendix includes the notations used in this thesis. Part of them are used in the Z specifications; others are used in describing the methodology. Informal semantics of these notations can be found in [J.M. Spivey, "The Z Notation: A Reference Manual", (Second Edition), Prentice Hall International Series in Computer Science, 1992.]

Definitions

$[]$	given set brackets for given types
$==$	abbreviation definition
$::=$	free type definition

Logic

$\neg$	logical negation
$\wedge$	logical conjunction
$\vee$	logical disjunction
$\Rightarrow$	logical implication
$\Leftrightarrow$	logical equivalence
$\forall\, d \bullet p$	universal quantification (d is declaration, p is predicate)
$\exists\, d \bullet p$	existential quantification (d is declaration, p is predicate)
$\exists_1\, d \bullet p$	unique existential quantification (d is declaration , p is predicate)
$<, \leq, =, \geq, >$	logical operators

(note: = is defined for all data types)

Sets

$\in$	set membership
$\notin$	set non-membership
$\varnothing$	empty set
$\subseteq$	subset
$\subset$	proper subset
$\{\}$	set brackets
$()$	tuple brackets
$\times$	Cartesian product
$\mathbb{P}$	power set
$\cap$	set intersection
$\cup$	set union
$\setminus$	set difference
$\bigcap$	generalized intersection
$\bigcup$	generalized union
$\#$	cardinality

Relations

$\leftrightarrow$	binary relation
$\mapsto$	maplet
dom	domain of a relation
ran	range of a relation

Functions

$\pfun$	partial function
$\fun$	total function
$\bij$	bijective function
$\pinj$	partial injection
$\inj$	total injection
$\psurj$	partial surjection
$\surj$	total surjection
$\oplus$	functional overriding

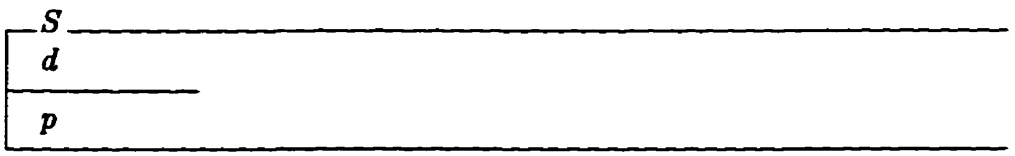
Numbers

$\mathbb{N}$	set of natural numbers
$\mathbb{N}_1$	set of non-zero natural numbers
$\mathbb{Z}$	set of integers
$+$	addition
$-$	subtraction
$*$	multiplication
$\mathbf{div}$	division
$..$	number range

Sequences

seq	sequence as a type
iseq	injective sequence
seq_1	non-empty sequence
$\langle\rangle$	sequence
$\frown$	sequence concatenation
head	first element of a sequence
tail	all but the head element of a sequence

Schema Notation

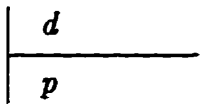


.	component selection in a schema
S'	state S after an operation
$[a/b]$	renaming schema components (rename b to a)
$\wedge$	schema conjunction
$\vee$	schema disjunction
$\neg$	schema negation
$\fatsemi$	schema composition
?	decoration of input variable
!	decoration of output variable

$$S \mathrel{\widehat=} [d \mid p]$$

The former denotes vertical notation of a schema and the latter denotes it horizontal version.

Axiomatic Definition



Notation for axiomatic definition is used for both functions and global constants.