Robotic Agents and Assembly Process : A Formal Specification Case Study

Kasilingam Periyasamy

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

June 1991

ISBN 0-315-68764-9

Abstract

Robotic Agents and Assembly Process:
A Formal Specification Case Study

Kasilingam Periyasamy, Ph.D.
Concordia University, 1991.

Formal specification enables a software developer to detect and eliminate inconsistencies and ambiguities in the requirements and promotes reasoning about the behavior of the software being developed. Formal specification case studies exist now for a wide range of problems varying in complexity from very simple applications such as floating point arithmetic in microprocessors to reactive systems such as robotics.

This thesis is devoted to developing formal behavior specification for a large scale industrial software system. Some of the inherent difficulties in specifying such large scale project are brought out and some solutions are suggested.

The case study chosen for this thesis is the problem of performing automated assembly of mechanical parts in a single static robot environment. Specifications for problems in the three subdomains *solid modeling, robot kinematics* and *assembly environment* are provided in this thesis. The model-based specification technique VDM is chosen for specifying the problems. A new methodology to derive an object-oriented design from a model-oriented specification is proposed and is illustrated for kinematics and solid modeling specifications. The current limitations of VDM, further extension to the specification language and possible specification refinements are mentioned.

To

M.N. Periyasamy
Seethalakshmi Periyasamy
S. Rajendran
Vijayalakshmi Rajendran

ACKNOWLEDGEMENTS

I sincerely express my gratitude to my supervisors Dr. V.S. Alagar and Dr. T.D. Bui. I have received an abundant measure of technical guidance, constant encouragement and frequent criticisms which made me possible to complete this work. I also thank them for generous financial support for the duration of my graduate studies.

I am grateful to the Quebec Ministry of Education for awarding me the international fee remission award and Centre de Researche Informatique de Montreal for their three year bursary to conduct this research.

It is my pleasure to personally thank my wife and my family members for their support and encouragement.

Finally I wish to thank all my friends, especially Dr. K. Arumugam and his family members for their moral support during my stay at Montreal.

Contents

List of Figures

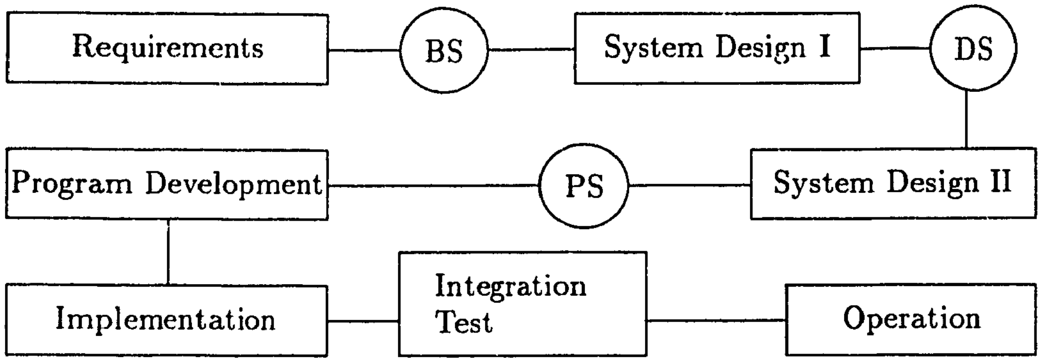
Chapter 1

Introduction

1.1 Need For Formal Specifications

Computer scientists, software engineers and managers of software systems have been expressing real concern with the lack of rigor in the practice of software development. As a solution, a recent software development paradigm treats the development as a process that proceeds from a formal specification to a final product. That is, the entire process may be considered as a transformation of the formal specification into a code that functions *correctly* when executed. The final product may be declared correct only if it is shown to meet the initial requirements and the transformation used to convert the formal specification (of the requirements) to the final product is proved to be correct. These two activities in any software development process model are known as *validation* and *verification*. This thesis argues for and provides formal specifications for a large scale industrial system platform to perform robotic assembly.

The terms verification and validation are misunderstood by a majority of novice software users to mean the same thing. It is to be emphasized that validation answers the question 'Are we producing the right product?' and verification answers the question 'Are we producing the product right?'. Ideally, verification performed at every stage of development removes errors that otherwise might occur in a software product. Verification demands a formal medium for expressing user requirements, design and program implementation details so that the outcome at any stage can be formally proved to be the same as the expected result. Such a mathematical proof



BS — Behavior Specification
DS — Design Specification (modules, interfaces)
PS — Program Specification

Figure 1.1: A Simple Life-Cycle Model with Specification Phases

indicates the correctness of the processes in that stage. It is generally agreed that verification is a difficult task and formal methods must provide sufficiently powerful and yet easy to use proof systems to conduct formal verification.

The formal verification medium of a process must be flexible enough to reason about the behavior of the process. A computer program in itself can be considered to be a formal representation of the user needs since the programming language itself is formal; however, it is difficult to reason about this form of representation and hence is not suitable for verification. What is required is a more abstract representation of the user needs from which a computer program can be derived either manually or automatically. Being formal (mathematical), this abstract representation enables the developer to assure correctness of the final program, by eliminating ambiguities, inconsistencies and contradictions, if any, present in the user requirements, design and implementation. In software engineering terminology, such a formal representation is called 'specification'. This term has the same meaning as is conventionally used in engineering applications; that is, the specification of a system refers to a theory about the behavior of the system. Depending on what stage of the software development process is being verified, the specification is named as *behavior specification, design specification* and *program specification* [Ala91].

Figure 1.1 [Ala91] represents a simplified software life cycle model which is an abstraction of several classical software process models augmented by the three stages of specifications : behavior specification, design specification and program specification. The behavior specification is a formal statement of *what* the system does, and is written in a declarative style. This stage of specification will eliminate ambiguities and contradictions present in the user requirements. The design specification contains more information on the structuring of the system, preserving the properties stated in the behavior specification. When compared to the behavior specification, the design specification is more detailed in the sense that it carries additional information pertaining to the system architecture, modules and their interfaces. Verification performed using design specification will remove inconsistencies in the design. Program specification deals with correctness of the implementation. Program verification is usually carried out during coding and may reveal errors such as non-termination of the program.

1.2 Formal Methods : Some Recent Trends

Formal methods are gaining importance in software development. Leveson [Lev90] has briefly discussed the current state of the art and the future potential of formal methods in software engineering. Gries [Gri91] has forcefully argued for the teaching of formal methods in computer science and pleaded for the use of such methods by software engineers in large scale software projects. Froome and Monahan [FrM88] discuss the role of mathematically formal methods in the development and in the assessment of safety critical systems. They have discussed pertinent industrial experience gained on using some of the most matured formal methods such as VDM, Z and CCS. Most importantly they mention that UK Defense Standard on safety critical software will make the use of formal methods mandatory for software to be used by the Ministry of Defense. It is also worthy of mentioning here that in Britain [CRN90] the Queen's Award for technological achievement for the year 1990 has been given to Oxford University for the development of formal methods in the specification and design of microprocessors. In particular, it is reported that the use of formal methods in the design of floating point arithmetic unit has enabled the development time to be reduced by 12 months. For example, the formal design of microprocessors

[CRN90] pointed to a number of errors in the informal (ad hoc) design that had not shown up in months of testing. Formal methods are applicable to a wide range of problems - from very simple applications such as floating point arithmetic to most complex systems such as robotics.

1.3 Some Issues In Specifying Large Scale Software

This thesis is devoted to developing formal behavior specification for a large scale industrial software system and its correctness proofs. Since the architectural design is to be derived from the formal specifications, we also describe a transformation for deriving a correct design from the specifications. Our study brings out some of the problems inherent in specifying a large scale project and suggests solutions for them.

Large scale industrial software development encompasses multiple application domains. For example, the software for a flight navigation system consists of the components : *software for sensor subsystems, software for visual display of the controlling activities* and *software for manipulating the control system elements*. It is easy to see that developing software for each one of these domains is an independent problem by itself. Developing a specification for large scale projects is inherently difficult due to the following reasons :

Design Complexity Specification for problems in individual domains may have to be developed independently and combined later.

Abstraction Levels The specification for the problem in one domain may not be at the same level of abstraction as the specification for a problem in another domain. Hence inconsistencies will arise if they are combined directly and permanently. On the other hand, finding the same level of abstraction for two problems in two different domains and/or determining the exact stage for combining the independent domain specifications are not trivial tasks and often requires the guidance of an application domain expert.

Specification Approaches Often a different specification technique is required or used to specify problems in two different domains. This is partly because of the nature of the application domain itself and partly because of the limitations and versatility of the existing specification techniques. Consequently, combining these specifications into a single framework is a complicated problem. One has to ensure semantic consistency between several specification layers.

A conventional approach is to study the behavior of each application domain independently and then derive an implementation for each domain and ultimately combine them at the coding stage. A disadvantage of this approach is that the behavior of the total system, in general, need not be the sum of the individual behaviors. Moreover, this will lead to a potential danger if there arises an error in the communication between two modules belonging to two different domains of the application.

The primary motivation of the thesis comes from the intent to alleviate these problems. Even though we do not provide a complete solution for all these problems, our attempt reveals the potentials and limitations of formal specifications in industrial software environment and brings out the need for further research in this area.

1.4 Case Study : Automated Assembly Plant

The case study chosen in this thesis is the problem of performing automated assembly of mechanical parts in a single static robot environment. The three major component domains in the application are *solid modeling, robotics* and *assembly operations*.

Solid modeling deals with the representation and manipulation of complex solids in computer systems. The correctness problem here requires the resolution whether the complex solid as obtained by the implementation (derived from our specifications) is indeed a practically realizable object as expressed in the requirements. According to Requicha [Req80], it is always possible to provide a computer representation for a solid which may not exist in the real world. The correctness proofs make sure that this does not happen. The key part of solid modeling responsible for this problem is the regularized boolean operations using which primitive solids can be combined into compound solids. Therefore regularized boolean operations are specified in this thesis. For simplicity, we restrict ourselves to polyhedral solids; however the given specifications can be extended to include solids with curved faces as well.

In robotics, especially when dealing with a single static robot, the fundamental problems call for specifying forward and inverse kinematics. Every movement performed by a robot relies on the correct implementation of forward and inverse kinematic operations and hence we specify these two operations in this thesis.

Research in automated assembly is continuously evolving. At present, no standard definitions and operations have come to exist; see for example [BAL91, HoS89, HuL90, PGL89, Wol89]. Therefore verification of automated assembly operations requires a formal definition of an object, formal definition of assembly and the specification of assembly operations themselves. In this thesis, we address the first two parts, namely formally specifying objects in assembly environment and defining the concept of assembly. Automated assembly of mechanical parts as opposed to other types of assembly such as assembly of electronic components in a printed circuit board is treated in the thesis.

Some people design the software first from the informal set of requirements and then specify the design using formal techniques [Gio90]. Such specifications are called *design specifications* and the correctness of such a design is derived or argued from the design specifications. This approach is iterative in the sense that anomalies in user requirements can be found only in the design stage and hence a major change in both the specification and design will be required every time an error is found in the design phase. We therefore recommend the derivation of design from formal specifications using mathematical principles rather than from the set of informal requirements. In this context, we provide a new methodology to derive an object-oriented design from behavioral specifications. Object-oriented design methods are known to be superior to the traditional functional design methods [Boo86, Cox86, Mey88a, Mey88b]. Consequently our approach has an added advantage of designing either a functional or an object-oriented design.

1.5 Specification Techniques

The existing formal techniques can be classified into three major categories - *axiomatic approaches*, *operational specification techniques* and *definitional specification techniques*. Specifications based on propositional, predicate and temporal logic belong to the first category. The specifications in this category are expressed as a set of axioms or logical assertions. On the other hand, operational specification techniques build an abstract model of the application and specify a number of operations on the model portraying the behavior of the model. VDM (Vienna Development Method) [BjJ82] and Z [Suf82] are two well known model-based specification techniques. The third approach namely definitional approach builds a mathematical theory of the object being specified which is analogous to an algebra in mathematics. The algebraic specification technique developed by Burstall and Goguen [BuG80] and Guttag [GuH80] are examples of definitional specification technique.

From these three approaches, we have chosen the operational approach for the following reasons : (i) Operational approaches builds an abstract model of the application which is required for realization in applications such as robotics and assembly. (ii) There exist refinement techniques such as operation decomposition and data refinement which provide stepwise refinement of the model-based specifications into design. (iii) Model-based specification techniques use simple mathematical primitives to build a high-level model of the object and hence it is easier to refine the abstract model into standard structures in existing programming languages. (iv) Model-based techniques provide formal and rigorous proof techniques for reasoning about the behavior of the system being specified.

The model-based approach VDM (Vienna Development Method) is our choice for specifying the three application domains stated earlier and hence the entire assembly plant. The reasons for choosing VDM are as follows :

Syntax The language for VDM, called *Meta-IV*, has very few syntactic constructs [CHJ86, And90] and so VDM is easy to learn and use. Moreover, British Standards Institute is now standardizing the syntax of Meta-IV.

Semantics and Reasoning The goal is to specify only the behavioral aspects of the assembly plant and reason about correctness. VDM provides a sufficient set of abstract data types to model the objects in the application domains.

Top-down-design VDM is a top-down approach. Hence it is possible to state the most abstract specification of a problem in VDM and refine it stepwise into more concrete specifications. Techniques such as *operation decomposition* and *data reification* can be used to refine VDM specification.

Real-time aspects and concurrency are two important attributes which coexist in robot manipulations. However, these issues are to be associated more with the design and in particular with an implementation of the software for robot manipulations. Consequently they have to be dealt with at a lower level of the software development for robotic applications. We claim that higher level behavioral specifications for robot manipulations such as the one described in this thesis need not address the issues related to real-time and concurrent aspects. However, if the goal is to specify the control software of an actual robot, then mechanisms must be provided to map higher level specifications into real-time and concurrent specifications, the former capturing correctness issues and the latter addressing the performance aspects related to timing constraints.

1.6 Thesis Organization

The thesis is organized as follows : Chapter 2 gives a brief introduction to VDM specification technique. Only the necessary syntactic structures and their associated semantics that are used in this thesis are given; In addition, the language for VDM, **Meta-IV**, is continuously evolving and there are no standards published at the time of writing this thesis. We, however, follow the notations given in [CHJ86] which gives a fairly complete syntax of VDM published till then. The next three chapters are devoted to the specifications for each one of the application domains mentioned earlier.

The specifications for regularized boolean operations for combining polyhedral solids are given in Chapter 3. In Chapter 4, we give specifications for the structure of a general-purpose robotic agent and specify forward and inverse kinematic operations. The specifications for a robot and its operations use specifications for rigid solids and their primitive operations *translation* and *rotation*. Chapter 5 provides a formal definition of objects, their surface characteristics which are used in defining assembly between objects and outlines the verification process of an assembly. The notion of shape operators is introduced in the context of assembly. This is a new concept which distinguishes our work from others in this area. In Chapter 6, we describe a methodology for deriving an object-oriented design from a model-based specification. The methodology is illustrated for a subset of the specifications given in earlier chapters. Chapter 7 outlines future research work and provides a conclusion of the thesis. Some of the specifications stated in the Appendix are more general and can be used for a variety of applications. Consequently, they may be treated as a reusable set of library specifications.

Chapter 2

Vienna Development Method - A Brief Summary

The *Vienna Development Method* (VDM) is a high-level model-based specification technique, designed in the late 70's after the success of its predecessor *The Vienna Definition Language (VDL)*. Initially, it was designed to specify the semantics of a programming language; however it is widely used now for specifying many other applications as well.

The main advantage of VDM over its competitor Z is that VDM follows a top-down approach. This enables the specification analyst to provide a very high level abstraction of the problem and stepwise refine it towards design and implementation. Moreover, VDM uses very few simple mathematical primitives compared to the richer syntax of Z and hence is easier to read and understand. It is for these reasons that we chose VDM for the current work. One advantage of Z over VDM is that Z provides the schema calculus capability for combining two or more state space specifications into one. This is in contrast to the inherent limitation in VDM wherein everything must be stated within one state space specification. Even though this does not pose problems for several information processing applications as it is currently used, we find that it is one of the problems which future versions of VDM should address. We discuss this problem in more detail in a later chapter. A detailed treatment of VDM can be found in [BjJ87, CHJ86]; consult [Spi89] for Z syntax and refer [Hay88] for several case studies written in Z.

2.1 Primitives of a VDM Specification

VDM is based on well-defined mathematical primitives - *sets, maps, lists* and *trees*. In addition, it also provides facilities for the user to define new data types which are aggregations of the primitive types. VDM is based on the approach to programming language theory known as *denotational semantics* [CHJ86]. The behavior of an entity in VDM is specified by a set of operations defined on the abstract model of the entity. Operations are specified as a collection of predicates, grouped into two major categories, namely *pre-conditions* and *post-conditions*. Predicates are combined using the logical connectives **and** ($\wedge$), **or** ($\vee$) and **not** ($\sim$). The pre-condition for an operation is a logical formula stating the system constraints to be satisfied before the operation is invoked; that is, if any one of the predicates in the formula is false or undefined, the operation fails. The post-condition specifies the constraints that are to be satisfied after the operation successfully terminates. There are properties that are to be satisfied at every instant. These properties, called *invariants*, are also specified in VDM as a set of constraints. Invariants are further classified into *type invariants* and *state invariants*; the former refers to the properties of each data type which are to be retained at every instant of that data type while the latter corresponds to the properties of the system state.

2.1.1 Notations

Specifications in VDM use notations from logic and set theory with their conventional semantics. There is no standard notation yet for VDM. Recently, efforts are underway at National Physical Laboratory, UK for standardizing VDM notations. We follow the notations given in [CHJ86]. Some of the standard notations used in this thesis are the following :

$A \Rightarrow B$	A implies B
$A \Leftrightarrow B$	A is equivalent to B
$\forall$	for all
$\exists$	there exists
$\exists!$	there exists exactly one
$\wedge$	logical AND
$\vee$	logical OR
$\sim$	logical NOT
$\cup$	union
$\in$	set membership
$\mid$	discriminated union
$\triangleq$	defined as

In addition to these standard notations, we introduce the following additional notations and conventions in writing specifications :

- The symbol $\oplus$ is used for logical Exclusive-OR operation with the semantics that for two predicates p and q,

  $$p \oplus q = \begin{cases} false, & p = q = true\ (OR)\ p = q = false \\ true, & otherwise \end{cases}$$

- The **case** construct is used with its semantics as defined in Pascal.

- We use the notation $T_1 \times T_2 \rightarrow T$ in the signature part of auxiliary functions (explained later) to mean that there are two input variables of types T_1 and T_2 and the output variable is of type T. Consequently, when there are k (≥ 2) input variables, the notation used is

  $T_1 \times T_2 \times \ldots \times T_k \rightarrow T$

- Upper case alphabets are used for variables in declaration, while the same variables are written in lower case within expressions. Type names are written with their first letter in upper case. Individual fields of a composite type are written in upper case. When used, they are still written in upper case with the name of the composite variable in lower case; this is the selector operation in VDM.

- The variable A' in an expression indicates that variable A is updated by that expression.

- The '=' operator is overloaded in the sense that it is used both for assignment as well as for comparison. Composite objects of the same type are comparable and are compared component by component recursively until a resolution is obtained.

The following example illustrates these conventions :

Example 1 :

Primitive geometric objects such as Points and Line segments can be described in VDM using the primitive data type Nat0, the natural numbers. In the specifications given below, the two composite types 'Point' and 'Line-segment' are defined using the primitive type 'Nat0'. 'Distance' is an operation defined for points which specifies the distance between two points in space. 'Line-length' is an operation defined for line-segments which specifies the length of a line segment using the 'distance' operation for points. Thus complex geometric objects such as square and rectangle can be specified in terms of the simple ones such as point and line segment. For both 'Line-length' and 'Distance', the pre-condition is not given. This means that the pre-condition for these operations is always true. The type invariant for a line segment asserts that its end points are distinct.

$$\begin{array}{lll} \text{Point} & :: & \text{X : Nat0} \\ & & \text{Y : Nat0} \\ & & \text{Z : Nat0} \\ \text{Line-segment} & :: & \text{END-POINT1 : Point} \\ & & \text{END-POINT2 : Point} \end{array}$$

Line-length : Line-segment $\rightarrow$ Nat0

post-Line-length (l, ll) $\triangleq$ ll' = distance (END-POINT1(l), END-POINT2(l))

Distance : Point $\times$ Point $\rightarrow$ Nat0

post-Distance (p, q, d) $\triangleq$

$$d' = \sqrt{(X(p) - X(q))^2 + (Y(p) - Y(q))^2 + (Z(p) - Z(q))^2}$$

inv-Line-segment (l) $\triangleq$ END-POINT1(l) $\neq$ END-POINT2(l)

2.1.2 Consistency of VDM Specifications

Consistency of specifications can be checked by asserting that the state invariant is respected by every operation. That is,

$$\text{pre-OP} \wedge \text{inv (S)} \wedge \text{post-OP} \Rightarrow \text{inv (S')}$$

where "inv (S)" and "inv (S')" refer to the state invariants before and after the operation OP respectively and "pre-OP" and "post-OP" denote the pre- and post-conditions of OP. If every operation in the specification respects this condition, then the specification is consistent. Often it is tedious to derive formal consistency proof for every operation in the specification because additional predicates describing facts about the environment may be required. Hence a rigorous approach as suggested by Jones [Jon86] is taken. The following example illustrates these concepts :

Example 2 :

A Database for a Personal Address-Phone Book is specified below :

Assume that the names are unique and each page of the book contains information about only one person. It is also assumed that the pages are sorted using the names. The book is modeled as a list of pages.

State ::

BOOK	: Page–list
Page	:: NAME : String ADDRESS : String PHONE : Nat

INIT ()

(∗ Initialize the Book ∗)

ext BOOK : **wr** Page–list

Post book' = <>

ADD (P : Page)

(∗ Add a new page to the book. It must be assured that the name does not exist in the book before adding the page. The new name is inserted into a position such that the name before the new name is alphabetically less than the new name and the one after the new name is greater than the new name and the other pages in the book are not corrupted due to insertion. ∗)

ext BOOK : **wr** Page–list

Pre

$(\forall \, i \in \{1 \cdot\cdot \textbf{ len } \text{book}\}) \; (\text{NAME (book[i])} \neq \text{NAME(p)})$

Post

$(\sim (\exists\ k \in \{1 \cdot\cdot\ \textbf{len}\ book\})\ (NAME(book[k]) < NAME(p))$

$\Rightarrow (book'[1] = p) \wedge$

$(\forall\ i \in \{1 \cdot\cdot\ \textbf{len}\ book\})\ (book'[i+1]) = book[i])$

$) \oplus$

$(\sim (\exists\ k \in \{1 \cdot\cdot\ \textbf{len}\ book\})\ (NAME(book[k]) > NAME(p))$

$\Rightarrow (\forall\ i \in \{1 \cdot\cdot\ \textbf{len}\ book\})\ (book'[i] = book[i]) \wedge$

$(book'[\textbf{len}\ book + 1] = p)$

$) \oplus$

$(\exists!\ k \in \{1 \cdot\cdot\ \textbf{len}\ book\})$

$((NAME(book[k]) < NAME(p)) \wedge$

$(NAME(book[k+1]) > NAME(p)) \Rightarrow$

$(\forall\ j \in \{1 \cdot\cdot\ k\})\ (book'[j] = book[j]) \wedge$

$(book'[k+1] = p) \wedge$

$(\forall\ j \in \{k+1 \cdot\cdot\ \textbf{len}\ book\})\ (book'[j+1] = book[j]))$

DELETE (N : String)

(* Delete an already existing page corresponding to the given name; if the page does not exist, the operation fails. The post condition asserts that the intended page does not appear in the updated book. In addition, it is also to be stated that pages are not corrupted by the DELETE operation. The reason for introducing such a strong post condition is that it is possible to have several implementations satisfying the first condition, but corrupting the book; for example, some other page may also get deleted or pages might have been rearranged changing the order.*)

ext BOOK : **wr** Page–list

Pre

$(\exists!\ i \in \{1 \cdot\cdot\ \textbf{len}\ book\})\ (NAME(book[i]) = n)$

Post

$\sim (\exists\ k \in \{1 \cdot\cdot\ \textbf{len}\ book')\ (NAME(book'[k]) = n) \wedge$

$(\forall\ i \in \{1 \cdot\cdot\ \textbf{len}\ book\})$

$((NAME(book[i]) < n \Rightarrow book'[i] = book[i]) \wedge$

$(NAME(book[i]) > n \Rightarrow book'[i-1] = book[i]))$

GET-PHONE (N : String) P : Nat

(* Get the phone number for the given name. *)

ext BOOK : **rd** Page–list

Pre

$(\exists!\ i \in \{1 \cdot\cdot\ \mathbf{len}\ book\})\ (NAME(book[i]) = n)$

Post

$(\exists!\ i \in \{1 \cdot\cdot\ \mathbf{len}\ book\})$

$((NAME(book[i]) = n) \wedge (p' = PHONE(book[i])))$

<u>State Invariant :</u> The state invariant in this case asserts that the names are unique at any instant and they are alphabetically ordered at all times.

inv-State $\triangleq$

$(\forall\ i,j \in \{1 \cdot\cdot\ \mathbf{len}\ book\})$

$((i \neq j \Rightarrow NAME(book[i]) \neq NAME(book[j])) \wedge$

$(i < j \Rightarrow NAME(book[i]) < NAME(book[j]))$

)

<u>Consistency Check :</u>

We give the proof for only one operation – ADD. The proofs for other operations are similar.

We prove the consistency using the convention

pre-Op $\wedge$ inv (s) $\wedge$ post-Op $\Rightarrow$ inv (s')

Since the post-condition consists of three predicates connected by $\oplus$, only one of them can occur at any one time. Let us take the most general case; proofs for the other two cases are most straightforward. Thus, it is be proved that

$(\forall\ i \in \{1 \cdot\cdot\ \mathbf{len}\ book\})\ (NAME(book[i]) \neq NAME(book[p])) \wedge$

$(\forall\ i,j \in \{1 \cdot\cdot\ \mathbf{len}\ book\})$

$((i \neq j \Rightarrow NAME(book[i]) \neq NAME(book[j])) \wedge$

$(i < j \Rightarrow NAME(book[i]) < NAME(book[j]))) \wedge$

$(\exists!\ k \in \{1 \cdot\cdot\ \mathbf{len}\ book\})$

$((\mathrm{NAME}(\mathrm{book}[k]) < \mathrm{NAME}(p)) \wedge$

$(\mathrm{NAME}(\mathrm{book}[k+1]) > \mathrm{NAME}(p)) \Rightarrow$

$(\forall j \in \{1 \cdots k\})\ (\mathrm{book}'[j] = \mathrm{book}[j]) \wedge$

$(\mathrm{book}'[k+1] = p) \wedge$

$(\forall j \in \{k+1 \cdots \textbf{len}\ \mathrm{book}\})\ (\mathrm{book}'[j+1] = \mathrm{book}[j])) \Rightarrow$

$(\forall i,j \in \{1 \cdots \textbf{len}\ \mathrm{book}'\})$

$((i \neq j \Rightarrow \mathrm{NAME}(\mathrm{book}'[i]) \neq \mathrm{NAME}(\mathrm{book}'[j])) \wedge$

$(i < j \Rightarrow \mathrm{NAME}(\mathrm{book}'[i]) < \mathrm{NAME}(\mathrm{book}'[j])))$

From the post-condition, it can be easily observed that all the pages in the old book are retained and the new page is added among them. Since the invariant is true before the operation, for any indexes i and j in the new book, if book[i] and book[j] are pages already available in the old book, then it is true that NAME[i] $\neq$ NAME[j]. Hence it is to be proved that there is no page in the old book that has the same name as the new name to be added. This is assured by the pre-condition and hence it is proved that

$(\forall i, j \in \{1 \cdots \textbf{len}\ \mathrm{book}'\})\ (i \neq j \Rightarrow (\mathrm{NAME}(\mathrm{book}'[i] \neq \mathrm{NAME}(\mathrm{book}'[j])))$

For the second part of the proof, we have to show that

$(\forall i, j \in \{1 \cdots \textbf{len}\ \mathrm{book}')$

$(i < j \Rightarrow \mathrm{NAME}(\mathrm{book}'[i]) < \mathrm{NAME}(\mathrm{book}'[j]))$

There are four different situations to be considered here; let the new page be added at the $(k+1)^{th}$ position :

<u>Case 1</u> : i,j < (k+1)

In this case, NAME(book'[i]) < NAME(book'[j]), since all the pages from 1 to k in the new book are retained from the old book as stated in the post-condition and the invariant is true before the operation.

<u>Case 2</u> : i,j > (k+1)

The same arguments in Case 1 are applicable here as well.

<u>Case 3</u> : i < (k+1) and j = (k+1)

From the post-condition, it is clear that the new page is greater than the k^{th} page; i.e.,

NAME(book[k]) < NAME(p)

and the first k pages are retained. Hence the following are true :

$(1 \leq i \leq k) \wedge (NAME(book[i]) < NAME(p))$

Case 4 : $i = (k+1)$ and $j > (k+1)$

The same arguments as in Case 3 are applicable.

Therefore we have proved the consistency of the specification with respect to the stated invariants. In addition to consistency checking, we used a rigorous approach to reason about the system. A number of such rigorous proofs are shown, for example, on solid modeling, robotic motion and assembly in subsequent sections.

2.1.3 Some more conventions

The **let** ...**tel** clause is used to simplify expressions. For example,

let r = RADIUS (Cir) **in**

A' = $\Pi * r^2$

tel

is an abbreviation for A' = $\Pi *$ (RADIUS (Cir))^2.

We also introduce *type equivalences* such as

Line = Axis–Rep

Solid = Structure

in our specifications. Types T_1 and T_2 are type equivalent in the sense that a variable of type T_1 can be used in a place where a variable of type T_2 is expected.

In VDM, two different styles of specifications, namely pure *functional style* and *model-based style* can be followed. If a specification does not require an explicit reference to a global variable, then it can be written using the functional style. However, when a specification requires an explicit reference to one or more global variables, which incidentally define the *state* of the system, it is written in model-based style. Another major difference between these two styles is the presence of **ext** clause in the model-based style that lists all the global variables accessed in that operation with their read/write attributes. Invariably, both styles of specifications will be required for a large problem specification.

We use a number of *auxiliary functions* in the specifications. These are, in fact, modules of a complex specification. They are generally written using the functional style of VDM. In this thesis, auxiliary functions represent actions returning some values as opposed to the constraints which always return logical values. The following example illustrates the use of auxiliary functions.

Example 3 :

From plane geometry, it is well known that two intersecting line segments have only one point in common to both line segments. In designing a geometric reasoning system, this requirement can be specified as follows :

Intersect : Line-segment $\times$ Line-segment $\rightarrow$ Point

post-Line-segment (l_1, l_2, p) $\triangleq$

(p' = convex-comb (END-POINT1(l_1), END-POINT2(l_1))) $\wedge$

(p' = convex-comb (END-POINT1(l_2), END-POINT2(l_2)))

Convex-comb : Point $\times$ Point $\rightarrow$ Point

post-Convex-comb (p, q, r) $\triangleq$ $(\exists\, \lambda,\ 0 \leq \lambda \leq 1)\ (r' = \lambda\, p + (1 - \lambda)\, q)$

Here, 'Convex-comb' is an auxiliary function returning a point which is the convex combination of two points p and q.

Chapter 3

Regularized Boolean Operations

One of the important tasks in solid modeling is the creation of complex solids from primitive solids. Almost all solid modeling techniques use regularized boolean operations to combine solids into compound solids [Req80]. However, they are all algorithmic in nature; i.e., they describe how the compound solid can be created and/or represented in the modeling system. Our goal is to study the behavior of compound solids and reason about their behavior in an assembly environment. Hence it is necessary to build abstract models of primitive solids and specify the regularized boolean operations using these abstract models. In this chapter, we provide formal specifications for regularized operations for combining polyhedral solids. Generalization to solids with curved faces is not treated in this thesis.

3.1 Solid Modeling

Solid modeling is a design aid for the construction of complex mechanical structures. Software for solid modeling deals with representation and manipulation of solids within computer systems. Existing solid modeling techniques such as Constructive Solid Geometry (CSG), Boundary Representation (B-Rep) and Cell Decomposition are all constructive in the sense that they describe algorithmically how to create and represent complex solids from primitive solids, but do not address the question of correctness of those algorithms. It is mandatory to ensure correctness of the solid modeler; otherwise the software may represent objects which are not practically realizable. Requicha claims that in computer systems it is possible to represent solids which do not exist in reality. We therefore formally characterize the representations and specify operations of a solid modeler. The most important operation of a solid modeler which exhibits its behavior is that which combines primitive solids into compound solids. Other operations such as boundary approximation, boundary evaluation and membership classification [ReV85, Til80] are representation dependent and deal with the implementation of the solid modeler rather than with its behavior. Almost all solid modeling techniques use regularized boolean operations for combining solids [Req80]. Hence in this chapter, we formally represent a solid and specify the regularized boolean operations within this abstract model.

3.1.1 Regularized Operations

The three Boolean operations supported by a majority of solid modelers are *union*, *intersection* and *difference*. A straightforward application of these set theoretic operations on solids may produce a *dangling edge* or a *dangling face* (see Figure 3.1 and Figure 3.2). Hence Requicha [Req80] has considered the theory of r-sets and r-set operations for modeling solids. This theory views a solid as a closed point set and mandates that the result of every regularized operation is also a closed set.

For two point sets X and Y, the regularized operations are defined as

$X \cup^* Y = k\, i\,(X \cup Y)$

$X \cap^* Y = k\, i\,(X \cap Y)$

$X -^* Y = k\, i\,(X - Y)$

$c^*X = k\ i\ (cX)$

where k refers to *closure* and i refers to *interior* of a point set. This definition can be viewed as a very high level specification for regularized operations. Algorithms of Tilove [Til80, Til84] and Requicha [ReV85] are naive one step implementations of the above definition. However, using these algorithms, it is difficult to reason about the behavior of the operations which combine the solids. For example, proving that the implemented operations do not create dangling edges or dangling faces, requires neighborhood approximations and hence is a tedious task. Hence we move to the next level of abstraction to define the solid and the regularized operations and prove that these abstract operations will not produce any dangling edges or dangling faces. Any implementation derived from these formal specifications will therefore respect

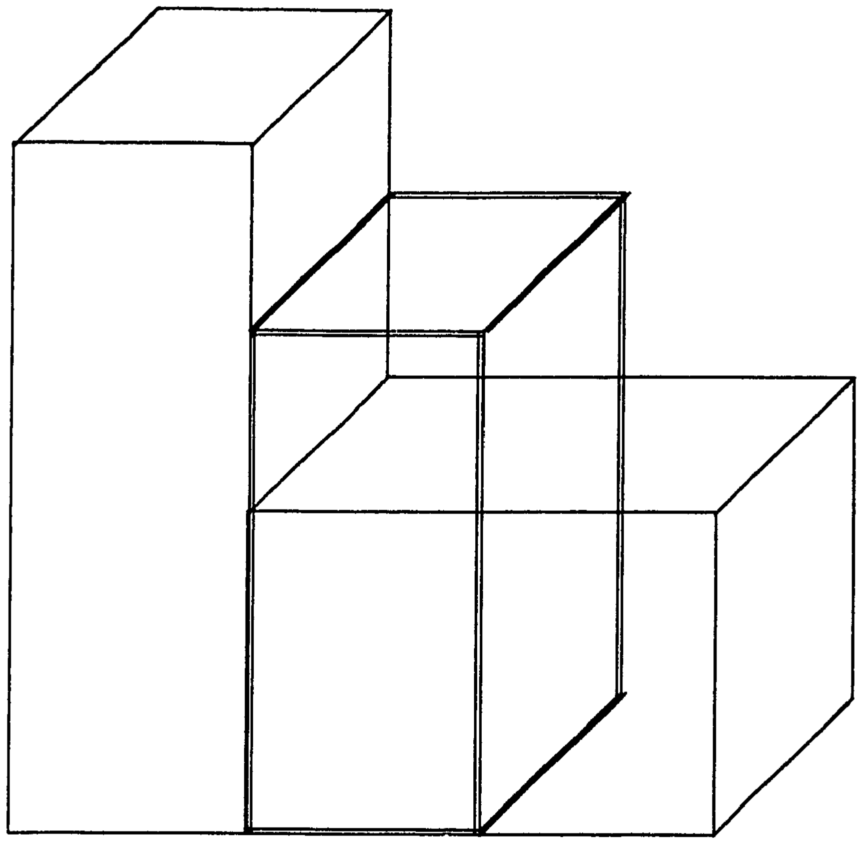


Figure 3.1: Intersection of Solids

the r-set theory. In addition, we have also discussed extensions to the representation techniques to include physical properties of solids [ABP88b, ABP90].

3.2 Abstract Model of a Solid

As mentioned earlier, we consider only polyhedral solids in this thesis. An abstract polyhedral solid is defined in terms of its bounding surfaces, called *faces*. It is to be noted that the abstract model is not a boundary representation of a solid; it provides only an abstract view of the solid. Each face is identified by the set of its vertices and by its ordered collection of *directed edges*. An edge, in turn, consists of two *vertices* V1 and V2, and is directed, say from V1 to V2. The ordering imposed on the edges of a face and the direction of each edge, enables us to find out the interior of a face. Associated with each face is a vector, the *normal* to the face whose

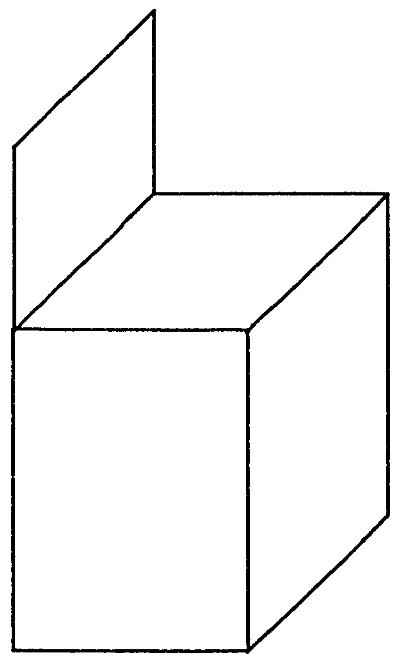


Figure 3.2: Dangling Face due to Intersection

direction determines the side of the face contributing to the interior of the solid. *This normal is a directed line segment perpendicular to the plane of the face (and hence perpendicular to every edge belonging to this face) such that looking from the other end of the normal towards the face enables one to see the direction of edges follow one convention (say clockwise).*

This definition of normal allows us to view the interior of an object from its collection of faces. See Figure 3.3 and Figure 3.4. The directed edges of the face with vertices (1,2,3,8) are {<1,8>, <8,3>, <3,2>, <2,1>}. Similarly, the directed edges of the face with vertices (1,2,5,6) are {<1,2>, <2,5>, <5,6>, <6,1>}. Notice that the edge common to these two faces has opposite directions as shown in Figure 3.3. In Figure 3.4, the hollow solid is obtained by scooping out the solid portion with vertices (9,10,...,16) from the solid with vertices (1,2,...,8). The resulting solid has sixteen vertices (1,2,...,16) and ten faces. Notice that the directed edge <13,12> of the face with vertices (12,13,14,15) of the inner solid becomes the directed edge <12,13> of the face with vertices (4,5,6,7,12,13,14,15) in the resulting solid. Similar remarks apply to other edges. The concept of normal as defined here is central to our discussion in later sections.

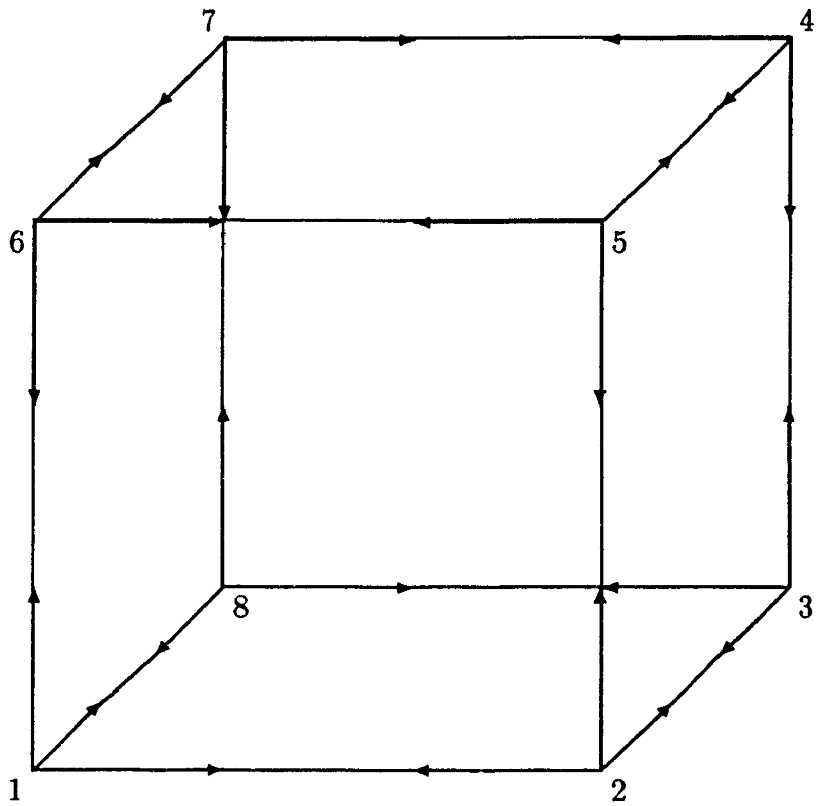


Figure 3.3: Directed Edges in a Solid

In order to state and prove type invariants, we need an initial set of hypotheses. These are enumerated below :

3.2.1 Initial Hypotheses

H1 Every input solid is polyhedral with no holes and has a well-defined interior. It is enclosed by at least four bounded planes called *faces.*

H2 A face is a bounded plane defined by the enclosure of at least three line segments called *edges.* The edges of a given face lie in the same plane.

H3 Associated with each face is a directed line segment called *normal* which points to the interior of the solid of which this face is a constituent. By the definition

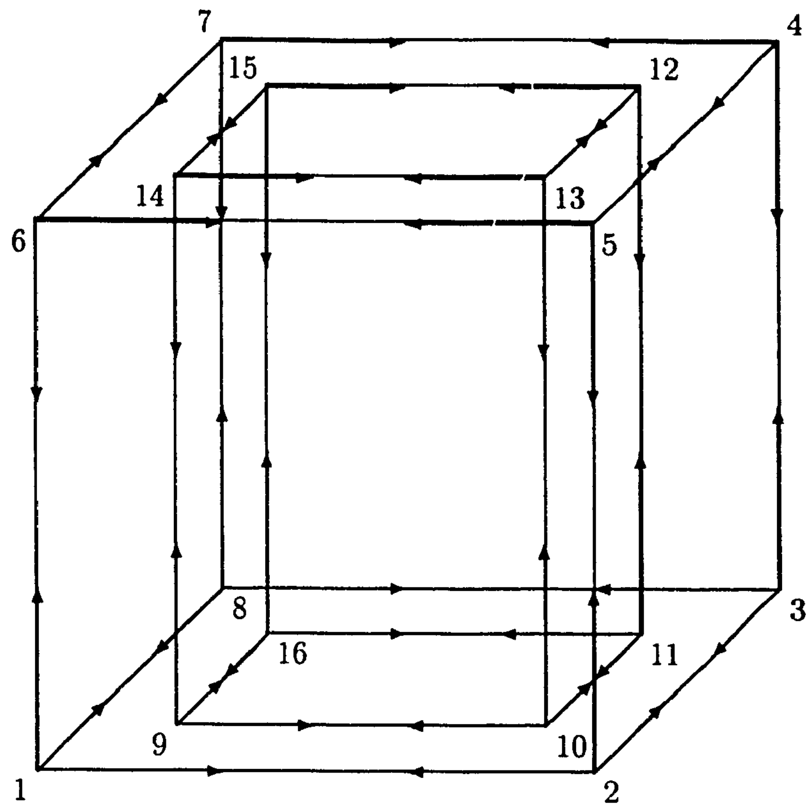


Figure 3.4: Directed Edges in a Hollow Solid

of normal, this directed line segment is perpendicular to the bounded plane of the face and hence is perpendicular to every edge of the face.

H4 Every face is *simply connected*; that is, starting from a vertex of an edge, it is possible to traverse sequentially through the adjacent edges (recall that the edges are directed) and return to the same vertex.

H5 Two faces sharing an edge are adjacent and normals to adjacent faces are distinct.

H6 An edge is incident to two distinct points called *vertices*; these are the end points of the edge.

H7 Two edges of a face having one vertex in common are adjacent and vertices of adjacent edges are non-collinear.

H8 Every input solid s is a regular solid; i.e., s does not have a dangling edge or a dangling face. Every vertex in s is shared by at least two edges and every edge in s is shared exactly by two faces of s.

3.2.2 Illustration of the Boolean Operations

In this section we illustrate the three regularized Boolean operations - UNION, DIFFERENCE and INTERSECTION with an example. In general, the result of applying one of these operations to two solids s_1 and s_2 may produce a single solid s_3 or a collection of solids s_3. The input solids satisfy the properties H1-H8; however, the resulting solid s_3 may not inherit all these properties. For example, the difference operator may produce a solid with holes in some faces. That is, some faces may enclose a *multiply connected region*, which is a region defined by a disjoint collection of edges with each collection of edges satisfying H4. See Figure 3.5. However all the other stated properties, in particular the regularity property H8, will be inherited by the resulting solid.

A face f of s_3 is created from only one of the following possibilities : (1) It is an unmodified face of s_1 or an unmodified face of s_2; (2) It is a face of s_1 modified due to the interference of one or more faces of s_2; (3) It is a face of s_2 modified due to the interference of one or more faces of s_1.

Example 1 :

Consider the two objects s_1 and s_2 shown in Figure 3.6. For notational convenience, the vertices of s_1 are given in upper case letters and those of s_2 are given in lower case letters. We denote a face by its bounding vertices. The table following the figure gives the faces of s_3, created by each one of the operations on s_1 and s_2. The ordering of faces and the starting vertex of each face is immaterial for the discussion. For each face the direction of edges are given in clockwise direction as seen from its normal.

UNION

Face 1	-	V1,V2,V3,V4,V1	unmodified
Face 2	-	V1,V4,V8,V5,V1	unmodified
Face 3	-	V1,V5,V6,V2,V1	unmodified
Face 4	-	V5,V8,V7,V6,V5	unmodified

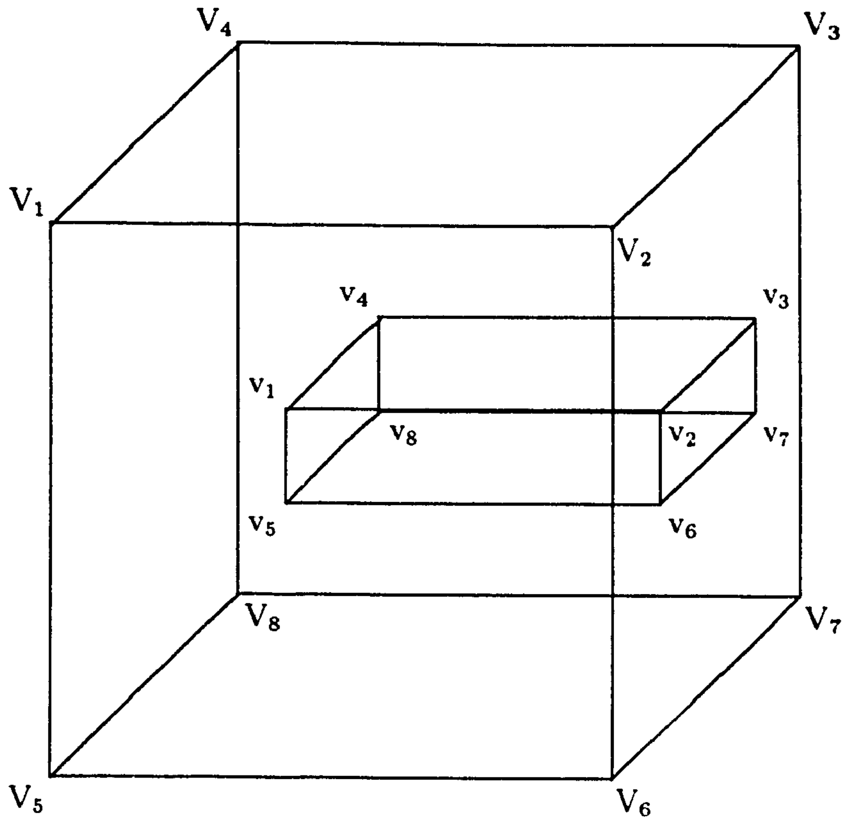


Figure 3.5: Solid Obtained as the Difference of Two Solids

Face 5	-	V4,V3,v14',v4',v8',v58',V7,V8,V4	modified from V4,V3,V7,V8,V4
Face 6	-	V3,V2,V6,V7,v58',v5',v1',v14',V3	modified from V3,V2,V6,V7,V3
Face 7	-	v3,v2,v6,v7,v3	unmodified
Face 8	-	v2,v3,v4',v14',v1',v2	modified from v2,v3,v4,v1,v2
Face 9	-	v3,v7,v8',v4',v3	modified from v3,v7,v8,v4,v3
Face 10	-	v7,v6,v5',v58',v8',v7	modified from v7,v6,v5,v8,v7
Face 11	-	v2,v1',v5',v6,v2	modified from v2,v1,v5,v6,v2

<u>DIFFERENCE</u>

Face 1	-	V1,V2,V3,V4,V1	unmodified
Face 2	-	V1,V4,V8,V5,V1	unmodified
Face 3	-	V1,V5,V6,V2,V1	unmodified
Face 4	-	V5,V8,V7,V6,V5	unmodified

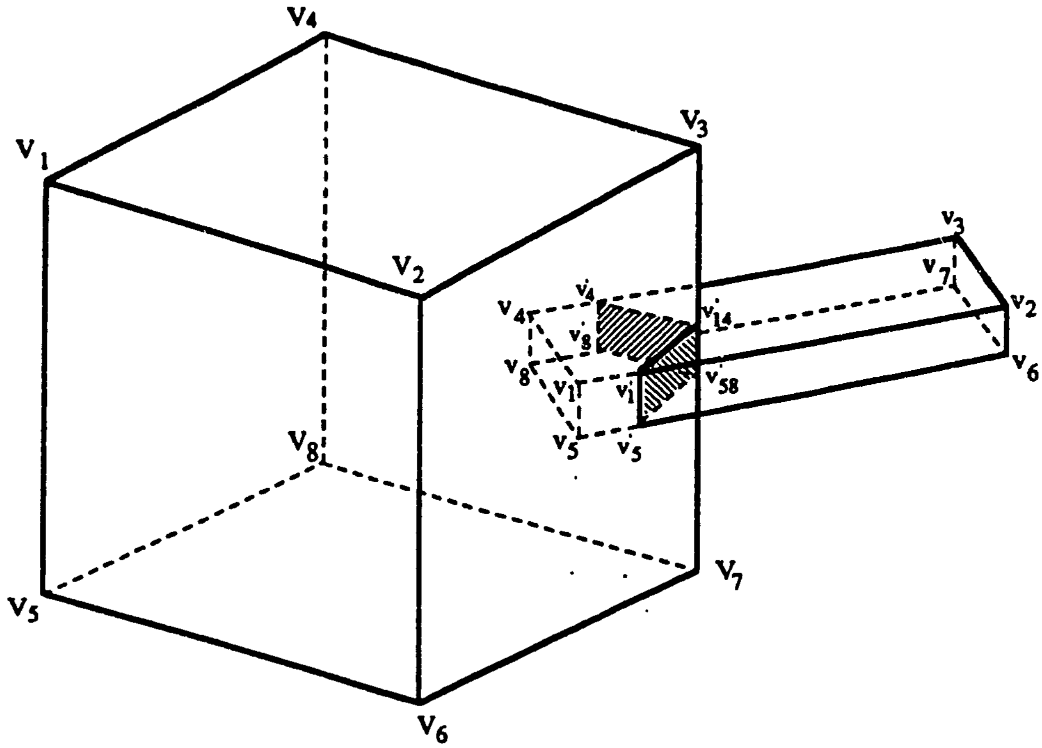


Figure 3.6: Regularized Boolean Operations on Solids

Face 5	-	V4,V3,v14',v4',v8',v58',V7,V8,V4	modified from V4,V3,V7,V8,V4
Face 6	-	V3,V2,V6,V7,v58',v5',v1',v14',V3	modified from V3,V2,V6,V7,V3
Face 7	-	v4,v1,v5,v8,v4	unmodified
Face 8	-	v1,v1',v5',v5,v1	modified from v2,v1,v5,v6,v2
Face 9	-	v8,v5,v5',v58',v8'v8	modified from v8,v7,v6,v5,v8
Face 10	-	v4,v8,v8',v4',v4	modified from v4,v3,v7,v8,v4
Face 11	-	v4',v14',v1',v1,v4,v4'	modified from v1,v2,v3,v4,v1

<u>INTERSECTION</u>

Face 1	-	v1,v4,v8,v5,v1	unmodified
Face 2	-	v1,v5,v5',v1',v1	modified from v2,v1,v5,v6,v2
Face 3	-	v1',v5',v58',v14',v1'	modified from V3,V2,V6,V7,V3
Face 4	-	v4',v14',v58',v8',v4'	modified from V4,V3,V7,V8,V4
Face 5	-	v4,v4',v8',v8,v4	modified from v4,v3,v7,v8,v4
Face 6	-	v8,v8',v58',v5',v5,v8	modified from v8,v7,v6,v5,v8
Face 7	-	v4,v1,v1',v14',v4',v4	modified from v1,v2,v3,v4,v1

It is clear from this example that the resulting solid s_3 can be defined unambiguously if s_1 and s_2 have been defined in terms of their respective faces and normals. Since each face f of s_3 is either a modified or unmodified face f of s_1 or s_2, the normal of f can be easily computed from f; this is because of the fact that the normal to a face is invariant due to the modification of the face. Hence the interior of the resulting solid s_3 can be defined using the normals to its faces. In section 3.5, we use these invariant properties to assert the validity of the specifications.

3.3 Specifications for Boolean Operations

We follow a top-down approach in providing the specifications for the three operations - UNION, DIFFERENCE and INTERSECTION. The constraints may include auxiliary functions whose specifications are given. Simple functions are not specified and only their signatures are stated at the end. The data type modeling an object is given below :

Object	::	TYPE : {SIMPLE, COMPOSITE} FACES : Facetype-set
Facetype	::	FACEID : Nat0 NORMAL : Direction EDGES : Edgetype-list-set
Edgetype	::	EDGEID : Nat0 VERTICES : Point-list
Surface	=	Point-set
Operation	=	{UNION, INTERSECT, DIFFERENCE}

An object can be a simple polyhedron (satisfying the properties H1-H8) or a composite polyhedron (a polyhedron with holes in some faces). For simplicity of discussions, we assume that every input object type is SIMPLE, although the type of the result may be COMPOSITE. It is easy to modify our specifications for COMPOSITE type inputs as well.

An object is defined in terms of its faces without regard to the ordering of faces and so the field 'FACES' is defined as a set. However, for each face, there is an ordering imposed on the edges due to the definition of normal (see Figure 3.3) and so 'EDGES' is defined as a list-set. The cardinality of this set is 1 for each face of a SIMPLE type object and for COMPOSITE type, the cardinality of the set is greater than 1 for at least one face. Each element of EDGES is a list of vertices ordered by the direction of traversal of edges required by the normal to this face. The identification fields in Facetype and Edgetype are essential. An edge may be shared by two faces of the same object type and two faces of two different object types may be touching each other. In such situations we need to identify each edge or face without any ambiguity. So, we insist on unique EDGEIDs and FACEIDs. This is also justified because an edge e shared by two faces f_1 and f_2 has opposite directions when viewed from their normals; hence EDGEID (e) in $f_1 \neq$ EDGEID (e) in f_2 must hold. So, we add the additional hypothesis :

H9 Every edge in the input solid has a unique EDGEID and every face has a unique FACEID. Consequently the cardinality of the set of EDGEIDs representing the edges of a given face equals the number of edges in that face.

Notice that no identification field is associated with the object definition given above. However, when this specification is embedded in the specification of an environment such as robotic assembly of mechanical parts, the identification field for objects may become necessary.

Types 'Point' and 'Direction' in the above specifications are assumed to have been defined already. 'Nat0' refers to the set of natural numbers including zero. It is to be noted that the above types capture the inherent structure of any polyhedral object in terms of its bounding faces, edges and vertices without regard to any specific representation.

The specifications for the three regularized Boolean operations - union, intersection and difference are given next. Since the two operations difference and intersection can produce multiple objects, the result of Boolean operations is of type Object-set.

BOOLEAN-SOLIDS (s_1, s_2 : Object, OPR : Operation) s_3 : Object-set
(* The specifications cover the boolean operations UNION, INTERSECTION and

DIFFERENCE. The type of operation is passed as a parameter OPR. The result is a set of objects. *)

Post

$s_3 = \{s \mid s \in \text{Object} \wedge \text{validate}\ (s, s_1, s_2, \text{opr})\}$

(* The operation 'Validate' ensures that the resulting solid s obtained by the operation Opr on solids s_1 and s_2, indeed satisfies the properties of a SIMPLE or COMPOSITE polyhedron. *)

Validate : Object × Object × Object × Operation ⇒ Boolean

pre-Validate (s, s_1, s_2, opr) $\triangleq$

(* the input solids are of type SIMPLE. *)

$(\forall f_1 \in \text{FACES}\ (s_1))$ (**card** EDGES(f_1) = 1)

$\wedge$ $(\forall f_2 \in \text{FACES}(s_2))$ (**card** EDGES(f_2) = 1)

post-Validate (s, s_1, s_2, opr, b) $\triangleq$

b' ⇔ **let** f-col$_1$ = FACES (s_1), f-col$_2$ = FACES (s_2),

f-col$_3$ = FACES (s) **in**

case opr **of**

(* For the 'UNION' operation, each face of the new object must be either from s_1 or from s_2 or newly created from the faces of s_1 and s_2; however, the three domains from which this face is created, must all be distinct. *)

UNION : (* post-conditions for UNION *)

(* assert the volumetric properties *)

volu-space (s_1) ⊂ volu-space (s) ∧

volu-space (s_2) ⊂ volu-space (s) ∧

($\forall f_3 \in$ f-col$_3$)

(($\exists! f_1 \in$ f-col$_1$)

(f_3 = *mk-facetype* (get-new-faceid, NORMAL (f1),

copy-edges (f1)))

⊕ ($\exists! f_2 \in$ f-col$_2$)

(f_3 = *mk-facetype* (get-new-faceid, NORMAL (f2), copy-edges (f2)))

$\oplus$ $f_3 \in$ create-faces (s_1, s_2, opr))

(* For the 'DIFFERENCE' operation, every face f_3 of the newly created solid s must be a face f_1 of solid s_1, or it can be a face f_2 of the solid s_2, provided that it is within the solid s_1 or a newly created face. Note that the 'DIFFERENCE' operation subtracts volume of solid s_2 from s_1 and so s is a part or whole of s_1. This implies that if a face f_3 in s is an unmodified face f_2 of s_2, then it will have its NORMAL reversed. *)

DIFFERENCE : (* post-conditions for DIFFERENCE *)

(* assert the volumetric properties *)

volu-space (s) $\subset$ volu-space (s_1) $\wedge$

($\forall$ $f_3 \in$ f-col_3)

(($\exists!$ $f_1 \in$ f-col_1)

(f_3 = *mk-facetype* (get-new-faceid, NORMAL (f1),

copy-edges (f1)))

$\oplus$ ($\exists!$ $f_2 \in$ f-col_2)

(bounded-plane (f_3) $\subseteq$ volu-space (s_1) $\wedge$

f_3 = *mk-facetype* (get-new-faceid, reverse (NORMAL (f2)),

copy-edges (f2)))

$\oplus$ $f_3 \in$ create-faces (s_1, s_2, opr))

(* For the 'INTERSECT' operation, every face f_3 of solid s must be a face f_1 of solid s_1, provided it is within solid s_2, or a face f_2 of solid s_2 provided it is within solid s_1 or it belongs to the newly created set of faces. *)

INTERSECT : (* post-condition for INTERSECTION *)

(* assert the volumetric properties *)

volu-space (s) $\subset$ volu-space (s_1) $\wedge$

volu-space (s) $\subset$ volu-space (s_2) $\wedge$

($\forall$ $f_3 \in$ f-col_3)

(($\exists!$ $f_1 \in$ f-col_1)

(f_3 = *mk-facetype* (get-new-faceid, NORMAL (f1),
copy-edges (f1)) $\wedge$
bounded-plane (f_3) $\subseteq$ volu-space (s_2))

$\oplus$ ($\exists!$ $f_2 \in$ f-col_2)
(f_3 = *mk-facetype* (get-new-faceid, NORMAL (f2),
copy-edges (f2)) $\wedge$
bounded-plane (f_3) $\subseteq$ volu-space (s_1))

$\oplus$ $f_3 \in$ create-faces (s_1, s_2, opr))

endcase

(* assert that adjacent faces are not coplanar. *)

$\wedge$ ($\forall$ f_{31}, $f_{32} \in$ f-col_3)
(Adjacent-faces (f_{31}, f_{32}) $\Rightarrow$
NORMAL (f_{31}) $\neq$ NORMAL (f_{32}) $\wedge$
NORMAL (f_{31}) $\neq$ reverse (NORMAL (f_{32})))

(* construct the new object s. *)

$\wedge$ <u>if</u> **card** EDGES (f-col_3) = 1 <u>then</u>
s' = *mk-object* (TYPE = 'SIMPLE', f-col_3)
<u>else</u> s' = *mk-object* (TYPE = 'COMPOSITE', f-col_3)

tel

(* Function 'copy-edges' produces a copy of all edges of the input face f, with EDGEIDs replaced by new ids. *)

Copy-edges : Facetype $\rightarrow$ Edgetype-list-set
pre-Copy-edges (f) $\triangleq$ f $\neq$ NIL
post-Copy-edges (f, set-of-e-col) $\triangleq$
($\forall$ e-col $\in$ set-of-e-col)
(($\exists!$ e-col_1 $\in$ EDGES (f))
(($\forall$ e $\in$ **elems** e-col)
(($\exists!$ e_1 $\in$ **elems** e-col_1)
(e'= *mk-edgetype* (get-new-edgeid, VERTICES (e_1))))))

(* The function 'Adjacent-faces' checks whether the two faces f_1 and f_2 share a common edge in reverse directions. *)

Adjacent-faces : Facetype $\times$ Facetype $\rightarrow$ Boolean

pre-Adjacent-faces $(f_1, f_2) \quad \triangleq \quad (f_1 \neq \text{NIL}) \wedge (f_2 \neq \text{NIL})$

post-Adjacent-faces $(f_1, f_2, b) \quad \triangleq$

$$b' \Leftrightarrow (\exists\, e_1 \in \textbf{elems}\ el_1 \mid el_1 \in \text{EDGES}\ (f_1))$$
$$((\exists!\ e_2 \in \textbf{elems}\ el_2 \mid el_2 \in \text{EDGES}\ (f_2))$$
$$(\text{VERTICES}\ (e_1) = \textbf{rev}\ \text{VERTICES}\ (e_2)))$$

(* 'create-faces' is a function which creates a new set of faces from two collections of faces belonging to two different solids. The specification for this function shown below also checks for the relationships between the three sets of faces. *)

Create-faces : Object $\times$ Object $\times$ Operation $\rightarrow$ Facetype-set

post-Create-faces $(s_1, s_2, opr, \text{f-col}_3) \quad \triangleq$

let $\text{f-col}_1 = \text{FACES}\ (s_1), \text{f-col}_2 = \text{FACES}\ (s_2)$ **in**

(* f-col_3 denotes only newly created faces. *)

$(\text{f-col}_1 \cap \text{f-col}_3 = \{\}) \wedge (\text{f-col}_2 \cap \text{f-col}_3 = \{\}) \wedge$

$(\forall\, f_3 \in \text{f-col}_3)$

(**case** opr **of**

UNION :

$(\exists!\ f_1 \in \text{f-col}_1)$

$(f_3 = \text{intersect-face}\ (f_1, \text{f-col}_2) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_2))$

$\oplus\ (\exists!\ f_2 \in \text{f-col}_2)$

$(f_3 = \text{intersect-face}\ (f_2, \text{f-col}_1) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_1))$

DIFFERENCE :

$(\exists!\ f_1 \in \text{f-col}_1)$

$(f_3 = \text{intersect-face}\ (f_1, \text{f-col}_2) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_2))$

$\oplus\ (\exists!\ f_2 \in \text{f-col}_2)$

$(f_3 = \text{intersect-face}\ (f_2, \text{f-col}_1) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_1) \wedge$

$\text{NORMAL}\ (f_3) = \text{reverse}\ (\text{NORMAL}\ (f_2)))$

INTERSECT :

$((\exists!\ f_1 \in \text{f-col}_1)$

$(f_3 = \text{Intersect-face}\ (f_1,\ \text{f-col}_2) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_2))$

$\vee\ (\exists!\ f_2 \in \text{f-col}_2)$

$(f_3 = \text{intersect-face}\ (f_2,\ \text{f-col}_1) \wedge$

$\text{bounded-plane}\ (f_3) \subseteq \text{volu-space}\ (s_1)))$

(* If the newly created face can be obtained as a modified face f_1 of s_1 and can also be obtained as a modified face f_2 of s_2, then normals of f_1 and f_2 should point in the same direction in order to assure that the resulting face contributes to the interior of s_3. *)

$\wedge$ <u>if</u> $f_3 = \text{intersect-face}\ (f_1,\ \text{f-col}_2) \wedge$

$f_3 = \text{intersect-face}\ (f_2,\ \text{f-col}_1)$ <u>then</u>

$\sim(\text{opposite-direction}\ (\text{NORMAL}\ (f_1),\ \text{NORMAL}\ (f_2)))$

endcase)

tel

(* The function 'Intersect-face' computes a new face created by the intersection of a face with another set of faces. Since the newly created face is the original face modified only by the intersection of all faces in the second parameter, the NORMAL of the newly created face is set to that of the original face. *)

Intersect-face : Facetype $\times$ Facetype-set $\longrightarrow$ Facetype

post-Intersect-face $(f_i,\ \text{f-col},\ f_r)\ \triangleq$

$\sim$ unmodified-face $(f_i,\ f_r) \wedge$

let set-of-e-col$_i$ = EDGES (f_i), set-of-e-col$_r$ = EDGES (f_r) **in**

$(\forall\ \text{e-col}_r \in \text{set-of-e-col}_r)$

$((\forall\ i \in 1 \cdot\cdot$ **len** e-col$_r$)

(**let** e = e-col$_r$(i) **in**

$(\exists!\ \text{e-col}_i \in \text{set-of-e-col}_i)$

$((\exists!\ e_1 \in$ **elems** e-col$_i$)

(e' = *mk-edgetype* (get-new-edgeid, VERTICES (e_1)))))

(* unmodified edges of f_i *)

$\oplus$ ($\exists!$ f $\in$ f-col)

(~ unmodified-face (f, f_r) $\wedge$

($\exists!$ e-col_2 $\in$ EDGES (f))

(($\exists!$ e_2 $\in$ **elems** e-col_2)

(planar (f_i, e_2)

(* assert that e_2 lies in the plane of f_i. *)

$\wedge$ e' = *mk-edgetype* (get-new-edgeid, VERTICES (e_2))))

(* unmodified edges of some f *)

$\oplus$ (e $\in$ create-edges (f_i, f) $\wedge$

planar (f_i, e))))

(* assure the connectivity information between the edges. *)

$\wedge$ ($\forall$ i $\in$ 2 .. **len** e-col_r)

(**let** ver1 = VERTICES (e-col_r(i 1)),

ver2 = VERTICES (e-col_r(i)) **in**

hd ver2 = **hd tl** ver1) $\wedge$

hd e-col_r(1) = **hd tl** e-col_r (**len** e-col_r)

tel

(* assure that vertices of adjacent edges are non-collinear. *)

$\wedge$ ($\forall$ v,u,w $\in$ Point)

((v,u,w $\in$ **union** {**elems** VERTICES (e) | e $\in$ **elems** e-col_r} $\wedge$

(u $\neq$ v) $\wedge$ (w $\neq$ v)) $\Rightarrow$ (v $\neq$ convex-comb (u,w)))

(* assure that the elements of set-of-e-col_r are not connected. *)

$\wedge$ ($\forall$ e-col_{r1}, e-col_{r2} $\in$ set-of-e-col_r)

(e-col_{r1} $\neq$ e-col_{r2} $\Rightarrow$

($\forall$ e $\in$ e-col_{r1})

(($\forall$ e_1 $\in$ e-col_{r2})

(**elems** VERTICES (e) $\cap$ **elems** VERTICES (e_1) = {}))) $\wedge$

f_r = *mk-facetype* (get-new-faceid, NORMAL (f_i), set-of-e-col_r)

tel

(* 'unmodified-face' asserts that the face f_i is a copy of the face f_r, except for the face and edge identification fields. *)

Unmodified-face : Facetype $\times$ Facetype $\rightarrow$ Boolean

post-Unmodified-face (f_i, f_r, b) $\triangleq$

b' $\Leftrightarrow$ **let** set-of-e-col$_i$ = EDGES (f_i), set-of-e-col$_r$ = EDGES (f_r) **in**

($\forall$ e-col$_i$ $\in$ set-of-e-col$_i$)

(($\forall$ e_i $\in$ **elems** e-col$_i$)

(($\exists!$ e-col$_r$ $\in$ set-of-e-col$_r$)

(($\exists!$ e_r $\in$ **elems** e-col$_r$)

(VERTICES (e_i) = VERTICES (e_r)))))

tel

(* 'create-edges' function creates a new set of edges from the two faces passed as input. *)

Create-edges : Facetype $\times$ Facetype $\rightarrow$ Edgetype-set

post-Create-edges (f_1, f_2, e-col) $\triangleq$

($\forall$ e $\in$ e-col)

(($\exists!$ e-col$_1$ $\in$ EDGES (f_1) $\wedge$

($\exists!$ e-col$_2$, e-col$_3$ $\in$ EDGES (f_2))

(($\exists!$ e_1 $\in$ **elems** e-col$_1$ $\wedge$

$\exists!$ e_2 $\in$ **elems** e-col$_2$ $\wedge$

$\exists!$ e_3 $\in$ **elems** e-col$_3$) $\wedge$

($\sim$ unmodified-edge (e_1, e) $\wedge$ $\sim$ unmodified-edge (e_2, e) $\wedge$

$\sim$ unmodified-edge (e_3, e) $\wedge$

(e = part-of (e_1, e_2) $\oplus$

e = extension-of (e_1, e_2) $\oplus$ e = in-between (e_1, e_2) $\oplus$

e = (in-between (e_2, e_3) $\wedge$ $e_2 \neq e_3$)))))

(* 'unmodified-edge' asserts that **e** is a copy of e_1 except for the identification field. *)

Unmodified-edge : Edgetype $\times$ Edgetype $\rightarrow$ Boolean

post-Unmodified-edge (e_1, e, b) $\triangleq$

b' $\Leftrightarrow$ (**hd** VERTICES (e) = **hd** VERTICES (e_1) $\wedge$

hd tl VERTICES (e) = **hd tl** VERTICES (e_1))

$\vee$ (**hd** VERTICES (e) = **hd tl** VERTICES (e_1) $\wedge$
hd tl VERTICES (e) = **hd** VERTICES (e_1))

(* 'part-of' asserts that edge e is a portion of the edge e_1 by the intersection of e_2. *)

Part-of : Edgetype $\times$ Edgetype $\rightarrow$ Edgetype
post-Part-of (e_1, e_2, e) $\triangleq$
let v_1, v_2 = VERTICES (e) **in**
$v_1 \in$ **elems** VERTICES (e_1) $\wedge$
v_2 = convex-comb (**hd** VERTICES (e_2), **hd tl** VERTICES (e_2)) $\wedge$
($v_1 \neq v_2$) $\wedge$ e' = *mk-edgetype* (get-new-edgeid, v_1, v_2)
tel

(* 'extension-of' asserts that e is obtained as the extension of the edge e_1. This case occurs only when two edges each belonging to different faces are adjacent and their vertices are collinear. *)

Extension-of : Edgetype $\times$ Edgetype $\rightarrow$ Edgetype
pre-Extension-of (e_1, e_2) $\triangleq$
let v_1, v_2 = VERTICES (e_1),
v_3, v_4 = VERTICES (e_2) **in**
($v_2 = v_3$) $\wedge$ (collinear (v_1, v_2, v_4))
tel
post-Extension-of (e_1, e_2, e) $\triangleq$
let v_1, v_2 = VERTICES (e) **in**
v_1 = **hd** VERTICES (e_1) $\wedge$
v_2 = **hd tl** VERTICES (e_2) $\wedge$
$v_1 \neq v_2$ $\wedge$
e' = *mk-edgetype* (get-new-edgeid, v_1, v_2)
tel

(* 'in-between' asserts that the vertices of the edge e are obtained as the convex combinations of the vertices of the edges e_1 and e_2. Note that e_1 and e_2 must be distinct in this case.*)

In-between : Edgetype × Edgetype → Edgetype

pre-In-between (e_1, e_2) $\triangleq$

(* assure that e_1 and e_2 are not intersecting. *)

let v_1, v_2 = VERTICES (e_1),

v_3, v_4 = VERTICES (e_2) in

$(v_3 \neq \text{convex-comb}(v_1, v_2)) \wedge (v_4 \neq \text{convex-comb}(v_1, v_2))$

tel

post-In-between (e_1,e_2,e) $\triangleq$

let v_1, v_2 = VERTICES (e) in

v_1 = convex-comb (**hd** VERTICES (e_1), **hd tl** VERTICES (e_1)),

v_2 = convex-comb (**hd** VERTICES (e_2), **hd tl** VERTICES (e_2))$\wedge$

$v_1 \neq v_2 \wedge$

e'= *mk-edgetype* (get-new-edgeid, v_1, v_2)

tel

(* 'Collinear' asserts that three points passed as arguments are, in fact, collinear. *)

Collinear : Point × Point × Point → Boolean

post-Collinear (p, q, r) $\triangleq$

p = convex-comb (q, r) $\vee$ q = convex-comb (p, r) $\vee$ r = convex-comb (p, q)

(* 'convex-comb' asserts that point w divides the line segment defined by the points u and v, into two parts. *)

Convex-comb : Point × Point → Point

post-Corvex-comb (u, v, w) $\triangleq$ $(\exists\, \lambda, 0 \leq \lambda \leq 1)\ (w' = \lambda\, u + (1 - \lambda)\, v)$

(* The auxiliary functions 'get-new-edgeid' and 'get-new-faceid' return new identification numbers for newly created edges and faces respectively, every time when they are called. These functions can be specified as below *)

Get-new-edgeid : → Nat0

post-Get-new-edgeid (n) $\triangleq$ $(\forall\, x \in \text{Edgetype})\ (\text{EDGEID}(x) \neq n)$

Get-new-faceid : → Nat0

post-Get-new-faceid (n) $\triangleq$ $(\forall\, x \in \text{Facetype})\ (\text{FACEID}(x) \neq n)$

Bounded-plane : Facetype $\rightarrow$ Surface

Volu-space : Object $\rightarrow$ Surface

Opposite-direction : Direction $\times$ Direction $\rightarrow$ Boolean

Reverse : Direction $\rightarrow$ Direction

Planar : Facetype $\times$ Edgetype $\rightarrow$ Boolean

3.4 Type Invariants

One of our major goals is to prove the correctness of the formal specifications in the sense that only regular solids are produced as a result of BOOLEAN-SOLIDS. In this section, we state and prove the invariants for *Edgetype* and *Facetype*; these type invariants are used in the next section for proving the regularity of resulting objects.

In the following proofs, we use the style of Jones [Jon86], number the equations on the left and show the references on the right following three 'dots'.

Edgetype Invariant

Every edge of the newly created solid has exactly two distinct vertices. Formally stated,

inv-Edgetype (*mk-edgetype* (eid, ver)) $\triangleq$ (**len** ver = 2) $\wedge$ (ver(1) $\neq$ ver(2))

Proof :

In 'copy-edges',

e' = *mk-edgetype* (get-new-edgeid, VERTICES (e_1)) ...Copy-edges

$\Rightarrow$ VERTICES (e) = VERTICES (e_1)

(1.1) $\Rightarrow$ **len** VERTICES (e) = **len** VERTICES (e_1)

from $e_1 \in$ **elems** e-col$_1$ | e-col$_1 \in$ EDGES (f) ...Copy-edges

$\Rightarrow$ $e_1 \in$ **elems** e-col$_1$ | e-col$_1 \in$ EDGES (f_1) $\wedge$ $f_1 \in$ FACES (s_1) ...Validate

OR

$\Rightarrow$ $e_1 \in$ **elems** e-col$_2$ | e-col$_2 \in$ EDGES (f_2) $\wedge$ $f_2 \in$ FACES (s_2) ...Validate

infer e_1 is an edge of an input solid.

from (1.1) and **len** VERTICES (e_1) = 2 ...(H6)

infer **len** VERTICES (e) = 2

from (1.1) and VERTICES (e_1)(1) $\neq$ VERTICES (e_1)(2) ...(H6)

infer VERTICES(e)(1) $\neq$ VERTICES(e)(2)

In 'Intersect-face',

$e' = \textit{mk-edgetype}$ (get-new-edgeid, VERTICES (e_1)) ...Intersect-face

from $e_1 \in$ **elems** e-col$_i$ ...Intersect-face

$\Rightarrow$ $e_1 \in$ **elems** e-col$_i$ | e-col$_i$ $\in$ EDGES $(f_1) \wedge f_1 \in$ FACES (s_1)

OR

$\Rightarrow$ $e_1 \in$ **elems** e-col$_i$ | e-col$_i$ $\in$ EDGES $(f_2) \wedge f_2 \in$ FACES (s_2) ...Create-faces

infer e_1 is an edge of an input solid.

from len VERTICES $(e_1) \wedge$ VERTICES$(e_1)(1) \neq$ VERTICES$(e_1)(2)$ ...(H6)

infer len VERTICES(e) = 2 $\wedge$ VERTICES(e)(1) $\neq$ VERTICES(e)(2)

The proof is similar for

$e' = \textit{mk-edgetype}$ (get-new-edgeid, VERTICES(e_2)) ...Intersect-face

In 'Part-of', 'Extension-of' and 'In-between',

$e' = \textit{mk-edgetype}$ (get-new-edgeid, $<v_1, v_2>$)

(len VERTICES (e) = **len** $<v_1, v_2> = 2) \wedge (v_1 \neq v_2)$ ...Tuple

Facetype Invariant

Every newly created face f satisfies the following properties :

1. Every edge e in f has a unique EDGEID.
2. The sets of edges in f are disjoint.
3. Within each set of edges in f,
   - All edges must be *simply connected* as defined in H4.
   - Vertices of adjacent edges are non-collinear.
   - The normal computed from the vertices of every pair of adjacent edges has either the same direction or the opposite direction of the normal to the face.

Formally

inv-Facetype (*mk-facetype* (fid, direct, sel)) $\triangleq$

$$\sum_{el \in sel} \textbf{len } el = \textbf{card } \{\text{EDGEID}(e) \mid (e \in \textbf{elems } el) \wedge (el \in sel)\} \wedge$$

$(\forall$ el $\in$ sel)

$((\textbf{len}\ \text{el} \geq 3) \wedge$

$(\forall\ \text{i} \in \{2 \cdots \textbf{len}\ \text{el}\})$

$(\textbf{let}\ \text{ver}_1 = \text{VERTICES (el(i}-1)),$

$\text{ver}_2 = \text{VERTICES(el(i))}\ \textbf{in}$

$\textbf{hd tl}\ \text{ver}_1 = \textbf{hd}\ \text{ver}_2$ (* connected *)

$\wedge\ \sim \text{collinear} (\textbf{hd}\ \text{ver}_1, \textbf{hd}\ \text{ver}_2, \textbf{hd tl}\ \text{ver}_2)$

(* vertices of adjacent edges non-collinear *)

$\wedge\ \textbf{let}\ \text{dir} = \text{compute-normal} (\textbf{hd}\ \text{ver}_1, \textbf{hd}\ \text{ver}_2, \textbf{hd tl}\ \text{ver}_2)\ \textbf{in}$

$(\text{dir} = \text{direct}) \vee (\text{dir} = \text{reverse(direct)})$

tel

$\wedge\ \textbf{hd}\ \text{VERTICES(el(1))} = \textbf{hd tl}\ \text{VERTICES (el(}\textbf{len}\ \text{el)))}$

tel)

(* first and last edges are connected. *)

$\wedge\ (\forall\ \text{el}_1, \text{el}_2 \in \text{sel})$

$(\text{el}_1 \neq \text{el}_2 \Rightarrow$

$(\forall\ \text{e}_1 \in \textbf{elems}\ \text{el}_1, \text{e}_2 \in \textbf{elems}\ \text{el}_2)$

$(\textbf{elems}\ \text{VERTICES (e}_1) \cap \textbf{elems}\ \text{VERTICES(e}_2) = \{\}))$

We state and prove the following Lemmas which constitute the proofs for the Facetype invariant.

Lemma N1 : The number of distinct edges returned by Copy-edges (f) is equal to the number of distinct edges in EDGES (f) and it is also equal to the number of unique EDGEIDs returned by Copy-edges (f). Formally

$$\sum_{el \in \, copy-edges(f)} \textbf{len}\ \text{el} = \sum_{el_1 \, \in \, EDGES(f)} \textbf{len}\ \text{el}_1$$
$$= \textbf{card}\ \{\text{EDGEID (e)} \mid (\text{e} \in \textbf{elems}\ \text{el}) \wedge (\text{el} \in \text{copy-edges(f)})\}$$

Proof :

from *post*-Copy-edges

infer copy-edges(f) $\leftrightarrow$ EDGES (f) is bijective.

$\Rightarrow \textbf{card}\ \text{copy-edges(f)} = \textbf{card}\ \text{EDGES (f)}$

$$(1.1) \quad \Rightarrow \sum_{el \, \in \, copy-edges(f)} \textbf{len}\ \text{el} = \sum_{el_1 \, \in \, EDGES(f)} \textbf{len}\ \text{el}_1$$

from *post*-Copy-edges and *post*-Get-new-edgeid

infer $\textbf{card}\ \text{copy-edges(f)} = \textbf{card}\ \text{EDGES (f)} = 1$ ...Pre-Validate

(1.2) $\quad \wedge \quad \sum_{el \ in \ copy-edges(f)}$ **len** el = **card** {EDGEID(e) | (e ∈ **elems** el) ∧

(el ∈ copy-edges(f))}

from (1.1) and (1.2)

infer $\sum_{el \ in \ copy-edges(f)}$ **len** el = $\sum_{el_1 \in EDGES(f)}$ **len** el_1

= **card** {EDGEID(e) | (e ∈ **elems** el) ∧ (el ∈ copy-edges(f))}

Lemma N2 : For every face f in the newly created solid, the number of distinct edges is equal to the number of unique EDGEIDs of all edges in f. Formally,

$\sum_{el \in EDGES(f)}$ **len** el = **card** {EDGEID (e) | (e ∈ **elems** el) ∧ (el ∈ EDGES (f))}

Proof :

Case 1 : f is copied from an input solid.

from *post*-Validate

infer EDGES (f) = {copy-edges(f_1) | f_1 ∈ FACES(s_1)}

OR {copy-edges(f_2) | f_2 ∈ FACES(s_2)}

Now the proof follows from Lemma N1.

Case 2 : f is a newly created face.

from *post*-Intersect-face

(2.1) *infer* $\sum_{el \in EDGES(f)}$ **len** el = $\sum_{e-col_r \in set-of-e-col_r}$ **len** e-col_r

(∀ e-col_r ∈ set-of-e-col_r)

((∀ k ∈ {1 ·· **len** e-col_r} ∧ e = e-col_r(k))

(2.2) (EDGEID(e) = (EDGEID (e) | e ∈ A) ⊕

(EDGEID (e) | e ∈ B) ⊕ (EDGEID (e) | e ∈ C))

where

A = {e | e = *mk-edgetype* (get-new-edgeid, VERTICES (e_1)) ∧

(e_1 ∈ **elems** el_1) ∧ (el_1 ∈ EDGES (f_i))} ...Intersect-face

B = {e | e = *mk-edgetype* (get-new-edgeid, VERTICES (e_2)) ∧

(e_2 ∈ **elems** el_2) ∧ ($e2_1$ ∈ EDGES (f)) ∧ ...Intersect-face

(f ∈ FACES(s_1) ∨ f ∈ FACES (s_2))} ...Create-faces

C = {e | e ∈ Create-edges (f_i, f) ∧ ...Intersect-face

(f ∈ FACES (s_1) ∨ f ∈ FACES (s_2))} ...Create-faces

from post-Create-edges, post-Part-of, post-Extension-of and post-In-between

(2.3) *infer* $e \in$ Create-edges (f_1, f_2)

$\Rightarrow e =$ *mk-edgetype* (get-new-edgeid, $<v_1, v_2>$)

from (2.1), (2.2), (2.3) and *post*-Get-new-edgeid

infer $\sum_{el \in EDGES(f)}$ **len** el $= \sum_{e\text{-}col_r \in set\text{-}of\text{-}e\text{-}col_r}$ **len** e-col$_r$ $=$

card $\{EDGEID(e_1) \mid e_1 \in A\}$ + **card** $\{EDGEID(e_2) \mid e_2 \in B\}$ +

card $\{EDGEID(e_3) \mid e_3 \in C\}$

= **card** $\{EDGEID(e) \mid (e \in$ **elems** e-col$_r) \wedge$ (e-col$_r \in$ set-of-e-col$_r)\}$

= **card** $\{EDGEID\ (e) \mid (e \in el) \wedge (el \in EDGES\ (f))\}$

<u>Lemma N3</u> : The normal computed from the vertices of every pair of adjacent edges of a face f belonging to the newly created solid, must be in the same direction or in the opposite direction to the normal of the face f. Formally it could be stated as

($\forall$ f $\in$ FACES(s) $\wedge$ $\forall$ e-col $\in$ EDGES (f))

(($\forall$ e_1, e_2 $\in$ **elems** e-col)

(($\exists!$ v $\in$ Point)

((v $\in$ **elems** VERTICES(e_1) $\cap$ **elems** VERTICES(e_2))

$\Rightarrow$ **let** d compute-normal (**hd** VERTICES(e_1), **hd** VERTICES(e_2),

hd tl VERTICES(e_2)) **in**

(d = NORMAL (f)) $\vee$ (d = reverse (NORMAL (f))))))

tel

(* The auxiliary function 'compute-normal' returns the direction of a normal to three non-collinear points passed as parameters. *)

<u>Proof :</u>

<u>Case 1</u> : f is copied from a face f_x of an input solid.

from *post*-Validate and *post*-Copy-edges

(3.1) *infer* EDGES(f) $\leftrightarrow$ EDGES (f_x) is bijective.

(3.2) $\Rightarrow$ **card** EDGES(f) = **card** EDGES(f_x) = 1 ...(H1)

from (3.2)

infer ($\forall$ e_1, e_2 $\in$ **elems** el | el $\in$ EDGES(f_x))

(($\exists!$ v $\in$ Point)

((v $\in$ **elems** VERTICES(e_1) $\cap$ **elems** VERTICES(e_2))

$\Rightarrow$ compute-normal (**hd** VERTICES(e_1), **hd** VERTICES(e_2), hd tl VERTICES(e_2))

= NORMAL (f_x) ...(H7, H3)

= NORMAL (f)) ...Validate

Case 2 : f is a newly created face.

from *post*-Intersect-face

(3.3) *infer* NORMAL(f) = NORMAL(f_i)

= (NORMAL (f_1) | $f_1 \in$ FACES(s_1))

OR (NORMAL (f_2) | $f_2 \in$ FACES(s_2)) ... Create-faces

($\forall$ e_1, e_2 $\in$ Edgetype)

(($e_1 \in$ **elems** el_1 | $el_1 \in$ EDGES(f)) $\wedge$

($e_2 \in$ **elems** el_2 | $el_2 \in$ EDGES(f)) $\wedge$

($\exists!$ v $\in$ Point)

((v $\in$ **elems** VERTICES(e_1) $\cap$ **elems** VERTICES(e_2)) $\Rightarrow$ $el_1 = el_2$

$\wedge$ $\sim$ collinear (**hd** VERTICES(e_1), **hd** VERTICES(e_2),

hd tl VERTICES(e_2)) ... Intersect-face

$\wedge$ planar(f_i, e_1) $\wedge$ planar (f_i, e_2) ... Intersect-face

$\wedge$ planar(f, e_1) $\wedge$ planar (f, e_2) ... (3.3)

$\Rightarrow$ compute-normal (**hd** VERTICES(e_1), hd VERTICES(e_2),

hd tl VERTICES(e_2))

= NORMAL (f_i) ...(H7, H3)

= NORMAL (f)) ... (3.3)

Lemma N4 : For each face of a newly created solid, there are at least *three* edges in each list of edge-list-set. Formally,

($\forall$ f $\in$ FACES (s) $\wedge$ $\forall$ el $\in$ EDGES (f)) (**len** el $\geq$ 3)

Proof :

Case 1 : f is a copied face.

EDGES(f) = (copy-edges(f_1) | $f_1 \in$ FACES(s_1)) **OR**

(copy-edges(f_2) | $f_2 \in$ FACES(s_2)) ...Validate

$\Rightarrow$ **card** EDGES(f) = **card** EDGES(f_1) **OR card** EDGES(f_2) ...Copy-edges

= 1 ...(H1)

$\Rightarrow$ (**len** el | el $\in$ EDGES(f) = (**len** el_1 | $el_1 \in$ EDGES(f_1)) **OR**

$(\textbf{len}\ el_2 \mid el_2 \in EDGES(f_2))$

≥ 3 ...(H2)

Case 2 : f is a newly created face.

This part of the proof is given by contradiction.

Let el $\in$ EDGES (f).

Case 2.1 : **len** el = 1

hd VERTICES (el(1)) $\neq$ **hd tl** VERTICES (el (**len** el))

$\Rightarrow$ violates the connectivity property $\Rightarrow$ contradiction.

Case 2.2 : **len** el = 2

Case 2.2.1 : VERTICES (el(1)) = **rev** VERTICES (el(2))

$\Rightarrow$ the same edge is viewed in opposite directions with respect to NORMAL (f).

$\Rightarrow$ violates the property as stated in the definition of *normal.*

$\Rightarrow$ contradiction.

Case 2.2.2 : el(1) and el(2) are adjacent and are distinct.

$\Rightarrow$ **hd tl** VERTICES (el(1)) = **hd** VERTICES (el(2))

$\wedge$ **hd tl** VERTICES (el(**len** el)) $\neq$ **hd** VERTICES (el(1))

$\Rightarrow$ violates connectivity $\Rightarrow$ contradiction.

3.5 Behavior

In this section, two important properties of the specifications are stated and proved. We show that the specifications remain valid when one of the input solids is empty; that is, the validity of the specifications for the boundary cases is established. Secondly, we show that only regular solids can result from the specifications when the input solids are regular. Once again, we follow Jones' rigorous approach [Jon86] for the proofs.

Lemma R1 : The specification

BOOLEAN-SOLIDS (s_1,s_2 : Object; OPR : Operation) s_3 : Object-set

is correct when either s_1 or s_2 is null.

Proof :

The proof for UNION operation is shown below; the proofs for DIFFERENCE and

INTERSECTION are similar.

Case 1 : Let s_1 be empty.

In function 'Validate',

f-col$_1$ = {} since s_1 is empty.

Hence $(\exists!\ f_1 \in$ f-col$_1) (\cdots)$ is false.

We will prove that $f_3 \in$ create-faces (s_1,s_2,opr) is also false and therefore, only the second clause of UNION will hold good; it is to be noted that this clause does not concern with f-col$_1$.

In function 'create-faces',

f-col$_1$ = {} since s_1 is empty.

Hence $(\exists!\ f_1 \in$ f-col$_1) (\cdots)$ is false.

In the second clause, there are two predicates connected by $\wedge$. The second predicate

bounded-plane(f_3) $\not\subseteq$ volu-space(s_1)

is vacuously true since s_1 is empty. Hence in order to show that the result of create-faces is false, it is to be shown that the first predicate

f_3 = Intersect-face (f_2, f-col$_1$)

is false.

In function 'Intersect-face',

f-col = {}

Therefore, $(\exists!\ f \in$ f-col$) (\cdots)$ is false.

This indicates that all edges in EDGES (f_r) should have been obtained from EDGES (f_i) as stated in Intersect-face; i.e.,

($\forall$ e-col$_r$ $\in$ EDGES (f_r))
 (($\forall$ k $\in$ {1 $\cdot\cdot$ len e-col$_r$})
  (let e = e-col$_r$(k) in
   ($\exists!$ e-col$_i$ $\in$ EDGES (f_i))
    (($\exists!$ e_1 $\in$ **elems** e-col$_i$)
     (e' = *mk-edgetype* (get-new-edgeid, VERTICES (e_1))))))
      (* direct or unmodified edges *)

This shows that f_r is exactly similar to f_i, which is f_2 (in 'create-faces') belonging to s_2. However, the predicate

$\sim$ unmodified (f_i, f_r)

shows that f_r cannot be a copy of f_i. Due to this contradiction, the result returned by 'Intersect-face' as well as the result returned by 'create-faces' are FALSE. It is thus proved that the resulting solid s_3 is nothing but the solid s_2 (from 'BOOLEAN-SOLIDS').

Case 2 : Let s_2 be empty.

The same arguments as in Case 1 apply here with the roles of s_1 and s_2 interchanged.

Lemma R2 : The newly created solid s does not contain any dangling edge.

Proof : From the post-conditions of 'Validate', it can be observed that only one of the following is true for every face of solid s.

(i) copied from a face f_1 of solid s_1

(ii) copied from a face f_2 of solid s_2

(iii) newly created.

When Case (i) or (ii) applies, the result follows directly from H8. So, it is sufficient to prove for Case (iii).

From the post-conditions of 'create-faces', notice that f is obtained due to the modification of either a face f_1 of s_1 or a face f_2 of s_2. The modification is presented in 'Intersect-face'. From the post-conditions of 'Intersect-face', it can be observed that the modified face consists of a set of list of edges set-of-e-col$_r$. For each list e-col$_r$ in this set, the connectivity between the vertices of all edges is assured in the post-conditions. Hence e-col$_r$ does not contain any dangling edge; this implies that f does not contain any dangling edge.

Lemma R3 : The newly created solid s does not contain any dangling face.

Proof :

As stated earlier, a dangling face is one not contributing to the interior of the solid. It is therefore to be proved that every face f of the newly created solid s contributes to the interior of s.

The post-conditions of 'Validate' assert that every face f of the newly created solid s is created in exactly one of the following ways :

a) it is copied from a face f_1 of solid s_1.

b) it is copied from a face f_2 of solid s_2.

c) it is a modified face f_1 of solid s_1 (as given by the post-conditions of 'create-faces').

d) it is a modified face f_2 of solid s_2 (post-conditions of 'create-faces').

e) it is a modified face f_1 of solid s_1 **and** a modified face f_2 of solid s_2 (**and** here indicates that it could be obtained in either way); this is possible only when the operation is 'INTERSECT'.

From the post-conditions of 'Validate' and 'create-faces', observe the two facts : (i) the normal of a copied (or modified) face is retained; (ii) the volumetric space and hence the interior of the resulting solid s is obtained from those of its constituents. The situations corresponding to the three Boolean operations are as follows :

1. UNION : From post-Validate, it is clear that the volumetric space of S includes both the volumetric spaces of s_1 and s_2. This implies that the normals of faces of s must point to the interior of either s_1 or s_2. Faces of s which are copied faces of s_1 contribute to the interior of s_1 (post-conditions of 'Validate'). The same is true for copied faces of s_2. Faces of s which are modified faces of s_1 also contribute to the interior of s_1 (post-conditions of 'create-faces') since their respective normals remain unchanged. The same remark is true for modified faces of s_2 as well. Hence every face f in s will contribute to the interior of s.

2. DIFFERENCE : From post-Validate, notice that the interior of s should be part or whole of s_1. When s is the whole of s_1, nothing is to be proved. When S is a part of s_1, there is a portion S' of s_1 such that

   volu-space(s_1) = volu-space (S) $\cup$ volu-space (S') and

   volu-space(S') $\subseteq$ volu-space (s_2)

   Every normal to a face of s obtained from the faces of s_1 (copied or modified) points to the interior of s_1 (post-Validate, post-create-faces). Normals to faces of s which are obtained from faces of s_2 (copied or modified) have their directions reversed (post-validate, post-create-faces); this implies that they point to the interior of s and not to the interior of S'. Hence every normal to a face of s points to the interior of s.

3. INTERSECT : The interior of s should be common to the interior of s_1 as well as s_2. If a face of s is copied from a face of s_1, then this face also lies in the interior of s_2 (post-Validate). This implies that the normal to this copied face points to the interior of s_1 as well as to the interior of s_2; i.e., it points to the interior of s. Similar arguments apply to a face of S copied from a face of s_2 (post-Validate). Faces of s which are modified faces of s_1 or s_2 have to satisfy the same constraints as their copied counterparts (post-conditions for INTERSECT in 'create-faces'). But here arises the situation when a face f in the result can be obtained as a modification of f_1 belonging to s_1 and also as a modification of f_2 belonging to s_2. If the normals of f_1 and f_2 are in opposite directions, the face f is not created due to the post-condition for INTERSECT in 'create-faces'. However, if the normals are in the same direction, the interior of s_1 and s_2 when viewed from f_1 and f_2 contribute to the interior of s and hence the normal to f is also assigned the same direction. Hence every face f in s contributes to the interior of s.

This completes the proof that no dangling face is created as a result of any of the operations - UNION, DIFFERENCE and INTERSECTION.

We remark that the following type invariants are associated with type *Object* and every solid s produced by BOOLEAN-SOLIDS satisfies these invariants.

1. Every newly created solid s has at least *four* faces.

2. FACEID of every face f of s is unique.

3. Every edge e in s is shared by exactly two faces f_1 and f_2 of s.

4. Adjacent faces f_1 and f_2 in s are not coplanar.

Formally these invariants can be expressed as inv-Object (*mk-object* (type, fset)) $\triangleq$

$$
\begin{array}{l}
\quad \textbf{card}\ \text{fset} \geq 4 \wedge \\
\quad \textbf{card}\ \text{fset} = \textbf{card}\ \{\text{FACEID (f)} \mid \text{f} \in \text{fset}\} \wedge \\
\quad (\forall\ \text{f}_1 \in \text{fset}) \\
\qquad ((\forall\ \text{e}_1 \in \textbf{elems}\ \text{el}_1 \mid \text{el}_1 \in \text{EDGES } (\text{f}_1)) \\
\qquad\quad ((\exists!\ \text{f}_2 \in \text{fset}) \\
\qquad\qquad ((\exists\ \text{e}_2 \in \textbf{elems}\ \text{el}_2 \mid \text{el}_2 \in \text{EDGES } (\text{f}_2)) \\
\qquad\qquad\quad (\text{VERTICES } (\text{e}_1) = \textbf{rev}\ (\text{VERTICES } (\text{e}_2)))))) \wedge \\
\quad (\forall\ \text{f}_1, \text{f}_2 \in \text{fset}) \\
\qquad ((\forall\ \text{e}_1, \text{e}_2 \in \text{Edgetype}) \\
\qquad\quad ((\text{e}_1 \in \textbf{elems}\ \text{el}_1 \mid \text{el}_1 \in \text{EDGES } (\text{f}_1)) \wedge \\
\qquad\quad\ (\text{e}_2 \in \textbf{elems}\ \text{el}_2 \mid \text{el}_2 \in \text{EDGES } (\text{f}_2)) \wedge \\
\qquad\quad\ (\text{VERTICES } (\text{e}_1) = \textbf{rev}\ (\text{VERTICES } (\text{e}_2)) \\
\qquad\quad \Rightarrow \quad \text{NORMAL } (\text{f}_1) \neq \text{NORMAL } (\text{f}_2) \\
\qquad\quad \wedge \quad \text{NORMAL } (\text{f}_1) \neq \text{reverse (NORMAL } (\text{f}_2))))
\end{array}
$$

The following lemmas constitute the proofs for the first three type invariants; the last invariant is asserted in the post-condition of the function 'Validate'.

Lemma R4 : Every edge of the newly created solid s is shared by *exactly* two faces of s.

Proof : We prove this by contradiction. Let e be an edge belonging to a face f of S, which is not shared by any other face of s. Obviously f is a dangling face. However by Lemma R3, s does not contain any dangling face. This is a contradiction and hence every edge of S is shared by at least two faces of s.

Next, we show that every edge e of s is shared by *exactly* two faces of s. Once again it is proved by contradiction. Assume that there is an edge e of s shared by three faces f_1, f_2 and f_3 of s. There are two cases to be considered here :

Case 1 : Two of the faces, say f_1 and f_2, contribute to the interior of s and the third, f_3, lies outside the interior. Face f_3, in this case, will not contribute to the interior of s. But by Lemma R2, every face of s should contribute to the interior of solid S and hence f_3 will not be a face of s. Therefore edge e is shared by exactly two faces f_1 and f_2.

Case 2 : Two of the faces, say f_1 and f_2, contribute to the interior of s and the third, f_3, lies in the interior of s. By the definition of normal, there is only one side of face f_3 that can contribute to the interior of s. Since f_3 lies in the interior of S, a normal cannot be defined for f. Hence f_3 cannot be a face of s and therefore edge e is shared by exactly two faces f_1 and f_2. This completes the proof.

Lemma R5 : Adjacent faces of the newly created solid can share exactly one edge.

Proof :

We prove this Lemma by contradiction. Let f_1 and f_2 be two adjacent faces of s. Assume that e_1 and e_2 are two edges of f_1, shared by f_2. Then by Hypothesis H2, both e_1 and e_2 lie in the plane of f_1 and f_2. This implies that the faces f_1 and f_2 are coplanar. But, as stated in the post-conditions of 'Validate', adjacent faces of s are not coplanar. This is a contradiction and therefore, f_1 and f_2 can share at most one edge.

Lemma R6 : Every newly created solid s has at least four faces.

Proof :

By Lemma N4, each set of edges in a face f of s has at least three edges. Assume that each face has one edge-list with only three edges in each list. Similar arguments apply when a face has more than one edge-list, having more than three edges in each list.

Case 1 : There are only three faces in s.

Let f_1, f_2 and f_3 be the faces of s.

Let e-col$_1$ = $<e_{11}, e_{12}, e_{13}>$ and e-col$_1$ $\in$ EDGES (f_1) and
e-col$_2$ = $<e_{21}, e_{22}, e_{23}>$ and e-col$_2$ $\in$ EDGES (f_2) and
e-col$_3$ = $<e_{31}, e_{32}, e_{33}>$ and e-col$_3$ $\in$ EDGES (f_3).

By Lemma R4, every edge in s is shared by exactly two faces of s.

Let e_{11} be shared by the faces f_1 and f_2 and e_{12} be shared by f_1 and f_3.

Case 1.1 : Edge e_{13} is not shared by any other face.

This implies that face f_1 is a dangling face. But by Lemma R3, s does not contain any dangling face. Hence there is a contradiction in the initial assumption that e_{13} is not shared by any other face.

Case 1.2 : Edge e_{13} is shared by f_1 and f_2.

In this case, f_1 and f_2 share the two edges e_{11} and e_{13}. By Lemma R5, this leads to a contradiction.

Case 1.3 : Edge e_{13} is shared by f_1 and f_3.

Same arguments as in Case 1.2. Hence s cannot have only three faces.

Case 2 : s has two faces.

Let the faces be f_1 and f_2. By Lemma N3, each face should have at least three edges.

Let e-col$_1$ = $<e_{11}, e_{12}, e_{13}>$ and e-col$_1$ $\in$ EDGES (f_1) and

e-col$_2$ = $<e_{21}, e_{22}, e_{23}>$ and e-col$_2$ $\in$ EDGES (f_2).

By Lemma R5, only one of the edges of f_1 can be shared by f_2. This implies that the other two edges of f_1 are not shared and hence f_1 is a dangling face. This is a contradiction to Lemma R3. Hence s cannot have only two faces.

Case 3 : s has only one face f.

This implies that every edge of f is not shared by any other face and hence f is a dangling face. This is again a contradiction to Lemma R3.

Hence, s should have four or more faces.

Lemma R7 : For every newly created solid s,

$$\textbf{card}\ \text{FACES}\ (s) = \textbf{card}\ \text{FACEID}\ (f) \mid f \in s$$

Proof :

In 'Validate',

(7.1) **card** FACES (s) = **card** f-col_3

($\forall$ $f_3 \in$ f-col_3)

(7.2) (f_3 = *mk-facetype* (get-new-faceid, NORMAL (f_1), copy-edges (f_1))

$\wedge$ $f_1 \in$ FACES (s_1)

(7.3) **OR** (f_3 = *mk-facetype* (get-new-faceid, NORMAL (f_2), copy-edges (f_2))

$\wedge$ $f_2 \in$ FACES (s_2)

(7.4) **OR** (f_3 = *mk-facetype* (get-new-faceid, reverse (NORMAL (f_2)), copy-edges (f_2))

$\wedge$ $f_2 \in$ FACES (s_2)

from (7.1), (7.2), (7.3), (7.4) and post-get-new-faceid

infer **card** FACES (s) = **card** f-col_3 = **card** FACEID (f) | $f \in s$

The formal specification presented in this chapter constitutes the basis for verification of an offline programming environment for robotics and CAD applications.

Chapter 4

Specifications for Robot Kinematics

An *intelligent robot* is a physical machine endowed with computational mechanisms to plan, choose and execute actions and reason about the consequences of such choices. Intelligent robots are autonomous and they are required in environments where human interaction is hazardous or impossible. The computational mechanisms that make a robot intelligent are mainly software packages consisting of a variety of complex programs whose inputs and outputs are not just mathematical entities but represent physical objects. Ensuring correctness of the software used for intelligent robots is mandatory because online error recovery is almost impossible. Consequently, there is a need for a formal offline framework to specify the structure and properties of robots and their application domains so that (1) a static analysis can be conducted to reason about the behavior of the robotic system to be built and (2) the specifications can be transformed into robots implementing the specified tasks. In this chapter we describe formal specification supporting a rigorous analysis and a correct synthesis of robotic agents.

Online verification techniques are not generally preferred for robotic applications since they are expensive. Moreover, error recovery is sometimes impossible. Hence offline verification is an important development tool in robotic applications. Several offline techniques have been devised in recent years; among them, *simulation* and *prototyping* are the most notable ones. However, these techniques suffer from one basic disadvantage in that they do not provide reasoning capabilities to study the behavior of robots and are not sufficiently general to be used for all robotic applications. In addition, prototyping and simulation systems require additional resources apart from the development of the actual product and hence are very expensive.

4.1 Characteristics of a Robotic Agent

A robot abstracted away from the physical characteristics and particular physical environments of an actual robot is called an *agent*, a mathematical object endowed with operations whose manifestations in the real-world will drive a robot into its actions. Devoid of irrelevant details, an agent represents, in general, a class of real-life robots and the behavior of an agent permeates through this class of real-life robots. In this thesis, the term 'robot' is used to mean its 'agent'.

Our goal has been to formalize the structure of agents and the architecture of systems encompassing intelligent robots. Given the vastness and diversity of ideas that one has to gather and bring to bear on a problem of this magnitude and complexity for specification purposes, we have been very selective in the initial choice of subdomains. Brady [Bra89] has recently remarked that the automation of industrial processes such as mechanical parts assembly using robots is an important open problem. The most fundamental aspect that supports many robotic applications is "robot kinematics" which is to be well understood in order to study the behavior of a robotic agent. Robots are generally built using rigid solids and their kinematic operations are extended versions of the primitive operations on rigid solids such as *translation* and *rotation*. In subsequent sections, we specify rigid solids and their primitive operations and use them for specifying the structure of robotic agents and their forward and inverse kinematic operations. The formal specifications provided in this chapter can be used for offline verification of robotic applications.

4.2 Rigid Solids and Primitive Operations

A robot consists of several links joined together to form a chain, each link being a rigid solid. Robot kinematics is the study of the positional information and the associated transformations of these links in 3-dimensional space. Hence it is appropriate to study the behavior of rigid solids before specifying an actual robot. In this section, we provide the definition of a rigid solid without regard to physical properties such as *mass, density* and *type of material*, and give specifications for two primitive operations on rigid solids, namely *translation* and *rotation*. Theorems are stated and their proofs are derived from the specifications given in this section; we indicate, for important steps in a proof, the specification for which it is a consequence. These theorems will be useful in proving other theorems on robot kinematics.

4.2.1 Specification for a Rigid Solid

A rigid solid in space is defined by its shape and its position and orientation with respect to a global coordinate frame. Hence the type definition of a rigid solid can be stated in VDM as

Structure	::	POSI-ORIE : Transformation
		SHAPE : Primitive | Composite
Solid	=	Structure

One of the important properties used in subsequent specifications is the concept of equality of two vectors belonging to two different coordinate systems. Below, we give the specification for equality of vectors and state a theorem to assert the commutative and transitive properties of the function 'vector-equal'.

<u>Equality of Vectors</u> : Two vectors u and v belonging to the coordinate frames T_u and T_v respectively are said to be equal if

- norm (i.e., length) of u is equal to norm of v.
- angles subtended by u with the X, Y and Z axes of T_u are equal to the respective angles subtended by v with the X, Y and Z axes of T_v.

Vector-equal : Vec-Frame-Pair × Vec-Frame-Pair → Boolean

post-Vector-equal (A,B,b) $\triangleq$

b' = **let** u = VECTOR (A), v = VECTOR (B),
T_u = FRAME (A), T_v = FRAME (B) **in**
(norm (u) = norm (v)) $\wedge$ (angle-X (u,T_u) = angle-X (v,T_v)) $\wedge$
(angle-Y (u,T_u) = angle-Y (v,T_v)) $\wedge$
(angle-Z (u,T_u) = angle-Z (v,T_v))

tel

Theorem 1 *vector-equal (u,v) $\Rightarrow$ vector-equal (v,u);*
vector-equal (u,v) $\wedge$ vector-equal (v,w) $\Rightarrow$ vector-equal (u,w)

Proof : Follows immediately from the symmetric and transitive properties of equality ('=') for natural numbers.

We consider three primitive rigid solids in our specifications – Cube, Cone and Cylinder. A composite rigid solid is built from primitives or already defined composite objects by successive application of regularized boolean operations. Formally,

Primitive = Cube | Cylinder | Cone | $\cdots$

Composite :: OPERATION : Operation-types
LEFT : Structure
RIGHT : Structure

Operation-types = $\{\cup^*, \cap^*, -^*\}$

Specifications for primitive objects can be found in [PAB90].

Informally, a rigid solid can be defined as follows :

For every point p, distinct from the origin O of the local coordinate frame of the solid, the vector $\overrightarrow{Op}$ is an invariant; i.e., every new placement of the solid S, effected by a transformation T, produces a unique image of p, called p', such that vector-equal ($\overrightarrow{O'p'}$, $\overrightarrow{Op}$) is true where O' is the image of O under this transformation T.

Formally,

Rigid : Solid $\rightarrow$ Boolean

post-Rigid (S,b) $\triangleq$

b' = **let** O = position (POSI-ORIE (S)) **in**
(∀ p ∈ Point)
((p ≠ O) ∧ (on (p, S)) ⇒
(∀ T ∈ Transformation)
((∃! p',O' ∈ Point)
((p' = transform-point (T,p)) ∧

$(O' = \text{transform-point}\ (T,O)) \wedge$

(let u_T_u = const-vec-frame (point-vector (O',p'),

POSI-ORIE (S))),

v_T_v = const-vec-frame (point-vector (O,p),

POSI-ORIE (S'))) **in**

vector-equal (u_T_u,v_T_v)

tel)

)

)

)

tel

The totality of all points p' define S', which we call, the image of S under the transformation T.

Theorem 2 *The image S' of any rigid solid S is unique under a given transformation T.*

<u>Proof</u>

The proof follows from the specifications 'Rigid' and 'Vector-equal'.

Next let us consider the problem of determining whether or not two given solids S_1 and S_2 are images of each other. The following function determines the transform, if one exists, for two given solids S_1 and S_2.

Image : Solid $\times$ Solid $\rightarrow$ Transformation

pre-Image (S_1, S_2) $\triangleq$ rigid (S_1) $\wedge$ rigid (S_2)

post-Image (S_1, S_2, T) $\triangleq$

let O_1 = position (POSI-ORIE (S_1)), O_2 = position (POSI-ORIE (S_2)),

T_1 = POSI-ORIE (S_1), T_2 = POSI-ORIE (S_2) **in**

(O_2 = transform-point (T, O_1)) $\wedge$

($\forall$ p $\in$ Point)

(on (p, S_1) $\Rightarrow$

(on (transform-point (T, p), S_2)) $\wedge$

(let u_T_u = const-vec-frame (point-vector (O_1, p), T_1)

v_T_v = const-vec-frame (point-vector (O_2,
transform-point (T,p)), T_2) **in**
vector-equal (u_T_u, v_T_v)
tel)
)
tel

4.2.2 Specifications for Primitive Operations on Rigid Solids

In this section, we define two primitive operations with respect to rigid solids, namely *translation* and *rotation* and later apply them for robot kinematics.

Translation

Translation of an object is that transformation which defines the positional change in the object without change in its orientation. In space, an object can be translated parallel to a fixed plane or parallel to a fixed axis. In robotics, translations parallel to the axes of the joints in a manipulator arm are of interest. So we consider translations of a rigid body parallel to a fixed line, called the axis of translation. Informally,
The line joining the origin O of the local coordinate frame of S and the origin O' of the local coordinate frame of the translated solid S' is parallel to the axis of translation A. For every other point p on S, line Op is parallel to the line O'p' where p' is the image of p on S'.
Formally,
Translation : Solid × Axis–Rep → Solid

pre-Translation (S, A) $\triangleq$ rigid (S)
post-Translation (S, A, S') $\triangleq$
(rigid (S')) $\wedge$
(**let** O = position (POSI-ORIE (S)),
O' = position (POSI-ORIE (S')) **in**
(parallel (const-line (O, O'), A)) $\wedge$
(**let** T = image (S,S') **in**
(T $\neq$ NIL) $\wedge$

($\forall$ p $\in$ Point)

((p $\neq$ O) $\wedge$ (on (p, S)) $\Rightarrow$

parallel (const-line (O, p), const-line (O', transform-point (Γ,p)))

)

tel)

tel)

<u>Translation through a given distance</u>

The above specification for translation is more abstract in the sense that it captures the behavior of all instances of the translated solid. However, in practice, translation is specified for a particular distance. Below an enriched specification for translation is given which defines translation through a particular distance.

Translate-dist : Solid $\times$ Axis–Rep $\times$ Dist–Rep $\longrightarrow$ Solid

pre-Translate-dist (S, A, d) $\triangleq$ (rigid (S)) $\wedge$ (d $\neq$ 0)

post-Translate-dist (S, A, d, S') $\triangleq$

(S' = translation (S, A)) $\wedge$

(**let** O = position (POSI-ORIE (S)),

O' = position (POSI-ORIE (S')) **in**

distance (O, O') = d

tel)

Theorem 3 *Every point on a rigid solid S is moved through the same distance by* translation.

<u>Proof</u> :

The image S' of S under translation is rigid. $\cdots$ (post-Translation)

Let O and O' be the origins of the local coordinate frames of S and S' respectively.

Let p be a point on S and p' be its image on S'.

Since S and S' are rigid solids, from the specification of 'rigid' and 'vector-equal',

norm $(\overrightarrow{O'p'})$ = norm $(\overrightarrow{Op})$. $\cdots$ (1)

From post-Translation,

line Op is parallel to line O'p' and

line OO' is parallel to the axis of translation. $\cdots$ (2)

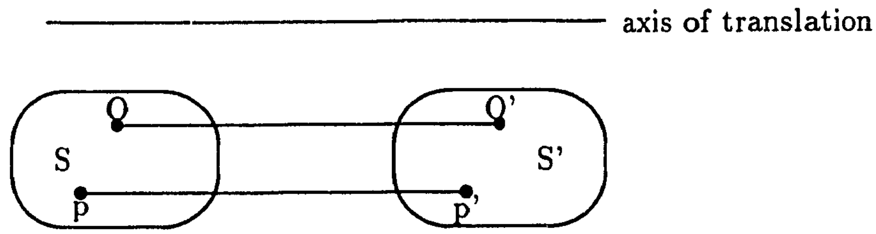


Figure 4.1: Translation parallel to an axis.

Consequently, line pp' is parallel to the axis of translation. $\cdots$ (3)

From (2) and (3),

distance (pp') = distance (OO').

i.e., O and p get translated through the same distance.

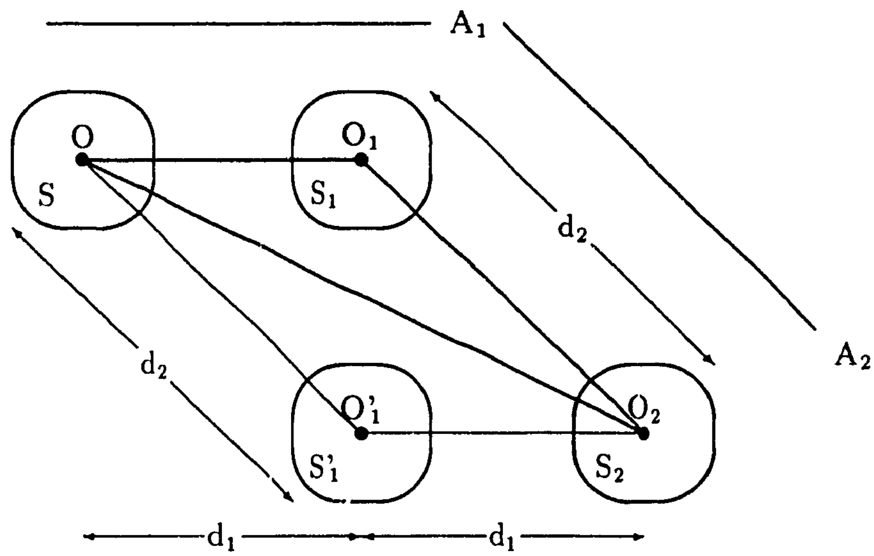


Figure 4.2: Additivity of Translation.

Theorem 4 *('Translate-dist' is additive.) Translation of a solid S by a vector* $\overrightarrow{d_1}$ *followed by another translation by a vector* $\overrightarrow{d_2}$ *is equivalent to translation by vector* $\overrightarrow{d} = \overrightarrow{d_1} + \overrightarrow{d_2}$.

Proof :

(Refer to Figure 4.2.)

Let S_1 be the image of a solid S translated along the axis A_1 through a distance d_1 and S_2 be the image of S_1, translated along the axis A_2 through a distance d_2.
Let O, O_1 and O_2 be the origins of S, S_1 and S_2 respectively.
By the specification of 'translate-dist',

norm $(\overrightarrow{OO_1}) = d_1$ and direction $(\overrightarrow{OO_1})$ is parallel to A_1.

Similarly, norm $(\overrightarrow{O_1O_2}) = d_2$ and direction of $(\overrightarrow{O_1O_2})$ is parallel to A_2.

By property of vector addition,

$\overrightarrow{OO_1} + \overrightarrow{O_1O_2} = \overrightarrow{OO_2}$.

i.e., $\overrightarrow{d_1} + \overrightarrow{d_2} = \overrightarrow{d}$.

Hence

S_2 can be obtained by a single translation using translate-dist (S, A, d).

Thus translate-dist (translate-dist (S, A_1, d_1), A_2, d_2) $\equiv$ translate-dist (S, A, d)
Hence the function 'translate-dist' is additive.

<u>Corollary</u> : ('Translate-dist' is commutative.)
translate-dist (translate-dist (S, A_1, d_1), A_2, d_2) $\equiv$
translate-dist (translate-dist (S, A_2, d_2), A_1, d_1).

<u>Proof</u> :

From Theorem 4,

translate-dist (translate-dist (S, A_1, d_1), A_2, d_2) $\equiv$ translate-dist (S, A, d).

By property of vector addition (See Figure 4.2),

translate-dist (translate-dist (S, A_2, d_2), A_1, d_1) $\equiv$ translate-dist (S, A, d).

Hence function 'translate-dist' is commutative.
Since the function 'transform-point' is bijective and hence has an inverse, the function 'translate-dist' is also bijective.

Theorem 5 *(There exists an inverse for 'Translate-dist' function.)If S_1 = translate-dist (S, A, d) then S = translate-dist (S_1, A, -d) where -d denotes the distance d in the direction opposite to axis A.*

Rotation

Rotation changes the orientation of an object continuously in such a way that every point on the object describes a circular path. Informally, *Any rotation of a rigid solid S with respect to an axis A takes every point p on S to its unique image p' on S' such that p and p' lie on the same circle having Q, the foot of the normal from p on A, as centre and Qp as radius.*

Rotation : Solid × Axis–Rep → Solid

pre-Rotation (S, A) $\triangleq$ rigid (S)

post-Rotation (S, A, S') $\triangleq$

(rigid (S') ∧
(**let** T = image (S, S') **in**
(T ≠ NIL) ∧
(∀ p ∈ Point)
(on (p,S) ⇒
(p' = transform-point (T, p)) ∧
(**let** circ = circle-pt-axis (p,A) **in**
lie-on-circle (p, circ) ∧ lie-on-circle (p', circ)
tel)
)
tel)

Circle-pt-axis : Point × Axis–Rep → Circle

post-Circle-pt-axis (p,A,Circ) $\triangleq$

let c = CENTRE (Circ), r = RADIUS (Circ) **in**
c = intersect (normal (p,A), A) ∧ r = distance (p,c)
tel

Rotation for a definite angle

As in the case of translation, we now specify rotation for a particular angle.

Rotate-angle : Solid × Axis–Rep × Angle–Rep → Solid

pre-Rotate-angle (S, A, θ) $\triangleq$ (rigid (S)) ∧ ($\theta \neq 0$)

post-Rotate-angle (S, A, θ, S') $\triangleq$

(S' = rotation (S, A)) ∧
(**let** T = image (S,S') **in**

(T $\neq$ NIL) $\wedge$

(<u>if</u> $\sim$ lie-on-axis (O, A) <u>then</u>

let O = position (POSI-ORIE (S)),

O' = transform-point (T, O)

Q = intersect (normal (O, A), A) **in**

angle (const-line (O, Q), const-line (O', Q)) = θ

tel)

<u>else</u>

($\forall$ p $\in$ Point)

((p $\neq$ O) $\wedge$ (On (p, S)) $\wedge$ ($\sim$ lie-on-axis (p, A)) $\Rightarrow$

let p' = transform-point (T, p),

R = intersect (normal (p, A), A) **in**

angle (const-line (p, R), const-line (p', R) = θ

tel

)

tel)

Similar to our remarks on the inverse of 'translate-dist', the function 'rotate-angle' is bijective and has an inverse.

Theorem 6 *(There exists an inverse for 'rotate-angle'.)If S' = rotate-angle (S, A, θ) then S = rotate-angle (S', A, -θ).*

<u>Lemma 1</u> : Every point on a rigid solid S is rotated through the same angle when the rotation is about an axis passing through the origin of the local coordinate frame of S.

<u>Proof</u>

Let T represent a coordinate frame whose origin coincides with the origin O of the local coordinate frame of S, such that axis OZ of T coincides with the axis of rotation. Any property of the solid computed with respect to the local coordinate frame can be derived from the properties computed with respect to T.

The proof for this lemma is given in two parts.

The first part proves that rotation of S through an angle θ causes the frame T to be rotated through an angle θ.

Since OZ coincides with the axis of rotation,

the image z' of every point z on OZ coincides with z itself due to rotation. (1)

Let x be a point on OX and x' be its image due to rotation through θ.

Similarly, let y be a point on OY and y' be its image due to rotation through θ.

All the four points x,y,x' and y' lie on the same plane (i.e., XOY plane).

Since $\angle xOx' = \angle yOy' = \theta$ and $\angle xOy = 90$, from the specification for 'rotation', we derive

$\angle x'Oy' = \angle x'Oy + \angle yOy' = \angle x'Oy + \angle xOx' = \angle xOy = 90.$

$\angle x'Oz' = \angle y'Oz' = \angle x'Oy' = 90.$ $\cdots$ (2)

From (1) and (2),

frame x'y'z' is the image of frame xyz under rotation through θ. $\cdots$ (3)

The second part proves that every point p is rotated through an angle θ.

Let p be an arbitrary point and let p' be its image due to rotation.

By specification of rigid solid,

vector-equal $(\overrightarrow{Op}, \overrightarrow{Op'})$ is true. i.e., $\angle pOx = \angle p'Ox'$

By specification of rotation, p and p' lie on a circle.

$\Rightarrow$ rotation of vector $\overrightarrow{Op}$ creates a cone whose vertex is O and

the base of the cone lies in a plane parallel to the XY-plane of T $\cdots$ rotation about OZ axis.

$\Rightarrow$ angles made by the vectors $\overrightarrow{Op}$ and $\overrightarrow{Op'}$ with the OZ axis is the same.

Consequently,

it is sufficient to prove that the angle between the vectors $\overrightarrow{Op_0}$ and $\overrightarrow{Op_0'}$ is θ where p_0 and p_0' are the projections of p and p' on the XY-plane.

$\angle xOp_0 = \angle x'Op_0' = \alpha$

$\angle p_0Op_0' = \angle xOp_0' - \angle xOp_0$

$= \angle xOx' + \angle x'Op_0' - \angle xOp_0$

$= \theta.$

Lemma 2 : Let O be the origin of the local coordinate frame of a solid S. Let S' be the image of S when S is rotated about an axis A and O' be the origin of the local

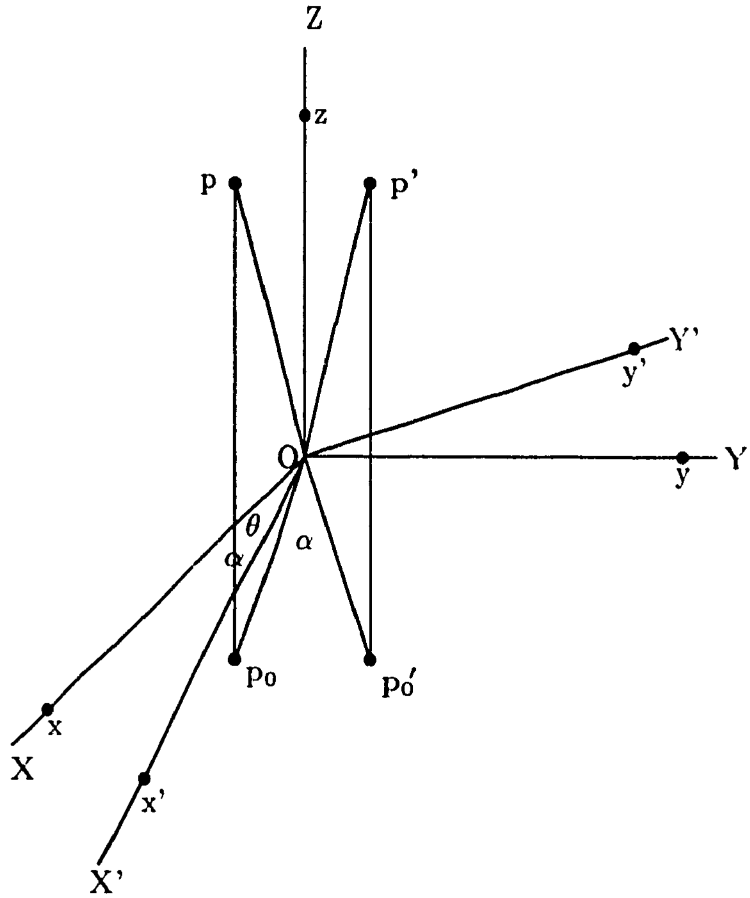


Figure 4.3: Rotation about an axis.

coordinate frame of S'. Let A' be the axis through O, parallel to A. Rotation of S through an angle θ about A is equivalent to rotation of S through θ about A' followed by a translation through $\overrightarrow{OO'}$.

<u>Proof</u>

Let S" be the image of S under a rotation of θ about the axis A' and p" be the image of an arbitrary point p on S.

If p' on S' is the image of p on S, then by specification of rigid solid,

vector-equal $(\overrightarrow{Op}, \overrightarrow{O'p'})$ and vector-equal $(\overrightarrow{Op}, \overrightarrow{Op''})$ $\cdots$ (1)

Let p"' be the image of p" due to translation of S" by a distance d = distance (O,O').

By specification of translation,

O' is the image of O under translation.

By specification of rigid solid,

vector-equal $(\overrightarrow{Op''}, \overrightarrow{O'p'''})$ $\cdots (2)$

From (1), (2) and Theorem 1,

vector-equal $(\overrightarrow{O'p'}, \overrightarrow{O'p'''}) \Rightarrow$ p"' coincides with p'.

$\Rightarrow$ p' is the image of p" under translation.

Since p is an arbitrary point on S, it follows that S' is the image of S" under translation. That is,

Rotation of S through θ about A $\equiv$

translation ((rotation of S through θ about A'), distance (O,O')).

Theorem 7 *Every point on a rigid solid S is rotated through the same angle by* rotation.

Proof :

The proof follows from Lemma 1 and Lemma 2.

Theorem 8 *('Rotate-angle' is additive.) Let S be a solid and O be the origin of its local coordinate frame. For any rotation of S through an angle θ_1 about an axis A_1 passing through O followed by any other rotation through an angle θ_2 about an axis A_2 passing through O taking S to S_2, there exists an angle θ_3 and an axis A_3 through O such that rotation of S through θ_3 about A_3 will take S to S_2.*

Proof :

From the specification of rotation, it is clear that the image of a point p due to rotation lies on a circular arc and points on the axis of rotation are unchanged.

The proof is based on the properties of spherical triangles constructed by the circular arcs of the images of points due to rotation [Par65]. Assume that OA_1 and OA_2 are of unit length so that A_1 and A_2 lie on the unit sphere about O.

Construct the spherical triangle $A_1A_2A_3$ with the $\angle A_1 = \frac{1}{2}\theta_1$ and $\angle A_2 = \frac{1}{2}\theta_2$ as shown in Figure 4.4. We claim that

- OA_3 is the resultant axis of rotation and
- $\theta_3 = -2\angle A_3$.

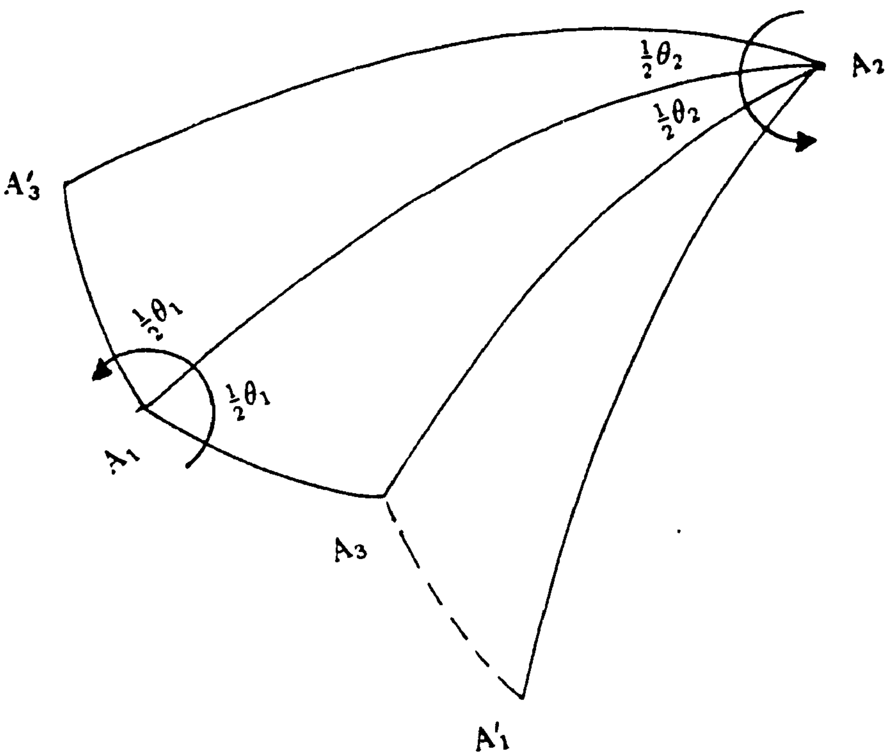


Figure 4.4: Additivity of Rotations.

The first claim is proved if we prove that the line OA_3 is an invariant under the composition of the two rotations. To prove this, consider the spherical triangle $A_1A_2A'_3$ which is the image of the triangle $A_1A_2A_3$ on the arc A_1A_2. The first rotation takes A_3 to A'_3 and the second rotation brings A'_3 back to A_3. This being true for every point on the line OA_3, it follows that OA_3 is the axis of the resultant rotation.

The proof for the second claim is as follows :

Let the spherical triangle $A'_1A_2A_3$ be the image of the triangle $A_1A_2A_3$ on the arc A_2A_3.

The first rotation will leave A_1 fixed and the second rotation will take A_1 to A'_1, leaving A_2 unchanged (A_3 remains unchanged as shown in the proof for the first part).

Hence,

$$\theta_3 = \angle A_1A_3A'_1 = 2\,(\Pi - \angle A_3) = -\,2\,\angle A_3.$$

$$\cos\tfrac{1}{2}\theta_3 = \cos\tfrac{1}{2}\theta_1\,\cos\tfrac{1}{2}\theta_2 - \sin\tfrac{1}{2}\theta_1\,\sin\tfrac{1}{2}\theta_2\,\cos\lambda$$

where λ is the angle between axes OA_1 and OA_2. The principal angle to the solution to this trigonometric equation gives θ_3, the angle of rotation about OA_3.

Corollary : ('Rotate-angle' is not commutative.)

rotate-angle (rotate-angle (S, A_1, θ_1), A_2, θ_2) $\not\equiv$

rotate-angle (rotate-angle (S, A_2, θ_2), A_1, θ_1).

Proof :

From Figure 4.4, notice that

rotate-angle (rotate-angle (S, A_1, θ_1), A_2, θ_2) will have OA_3

as the resultant axis of rotation, where as

rotate-angle (rotate-angle (S, A_1, θ_1), A_2, θ_2) will have OA'_3

as the resultant axis of rotation, where A'_3 is the image of A_3 on the arc A_1A_2.

Hence, rotate-angle is not commutative.

Theorem 9 *For any rotation of a solid S through an angle θ_1 about an arbitrary axis A_1 followed by any other rotation through an angle θ_2 about another arbitrary axis A_2 taking S to S_2, there exists an angle θ_3 and an axis A_3 such that rotation of S through θ_3 about A_3 will take S to S_2.*

Proof : The proof follows from Lemma 2, Theorem 4 and Theorem 8.

4.2.3 Specifications for Prismatic and Revolute Joints

Prismatic and Revolute joints are commonly used structures for building robotic manipulators. A joint connects two rigid solids in which one of them is fixed (with respect to the joint) and the other moves along/about an axis defined within the joint. This motion is a *translation in Prismatic joint* and a *rotation in Revolute joint.* Figures 4.5 and 4.6 show the structures of Prismatic and Revolute joints.

Prismatic Joint

Type Definition

```
Prisjoint        :: S1 : Solid
                    S2 : Solid
                    AXIS-OF-MOVE : Axis-Rep
                    MIN-DISPL : Dist-Rep
                    MAX-DISPL : Dist-Rep
```

Solid S2 can be moved by translation along the AXIS-OF-MOVE. MIN-DISPL and

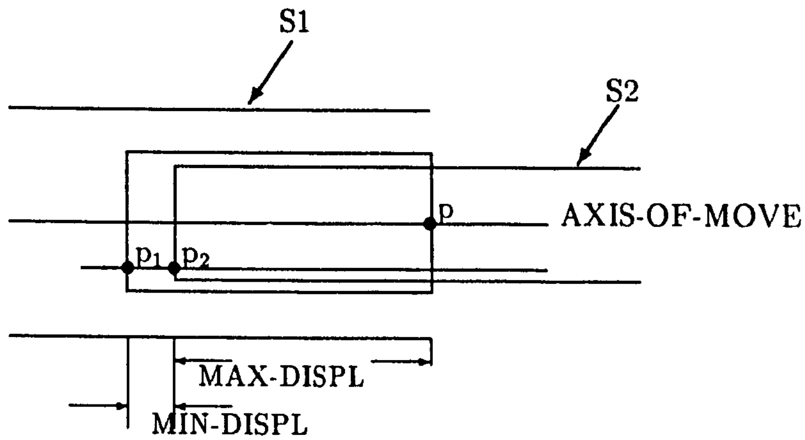


Figure 4.5: Structure of a Prismatic Joint.

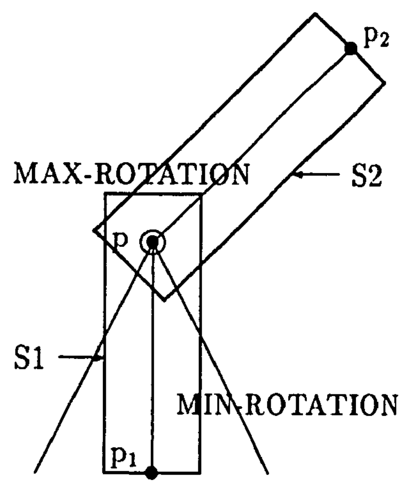


Figure 4.6: Structure of a Revolute Joint.

MAX-DISPL specify the minimum and maximum permissible displacements of S2. Informally, the main characteristics of a prismatic joint are

1. There exists a point p on the AXIS-OF-MOVE common to both S1 and S2. That is, the two links S1 and S2 are always connected.

2. There exist two points p_1 on S1 and p_2 on S2 such that

   - line p_1p_2 is parallel to the AXIS-OF-MOVE
   - MIN-DISPL $\leq$ distance $(p_1, p_2) \leq$ MAX-DISPL
   - distance $(p_1, p_2) >$ MIN-DISPL indicates that S2 has been translated along the AXIS-OF-MOVE by a distance d = distance (p_1, p_2) – MIN-DISPL.

Formally,

$$
\begin{array}{l}
\text{inv-Prisjoint (jn)} \quad \triangleq \\
\quad (\exists\ p \in \text{Point}) \\
\qquad ((\text{on } (p, \text{S1(jn)})) \wedge (\text{on } (p, \text{S2(jn)})) \wedge \\
\qquad (\text{lie-on-line } (p, \text{AXIS-OF-MOVE(jn)})) \wedge \\
\qquad (\exists\ p_1, p_2 \in \text{Point}) \\
\qquad\quad ((\text{on } (p_1, \text{S1(jn)})) \wedge (\text{on } (p_2, \text{S2(jn)})) \wedge \\
\qquad\quad (\text{parallel } (\text{const-line } (p_1, p_2)), \text{AXIS-OF-MOVE (jn)}) \wedge \\
\qquad\quad (\textbf{let } d = \text{distance } (p_1, p_2) - \text{MIN-DISPL (jn)} \textbf{ in} \\
\qquad\qquad (\text{MIN-DISPL (jn)} \leq d \leq \text{MAX-DISPL (jn)}) \wedge \\
\qquad\qquad d > 0 \Rightarrow \text{on } (p_2, \text{translate-dist (S2, AXIS-OF-MOVE (jn), d)}) \\
\qquad\quad \textbf{tel}) \\
\qquad\quad) \\
\qquad)
\end{array}
$$

Revolute Joint

Type Definition

Revoljoint	::	S1 : Solid S2 : Solid AXIS-OF-ROTATION : Axis–Rep MIN-ROTATION : Angle–Rep MAX-ROTATION : Angle–Rep

Solid S2 rotates with respect to the AXIS-OF-ROTATION. MIN-ROTATION and MAX-ROTATION specify the minimum and maximum permissible rotations by which S2 can be rotated.

Informally, the important characteristics of a revolute joint are

1. There exists a point p lying on the AXIS-OF-ROTATION common to both S1 and S2.

2. There exist two points p_1 on S1 and p_2 on S2 such that

   - $(p \neq p_1)$ and $(p \neq p_2)$

- lines pp_1 and pp_2 are perpendicular to the AXIS-OF-ROTATION

- MIN-ROTATION $\leq$ angle $(pp_1, pp_2) \leq$ MAX-ROTATION.

- angle $(pp_1, pp_2) >$ MIN-ROTATION indicates that S2 is rotated about the AXIS-OF-ROTATION through $\theta =$ angle (pp_1, pp_2) - MIN-ROTATION.

Formally,

inv-Revoljoint (jn) $\triangleq$
  ($\exists$ p $\in$ Point)
    ((on (p, S1)) $\wedge$ (on (p, S2)) $\wedge$
    (lie-on-line (p, AXIS-OF-ROTATION (jn))) $\wedge$
    ($\exists$ p_1, p_2 $\in$ Point)
      ((on (p_1, S1)) $\wedge$ (on (p_2, S2)) $\wedge$
      (perpendicular (const-line (p, p_1), AXIS-OF-ROTATION (jn))) $\wedge$
      (perpendicular (const-line (p, p_2), AXIS-OF-ROTATION (jn))) $\wedge$
      (**let** $\theta =$ angle (const-line (p, p_1), const-line (p, p_2)) **in**
        (MIN-ROTATION (jn) $\leq \theta \leq$ MAX-ROTATION (jn)) $\wedge$
        (θ - MIN-ROTATION (jn) $> 0 \Rightarrow$
          on (p_2, rotate-angle (S2, AXIS-OF-ROTATION (jn), θ)))
      tel)
)
)

4.3 Formalism of Robot Kinematics

In this section, we provide formal specifications for robot kinematics. Since there exists a variety of robots in practice, we restrict ourselves to a particular class of robots whose structure is discussed in the following section. We claim that our assumption is sufficiently general to formally capture the geometric structure and functionalities of many existing robots such as PUMA and Stanford Manipulators.

4.3.1 Robot Structure

In order to simplify the discussions, we consider a general-purpose multi-link robot with multi-fingered end-effector. One end of the first link of the robot is assumed to be fixed at a known location in the workspace. Two links are assumed to be joined by either a Prismatic joint or a Revolute joint. As explained in the previous section, a joint definition includes the two links joined at that place and the maximum and minimum permissible displacements of the moving link. The end-effector is attached to the last link of the robot and is treated differently from the links. In this report, the terms 'end effector' and 'gripper' are used interchangeably. The base of the gripper is called the 'wrist' to which the fingers or tools may be attached. In order to give a full picture of a robot, fingers are included in the state definition. However, we do not provide specifications for the fingers in subsequent sections since we only deal with kinematic aspects of robots. Specifications for fingers and tools, if any, will be considered in the context of specific applications such as grasping. Figure 4 7 gives the structure of a robot discussed in this section.

The shape of the links is usually described by a solid modeler. However we do not consider sweeping and volumetric aspects of the robot structure in this paper and hence we restrict ourselves to the primitive shapes *cone, cuboid* and *cylinder* and those composite structures that can be built from these primitives using regularized operations union, intersection and difference. Associated with every geometric entity is a position and an orientation. The position refers to the origin of the local coordinate frame embedded in the geometric entity.

4.3.2 Formal Model of Robots

For us, a robot environment consists of a robot and its coordinate frames. We do not consider the workspace of the robot in our discussions. The type definitions given below show the robot environment in **bold face**. The robot consists of a list of joints, a list of links and an end-effector (gripper). Each link has a unique identification number for reference. Its structure is defined by the variable 'GEOMETRY'. The base of the gripper is denoted by the variable 'WRIST' to which fingers are attached using prismatic and revolute joints. The types 'Prisjoint' and 'Revoljoint' have al-

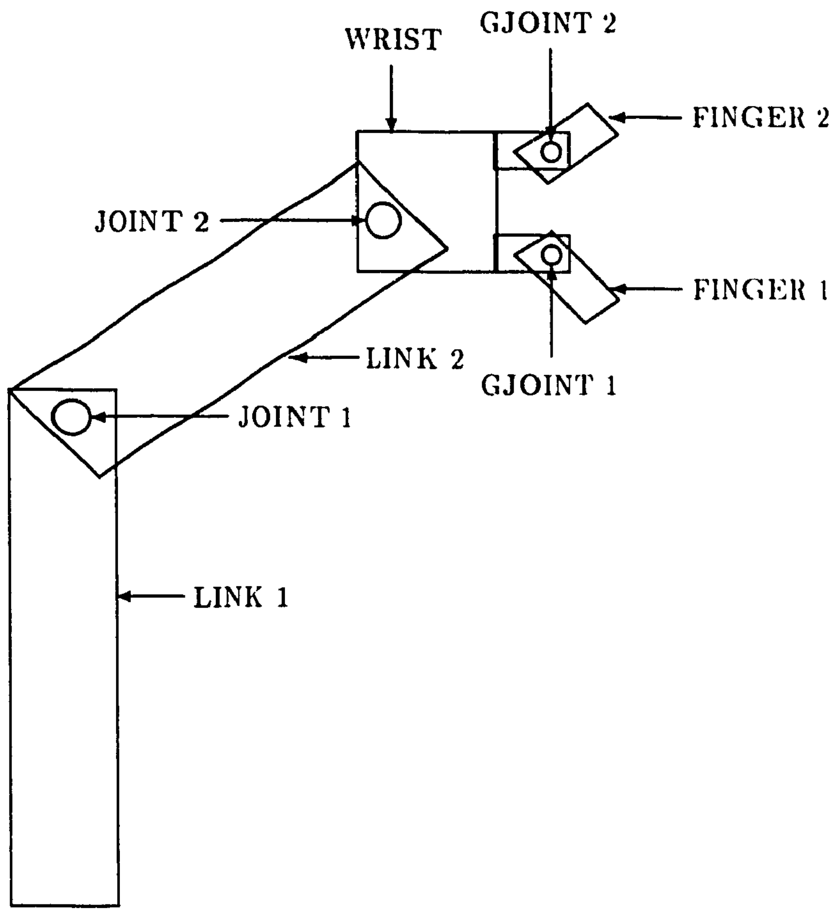


Figure 4.7: A Two-Link Manipulator (with only Revolute Joints).

ready been introduced in Section 3. There is a one-to-one correspondence between the fingers and the joints at the gripper, which is denoted by the map 'Fingertype → Jointtype'. Fingers will also have unique identification numbers.

State ::

BASE-COORD : Transformation

ROBOT-ARM : Manipulator

Manipulator :: LINKS : Armtype-list
JOINTS : Jointtype-list
GRIPPER : Grippertype

Armtype :: LINKID : ID-Rep
GEOMETRY : Structure

Jointtype	=	Prisjoint | Revoljoint
Grippertype	::	WRIST : Wristtype FINGER-GRIP-JOINTS : Fingertype $\rightarrow$ Jointtype
Wristtype	::	GEOMETRY : Structure
Fingertype	::	FINGERID : ID–Rep GEOMETRY : Structure

Type Invariants

The type invariants for the robot-arm are the following :

1. The links, the wrist and the fingers are all rigid.

2. LINKIDs as well as FINGERIDs are unique.

3. The number of joints is one less than the number of links since there is a joint between every two links. However, we treat the end-effector differently from the links and so there exists another joint between the last link and the end-effector. Hence the number of joints is *numerically* equal to the number of links.

4. The specification is concerned with linear chain of links; i.e., there is no closed structure formed by the links and the end-effector. This property can also be stated as follows :

   - Except for the first link, every link becomes part of exactly two joints. The first link is part of the first joint only. This is an assumption made as part of the robot structure. Thus the first link does not move and is static.

   - Except for the last joint, every joint connects exactly two distinct links. The last joint connects the last link and the end-effector.

Formally, these type invariants are stated as follows :

inv–Manipulator (lns, jns, grip) $\triangleq$

let fns = **dom** FINGER-GRIP-JOINTS (grip) **in**

(* Every link, the wrist and every finger is rigid. *)

($\forall$ i {1 ·· **len** lns }) (rigid (GEOMETRY (lns(i)))) $\wedge$

(rigid (GEOMETRY (WRIST (grip)))) $\wedge$

($\forall$ fn $\in$ fns) (rigid (GEOMETRY (fn))) $\wedge$

(* LINKIDs are unique *)

($\forall$ ln_1,ln_2 $\in$ **elems** lns)

(LINKID (ln_1) = LINKID (ln_2) $\Rightarrow$ ln_1 = ln_2) $\wedge$

(* FINGERIDs must be unique *)

($\forall$ fn_1,fn_2 $\in$ fns)

(FINGERID(fn_1) = FINGERID(fn_2) $\Rightarrow$ fn_1 = fn_2) $\wedge$

(* no.of joints = no.of links *)

len jns = **len** lns

(* Except for the first, every link is part of exactly two distinct joints. *)

($\forall$ i $\in$ {2 ·· **len** lns})

(($\exists!$ jn_1, jn_2 $\in$ **elems** jns)

((jn_1 $\neq$ jn_2) $\wedge$ (lns(i) = LINK2 (jn_1)) $\wedge$ (lns(i) = LINK1 (jn_2)))

) $\wedge$

(lns(1) = LINK1 (jns(1))) $\wedge$

(* Except for the last, every joint connects exactly two distinct links. *)

($\forall$ jn $\in$ **elems** (jns − jns(**len** jns)))

(($\exists!$ i,j $\in$ {1 ·· **len** lns})

((i $\neq$ j) $\wedge$ (LINK1 (jn) = lns(i)) $\wedge$ (LINK2 (jn) = lns(j))

)

) $\wedge$

(LINK1 (jns(**len** jns)) = lns (**len** lns)) $\wedge$ (LINK2 (jns(**len** jns)) = grip)

tel

4.3.3 Specifications for Forward Kinematics

In this section, we address the forward kinematics problem, namely "*given the change in positional information of the i^{th} link, determine the entire configuration of the* *robot*". Since there are two types of joints, we provide the specifications for two operations, namely *translate-link* (corresponding to the prismatic joint) and *rotate-link* (corresponding to the revolute joint). The specifications describe the effect on only one link; the consequences on the entire configuration are described by the theorems following the specifications.

Translation of a Link

TRANSLATE-LINK (TLINK : ID–Rep; DIST : Dist–Rep)
(∗ Translate the link with i.d. 'TLINK' through a distance 'DIST'. ∗)

ext ROBOT-ARM : **wr** Manipulator

Pre

  let lns = LINKS (robot-arm),
    jns = JOINTS (robot-arm) **in**
    (∗ Verify that the link i.d. passed as parameter is valid. ∗)
    (∃! k ∈ { 1 ·· **len** lns})
      ((LINKID (lns(k)) = tlink) ∧
      (∗ Verify that the corresponding joint is Prismatic. ∗)
      (∃! jn ∈ **elems** jns)
        ((jn ∈ Prisjoint) ∧
        (LINK2(jn) = lns(k))
)
)
  tel

Post

  let lns = LINKS (robot-arm) **in**
    (∃! k ∈ {1 ·· **len** lns})
      ((LINKID (lns(k)) = tlink) ∧
      (∃! jn ∈ **elems** jns)
        ((LINK2(jn) = lns(k)) ∧
        (**let** axis = AXIS-OF-MOVE (jn) **in**
          lns(k)' = *mk-Armtype* (LINKID (lns(k)),

```
                              translate-dist (GEOMETRY (lns(k)), axis, dist))
                tel)
              )
            )
    tel
```

Rotation of a Link

```
ROTATE-LINK (RLINK : ID-Rep; THETA : Angle-Rep)
(* Rotate the link with i.d. 'RLINK' through an angle 'THETA'. *)

ext    ROBOT-ARM : wr Manipulator
Pre
    let lns = LINKS (robot-arm),
        jns = JOINTS (robot-arm) in
        (* Verify that the link i.d. passed as parameter is valid. *)
        (∃! k ∈ { 1 ·· len lns})
            ((LINKID (lns(k)) = rlink) ∧
             (* Verify that the corresponding joint is Revolute. *)
             (∃! jn ∈ elems jns)
                ((jn ∈ Revoljoint) ∧
                 (LINK2(jn) = lns(k))
                )
            )
    tel
Post
    let lns = LINKS (robot-arm) in
        (∃! k ∈ {1 ·· len lns})
            ((LINKID (lns(k)) = rlink) ∧
             (∃! jn ∈ elems jns)
                ((LINK2(jn) = lns(k)) ∧
                 (let axis = AXIS-OF-ROTATION (jn) in
                    lns(k)' = mk_Armtype (LINKID (lns(k)),
```

```
                    rotate-angle (GEOMETRY (lns(k)), axis, theta))
          tel)
        )
      )
  tel
```

Let **len** lns = n and $(n+1)^{st}$ link be the wrist.

Theorem 10 *For $1 \leq i \leq n$, translating the i^{th} link by a distance d causes all the links from (i+1) to n, the wrist and all the fingers to be translated by the same distance d along the axis of translation of i^{th} link.*

Proof

Since $link_i$ is rigid, by Theorem 3, all the points of $link_i$ are translated by a distance d along the axis of translation. Since every joint j, $i < j \leq n$, is static, the relative position of the origin of the local coordinate frame of the j^{th} link with respect to the i^{th} link remains unchanged. Hence, by Theorem 3, every point on the configuration of the robot from link (i+1) to n is translated by a distance d along the axis of translation of i^{th} link. i.e., every link from (i+1) to n, the wrist and every finger is translated by a distance d along the axis of translation of i^{th} link.

Theorem 11 *For $1 \leq i \leq n$, rotating the i^{th} link by an angle θ causes all the links from (i+1) to n, the wrist and all the fingers to be rotated through the same angle θ about the axis of rotation of i^{th} link.*

Proof

The proof is similar to that of Theorem 10.

4.3.4 Specification for Inverse Kinematics

Task level descriptions of robot manipulations can be realized through inverse kinematics solutions. The problem of inverse kinematics can be informally stated as follows : "*given the positional change in end-effector, determine the valid configurations for the links of the robot*". Any tools or fingers attached to the end-effector are not considered in finding solutions for inverse kinematics problem. In general, there are many feasible solutions for this problem. We do not provide procedures for finding all possible solutions; rather, we characterize these solutions so that an offline formal verification of an algorithm computing these solutions can be carried out.

As stated earlier, commercial robots greatly vary in their structure. For example, a PUMA robot has only revolute joints. Hence any feasible solution for the inverse kinematics problem which requires a translation may not be successfully applied to a PUMA robot. Similarly, solutions requiring rotations cannot be applied to robots which have only prismatic joints. Therefore, in order to characterize all feasible solutions of inverse kinematics problem applicable to all types of robots, we categorize the problem into four parts :

1. Given two positions P_w and Q_w of the wrist having the same orientation, determine the valid configurations of the links to obtain Q_w from P_w using only *translations.*

2. Given two positions P_w and Q_w of the wrist having the same orientation, determine the valid configurations of the links to obtain Q_w from P_w using only *rotations.*

3. Given two positions P_w and Q_w of the wrist with different orientations, determine the valid configurations of the links to obtain Q_w from P_w using only *rotations.*

4. Given two positions P_w and Q_w of the wrist with different orientations, determine the valid configurations of the links to obtain Q_w from P_w using a combination of *translations* and *rotations.*

For each problem, we first informally describe a solution and then give a formal specification.

Problem 1

An Informal Description of Solution

Let A be the line joining P_w and Q_w. This is the required axis of translation with respect to the base coordinate frame and its direction is from P_w to Q_w.

Let d = distance (P_w, Q_w) and Pjn be the set of prismatic joints.

A feasible solution requires the existence of an arbitrary subset of prismatic joints SPjn such that

- SPjn $\subseteq$ Pjn.
- **card** SPjn = k.
- $A_1, A_2, \ldots, A_k$ are the axes of translation of the prismatic joints $SPjn_1$, $SPjn_2$, $\ldots$, $SPjn_k$.
- ${}^BA_1$, ${}^BA_2$, $\ldots$, ${}^BA_k$ are the axes of translation of the k joints with respect to the base coordinate frame.

It is now clear that Q_w can be obtained from P_w by translating the second link of $SPjn_i$ through a distance d_i along the axis ${}^BA_i$, $1 \leq i \leq k$, such that

- $d_i \leq$ MAX-DISPL ($SPjn_i$).
- $\sum_{i=1}^{k} \overrightarrow{d_i} = \overrightarrow{d}$.

where $\overrightarrow{d_i}$ is the vector of length d_i along the axis ${}^BA_i$ and $\overrightarrow{d}$ is the vector of length d along the axis A.

<u>Specification</u>

MOVE-WRIST-T (DESTINATION : Transformation)
(* Move the wrist to 'DESTINATION' using only translations. *)

ext
  ROBOT-ARM : **wr** Manipulator
  BASE-COORD : **rd** Transformation
Pre
 let jns = JOINTS (robot-arm),
  grip = GRIPPER (robot-arm),
  wrs = WRIST (grip) **in**
  (* Assure that the orientation of the wrist is unchanged. *)

let oldorie = orientation (POSI-ORIE (GEOMETRY (wrs))),
newore = orientation (destination) **in**
same-orientation (oldorie, neworie)
tel
$\wedge$ **let** oldposi = position (POSI-ORIE (GEOMETRY (wrs))),
newposi = position (destination),
d = vector (oldposi, newposi) **in**
(* jnlist is the set of prismatic joints *)
($\exists$ jnlist $\in$ Jointtype-list)
((**elems** jnlist $\subseteq$ **elems** jns) $\wedge$
($\forall$ i $\in$ {1 ·· **len** jnlist })
((jnlist(i) $\in$ Prisjoint) $\wedge$
($\forall$ j $\in$ {1 ·· **len** jnlist })
((i $\neq$ j) $\Rightarrow$ (jnlist(i) $\neq$ jnlist(j)))
(* vlist is the set of vectors applied to the links of jnlist.
^{B}vlist is the set of vectors vlist w.r.t the base. *)
($\exists$ vlist, ^{B}vlist $\in$ Vectortype-list)
((**len** vlist = **len** jnlist) $\wedge$
(**len** ^{B}vlist = **len** jnlist) $\wedge$
(norm (vlist(i)) $\leq$ MAX-DISPL (jnlist(i))) $\wedge$
(parallel (direction (vlist(i)), AXIS-OF-MOVE (jnlist(i)))) $\wedge$
(^{B}vlist(i) = vector-base (vlist(i),
POSI-ORIE (GEOMETRY (LINK2(jnlist(i)))),
base-coord)) $\wedge$
(v = vector-sum (^{B}vlist))
)
)
)
tel
tel
Post

let jns = JOINTS (robot-arm),
  grip = GRIPPER (robot-arm),
  wrs = WRIST (grip) **in**
  ($\exists$ jnlist $\in$ Jointtype-list,
    dlist $\in$ Dist–Rep-list)
    ((**len** jnlist = **len** dlist) $\wedge$
     (**elems** jnlist $\subseteq$ **elems** jns) $\wedge$
     ($\forall$ i $\in$ {1 $\cdot\cdot$ **len** jnlist})
      ((jnlist(i) $\in$ Prisjoint) $\wedge$
       (translate-link (LINKID (LINK2 (jnlist(i))), dlist(i)))
)
)
  $\Leftrightarrow$ POSI-ORIE (wrs)' = destination
tel

In the above specification, the variable 'jnlist' denotes the subset of prismatic joints SPjn, as stated in the informal description of the solution. We have chosen 'jnlist' as a list of joints rather than a set, so that it is easier to associate the corresponding elements from 'list' and '^{B}vlist' later. Moreover, we do not use the ordering property of elements within 'jnlist'. However, VDM allows duplication of elements in a list. In order to assure that 'jnlist' does not contain any duplicate elements, we have provided the predicate

  ($\forall$ j $\in$ {1 $\cdot\cdot$ **len** jnlsit})
    ((i $\neq$ j) $\Rightarrow$ (jnlist(i) $\neq$ jnlist(j)))

Problem 2

An Informal Description of Solution

Let the revolute joints be the set Rjn.

In any feasible solution, there exists an arbitrary subset of revolute joints SRjn satisfying the following constraints :

- SRjn $\subseteq$ Rjn.
- **card** SRjn = k.

- There exists a set of angles $\phi_1, \phi_2, \ldots, \phi_k$ such that rotating the second link of $SRjn_i$ through an angle ϕ_i causes a change in orientation to the second link of $SRjn_i$; call this change in orientation T_i and the corresponding linear displacement of the second link of $SRjn_i$ due to rotation through ϕ_i by d_i.

It is easy to see that Q_w can be obtained from P_w by rotating the second link of $SRjn_i$ through an angle ϕ_i about the axis of rotation of $SRjn_i$, $1 \le i \le k$, satisfying the constraints

- $\phi_i \le$ MAX-ROTATION ($SRjn_i$).

- $\sum_{i=1}^{k} T_{ix} = 0$, $\sum_{i=1}^{k} T_{iy} = 0$, $\sum_{i=1}^{k} T_{iz} = 0$ and $\sum_{i=1}^{k} \overrightarrow{d_i} = \overrightarrow{d}$
where $\overrightarrow{d}$ is the vector from P_w to Q_w.

In fact, problem 2 is a special situation of problem 3; i.e., if the change in orientation between P_w and Q_w is set to null, then problem 3 becomes problem 2. Hence we do not give the specifications for problem 2; rather, we give the specification for problem 3.

<u>Problem 3</u>

<u>Informal Description of the Solution for Problem 3</u>

The position Q_w can be obtained from P_w by rotating the second link of $SRjn_i$ through an angle ϕ_i along the axis of rotation of $SRjn_i$, $1 \le i \le k$, satisfying the constraints

- $\phi_i \le$ MAX-ROTATION ($SRjn_i$).

- $\sum_{i=1}^{k} T_{ix} = T_x$, $\sum_{i=1}^{k} T_{iy} = T_y$, $\sum_{i=1}^{k} T_{iz} = T_z$ and $\sum_{i=1}^{k} \overrightarrow{d_i} = \overrightarrow{d}$
where T_x, T_y and T_z are respectively the x, y and z components of the change in orientation of the wrist between P_w and Q_w with respect to the base coordinate frame.

<u>Specification</u>

MOVE-WRIST-OR (DESTINATION : Transformation)

(* Move the wrist to 'DESTINATION' by pure rotation with a change in orientation. *)

ext

ROBOT-ARM : **wr** Manipulator

BASE-COORD : **rd** Transformation

Pre

let grip = GRIPPER (robot-arm),

jns = JOINTS (robot-arm),

wrs = WRIST (grip),

oldorie = orientation (POSI-ORIE (GEOMETRY (wrs))),

oldposi = position (POSI-ORIE (GEOMETRY (wrs))),

newposi = position (destination),

neworie = orientation (destination),

(* T_w is the change in orientation of the wrist and x_w, y_w, z_w

are its x,y,z components. *)

T_w = change-in-orientation (oldorie, neworie),

x_w = X-component (T_w),

y_w = Y-component (T_w),

z_w = Z-component (T_w),

d = vector (oldposi, newposi) **in**

(* jnlist is the set of revolute joints. *)

($\exists$ jnlist $\in$ Jointtype-list)

((**elems** jnlist $\subseteq$ **elems** jns) $\wedge$

($\forall$ i $\in$ {1 ·· **len** jnlist})

((jnlist(i) $\in$ Revoljoint) $\wedge$

($\forall$ j $\in$ {1 ·· **len** jnlist})

((i $\neq$ j) $\Rightarrow$ (jnlist(i) $\neq$ jnlist(j)))

(* ϕlist is the set of angles applied to the second links of jnlist.

Tlist is the corresponding set of change in orientations after ϕlist applied.

dlist is the corresponding set of linear displacements due to ϕlist.*)

($\exists$ ϕlist $\in$ Angle–Rep-list,

Tlist $\in$ Transformation-list,

dlist $\in$ Vectortype-list)

((**len** ϕlist = **len** jnlist) $\wedge$

(**len** Tlist = **len** jnlist) $\wedge$

(ϕlist(i) $\leq$ MAX-ROTATION (jnlist(i))) $\wedge$

(**let** T_{old} = orientation (POSI-ORIE (GEOMETRY

(LINK2 (jnlist(i))))),

T_{new} = orientation (POSI-ORIE

(rotate-angle (GEOMETRY (LINK2 (jnlist(i)))

AXIS-OF-MOVE (jnlist(i)), ϕlist(i))))

Tlist(i) = change-in-orientation (T_{old}, T_{new})) $\wedge$

tel $\wedge$

(**let** P_{old} = position (POSI-ORIE (GEOMETRY

(LINK2 (jnlist(i))))),

P_{new} = position (POSI-ORIE

(rotate-angle (GEOMETRY (LINK2 (jnlist(i)))

AXIS-OF-MOVE (jnlist(i)), ϕlist(i))))

dlist(i) = vector (P_{old}, P_{new})) $\wedge$

tel $\wedge$

(* Xlist, Ylist, Zlist are the x,y,z components of Tlist. *)

($\exists$ Xlist, Ylist, Zlist $\in$ Real-list)

((**len** Xlist = **len** Tlist) $\wedge$

(**len** Ylist = **len** Tlist) $\wedge$

(**len** Zlist = **len** Tlist) $\wedge$

(Xlist(i) = X-component (transform-base (Tlist(i), base-coord)) $\wedge$

(Ylist(i) = Y-component (transform-base (Tlist(i), base-coord)) $\wedge$

(Zlist(i) = Z-component (transform-base (Tlist(i), base-coord)) $\wedge$

(Real-sum (Xlist) = x_w) $\wedge$

(Real-sum (Ylist) = y_w) $\wedge$

(Real-sum (Zlist) = z_w) $\wedge$

(vector-sum (dlist) = d)

)

)

```
        )
      )
    tel
  Post
    let jns = JOINTS (robot-arm),
      grip = GRIPPER (robot-arm),
      wrs = WRIST (grip) in
      (∃ jnlist ∈ Jointtype-list,
        ϕlist ∈ Angle–Rep-list)
        ((len jnlist = len ϕlist) ∧
         (elems jnlist ⊆ elems jns) ∧
         (∀ i ∈ {1 ·· len jnlist})
          ((jnlist(i) ∈ Revoljoint) ∧
           (rotate-link (LINKID (LINK2 (jnlist(i))), ϕlist(i)))
          )
        )
      ⇔ POSI-ORIE (wrs)' = destination
    tel
```

Problem 4

Below, specifications are given for two cases – interleaved activation (i.e., activation of only one link at a time) and concurrent activation (i.e., activation of more than one link at a time).

Case 1 : Interleaved activation.

An Informal Description of the Solution

Let the set of prismatic joints be Pjn and the set of revolute joints be Rjn.

Pjn ∪ Rjn = Jns, the set of joints.

Between Q_w and P_w, there exists a non-empty list of intermediate positions Plist such that

length of Plist = n, n > 0 and Plist(1) = P_w and Plist(n) = Q_w.

For all i, $1 \leq i \leq (n-1)$,

Plist(i+1) can only be obtained from Plist(i) by a translation imposed on one of the prismatic joints or by a rotation on one of the revolute joints. If the orientation between Plist(i) and Plist(i+1) remains unchanged, then translation is applied; otherwise, rotation is applied. Notice that a single rotation necessarily causes a change in orientation. Multiple rotations between successive positions cannot be applied in an interleaved activation.

Specification

MOVE-WRIST-OTR (DESTINATION : Transformation)

(* Move the wrist to 'DESTINATION' with change in orientation by a combination of translations and rotations applied to the links. Only one link is activated at a time. *)

ext

ROBOT-ARM : **wr** Manipulator

BASE-COORD : **rd** Transformation

Pre

let jns = JOINTS (robot-arm),

grip = GRIPPER (robot-arm),

wrs = WRIST (grip) **in**

($\exists$ Pjn, Rjn $\in$ Jointtype-list)

(($\forall$ i $\in$ {1 ·· **len** Pjn}) (Pjn(i) $\in$ Prisjoint) $\wedge$

($\forall$ j $\in$ {1 ·· **len** Rjn}) (Rjn(j) $\in$ Revoljoint) $\wedge$

((**elems** Pjn $\cup$ **elems** Rjn) = **elems** jns) $\wedge$

(**let** oldposi = position (POSI-ORIE (GEOMETRY (wrs))),

oldorie = orientation (POSI-ORIE (GEOMETRY (wrs))),

newposi = position (destination),

neworie = orientation (destination) **in**

($\exists$ Plist $\in$ Transformation-list)

((**len** Plist > 0) $\wedge$

(position (Plist(1)) = oldposi) $\wedge$

(orientation (Plist(1)) = oldorie) $\wedge$

(position (Plist (**len** Plist)) = newposi) $\wedge$

(orientation (Plist (**len** Plist)) = neworie) $\wedge$

```
        (∀ i ∈ {2 ·· len Plist})
         (POSI-ORIE (wrs) = Plist(i) ⇒
           (* If the orientation between successive positions remains
             the same, a translations is applied. *)
           (same-orientation (orientation (Plist(i-1)), orientation (Plist(i))) ⇒
             (∃! k ∈ {1 ·· len Pjn})
               (let v = vector (position (Plist(i-1)), position (Plist(i))) in
                 (parallel (AXIS-OF-MOVE (Pjn(k)), v)) ∧
                 ((norm (v) ≤ MAX-DISPL (Pjn(k)))
                 )
               tel)
           ) ∨
           (* If the orientation between successive positions changes,
             a rotation is applied. *)
           (∼ same-orientation (orientation (Plist(i-1)), orientation (Plist(i))) ⇒
             (∃! k ∈ {1 ·· len Rjn})
               (* Get the angle θ and axis of rotation from the
                 intermediate positions. *)
               (let θ = angle-from-orie (Plist(i-1), Plist(i)),
                  a = identify-axis (Plist(i-1), Plist(i)) in
                   (parallel (AXIS-OF-ROTATION (Rjn(k)), a)) ∧
                   ((θ ≤ MAX-ROTATION (Rjn(k)))
                   )
               tel)
           )
         )
        )
      tel)
     )
  tel
 Post
```

```
let jns = JOINTS (robot-arm),
  grip = GRIPPER (robot-arm),
  wrs = WRIST (grip) in
  (∃ Pjn, Rjn ∈ Jointtype-list)
    ((∀ i ∈ {1 ·· len Pjn}) (Pjn(i) ∈ Prisjoint) ∧
     (∀ j ∈ {1 ·· len Rjn}) (Rjn(j) ∈ Revoljoint) ∧
     ((elems Pjn ∪ elems Rjn) = elems jns) ∧
     (∃ dlist ∈ Dist-Rep-list,
      φlist ∈ Angle-Rep-list)
      ((len dlist = len Pjn) ∧
       (len φlist = len Rjn) ∧
       (∀ i ∈ {1 ·· len Pjn})
        (translate-link (LINKID (LINK2 (Pjn(i))), dlist(i))) ∧
       (∀ j ∈ {1 ·· len Rjn})
        (rotate-link (LINKID (LINK2 (Rjn(j))), φlist(j)))
      )
    ) ⇔ POSI-ORIE (wrs)' = destination
tel
```

The stated pre-condition for rotation can indeed be tested; see [Cra89] for details. The post-condition strongly asserts that the final destination is reached *if and only if* there exists a set of translations and rotations as implied in the predicates.

Interleaved activation of links obtain successive positions due to a single translation or a single rotation. Consequently, the path traversed by the wrist is a collection of piecewise line segments and circular arcs. The converse is also true; i.e., if the trajectory consists of piecewise line segments and circular arcs, then there exists an interleaved activation of links corresponding to this trajectory. However, if the trajectory is not composed of piecewise line segments and circular arcs, then an interleaved activation of the links cannot realize the trajectory. In this situation, a concurrent activation of links becomes necessary. We discuss this solution next.

<u>Case 2</u> : Concurrent activation

<u>Informal description of the solution</u>

Let Pjn and Rjn be respectively the set of prismatic and revolute joints such that (**card** Pjn = l) and (**card** Rjn = k), Pjn $\cup$ Rjn = Jns, the set of joints.
Between P_w and Q_w, there exists a non-empty list of intermediate positions Plist such that

length (Plist) = N, N > 0, Plist(1) = P_w and Plist(N) = Q_w.

For all t, $1 \leq t \leq (N-1)$,

> Plist(t+1) can be obtained only from Plist(t) by applying translation on some subset (possibly empty) of prismatic joints and/or rotation on some subset (possibly empty) of revolute joints simultaneously. The subsets cannot both be empty simultaneously. With reference to the base coordinate frame given in the state definition, this situation is mathematically described as below :
>
> - Let SPjn and SRjn respectively denote arbitrary subsets of prismatic and revolute joints chosen for activation.
> - Let (**card** SRjn = m) and (**card** SPjn = n).
> - ($0 \leq m \leq k$) and ($0 \leq n \leq l$).
> - Let T be the change in orientation of the wrist between Plist(t+1) and Plist(t) and let T_x, T_y and T_z respectively represent its X,Y,Z components.
> - Let $\overrightarrow{d}$ be the linear displacement vector between Plist(t+1) and Plist(t).
> - Let ϕ_i be the angle of rotation applied to the second link of $SRjn_i$, let T_i be the change in orientation of the second link due to ϕ_i and let T_{ix}, T_{iy} and T_{iz} be the X,Y,Z components respectively of T_i, $1 \leq i \leq m$.
> - Let $\overrightarrow{d_j}$ be the displacement vector applied to the second link of the prismatic joint $SPjn_j$, $1 \leq j \leq n$.
>
> The following conditions are to be satisfied for moving the wrist from Plist(t) to Plist(t+1).

- $\sum_{i=1}^{m} T_{ix} = T_x$, $\sum_{i=1}^{m} T_{iy} = T_y$ *and* $\sum_{i=1}^{m} T_{iz} = T_z$.
- $\sum_{j=1}^{n} \overrightarrow{d_j} = \overrightarrow{d_l}$.
- $\overrightarrow{d} = \overrightarrow{d_l} + \overrightarrow{d_\phi}$

  where $\overrightarrow{d_\phi}$ is the net displacement vector due to rotations of m links of SRjn.

Specification

MOVE-WRIST-OTRS (DESTINATION : Transformation)

(* Move the wrist to 'DESTINATION' with change in orientation, by a combination of translations and rotations applied to the links. More than one link may be activated simultaneously. *)

ext

ROBOT-ARM : **wr** Manipulator

BASE-COORD : **rd** Transformation

Pre

let jns = JOINTS (robot-arm),

grip = GRIPPER (robot-arm),

wrs = WRIST (grip) **in**

(∃ Pjn, Rjn ∈ Jointtype-list)

((∀ i ∈ {1 ·· **len** Pjn}) (Pjn(i) ∈ Prisjoint) ∧

(∀ j ∈ {1 ·· **len** Rjn}) (Rjn(j) ∈ Revoljoint) ∧

((**elems** Pjn ∪ **elems** Rjn) = **elems** jns) ∧

(**let** oldposi = position (POSI-ORIE (GEOMETRY (wrs))),

oldorie = orientation (POSI-ORIE (GEOMETRY (wrs))),

newposi = position (destination),

neworie = orientation (destination) **in**

(∃ Plist ∈ Transformation-list)

((**len** Plist > 0) ∧

(position (Plist(1)) = oldposi) ∧

(orientation (Plist(1)) = oldorie) ∧

(position (Plist (**len** Plist)) = newposi) ∧

(orientation (Plist (**len** Plist)) = neworie) $\wedge$

($\forall$ t $\in$ {2 ·· **len** Plist})

(POSI-ORIE (wrs) = Plist(t) $\Rightarrow$

(* dvec is the vector displacement and T is the change in orientation between successive positions. *)

let dvec = vector (position (Plist(t)), position (Plist(t+1))),

T = change-in-orientation (orientation (Plist(t)), orientation (Plist(t+1))),

T_x = X-component (T, base-coord),

T_y = Y-component (T, base-coord),

T_z = Z-component (T, base-coord) **in**

($\exists$ SPjn, SRjn $\in$ Jointtype-list)

(($\forall$ i $\in$ {1 ·· **len** SPjn}) (SPjn(i) $\in$ Prisjoint) $\wedge$

($\forall$ j $\in$ {1 ·· **len** SRjn}) (SRjn(j) $\in$ Revoljoint) $\wedge$

(**len** SPjn $\leq$ **len** Pjn) $\wedge$

(**len** SRjn $\leq$ **len** Rjn) $\wedge$

(**len** SPjn + **len** SRjn $\neq$ 0) $\wedge$

(* dlist is the list of vector displacements on prismatic joints. ϕlist is the list of angles, imposed on revolute joints, and Tlist is the corresponding list of orientations of the links of revolute joints. *)

($\exists$ dlist $\in$ Vectortype-list,

ϕlist $\in$ Angle–Rep-list,

Tlist $\in$ Transformation-list)

((**len** dlist = **len** SPjn) $\wedge$

(**len** ϕlist = **len** Tlist) $\wedge$

(**len** ϕlist = **len** SRjn) $\wedge$

($\forall$ i $\in$ {1 ·· **len** SPjn})

((norm (dlist(i)) $\leq$ MAX-DISPL (SPjn(i)))) $\wedge$

($\forall$ j $\in$ {1 ·· **len** SRjn})

((ϕlist(j) $\leq$ MAX-ROTATION (SRjn(j)))) $\wedge$

```
(let T_old = orientation (POSI-ORIE (GEOMETRY
 (LINK2 (SRjn(j))))),
 T_new = orientation (POSI-ORIE
   (rotate-angle (GEOMETRY (LINK2 (SRjn(j)))
   AXIS-OF-MOVE (SRjn(j)), φlist(j))))
 Tlist(j) = change-in-orientation (T_old, T_new)) ∧
 tel ∧
(* dvec⃗ = dlvec⃗ + dφvec⃗ where
 dφvec⃗ is the net displacement due to rotations φlist. *)
(let dlvec, dφvec ∈ Vectortype,
   φvec ∈ Vectortype-list in
 (dlvec = vector-sum (dlist) ∧
 (len φvec = len Tlist) ∧
 (∀ j ∈ {1 ·· len φvec})
    (let P_old = position (POSI-ORIE (GEOMETRY
          (LINK2 (Srjn(j))))),
       P_new = position (POSI-ORIE
          (rotate-angle (GEOMETRY (LINK2 (SRjn(j)))
             AXIS-OF-MOVE (SRjn(j)), φlist(j))))
       φvec(j) = vector (P_old, P_new)) ∧
      tel
   (dφvec = vector-sum (φvec)) ∧
   (let dvec_B = const-vec-frame (dvec, base-coord),
      dl_φ_B = const-vec-frame
          (vector-sum (dlvec, dφvec), base-coord) in
   vector-equal (dvec_B, dl_φ_B)
    tel)
   tel) ∧
 (∃ Xlist, Ylist, Zlist ∈ Real-list)
  ((len Xlist = len Tlist) ∧
   (len Ylist = len Tlist) ∧
```

(**len** Zlist = **len** Tlist) $\wedge$
($\forall$ j $\in$ {1 $\cdot\cdot$ **len** Tlist})
((Xlist(j) = X-component (Tlist(j), base-coord)) $\wedge$
(Ylist(j) = Y-component (Tlist(j), base-coord)) $\wedge$
(Zlist(j) = Z-component (Tlist(j), base-coord)) $\wedge$
(Real-sum (Xlist) = T_x) $\wedge$
(Real-sum (Ylist) = T_y) $\wedge$
(Real-sum (Zlist) = T_z)
)
)
)
)
tel
)
)
tel)
)
tel
Post
let jns = JOINTS (robot-arm),
grip = GRIPPER (robot-arm),
wrs = WRIST (grip) **in**
($\exists$ Pjn, Rjn $\in$ Jointtype-list)
(($\forall$ i $\in$ {1 $\cdot\cdot$ **len** Pjn}) (Pjn(i) $\in$ Prisjoint) $\wedge$
($\forall$ j $\in$ {1 $\cdot\cdot$ **len** Rjn}) (Rjn(j) $\in$ Revoljoint) $\wedge$
((**elems** Rjn $\cup$ **elems** Rjn) = **elems** jns) $\wedge$
($\exists$ dlist $\in$ Dist–Rep-list,
ϕlist $\in$ Angle–Rep-list)
((**len** dlist = **len** Pjn) $\wedge$
(**len** ϕlist = **len** Rjn) $\wedge$
($\forall$ i $\in$ {1 $\cdot\cdot$ **len** Pjn})

```
    (translate-link (LINKID (LINK2 (Pjn(i))), dlist(i))) ∧
   (∀ j ∈ {1 ·· len Rjn})
    (rotate-link (LINKID (LINK2 (Rjn(j))), φlist(j)))
   )
 ) ⇔ POSI-ORIE (wrs)' = destination
tel
```

4.4 Remarks on the Current Work

The additive properties for translation and rotation can be used to advantage in obtaining efficient solutions for inverse kinematics problems. The existence of feasible solutions is constrained by the geometric structure of the robot arm. For example, two translations along two different joint axes can be replaced by a single translation if and only if there exists a prismatic joint whose axis coincides with the resultant axis of these two translations. Similar remarks apply to revolute joints for combining rotations. Optimization schemes based on the additive properties must be investigated at the implementation level.

We remarked earlier that an agent represents a class of real-life robots. For a given application, there usually exists a set of trajectories constrained by the environment. Using the framework outlined in this chapter, we can compare the performances of robots having specifications that are consistent with the agent's specification and which can best accomplish the given task along the given set of trajectories. Consequently, from the specifications on rigid solids, joints and primitive operations, robots may be generated automatically and the specification for forward and inverse kinematic operations can be the basis for a formal verification of the software implementing kinematics. Forces, cognitive mechanisms, sensors and control aspects can be specified independently and then combined for a particular class of application.

For simplicity at the logical level, we have considered joints having only one degree of freedom. This is consistent with the mechanical design considerations being practised [Cra89]. It is only rarely that the joints with n degrees of freedom are used in practice. Such joints can be accommodated in our formalism by letting n joints, each with one degree of freedom, connecting links of negligible length. That is, the structural integrity of a joint with n degrees of freedom has been modeled without loss of generality in our formalism. We also remark that the behavior of the controller for the joint with n degrees of freedom is also faithfully captured in our formalism. A Controller that activates only one degree of freedom at a time corresponds to the interleaved specification model whereas the controller which activates more than one degree of freedom at a time corresponds to the concurrent model.

Chapter 5

Formal Definition of Assembly

In this chapter, we define abstract objects and their surface characteristics and use them in defining the notion of assembly. We start describing previous work done in this area and show how our approach is different from others. In all our discussions, we refer to assembly of mechanical parts as opposed to other types of assemblies such as assembly of electronic components. For simplicity, we restrict ourselves to assembly requiring no tools such as hammer and screw driver for the operation. However, the specifications can be extended to include these situations once the tools are modeled.

5.1 Previous Work

Assembly was thought of as a pure mechanical operation until late 70's. With the introduction of Computer Science principles into engineering applications these tasks became inter-disciplinary. With robots brought into several engineering tasks, the development of robotic software for each application became a crucial requirement for automation. For example, robots were placed in hazardous environments where human interaction was tedious and so research turned towards developing control software for those particular robots. In a similar way, assembly of mechanical parts was handled by only humans in the traditional approach. When robots took their place in assembly, researchers concentrated on developing task level robot programming languages pertinent to assembly tasks. Among the notable ones in this area are RAPT [PAB78], VAL and AUTOPASS [LiW77]. RAPT is the only language on which experimentation was continued after its initial design; no further improvements have been reported in the literature so far about VAL and AUTOPASS. Popplestone and his colleagues [LiP89, PAB80, PWL88, PGL89] have extensively reported on developing a complete working system for assembly of mechanical parts. Recently, Thomas [ThC88, ThC89] and others also looked into the same problem. Both Popplestone and Thomas used group theory concepts for defining the relationships between the objects being assembled. Our approach pursued in this thesis is different from them in that the notion of shape operators is introduced in defining assembly relationships.

Another interesting problem attempted by several researchers in the area of mechanical parts assembly is deriving the sequence of assembly operations, given the shape of the objects to be assembled. Homem de Mello and Sanderson [HoS88, HoS89] have demonstrated the use of AND/OR graph in deriving and representing the assembly sequences. Others [BAL91, HuL89, HuL90, Wol89] have followed a slightly different approach using precedence knowledge. In all these reports, the task was to devise a cost effective method to derive a valid sequence of assembly operations. However, they do not address the issues of defining or characterizing the geometric relationships between two objects being assembled.

In this thesis, objects, their surface characteristics and the notion of assembly as a relationship between the surface characteristics of the objects, are abstractly defined. We state and prove theorems which help in verifying the assembly of a given set of objects. In the next section, we give the definition of the shape operator and show the computation of shape operators for primitive surface characteristics. Subsequent sections define the concept of assembly based on shape operators of objects.

5.2 Abstract Definition of an Object

In a typical assembly environment, a solid modeler is used for the representation of the objects. The choice of representation techniques depend on the application and the versatility of the representation technique selected. However, at the abstract level, the description should be independent of the representation in the design/implementation phase and at the same time describe the properties of intended operations on the objects. With this view, we model abstract objects in terms of their surface characteristics and describe assembly operations as a set of relations on the surface characteristics. The primitive entity by which an abstract object is described is called a *feature.*

5.2.1 Feature of an Object

A *feature* is a visible patch on the surface of an object. Conversely, everything that is visible on the surface of an object is associated with a feature of that object. Below we define features more abstractly and state the conditions under which a given set of features constitute a closed boundary of an object.

The abstract definition of a feature requires the definitions of two other terms : 'shape operator' and 'directional derivative'. They are defined next.

Definition 1 *The shape operator of a surface M is a vector valued function defined at every point p on M along every tangent vector $\overrightarrow{v}$ at p to M.*

The shape operator of a surface M at a point p along a tangent vector $\overrightarrow{v}$ is denoted by $S_{p,\vec{v}}(M)$. This is the same as the *directional derivative* of the outward unit normal **U** at p, as p moves along $\overrightarrow{v}$.

Definition 2 *The directional derivative of a vector field U at a point p along a tangent vector $\overrightarrow{v}$ is defined to be the initial velocity of the curve $t \rightarrow U(p+tv)$.*

The directional derivative is denoted by $\nabla_{\vec{v}}U(p)$ and hence $\nabla_{\vec{v}}U(p) = U'(p+tv)_{t=0}$. A theory of shape operators and directional derivatives can be found in [Bar66].

Using the definitions of shape operator and directional derivative, a more rigorous definition of feature can be given as follows :

Definition 3 *A feature f is a surface for which the shape operator defined at every point on f satisfies the following conditions :*

1. *For every point p on f AND for every tangent vector $\overrightarrow{v}$ at p, $||S_{p,\vec{v}}(f)||$ is finite.*
2. *For any two points p and q on f AND a given direction $\overrightarrow{v}$, $S_{p,\vec{v}_1}(f) = S_{q,\vec{v}_2}(f)$, where $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are tangent vectors at p and q respectively, parallel to $\overrightarrow{v}$.*
3. *For a given point p on f, and for every tangent vector $\overrightarrow{v}$ at p, $S_{p,\vec{v}}(f)$ has a continuous derivative.*

Our concept of features is somewhat similar to that used in pattern recognition and vision systems. A feature in pattern recognition systems belongs to one of the two sets *stored features* and *extracted features*. Stored features constitute the model of the object being recognized while the extracted features are taken from the image of the same object. The primary goal in pattern recognition is to find a match between subsets of these two sets of features. Accordingly, the definition of feature is restricted by the information that can be extracted from the image of an object. The extracted features must ensure that the recognition process is view-independent, size invariant and orientation invariant. In [Pau88], the features are selected in such a way that they can be easily extracted or computed from the information in the image of an object and the shape of the object can be reconstructed from these features. Consequently, surfaces are treated as composite features and are defined by shape operators. These shape operators are uniquely determined from the *first* and *second fundamental forms* of surfaces which can be easily computed from the data in a range image of an object. It is therefore clear that the shape operator is a well defined mathematical property of a surface.

We do not deal with the representation or computation of shape operators; rather we use them to define features and assembly of objects based on features. In contrast, the purpose of shape operators in vision and pattern recognition systems is to define features which are used to recognize objects. They represent shape operators by means of fundamental forms of surfaces and compute these forms from range data.

5.2.2 Primitive and Composite Features

The features are divided into two categories : *primitive* and *composite*. Primitive features are in fact the features of primitive solids. We consider the following primitive solids in our discussion :

Primitive solids = {rectangular cube, cylinder, cone }

These primitive solids are sufficient to model a number of objects in the real world. Consequently, the set of primitive features are the following :

Primitive features = {rectangular face, circular face,
lateral surface of a cylinder, lateral surface of a cone}

Hereafter, we use the terms 'cylindrical surface' and 'conical surface' to refer to the lateral surfaces of a cylinder and a cone respectively.

Next we show the derivation of shape operators for the primitive features along their principal tangent vectors at every point p. The principal tangent vectors are any two orthogonal vectors in the tangent plane at p such that any other tangent vector at p can be expressed as a linear combination of the two principal tangent vectors. In deriving the shape operator for a feature, we use the following steps :

1. Define the equation for the feature F(x,y,z).
2. Obtain the equation for the normal vector fields $C_1U_1 + C_2U_2 + C_3U_3$ where C_1, C_2 and C_3 are the normalized coefficients obtained from the partial derivatives $\frac{\partial F}{\partial x}$, $\frac{\partial F}{\partial y}$ and $\frac{\partial F}{\partial z}$ respectively.
3. Choose a point $P(x_1, y_1, z_1)$ on the feature.
4. Derive the equation for the principal tangent vectors at P. These tangent vectors are expressed in the from $\overrightarrow{v}$: (ℓ, m, n) where ℓ, m and n are the direction numbers of $\overrightarrow{v}$.
5. The normal U_p at the point P and the principal tangent vector v_p at P are orthogonal and hence their dot product is zero. This condition is to be checked for both the principal tangent vectors.
6. Obtain the expression (p+tv) as $(x_1+t\ell, y_1+tm, z_1+tn)$.
7. Substitute (p+tv) in U(p+tv).
8. Differentiate U(p+tv) with respect to t and obtain the expression $U'_{t=0}$. This is the shape operator $S_{p,\vec{v}} = \nabla_{\vec{v}}U$.

Planar Primitives

Both the rectangle and the circle are planar primitives and hence the shape operator at every point on a rectangular surface is the same as the shape operator at every point on a circular surface.

The equation for a plane Π is $ax + by + cz + d = 0$.

The equation for the normal vector field is $aU_1 + bU_2 + cU_3 = 0$.
The normal vector field has constant coordinates and hence for any tangent vector at any point on the plane, the derivative is always zero. Hence $S_{p,\vec{v}}(\Pi) = 0$.

Cylindrical Surface

The equation for an infinite cylinder is $x^2 + y^2 = r^2$ where r is the radius of the circle. This equation assumes that the origin of the coordinate frame coincides with the centre of one of the circles of the cylinder and the axis of the cylinder is the Z-axis of the coordinate frame. A finite cylinder thus can be described with an additional constraint $0 \leq z \leq \ell$, where ℓ is the length of the cylinder. This additional constraint does not have any effect on the process of deriving the shape operator as shown below and hence the process for an infinite cylinder is the same for finite cylinder.

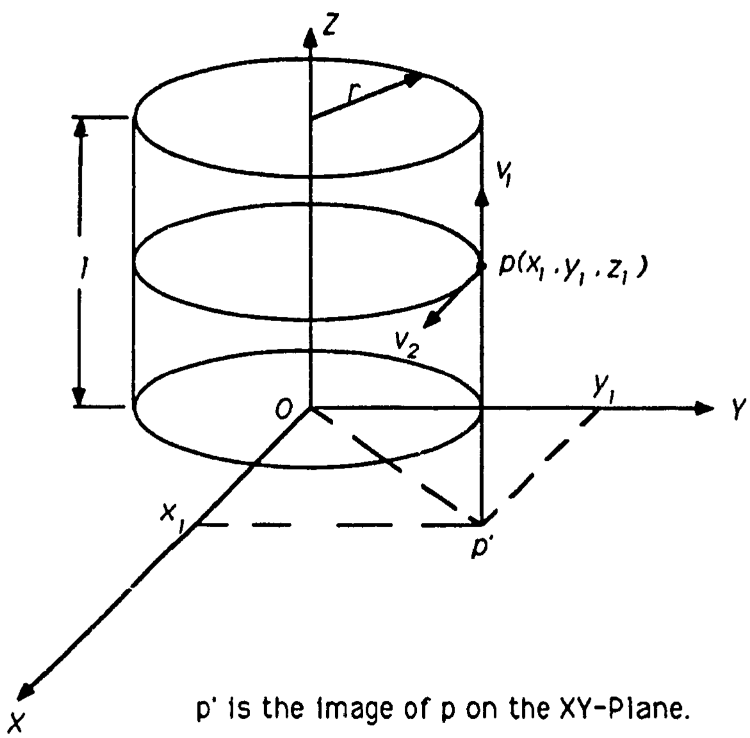


Figure 5.1: A finite Cylinder.

Equation for the cylinder : $x^2 + y^2 = r^2$.
Coefficients of normal vector field **U** derived from the partial derivatives : (2x, 2y, 0).
Normalized coefficients : $(\frac{x}{\sqrt{x^2+y^2}}, \frac{y}{\sqrt{x^2+y^2}}, 0) = (\frac{x}{r}, \frac{y}{r}, 0)$.

Equation for $\mathbf{U}$: $\mathbf{U} = \frac{x}{r}U_1 + \frac{y}{r}U_2$.

The principal tangent vector v_1 in Figure 5.1 coincides with the ruling line of the cylinder (a line parallel to the axis and is on the cylindrical surface) containing the point $P(x_1, y_1, z_1)$. Thus v_1 is parallel to z-axis and hence has the direction numbers $v_1 : (0, 0, 1)$.

It is obvious that the normal at P and v_1 are orthogonal to each other.

Expression $(p+tv) : (x_1, y_1, z_1+t)$

$U(p+tv) = \frac{x_1}{r}U_1 + \frac{y_1}{r}U_2$.

This is independent of t and hence the derivative of $\mathbf{U}$ is zero.

i.e., $U'_{t=0} = 0$.

Therefore $S_{p,\vec{v_1}}(Cyl) = 0$.

The principal tangent vector v_2 in Figure 5.1 is parallel to the XY-plane.

Let $(\ell, m, 0)$ be the direction numbers of v_2.

Clearly, $\ell^2 + m^2 = 1$.

v_2 is perpendicular to the radius of the circle at P and hence the dot product of the radius line and the tangent vector v_2 must be equal to zero.

The direction numbers of the projected line of the radius (see the figure 5.1) on the XY-plane is $(x_1, y_1, 0)$ which when normalized, becomes $(\frac{x_1}{r}, \frac{y_1}{r}, 0)$.

$$\ell\frac{x_1}{r} + m\,\frac{y_1}{r} = 0.$$

$$= \ell x_1 + m y_1 = 0$$

$$\Rightarrow \frac{\ell}{y_1} = -\,\frac{m}{x_1} = k, \text{ say.}$$

$$\Rightarrow \ell = k y_1 \text{ and } m = -\,k x_1.$$

$$\text{But } \ell^2 + m^2 = 1 \Rightarrow k^2\,(x_1^2 + y_1^2) = k^2 r^2 = 1.$$

$$\Rightarrow k = \frac{1}{r}.$$

Hence $\quad \ell = \frac{y_1}{r}$ and $m = -\,\frac{x_1}{r}$.

Thus $v_2 : (\frac{y_1}{r}, -\frac{x_1}{r}, 0)$.

<u>Check</u> : A simple calculation will show that v_2 and the normal at P are orthogonal to each other.

Expression $(p+tv) : (x_1+\frac{ty_1}{r}, y_1 - \frac{tx_1}{r}, z_1)$.

$U(p+tv) = \frac{1}{r}\,((x_1+\frac{ty_1}{r})\,U_1 + (y_1 - \frac{tx_1}{r})\,U_2)$.

Hence $S_{p,\vec{v_2}}$ (Cyl) = $U'_{t=0} = \frac{y_1}{r^2} U_1 - \frac{x_1}{r^2} U_2 = \frac{1}{r} \vec{v_2}$

Conical Surface

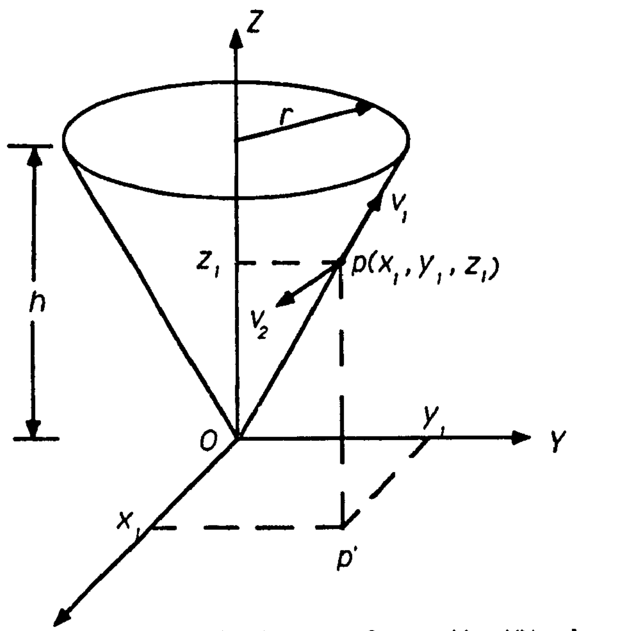


Figure 5.2: A finite Cone.

The equation for an infinite cone is $x^2 + y^2 = z^2$. However, a finite cone is constrained by its radius and height. Accordingly, the equation for a finite cone is $x^2 + y^2 = \lambda z^2$, where $\lambda = \frac{r^2}{h^2}$. These equations assume that the origin of the coordinate frame coincides at the vertex of the cone and the axis of the cone is the Z-axis.

Coefficients for the normal vector field derived from the partial derivatives :

$(2x, 2y, -2\lambda z)$.

Normalized coefficients : $(\frac{x}{\mu z}, \frac{y}{\mu z}, -\frac{\lambda}{\mu})$

$\mu = \frac{r}{h^2}\sqrt{r^2 + h^2}$

Thus $\mathbf{U} = \frac{x}{\mu z} U_1 + \frac{y}{\mu z} U_2 - \frac{\lambda}{\mu} U_3$.

The principal tangent vector v_1 in Figure 5.2 coincides with one of the slanting lines passing through the point $P(x_1, y_1, z_1)$ (a slanting line is the one, one of whose end points is the vertex of the cone and the other end point lies on the periphery of the base circle). The direction numbers of v_1 are (x_1, y_1, z_1) which when normalized becomes $(\frac{x_1}{\sqrt{1+\lambda}z_1}, \frac{y_1}{\sqrt{1+\lambda}z_1}, \frac{1}{\sqrt{1+\lambda}})$.

Expression (p+tv) : $(x_1 + \frac{tx_1}{\sqrt{1+\lambda}z_1}, y_1 + \frac{ty_1}{\sqrt{1+\lambda}z_1}, z_1 + \frac{t}{\sqrt{1+\lambda}})$.

$$U(p+tv) = \frac{x_1 + \frac{tx_1}{\sqrt{1+\lambda}z_1}}{\mu(z_1 + \frac{t}{\sqrt{1+\lambda}})} U_1 + \frac{y_1 + \frac{ty_1}{\sqrt{1+\lambda}z_1}}{\mu(z_1 + \frac{t}{\sqrt{1+\lambda}})} U_2 - \frac{\lambda}{\mu} U_3.$$

Differentiating **U**, we get,

$$U' = \frac{\mu(z_1 + \frac{t}{\sqrt{1+\lambda}})\frac{x_1}{\sqrt{1+\lambda}z_1} - (x_1 + \frac{tx_1}{\sqrt{1+\lambda}z_1})\frac{\mu}{\sqrt{1+\lambda}}}{\mu^2(z_1 + \frac{t}{\sqrt{1+\lambda}})^2} U_1 +$$

$$\frac{\mu(z_1 + \frac{t}{\sqrt{1+\lambda}})\frac{y_1}{\sqrt{1+\lambda}z_1} - (y_1 + \frac{ty_1}{\sqrt{1+\lambda}z_1})\frac{\mu}{\sqrt{1+\lambda}}}{\mu^2(z_1 + \frac{t}{\sqrt{1+\lambda}})^2} U_2$$

Thus $S_{p,\vec{v_1}}$ (Co) $= U'_{t=0} = 0$.

The other principal tangent vector v_2 in Figure 5.2 is parallel to the XY-plane. The derivation for the direction numbers of v_2 are

$v_2 : (\frac{1}{\sqrt{\lambda}}\frac{y_1}{z_1}, -\frac{1}{\sqrt{\lambda}}\frac{x_1}{z_1}, 0)$.

The derivation steps are similar to the tangent vector v_2 for the cylinder.

Expression (p+tv) : $(x_1 + \frac{ty_1}{\sqrt{\lambda}z_1}, y_1 - \frac{tx_1}{\sqrt{\lambda}z_1}, z_1)$.

$$U(p+tv) = \frac{x_1 + \frac{ty_1}{\sqrt{\lambda}z_1}}{\mu z_1} U_1 + \frac{y_1 - \frac{tx_1}{\sqrt{\lambda}z_1}}{\mu z_1} U_2 - \frac{\lambda}{\mu} U_3.$$

$$S_{p,\vec{v_2}} (Co) = U'_{t=0} = \frac{y_1}{\mu\sqrt{\lambda}z_1^2} U_1 - \frac{x_1}{\mu\sqrt{\lambda}z_1^2} U_2.$$

$$= \frac{1}{\mu z_1} \vec{v_2}.$$

Using the equation for cone and the expression for μ and λ, the magnitude of the shape operator becomes

$$S_{p,\vec{v_2}} (Co) = \frac{\sqrt{\lambda}}{\mu} (\frac{1}{\sqrt{x_1^2 + y_1^2}})$$

$$= \frac{\sqrt{\lambda}}{\mu} (\frac{1}{radius\ of\ circle\ at\ P}). \qquad \ldots(1)$$

$$= \frac{1}{\mu z_1} \qquad \ldots(2)$$

Expression (1) shows that the shape operator depends on the radius of the circle at P. Expression (2) indicates that the shape operator depends on the height of the partial cone at P.

5.2.3 Specification for Primitive and Composite Features

Though the mathematical definition of a feature is sound, the notations used to describe a feature and its properties are not conventional to the software community which rely on simple programming language-like notations. Hence the mathematical definition of features are to be restated in the notation of the specification language (in our case, VDM) in order to derive a valid design as well as to ensure its correctness. A formal definition of a primitive feature in VDM is given below :

Primitive-feature	=	Rectangle | Circle | Cylindrical-surface | Conical-surface
Rectangle	::	LENGTH : Line-segment BREADTH : Line-segment
Circle	::	CENTRE : Point RADIUS : Nat0 PLANE-OF-CIRCLE : Plane
Cylindrical-surface	::	CIRCLE1 : Circle CIRCLE2 : Circle
Conical-surface	::	BASE-CIRCLE : Circle VERTEX : Point

The invariants for the primitive features are given below. A more detailed treatment of these primitives can be found on [PAB90].

<u>Invariants</u> :

inv-Rectangle (Rec) $\triangleq$

(END-POINT1 (LENGTH (Rec)) = END-POINT1 (BREADTH (Rec))) $\wedge$

(perpendicular-line-segments (LENGTH (Rec), BREADTH (Rec)))

inv-Circle (Cir) $\triangleq$ pt-on-plane (CENTRE(Cir), PLANE-OF-CIRCLE(Cir))

inv-Cylinder (Cyl) $\triangleq$

let Pl_1 = PLANE-OF-CIRCLE (CIRCLE1 (Cyl)),

Pl_2 = PLANE-OF-CIRCLE (CIRCLE2 (Cyl)),

axis = *mk_Line-segment* (CENTRE (CIRCLE1 (Cyl)),

CENTRE (CIRCLE2 (Cyl))) **in**

($\sim$ line-on-plane (axis, Pl_1)) $\wedge$

($\sim$ line-on-plane (axis, Pl_2))

tel

inv-Cone (Co) $\triangleq$

let axis = *mk_Line-segment* (VERTEX (Co), CENTRE (BASE-CIRCLE (Co))) **in**

$\sim$ line-on-plane (axis, PLANE-OF-CIRCLE (BASE-CIRCLE (Co)))

tel

The specification for shape operator, based on its definition stated earlier, is given below :

Shape-operator : Feature $\times$ Point $\times$ Vector $\rightarrow$ Vector.

pre-Shape-operator (f, p, v) $\triangleq$ tangent-vector (f, p, v)

post-Shape-operator (f, p, v, S) $\triangleq$

(norm (S) $< \infty$) $\wedge$

($\forall$ v_1, v_2 $\in$ Vector, q, r $\in$ Point)

((tangent-vector (f, q, v_1)) $\wedge$

(tangent-vector (f, r, v_2)) $\wedge$

(parallel (v_1, v)) $\wedge$ (parallel (v_2, v)) $\Rightarrow$

shape-operator (f, q, v_1) = shape-operator (f, r, v_2)

) $\wedge$

($\forall\, \delta > 0$) ($\exists\, \lambda > 0$)

(norm (diff (shape-operator (f, p, v) –

diff (shape-operator (f, p+δ, v))) < λ)

In the above specification, the function 'diff' represents conventional differentiation and is not further defined here. Any further description of shape operator requires procedural details and these can be addressed on the design.

Composite features are obtained from primitive features or already constructed composite features using the functions 'Funion', 'Finter' and 'Fdiff'. These three functions respectively denote the union, intersection and difference operations among the features. In addition to the two input features f_1 and f_2, two additional parameters $area_1$ and $area_2$ representing the overlapping area between f_1 and f_2 must also be input. Areas are represented as point sets in the abstract level.

A composite feature can also be obtained from another feature f by chopping a portion of f. It is to be noted that the chopping plane and its positive normal should be given as arguments to the 'chopping' function along with the feature to be chopped in order to decide which portion of the original feature is to be retained after chopping.

A formal definition of composite feature obtained using these functions is given below. In all these specifications, we assert that every point in the resulting feature belongs to one or both of its constituents and the shape operator is also consistent at that point :

Funion : Feature × Point–set × Feature × Point–set → Composite-feature

pre-Funion (f_1, $pset_1$, f_2, $pset_2$) $\triangleq$

(* $pset_1$ and $pset_2$ refer to the areas on f_1 and f_2 respectively. *)

  $(\forall\ p_1 \in pset_1)\ (on\ (p_1, f_1)) \wedge$

  $(\forall\ p_2 \in pset_2)\ (on\ (p_2, f_2)) \wedge$

  $(\mathbf{card}\ pset_1 = \mathbf{card}\ pset_2)$

post-Funion (f_1, $pset_1$, f_2, $pset_2$, f) $\triangleq$

(* $pset_1$ and $pset_2$ are merged together. *)

  $(\forall\ p_1 \in pset_1)\ ((p_1 \in pset_2) \wedge on\ (p_1, f_1) \wedge (on\ (p_1, f_2)) \wedge$

  $(\forall\ p_2 \in pset_2)\ ((p_2 \in pset_1) \wedge on\ (p_2, f_1) \wedge (on\ (p_2, f_2)) \wedge$

  (* every point p on the resulting feature f is either a point on f_1 or a point on f_2 *)

  $(\forall\ p \in Point)$

    $(on\ (p, f) \Rightarrow$

      $(on\ (p, f_1) \vee on\ (p, f_2)) \wedge$

      (* every tangent vector v to every point p on the resulting feature f is also a tangent vector to f_1 at p or a tangent vector to f_2 at p and shape operator at p along v is preserved from the input feature. *)

$(\exists\ v_1, v_2 \in \text{Vector})$

$((\forall\ v \in \text{Vector})$

$(\text{tangent-vector}\ (f, p, v) \Rightarrow (\exists\ a, b \in \text{Nat})\ (v = av_1 + bv_2))$

$(\text{tangent-vector}\ (f, p, v_1) \wedge \text{tangent-vector}\ (f, p, v_2)) \wedge$

$(\text{on}\ (p, f_1) \Rightarrow$

$(\text{tangent-vector}\ (f_1, p, v_1)) \wedge$

$(\text{shape-operator}\ (f, p, v_1) = \text{shape-operator}\ (f_1, p, v_1)) \wedge$

$(\text{tangent-vector}\ (f_1, p, v_2)) \wedge$

$(\text{shape-operator}\ (f, p, v_2) = \text{shape-operator}\ (f_1, p, v_2))$

$) \wedge$

$(\text{on}\ (p, f_2) \Rightarrow$

$(\text{tangent-vector}\ (f_2, p, v_1)) \wedge$

$(\text{shape-operator}\ (f, p, v_1) = \text{shape-operator}\ (f_2, p, v_1)) \wedge$

$(\text{tangent-vector}\ (f_2, p, v_2)) \wedge$

$(\text{shape-operator}\ (f, p, v_2) = \text{shape-operator}\ (f_2, p, v_2))$

)

)

)

Finter : Feature $\times$ Point–set $\times$ Feature $\times$ Point–set $\rightarrow$ Composite-feature

pre-Finter $(f_1, pset_1, f_2, pset_2)$ $\triangleq$

$(\forall\ p_1 \in pset_1)\ (\text{on}\ (p_1, f_1)) \wedge$

$(\forall\ p_2 \in pset_2)\ (\text{on}\ (p_2, f_2)) \wedge$

$(\textbf{card}\ pset_1 = \textbf{card}\ pset_2)$

post-Finter $(f_1, pset_1, f_2, pset_2, f)$ $\triangleq$

$(\forall\ p_1 \in pset_1)\ ((p_1 \in pset_2) \wedge \text{on}\ (p_1, f_1) \wedge (\text{on}\ (p_1, f_2)) \wedge$

$(\forall\ p_2 \in pset_2)\ ((p_2 \in pset_1) \wedge \text{on}\ (p_2, f_1) \wedge (\text{on}\ (p_2, f_2)) \wedge$

$(\forall\ p \in \text{Point})$

$(\text{on}\ (p, f) \Rightarrow$

$(\text{on}\ (p, f_1) \wedge \text{on}\ (p, f_2)) \wedge$

$(\exists\ v_1, v_2 \in \text{Vector})$

$((\forall\ v \in \text{Vector})$

$(\text{tangent-vector } (f, p, v) \Rightarrow (\exists\, a, b \in \text{Nat})\ (v = av_1 + bv_2)) \wedge$

$(\text{tangent-vector } (f, p, v_1)) \wedge$

$(\text{tangent-vector } (f_1, p, v_1)) \wedge$

$(\text{tangent-vector } (f_2, p, v_1)) \wedge$

$(\text{shape-operator } (f, p, v_1) = \text{shape-operator } (f_1, p, v_1)) \wedge$

$(\text{shape-operator } (f, p, v_1) = \text{shape-operator } (f_2, p, v_1)) \wedge$

$(\text{tangent-vector } (f, p, v_2)) \wedge$

$(\text{tangent-vector } (f_1, p, v_2)) \wedge$

$(\text{tangent-vector } (f_2, p, v_2)) \wedge$

$(\text{shape-operator } (f, p, v_2) = \text{shape-operator } (f_1, p, v_2)) \wedge$

$(\text{shape-operator } (f, p, v_2) = \text{shape-operator } (f_2, p, v_2))$

)

)

Fdiff : Feature $\times$ Point–set $\times$ Feature $\times$ Point–set $\longrightarrow$ Composite-feature

pre-Fdiff $(f_1, pset_1, f_2, pset_2) \quad \triangleq$

$(\forall\, p_1 \in pset_1)\ (\text{on } (p_1, f_1)) \wedge$

$(\forall\, p_2 \in pset_2)\ (\text{on } (p_2, f_2)) \wedge$

$(\textbf{card } pset_1 = \textbf{card } pset_2)$

post-Fdiff $(f_1, pset_1, f_2, pset_2, f) \quad \triangleq$

$(\forall\, p \in \text{Point})$

$(\text{on } (p, f) \Rightarrow$

$(\text{on } (p, f_1) \wedge \sim (p \in pset_1) \wedge \sim \text{on } (p, f_2)) \wedge$

$(\exists\, v_1, v_2 \in \text{Vector})$

$((\forall\, v \in \text{Vector})$

$(\text{tangent-vector } (f, p, v) \Rightarrow (\exists\, a, b \in \text{Nat})\ (v = av_1 + bv_2)) \wedge$

$(\text{tangent-vector } (f, p, v_1)) \wedge$

$(\text{tangent-vector } (f_1, p, v_1)) \wedge$

$(\text{shape-operator } (f, p, v_1) = \text{shape-operator } (f_1, p, v_1)) \wedge$

$(\text{tangent-vector } (f, p, v_2)) \wedge$

$(\text{tangent-vector } (f_1, p, v_2)) \wedge$

$(\text{shape-operator } (f, p, v_2) = \text{shape-operator } (f_1, p, v_2))$

)

)

Composite features may also be obtained by cutting a feature by a plane. We use the term 'chopping' in a very strict sense such that the chopping plane cuts the feature into exactly two halves. This condition eliminates the situations where the chopping plane touches the feature or stays away from the feature. The function 'chopping' is informally defined as follows :

> Let f_i denote the feature to be chopped, Pl the chopping plane and N the positive normal of the plane Pl. Let f_o denote the feature obtained after chopping. There exist two points p and q on the periphery of f_i such that for any two points p_1 and q_1 on the periphery of f_i, the line pq is always longer then the line p_1q_1. In some sense, the line pq denotes the major axis of the feature f_i. In addition, p and q should be on the opposite sides of the chopping plane Pl and their respective distances from the plane are not zero. After chopping, for every point r on f_o, the vector $\overrightarrow{rs}$ points to the same direction as that of the positive normal N to the chopping plane Pl, where s refers to the foot of the perpendicular line from r to the plane Pl.

Summarizing these informal conditions, the formal definition of the function 'chopping' is obtained as follows :

Chopping : Feature $\times$ Point $\times$ Point $\times$ Plane $\times$ Vector $\rightarrow$ Composite-feature

pre-Chopping (f_i, p, q, Pl, N) $\triangleq$

($\forall$ p_1, q_1 $\in$ Point)

(distance (p, q) $\geq$ distance (p_1. q_1)) $\wedge$

(**let** r = intersect-line-plane (const-line (p, q), Pl) **in**

(r $\neq$ p) $\wedge$ (r $\neq$ q) $\wedge$

(distance (r, p) $\neq$ 0) $\wedge$ (distance (r, q) $\neq$ 0)

tel) $\wedge$

($\forall$ ℓ $\in$ Line-segment) (line-on-plane (ℓ, Pl) $\Rightarrow$ perpendicular (ℓ, N)) *post*-Chopping (f_i, p, q, Pl, N, f_o) $\triangleq$

(∀ p ∈ Point)

(on (p, f_o) ⇒

(on (p, f_i) ∧

let ℓ = normal-pt-plane (p, Pl) **in**

let q = intersect-line-plane (ℓ, Pl) **in**

let v = vector (q, p) **in**

same-direction (v, N)

tel

tel

tel

) ∧

(∀ v_1 ∈ Vector)

((tangent-vector (f_o, p, v_1) ⇒ tangent-vector (f_i, p, v_1)) ∧

(shape-operator (f_o, p, v_1) = shape-operator (f_i, p, v_1))

)

)

Tangent-vector : Feature × Point × Vector ⟶ Boolean

pre-Tangent-vector (f, p, v) $\triangleq$ on (p, f) ∧ pt-on-vector (p, v)

post-Tangent-vector (f, p, v, b) $\triangleq$

∼ (∃ q ∈ Point) (on (q, f) ∧ pt-on-vector (q,v)

Lemma 1 *A composite feature is a feature.*

Proof : In order to prove that a composite feature f is a feature, we have to prove that the shape operator at every point p on f is defined and satisfies the conditions for the shape operator as defined earlier. In addition, if the point p is common to both the input features (in case of funion and finter), then we need to show that the shape operator to f_1 and f_2 at p along every tangent vector v must be the same (consistent). We will prove the lemma for each one of the operators funion, finter, fdiff and Chopping separately.

Case 1 : in function 'funion'

Every point p on f is either a point on f_1 (exclusively) or a point on f_2 (exclusively) or a point on both f_1 and f_2.

Case 1.1 : $(p \in f_1) \wedge (p \notin f_2)$

As mentioned in post-funion, every tangent vector v to f at p is also a tangent vector to f_1 at p. Moreover, the shape operator (f, v, p) is the same as the shape operator (f_1, v, p). Hence the shape operator at p is well defined.

Case 1.2 : $(p \notin f_1) \wedge (p \in f_2)$

The arguments are similar to that of Case 1.1.

Case 1.3 : $(p \in f_1) \wedge (p \in f_2)$

In this case, as seen from post-funion, every tangent vector to f at p is also a tangent vector to f_1 at p as well as a tangent vector to f_2 at p. In addition, shape-operator (f, v, p)

= shape-operator (f_1, v, p)

= shape-operator (f_2, v, p).

The three-way equality constraint implies that the shape of f_1 and f_2 is the same at every common point p on f. Hence the lemma holds good.

Case 2 : in function 'finter'

As seen from post-finter, every point p on f is a point on f_1 as well as a point on f_2. The rest of the proof then follows from Case 1.3.

Case 3 : in function 'fdiff'

From post-fdiff, it can be observed that every point on f is only a point on f_1 and every tangent vector v to f at p is also a tangent vector to f_1 at p. Moreover, the shape operator at p for f is also the same as the shape operator to f_1 at p along v. Therefore, it is easy to see that the shape operator of f at every point p is the same as the shape operator of f_1 at p, which in turn is well defined.

Case 4 : Chopping

The arguments are very similar to that of Case 3.

The following observations can be made regarding the composite features :

- a composite feature cannot be constructed using a flat feature (rectangular or circular) and a curved feature (cylindrical or conical), since the shape operator will not have a continuous derivative along every tangent vector at the meeting point between the features.
- a cylindrical feature cannot be joined with a conical feature to form a composite feature for the same reason mentioned above.
- a cylindrical feature can be joined with another cylindrical feature to form a composite feature only if they are of same dimensions and their axes aligned.
- a conical feature can be joined with another conical feature to form a composite feature only if both of them have the same angle at the vertex and their vertices coincide.

Lemma 2 *With respect to composition, the primitive features are partitioned into three disjoint subsets {{rectangular face, circular face}, {cylindrical surface}, {conical surface}}.*

Proof : It is already proved that a composite feature is a feature. Hence the shape operator at every point on the composite feature along every tangent vector is uniquely determined. It is therefore sufficient to prove that each member of one partition cannot produce a composite feature with a member from another partition.

Let C be the composite feature produced from the two features f_1 and f_2.

Case 1 : f_1 is rectangular and f_2 is cylindrical.

For rectangular features, the shape operator at any point along every tangent vector at that point is 0. However, the shape operator at a point p on the cylindrical feature is 0 only along the tangent vectors that are parallel to the axis of the cylinder; the shape operator for the same point p along the tangent vector $\overrightarrow{v}$ perpendicular to the axis is $\frac{1}{r}\overrightarrow{v}$. For any other tangent vector which is a linear combination of these two tangent vectors, the shape operator is definitely not 0 since there is at least one non-zero result. Hence the shape operator for the same point on a rectangular face is different from that for the same point on a cylindrical surface. Hence C cannot be obtained from f_1 and f_2.

Case 2 : f_1 is rectangular and f_2 is conical.

The arguments of Case 1 are equally applicable to Case 2 as well.

Case 3 : f_1 is cylindrical and f_2 is conical.

In this case, there is no value for the shape operator common to both cylindrical and conical surfaces. Obviously C cannot be composed from cylindrical and conical surfaces.

It is also to be observed that composite features obtained by chopping will not modify the shape of the original feature and hence the shape operator at every point on a composite feature obtained by chopping is the same as that of the original feature at that point.

5.2.4 Normal to a Feature

Associated with each feature is a *normal* which always points to the exterior of the object. The normal to a feature f at a point p on f is the same as the normal to the tangent plane to f at p. Consequently, the normal to a planar feature f_p at every point p is perpendicular to every line on f_p passing through p. For cylindrical feature f_{cl}, the normal at any point p is perpendicular to the axis of the cylinder and for conical feature f_{co}, the normal at any point p is perpendicular to a line ℓ passing through p and making an angle with the axis equal to half of the conical angle. A formal definition of axis of a cylinder, axis of a cone and angle of a cone are given in [PAB90].

Normal : Feature $\times$ Point $\rightarrow$ Vector

pre-Normal (f, p) $\triangleq$ on (p, f)

post-Normal (f, p, N) $\triangleq$

(f $\in$ Rectangle $\Rightarrow$

```
    let Pl = derive-plane-pts (END-POINT1 (LENGTH (f)),
                END-POINT2 (LENGTH(f)), END-POINT2 (BREADTH (f))) in
      perpendicular-line-plane (N, Pl)
    tel
) ∧
(f ∈ Circle ⇒ perpendicular-line-plane (N, PLANE-OF-CIRCLE (f))) ∧
(f ∈ Cylindrical-surface ⇒
    let axis = axis-of-cylinder (f) in
      (∃! q ∈ Point)
        ((pt-on-line (q, axis) ⊕ pt-on-line (q, extrapolate (axis))) ∧
         (* q is the foot of perpendicular from p to the axis. *)
         (same-direction (N, vector (p,q)))
        )
    tel
) ∧
(f ∈ Conical-surface ⇒
    (p ≠ VERTEX (f)) ∧
    (perpendicular-line-line (N, line (p,VERTEX(f))))
) ∧
(f ∈ Composite-feature ⇒
    (∃ f1, f2 ∈ Feature, pset1, pset2 ∈ Point-set)
      (f = funion (f1, pset1, f2, pset2) ⇒
        (on (p, f1) ⇒ N = normal (f1, p)) ∨ (on (p, f2) ⇒ N = normal (f2, p))
      ) ⊕
    (∃ f1, f2 ∈ Feature, pset1, pset2 ∈ Point-set)
      (f = finter (f1, pset1, f2, pset2) ⇒
        (on (p, f1) ∧ on (p, f2) ∧ (N = normal (f1, p)) ∧ (N = normal (f2, p))
      ) ⊕
    (∃ f2, f2 ∈ Feature, pset1, pset2 ∈ Point-set)
      (f = fdiff (f1, pset1, f2, pset2) ⇒ (on (p, f1) ∧ N = normal (f1, p))
      ) ⊕
```

$(\exists\ f_3 \in \text{Feature}, r, s \in \text{Point}, Pl \in \text{Plane}, N_1 \in \text{Vector})$

$(f = \text{chopping}\ (f_3, r, s, Pl, N_1) \Rightarrow (\text{on}\ (p, f_3) \wedge (n = \text{normal}\ (f_3, p)))$

$))$

5.2.5 Functional Description of Features

Although our choice of primitives will be sufficient to describe a large number of objects in the real world, they are still inadequate to describe some features belonging to objects that are commonly encountered in the real world. Therefore it is sometimes necessary to functionally describe a feature using the primitive features; see the example below :

Example 1 :

Intersecting Cylinders.

Consider the penetration of a cylinder A into another cylinder B as shown in Figure 5.3. For simplicity, we consider only regular cylinders and only the situation where cylinder A penetrates cylinder B such that their axes are at right angles. Let p_t be the point of intersection of the axes. The distance of p_t from the centre of CIRCLE1 of A as well as the distance of p_t from the centre of CIRCLE1 of B are also passed as input parameters in order to define the penetration. The common curve comprising the points of intersection is called a 'profile'. The profile is formally defined as a list of points. Below, we give a formal definition for 'penetration'.

Profile = Curve = Point–list

Penetration : Cylindrical-surface $\times$ Cylindrical-surface $\times$ Nat $\times$ Nat $\rightarrow$ Profile

pre-Penetration (Cyl_1, Cyl_2, d_1, d_2) $\triangleq$

let $axis_1$ = axis-of-cylinder (Cyl_1),

$axis_2$ = axis-of-cylinder (Cyl_2) **in**

(perpendicular-line-segments $(axis_1, axis_2) \wedge$

let p_t = intersect-line-segments $(axis_1, axis_2)$ **in**

$(p_t \neq \text{NIL}) \wedge$

$(d_1 = \text{distance}\ (p_t, \text{CENTRE}\ (\text{CIRCLE1}\ (Cyl_1)))) \wedge$

$(d_2 = \text{distance}\ (p_t, \text{CENTRE}\ (\text{CIRCLE1}\ (Cyl_2)))) \wedge$

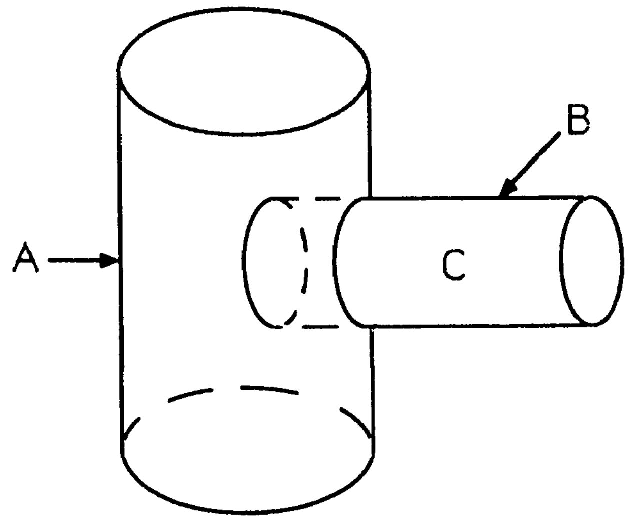


Figure 5.3: Intersecting Cylinders.

$(d_1 \neq 0) \wedge (d_2 \neq 0)$

tel

tel

post-Penetration (Cyl_1, Cyl_2, d_1, d_2, pf) $\triangleq$

(* every point on the profile is obtained as the intersection of two circles: cir_1 and cir_2, $cir_1 \in Cyl_1$ and $cir_2 \in Cyl_2$. *)

let $axis_1$ = axis-of-cylinder (Cyl_1),

$axis_2$ = axis-of-cylinder (Cyl_2) **in**

($\forall$ i $\in$ {1 ·· **len** pf})

(($\exists!$ cir_1, cir_2 $\in$ Circle)

((pt-on-line (CENTRE (cir_1), $axis_1$)) $\wedge$

(RADIUS (cir_1) = RADIUS ($CIRCLE_1$ (Cyl_1))) $\wedge$

(perpendicular-line-plane ($axis_1$, PLANE-OF-CIRCLE (cir_1)) $\wedge$

(pt-on-line (CENTRE (cir_2), $axis_2$)) $\wedge$

(RADIUS (cir_2) = RADIUS ($CIRCLE_1$ (Cyl_2))) $\wedge$

(perpendicular-line-plane ($axis_2$, PLANE-OF-CIRCLE (cir_2)) $\wedge$

(pf(i) $\in$ intersect-circles (cir_1, cir_2)) $\wedge$

(distance (pf(i), CENTRE (cir_1)) = RADIUS (cir_1)) $\wedge$

(distance (pf(i), CENTRE (cir_2)) = RADIUS (cir_2))

)

)

tel

Now, the specification for the composite feature C (portion of the cylindrical surface A only) can be given as below :

Composite-cyl-inter-1 : Cylindrical-surface $\times$ Cylindrical-surface $\rightarrow$ Feature

pre-Composite-cyl-inter-1 (Cyl_1, Cyl_2) $\triangleq$

($\exists$ d_1, d_2 $\in$ Nat)

(**let** pf = penetration (Cyl_1, Cyl_2, d_1, d_2) **in**

pf $\neq$ NIL

tel)

post-Composite-cyl-inter-1 (Cyl_1, Cyl_2, f) $\triangleq$

(* The portion C of the cylinder is composed of a set of line segments all of which are parallel to the axis of the cylinder A and are at a distance equal to the radius of CIRCLE1 of the cylinder. One end point of each line segment lies on the circumference of CIRCLE1 and the other end point lies on the profile, formed by the intersecting cylinders. *)

($\forall$ p $\in$ Point)

(on (p, f) $\Rightarrow$

((on-cyl-surface (p, Cyl_1)) $\wedge$

($\exists$ ℓ $\in$ Line-segment)

((lie-on-periphery (END-POINT1 (ℓ), CIRCLE1 (Cyl_1))) $\wedge$

(**let** pf = define-profile (Cyl_1, Cyl_2) **in**

($\exists!$ i $\in$ {1 $\cdot\cdot$ **len** pf}) (END-POINT2 (ℓ) = pf(i))

tel) $\wedge$

dist-line-line (axis-of-cylinder (Cyl_1), l) =

RADIUS (CIRCLE1 (Cyl_1))

)

)

)

It is easy to prove that Composite-cyl-inter-1 results in a composite-feature and hence a feature; the proof is similar to that of Lemma 1.

5.2.6 Closed boundary of an Object

In order to prove that an object can be uniquely determined by its features, we have to show that every feature in the set of features of the object is *completely connected* to form a closed boundary. This can be shown by first establishing the connectives (point sets) of a feature and then stating that every connective of a feature should be joined to exactly one other connective of same type, belonging to another feature of the same object. We define a connective to be a curve so that line segments, circles and circular arcs can be also be defined.

Connective = Curve

Connectives-of-feature : Feature $\rightarrow$ Connective-list

post-Connectives-of-feature (f, Cfs) $\triangleq$

(CONSTRUCT(f) $\in$ Rectangle $\Rightarrow$

(**len** Cfs = 4) $\wedge$ (Cfs $\in$ Line-segment–list) $\wedge$

(Cfs(1) = side1 (f) $\wedge$ Cfs(2) = side2 (f) $\wedge$

Cfs(3) = side3 (f) $\wedge$ Cfs(4) = side4 (f))

) $\wedge$

(CONSTRUCT(f) $\in$ Circle $\Rightarrow$

(**len** Cfs = 1) $\wedge$ (Cfs $\in$ Periphery-of-circle–list) $\wedge$

(Cfs(1) = periphery (f))

) $\wedge$

(CONSTRUCT(f) $\in$ Cylindrical-surface $\Rightarrow$

(**len** Cfs = 2) $\wedge$ (Cfs $\in$ Periphery-of-circle–list) $\wedge$

(Cfs(1) = periphery (CIRCLE1 (f))) $\wedge$ (Cfs(2) = periphery (CIRCLE2 (f)))

) $\wedge$

(CONSTRUCT(f) $\in$ Conical-surface $\Rightarrow$

(**len** Cfs = 1) $\wedge$ (Cfs $\in$ Periphery-of-circle–list) $\wedge$

(Cfs(1) = periphery (BASE-CIRCLE (f)))

$) \wedge$

$(\text{CONSTRUCT}(f) \in \text{Composite-feature} \Rightarrow \text{Cfs} = \text{compute-connectives } (f))$

As remarked earlier, composite features obtained from cylindrical features will result in a cylindrical (composite) feature; the same argument is true for conical features as well. Hence the connectives of a cylindrical composite feature are directly derived from those of the cylindrical primitive features; similarly, the connectives of a conical composite feature are also derived from its constituents. However, for composite features obtained from planar primitives or 'chopping', the determination of connectives is quite complex.

We next state the conditions to assert whether or not a given set of features form the closed boundary of an object. Informally, *every feature in the set is completely connected; i.e., every connective c of a feature f is connected to exactly one other connective c_1 of another feature f_1, leaving no connective open (unconnected).* Formally,

Closed : Object $\rightarrow$ Boolean

post-Closed (Obj, b) $\triangleq$

$$
\begin{array}{l}
b \Leftrightarrow \ (\forall\, f_1 \in \text{FEATURES (Obj)}) \\
\qquad ((\forall\, cf_1 \in \textbf{elems} \text{ connectives-of-feature } (f_1)) \\
\qquad\quad ((\exists!\ f_2 \in \text{FEATURES (Obj)}) \\
\qquad\qquad ((f_1 \neq f_2) \\
\qquad\qquad\ (\exists!\ cf_2 \in \textbf{elems} \text{ connectives-of-feature } (f_2)) \\
\qquad\qquad\quad ((\forall\, p \in \text{Point}) \\
\qquad\qquad\qquad (\text{on } (p, cf_1) \Leftrightarrow \text{on } (p, cf_2)) \\
\qquad\qquad\quad) \\
\qquad\qquad) \\
\qquad\quad) \\
\qquad)
\end{array}
$$

5.2.7 Relationship between Features of the Same Object

The normals to features play an important role in identifying the relationship between adjacent features of the same object (two features are said to be adjacent if they have a common connective). Since, in our discussion, objects are described by features, portions of an object such as a 'hole' is also defined in terms of relationship between adjacent features of the same object. Since, strictly speaking, 'hole' is a term to be defined with respect to something, we define a hole with respect the interior of the object. Our notion of hole is based on the normals to features which always point to the exterior of the object.

The curved features and flat features have distinct characteristics in composing a hole (A curved feature is a cylindrical surface, conical surface or any composite feature obtained using one of these two primitive features. A planar feature is a rectangle, circle or any composite feature obtained using one or both of these primitive features). For example, a cylindrical or conical feature can by itself represent a hole whereas a flat feature (rectangular or circular) cannot by itself represe ole.

A feature is said to 'participate' in a hole if it by itself rep nts a hole or forms part of a hole. The participation of a flat feature in forming a hole is identified from its neighbors or adjacent features.

- For a curved feature f, let Π be the plane perpendicular to the axis of the curved feature and cut the axis at a point p. If for every point q common to the feature and plane Π, the normal to f at q converges towards p, then the curved feature f participates in a hole (in this case it by itself represents the hole).

- For a flat feature f, if there is at least one adjacent feature f_1 which is curved and participates in a hole, then f also participates in a hole.

- For a flat feature f, if there is at least one adjacent feature f_1 which is flat and normals defined at any two points p and q such that $p \in f$ and $q \in f_1$ intersect, then f participates in a hole. It can be observed that if f participates in a hole having met the constraints above, then f_1 which is also flat, participates in a hole.

- In all other cases, f does not participate in a hole.

A formal description of participation of a feature in a hole is given below :

Participate-in-hole : Feature $\times$ Boolean

post-Participate-in-hole (f, b) $\quad \triangleq$

((f $\in$ Cylindrical-surface) $\vee$ (f $\in$ Conical-surface) $\Rightarrow$

b $\Leftrightarrow$ ($\exists$ Π $\in$ Plane)

((perpendicular-line-plane (axis(f), Π)) $\wedge$

(**let** p = intersect-line-plane (axis (f), Π) **in**

($\forall$ q $\in$ Point)

((on (q, f) $\wedge$ pt-on-plane (q, Π) $\Rightarrow$ vector (p,q) = normal (f, q))

tel)

)

) $\wedge$

((f $\in$ Circle) $\vee$ (f $\in$ Rectangle) $\Rightarrow$

b $\Leftrightarrow$ ($\exists$ f_1 adjacent-features (f))

(participate-in-hole (f_1)) $\vee$

($\exists$ f_1 $\in$ adjacent-features (f))

(($\exists$ p, p_1, q $\in$ Point)

(on (p, f) $\wedge$ on (p_1, f_1) $\wedge$

q = intersect (normal (f, p), normal (f_1, p_1))

)

)

) $\wedge$

(f $\in$ Composite-feature $\Rightarrow$

(($\exists$! f_1, f_2 $\in$ Feature, $pset_1$, $pset_2$ $\in$ Point–set)

(f = funion (f_1, $pset_1$, f_2, $pset_2$) $\Rightarrow$

b $\Leftrightarrow$ participate-in-hole (f_1) $\vee$ participate-in-hole (f_2)) $\oplus$

(f = finter (f_1, pset$-$1, f_2, $pset_2$) $\Rightarrow$

b $\Leftrightarrow$ participate-in-hole (f_1) $\wedge$ participate-in-hole (f_2)) $\oplus$

(f = fdiff (f_1, $pset_1$, f_2, $pset_2$) $\Rightarrow$

b $\Leftrightarrow$ participate-in-hole (f_1)) $\wedge$ $\sim$ participate-in-hole (f_2) $\oplus$

($\exists$ p,q $\in$Point, Π_1 $\in$ Plane, N_1 $\in$ Vector)

$(\mathrm{f} = \text{chopping } (\mathrm{f}_1, \mathrm{p}, \mathrm{q}, \Pi_1, \mathrm{N}_1) \Rightarrow \mathrm{b} \Leftrightarrow \text{participate-in-hole } (\mathrm{f}_1))$

)

)

5.3 Formal Description of Assembly

At the abstract level, assembly is defined as a set of contacts between the features of the objects to be assembled. These contacts are classified into three categories, namely *point contact, line contact* and *area contact.* Although it is possible to formally define all the three types of contacts, we restrict ourselves to only area contacts because the other two types of contacts are practically of no interest in the context of assembly. The reason is that the stability of the object resulting from point contact and line contact types of assembly cannot be assured. However, these two types of contacts are still of interest in the case of grasping an object by a robot. In this case, stability is achieved with the result of forces applied at the gripper and frictional forces at the points of contact.

5.3.1 Assembly Requirements

In all our discussions below, we assume that assembly can be defined only between two objects at a time; the objects may themselves represent subassemblies as well.

Assembly is defined by a set of area contacts between two non-empty subsets of features, one belonging to each component being assembled. These two subsets put together are called *assembly features.* The set of assembly features are classified into two groups, namely *mating features* and *consequent features.* Mating features are those described by user requirements while consequent features represent the set of features which automatically make area contacts as a result of joining the mating features. It will be shown later that every pair of consequent features results due to the joining of a pair of mating features which are of same type and are of same dimensions.

Informal Description of Assembly

Let Obj_1 and Obj_2 be the two objects being assembled and let Obj_3 be the resulting object. Let F_1, F_2 and F_3 denote the set of features of the objects Obj_1, Obj_2 and Obj_3 respectively. Let the mating features be denoted as $fset_1$ and $fset_2$ and the consequent features be denoted as $cset_1$ and $cset_2$. Both $fset_1$ and $fset_2$ are given by the users and $cset_1$ and $cset_2$ are automatically generated.
It is known that ($fset_1 \subseteq F_1$), ($fset_2 \subseteq F_2$), ($cset_1 \subset F_1$) and ($cset_2 \subset F_2$)
For every feature f_1 in $fset_1$, there exists exactly one feature f_2 in $fset_2$ such that f_1 and f_2 are of same type and are defined to make area contacts. Note that, there may exist another feature f'_2 in $fset_2$ such that f_1 and f'_2 are of same type and area contact is possible between them. This situation indicates that two objects can be assembled in more than one way.

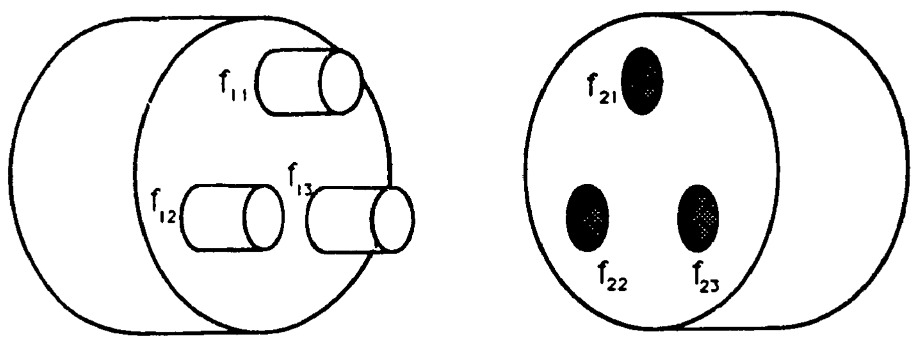


Figure 5.4: Possibility of more than one assembly between two given objects.

To illustrate, consider the assembly of two objects shown in Figure 5.4. Assume that the cylinders are positioned in the corners of an equilateral triangle and the holes in the other object are located at the corners of another equilateral triangle of the same size. Let these two triangles be centered with respect to the circular faces in the two objects. Assume that all the three cylinders in Obj_1 and the three holes in Obj_2 are of same dimensions. It is easy to see that a cylinder in Obj_1 can fit into any one of the three holes in Obj_2. However, once it is decided to mate the feature f_{11} with f_{21}, say, then the other two mating features are fixed as f_{12} with f_{22} and f_{13} with f_{23}. Thus there exist three possible mutually independent assemblies for these two objects as described below :

$<f_{11}, f_{21}>, <f_{12}, f_{22}>, <f_{13}, f_{23}>$

$<f_{11}, f_{22}>, <f_{12}, f_{21}>, <f_{13}, f_{23}>$

$<f_{11}, f_{23}>, <f_{12}, f_{22}>, <f_{13}, f_{21}>$

At the abstract level, our specifications capture the geometric relationships that are common to all these sequences.

As a consequence of assembly, some features in both F_1 and F_2 do not appear in F_3 and some other features appear in F_3 which do no exist in F_1 or in F_2. We denote these two sets of features as 'flost' and 'fnew' respectively. We can observe the following relationships among the various sets of features described thus far :

- There exists a one-to-one correspondence between the subsets $fset_1$ and $fset_2$; a similar relationship exists between $cset_1$ and $cset_2$ as well. By one-to-one correspondence, we mean that for every feature f_1 in $fset_1$, there exists a unique feature f_2 in $fset_2$ such that f_1 and f_2 make area contacts.

  $\mathbf{card}\ fset_1 = \mathbf{card}\ fset_2$

  $\mathbf{card}\ cset_1 = \mathbf{card}\ cset_2$

  $(\forall\ f_1 \in fset_1)\ (\exists!\ f_2 \in fset_2)\ (\text{area-contact}\ (f_1, f_2) \neq \text{NIL})$

  $(\forall\ c_1 \in cset_1)\ (\exists!\ c_2 \in cset_2)\ (\text{area-contact}\ (c_1, c_2) \neq \text{NIL})$

- All these four subsets do not appear in the resulting object and hence they are all lost. i.e.,

  $\mathbf{union}\ (fset_1, fset_2, cset_1, cset_2) = \text{flost}$

- The features which are lost are part of the features of the components Obj_1 and Obj_2 and do not appear in the resulting object. Consequently, 'flost' can be defined as

  $\text{flost} = (F_1 \cup F_2) - F_3$

- Similarly, the set of features 'fnew' can be described as

  fnew $= F_3 - (F_1 \cup F_2)$

- Both 'flost' and 'fnew' will never be empty; i.e., as a result of the assembly, some features are always lost and some are always created. Hence

  $F_3 \neq (F_1 \cup F_2)$

- In the resulting object, there is at least one feature which is preserved as such (unmodified) from the original feature set of either Obj_1 or Obj_2. Thus,

  $F_3 \cap (F_1 \cup F_2) \neq \{\}$

These descriptions belong to the quantitative analysis of assembly. They are useful in reducing the complexity of computation in assembly operations, an important requirement in this context.

Next, we describe the relationships between the set of consequent features and mating features. When a feature f_1 in $fset_1$ is joined with a feature f_2 in $fset_2$, two cases arise :

Case 1 : f_1 and f_2 are of the same dimension.
In practice, for two features to fit, one of them should have more volume space than the other; in other words, a clearance space is to be given for fitting the two features. However, such tolerance limits are dealt with in the implementation phase of the software process model. Hence we ignore tolerances in the specification. Consequently, at the specification level, if two features are to be fitted together, it is sufficient that they be of same dimension. In the present case, f_1 fits exactly with f_2. By exact fit, we mean that no portion of either f_1 or f_2 is visible in the resulting object. Consequently, the features that are previously connected to f_1 and f_2 will automatically make area contacts and thus become the consequent features. If $gset_{11}$ represents the set of features connected to f_1 and $gset_{22}$ represents the set of features connected to f_2, then $gset_{11}$ is a subset of $cset_1$ and $gset_{22}$ is a subset of $cset_2$. That is,

$(gset_{11} \subseteq cset_1)$ and $(gset_{22} \subseteq cset_2)$

In addition, there exists a one-to-one correspondence between $gset_{11}$ and $gset_{22}$ and hence

$\textbf{card}$ $gset_{11}$ = $\textbf{card}$ $gset_{22}$ and

$(\forall\ g_1 \in gset_{11})\ (\exists!\ g_2 \in gset_{22})$ (area-contact $(g_1, g_2) \neq$ NIL)

By generalizing this concept for all the features in $fset_1$ and $fset_2$, we conclude

$cset_1 = \{g_i \mid g_i \in$ connected $(f_i) \wedge f_i \in fset_1 \wedge$

$(\exists!\ f_j \in fset_2)$(same-dimensions $(f_i, f_j))\}$ and

$cset_2 = \{g_j \mid g_j \in$ connected $(f_j) \wedge f_j \in fset_2 \wedge$

$(\exists!\ f_i \in fset_1)$(same-dimensions $(f_j, f_i))\}$

Case 2 : the dimensions of f_1 and f_2 are different.

Two situations arise in this case.

Case 2.1 : one feature, say f_2, is completely covered (and hence becomes invisible) by the other feature, f_1.

In such a situation, f_2 is completely lost. However, f_1 is modified and appears as a new feature in Obj_3. See Figure 5.5 which gives a pictorial description of this situation in case of primitive features.

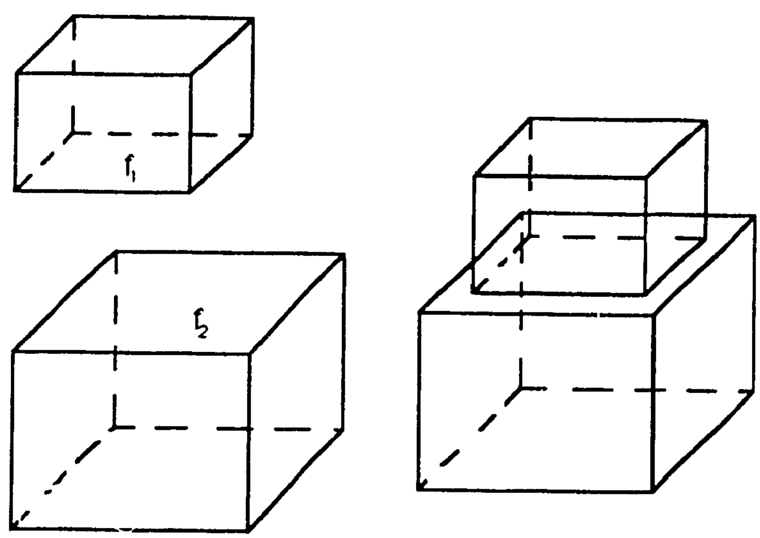


Figure 5.5: Example for one feature covering the other.

Case 2.2 : f_1 and f_2 will partially overlap with each other; i.e., some portion of f_1 and some portion of f_2 are still visible even after the assembly. This situation gives rise to two new features in the resulting object Obj_3 which are modified versions of f_1 and f_2. This is illustrated in Figure 5.6 for primitive features.

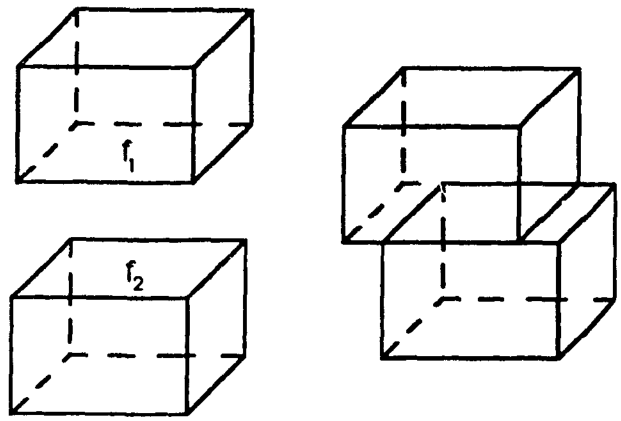


Figure 5.6: Example for partial overlapping of features.

From 2.1 and 2.2, we conclude that every new feature in Obj_3 is a modified version of a feature f where f belongs to either Obj_1 or Obj_2. Hence

$(\forall f_n \in \text{fnew})\ (\exists!\ f \in (\text{fset}_1 \cup \text{fset}_2 \cup \text{cset}_1 \cup \text{cset}_2))\ (f_n = \text{modified } (f))$

which also gives rise to the following relation

$(\forall f_n \in \text{fnew})\ (\exists!\ f \in \text{flost})\ (f_n = \text{modified } (f))$ and

card fnew $\leq$ card flost

Specifications for Assembly

Summarizing all the informal descriptions stated previously, we now provide the formal specifications for assembly relationships.

Object :: POSI-ORIE : Transformation
FEATURES : Feature–set

Mating-Features $\quad = \quad$ Feature $\rightarrow$ Feature

The VDM map data type is used to describe the mating features which automatically assures one-to-one correspondence between fset_1 and fset_2.

Assembly : Object $\times$ Object $\times$ Mating-Features $\rightarrow$ Object

pre-Assembly (Obj_1, Obj_2, Mfeatures) $\quad \triangleq$

$\quad$ **let** T_1 = POSI-ORIE (Obj_1), T_2 = POSI-ORIE (Obj_2),

$\quad\quad$ $\mathrm{fset}_1 = \mathbf{dom}$ Mfeatures, $\mathrm{fset}_2 = \mathbf{rng}$ Mfeatures **in**

$\quad\quad$ ($\mathrm{fset}_1 \subseteq$ FEATURES (Obj_1)) $\wedge$

$\quad\quad$ ($\mathrm{fset}_2 \subseteq$ FEATURES (Obj_2)) $\wedge$

$\quad\quad$ $(\mathrm{fset}_1 \neq \{\}) \wedge (\mathrm{fset}_2 \neq \{\}) \wedge$

$\quad\quad$ ($\mathbf{card}\ \mathrm{fset}_1 = \mathbf{card}\ \mathrm{fset}_2$) $\wedge$

$\quad\quad$ $(\forall\, f_1 \in \mathrm{fset}_1)$

$\quad\quad\quad$ (can-mat-eachother (f_1, Mfeatures(f_2)))

$\quad$ **tel**

post-Assembly (Obj_1, Obj_2, Mfeatures, Obj_3) $\quad \triangleq$

$\quad$ **let** F_1 = FEATURES (Obj_1), F_2 = FEATURES (Obj_2),

$\quad\quad$ F_3 = FEATURES (Obj_3), $\mathrm{fset}_1 = \mathbf{dom}$ Mfeatures,

$\quad\quad$ $\mathrm{fset}_2 = \mathbf{rng}$ Mfeatures **in**

$\quad\quad$ $(F_3 \neq (F_1 \cup F_2)) \wedge$

$\quad\quad$ $(F_3 \cap (F_1 \cup F_2) \neq \{\}) \wedge$

$\quad\quad$ $(\exists\, \mathrm{cset}_1, \mathrm{cset}_2, \mathrm{flost}, \mathrm{fnew} \in$ Feature-set)

$\quad\quad\quad$ $((\mathrm{fset}_1 \subseteq F_1) \wedge (\mathrm{fset}_2 \subseteq F_2) \wedge$

$\quad\quad\quad$ $(\mathrm{cset}_1 \subset F_1) \wedge (\mathrm{cset}_2 \subset F_2) \wedge$

$\quad\quad\quad$ $(\mathrm{fset}_1 \cap \mathrm{cset}_1 = \{\}) \wedge$

$\quad\quad\quad$ $(\mathrm{fset}_2 \cap \mathrm{cset}_2 = \{\}) \wedge$

$\quad\quad\quad$ ($\mathbf{card}\ \mathrm{fset}_1 = \mathbf{card}\ \mathrm{fset}_2$) $\wedge$

$\quad\quad\quad$ ($\mathbf{card}\ \mathrm{cset}_1 = \mathbf{card}\ \mathrm{cset}_2$) $\wedge$

$\quad\quad\quad$ $(\forall\, c_1 \in \mathrm{cset}_1)$

$\quad\quad\quad\quad$ $(\exists!\, f_1 \in \mathrm{fset}_1)$ (connected (c_1, f_1)) $\wedge$

$(\forall\ c_2 \in cset_2)$

$(\exists !\ f_2 \in fset_2)\ (connected\ (c_2, f_2)) \wedge$

$(flost = (F_1 \cup F_2) - F_3) \wedge$

$(fnew = (F_3 - (F_1 \cup F_2)) \wedge$

$(flost = \mathbf{union}\ (fset_1, fset_2, cset_1, cset_2)) \wedge$

$(\mathbf{card}\ fnew \leq \mathbf{card}\ flost) \wedge$

$(\forall\ f_1 \in fset_1)$

$(\exists !\ f_2 \in fset_2)$

$((\mathbf{let}\ pset_f = \text{area-contact}\ (f_1, f_2)\ \mathbf{in}$

$pset_f \neq \{\}$

$\mathbf{tel}) \wedge$

$(\text{same-dimensions}\ (f_1, f_2) \Rightarrow$

$(\exists\ gset_1, gset_2 \in \text{Feature-set})$

$((gset_1 \subseteq cset_1) \wedge (gset_2 \subseteq cset_2) \wedge$

$(gset_1 = \text{adjacent-features}\ (f_1)) \wedge$

$(gset_2 = \text{adjacent-features}\ (f_2)) \wedge$

$(\forall\ g_1 \in gset_1)$

$(\exists !\ g_2 \in gset_2)$

$(\mathbf{let}\ pset_g = \text{area-contact}\ (g_1, g_2)\ \mathbf{in}$

$pset_g \neq \{\}$

$\mathbf{tel})$

)

)

$) \wedge$

$(\forall\ f_3 \in fnew)$

$(\exists\ f_1, f_2 \in \text{Feature}, pset_1, pset_2 \in \text{Point-set})$

$((f_1 \in fset_1) \wedge (f_2 \in fset_2) \wedge$

$((f_3 = \text{funion}\ (f_1, pset_1, f_2, pset_2)) \oplus$

$(f_3 = \text{finter}\ (f_1, pset_1, f_2, pset_2)) \oplus)$

$(f_3 = \text{fdiff}\ (f_1, pset_1, f_2, pset_2))$

)

)

)

Connected : Feature $\times$ Feature $\rightarrow$ Boolean

post-Connected (f, g) $\triangleq$

(f $\in$ adjacent-features(g)) $\wedge$ (g $\in$ adjacent-features (f))

Adjacent-features : Feature $\rightarrow$ Feature-set

post-Adjacent-features (f, fset) $\triangleq$

(**card** fset = **len** connectives-of-features (f)) $\wedge$

($\forall$ cf $\in$ **elems** connectives-of-features (f))

($\exists!$ f_1 $\in$ fset)

($\exists!$ cf_1 $\in$ **elems** connectives-of-features (f_1))

(($\forall$ p $\in$ Point) (on (p, cf) $\Leftrightarrow$ on (p, cf_1)))

Area-contact : Feature $\times$ Feature $\rightarrow$ Point-set

post-Area-contact (f_1, f_2, pset) $\triangleq$

($\forall$ p $\in$ pset)

((on (p, f_1) $\wedge$ on (p, f_2) $\wedge$

opposite-direction (normal (f_1, p), normal (f_2, p))

) $\wedge$

(* the contacting area must be continuous *)

continuous (pset) $\wedge$

(* the contacting area is neither a point nor a line *)

($\exists$ p_1, p_2, p_3 $\in$ pset)

($\sim$ collinear (p_1, p_2, p_3))

The function 'continuous' specifies the continuity of a point set. This will be defined in detail only in the implementation.

Same-dimensions : Feature $\times$ Feature $\rightarrow$ Boolean

post-Same-dimensions (f_1, f_2, b) $\triangleq$

b $\Leftrightarrow$ (f_1 $\in$ Rectangle $\Rightarrow$

$(f_2 \in \text{Rectangle}) \wedge (\text{LENGTH}\ (f_2) = \text{LENGTH}\ (f_1)) \wedge$

$(\text{BREADTH}\ (f_2) = \text{BREADTH}\ (f_1))$

$) \oplus$

$(f_1 \in \text{Circle} \Rightarrow$

$(f_2 \in \text{Circle}) \wedge (\text{RADIUS}\ (f_2) = \text{RADIUS}\ (f_1))$

$) \oplus$

$(f_1 \in \text{Cylindrical-surface} \Rightarrow$

$(f_2 \in \text{Cylindrical-surface}) \wedge$

$(\text{RADIUS}\ (\text{CIRCLE1}\ (f_2)) = \text{RADIUS}\ (\text{CIRCLE1}\ (f_1)))$

$) \oplus$

$(f_1 \in \text{Conical-surface} \Rightarrow$

$(f_2 \in \text{Conical-surface}) \wedge$

$(\text{RADIUS}\ (\text{BASE-CIRCLE}\ (f_2)) = \text{RADIUS}\ (\text{BASE-CIRCLE}\ (f_1))$

$) \oplus$

$(f_1 \in \text{Composite-feature} \Rightarrow$

$(f_2 \in \text{Composite-feature}) \wedge$

$(\exists\ f_3, f_4, f_5, f_6 \in \text{Feature}, \text{pset}_1, \text{pset}_2, \text{pset}_3, \text{pset}_4 \in \text{Point-set})$

$((\text{same-dimensions}\ (f_3, f_5)) \wedge (\text{same-dimensions}\ (f_4, f_6)) \wedge$

$(\textbf{card}\ \text{pset}_1 = \textbf{card}\ \text{pset}_3) \wedge (\textbf{card}\ \text{pset}_2 = \textbf{card}\ \text{pset}_4) \wedge$

$((f_1 = \text{funion}\ (f_3, \text{pset}_1, f_4, \text{pset}_2) \Rightarrow$

$f_2 = \text{funion}\ (f_5, \text{pset}_3, f_6, \text{pset}_4) \oplus$

$(f_1 = \text{finter}\ (f_3, \text{pset}_1, f_4, \text{pset}_2) \Rightarrow$

$f_2 = \text{finter}\ (f_5, \text{pset}_3, f_6, \text{pset}_4) \oplus$

$(f_1 = \text{fdiff}\ (f_3, \text{pset}_1, f_4, \text{pset}_2) \Rightarrow$

$f_2 = \text{fdiff}\ (f_5, \text{pset}_3, f_6, \text{pset}_4) \oplus$

$(\exists\ p, q, r, s, \in \text{Point}, \Pi_1, Pi_2 \in \text{Plane}, N_1, N_2 \in \text{Vector})$

$(f_1 = \text{chopping}\ (f_3, p, q, Pi_1, N_1) \Rightarrow$

$f_2 = \text{chopping}\ (f_5, r, s, Pi_2, N_2))$

$)$

$)$

$)$

The function 'Can-mat-eachother' validates every pair of mating features for the assembly. It is first informally described as follows :
For every pair $<f_1, f_2>$ of mating features,

- If f_1 and f_2 are defined with respect to a common coordinate frame T such that the origin O of T is a common point between f_1 and f_2, then both f_1 and f_2 should make an area contact. Let this area be denoted as 'pset'.

- f_1 and f_2 must be of same shape.

- If there exists a feature g_1 adjacent to f_1 such that its connective to f_1 is completely enclosed in the contacting area pset and g_1 is not part of any hole, then there must exist another feature g_2 adjacent to f_2 such that connective between f_2 and g_2 is also completely enclosed within pset and g_1 and g_2 can-mat-eachother.

- If f_1 and f_2 are of same dimensions, then

  - the number of adjacent features to f_1 is equal to the number of adjacent features to f_2

  - for every adjacent feature g_1 to f_1, there exists a unique adjacent feature g_2 to f_2 such that g_1 and g_2 can-mat-eachother.

- If f_1 is part of a hole, then f_2 cannot be part of any hole.

A formal description of 'can-mat-eachother' now follows :

Can-mat-eachother : Feature $\times$ Feature $\rightarrow$ Boolean
post-Can-mat-eachother (f_1, f_2, b) $\triangleq$
    b $\Leftrightarrow$ ($\exists$ T $\in$ Transformation)
        (**let** O = position (T) **in**
            on (O, f_1) $\wedge$ on (O, f_2) $\Rightarrow$
                (**let** pset = area-contact (f_1, f_2) **in**
                    (pset $\neq$ {}) $\wedge$
                    ($\forall$ p $\in$ pset, v $\in$ Vector)
                        (tangent-vector (f_1, p, v) $\Rightarrow$

(tangent-vector (f_2, p, v) $\wedge$

(shape-operator (f_1, p, v) = shape-operator (f_2, p, v))

) $\wedge$

($\forall$ g_1 $\in$ adjacent-features (f_1))

(connectives-of-feature (g_1) $\subset$ pset) $\wedge$

$\sim$ participate-in-hole (g_1) $\Rightarrow$

($\exists!$ g_2 $\in$ adjacent-features (f_2))

(connectives-of-feature (g_2) $\subset$ pset) $\wedge$

(can-mat-eachother (g_1, g_2)

)

tel) $\wedge$

(participate-in-hole (f_1) $\Rightarrow$ $\sim$ participate-in-hole (f_2)) $\wedge$

(same-dimensions (f_1, f_2) $\Rightarrow$

(**card** adjacent-features (f_1) = bf card adjacent-features (f_2)) $\wedge$

($\forall$ g_1 $\in$ adjacent-features (f_1))

($\exists!$ g_2 $\in$ adjacent-features (f_2))

(can-mat-eachother (g_1, g_2))

)

tel)

The quantitative and qualitative properties of assembly are captured in the specification. We remark that the given specification is incomplete and requires several extensions before being found useful for assembly process verification.

Chapter 6

Deriving Design from Formal Specifications

A behavior specification describes WHAT the system is supposed to do and is therefore independent of any design or implementation details. In contrast, the design specification expresses the structure of various modules in the design, their exported components and the interface between the modules. Thus the design specification is more detailed than its behavior specification. Several methods have been reported recently in the literature on refinement of formal specifications into design [Jon86, CDD90, Gio90].

Rapid prototyping is one of the design approaches by which a quick implementation is derived from the specifications. This prototype captures most of the functionalities of the end product. However, this method is cost-effective only if it is possible to develop an inexpensive prototype. In addition, if more errors are found at later stages of implementation, this approach is not cost-effective; the prototyping must be redone. Stepwise refinement of the formal specifications to design and implementation is an alternate approach for software development.

Stepwise refinements of VDM specifications have been advocated by Jones [Jon86] using operation decomposition and data refinement techniques. Operation decomposition is a method of decomposing complex operations into more primitive operations; hence it is generally algorithmic. Data refinement, on the other hand, refines the abstract data types in the specification into more concrete data types. The selection of concrete data types and the mapping from abstract data types to concrete data types depend on the application. We therefore recommend a refinement approach such as the one shown in Figure 6.1.

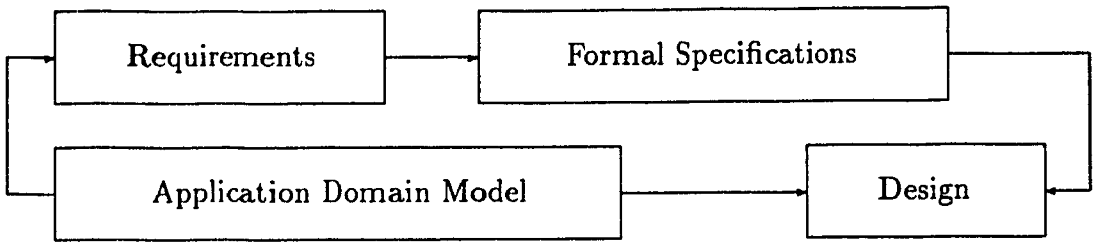


Figure 6.1: Role of Formal Specifications and Application Domain Model in Software Development.

6.1 Refining to an Object-Oriented Design

In this chapter, we propose a methodology to refine model-based specifications into object-oriented design. Advantages of object-oriented design over traditional functional design have been extensively discussed in the literature [Boo86, Cox86, Mey88a, Mey88b]. Researchers who attempted to derive an object-oriented design for a set of requirements have either first informally designed and then specified the design [Gio90] or extended the specification style to suit their lower level design needs [CDD90]. In both the attempts, the design decisions preempts formal specifications. Consequently, they do not effectively make use of the power of formal specifications in verifying the ultimate design. We claim that our attempt is general in the sense that it is applicable to any problem domain and is also independent of any particular object-oriented design method.

In the following sections, we give schema for model-oriented specification and object-oriented design paradigm, both abstracting most of the existing techniques in their respective domains (VDM, Z, Eiffel and HOOD). The proposed transformation requires user interaction at critical stages and can be automated with support from a knowledge-based system. See Appendix where the designs for the specifications discussed in Chapters 3 and 4 are derived using the proposed methodology.

6.2 Schema for Object Oriented Design Paradigm

Informally, an object is a software unit whose behavior can be completely characterized by the actions that it suffers and the actions that it imports. The importation gives the relationships among objects and the extent to which one affects the other. For us, there are two kinds of objects, application domain objects and system objects. There is no distinction between them in terms of their formal representations.

Objects exhibiting similar properties are grouped into a *class*. Consequently, a class is the abstract data type of an object-oriented design paradigm and objects belonging to a class are the instantiations of this abstract data type. Every class has a non-empty set of *features* which is the union of two non-empty sets, namely the set of *operations* affecting an object of the class and the set of variables, also called *attributes*, over which the operations are defined. Figure 6.2 represents a schema of an object.

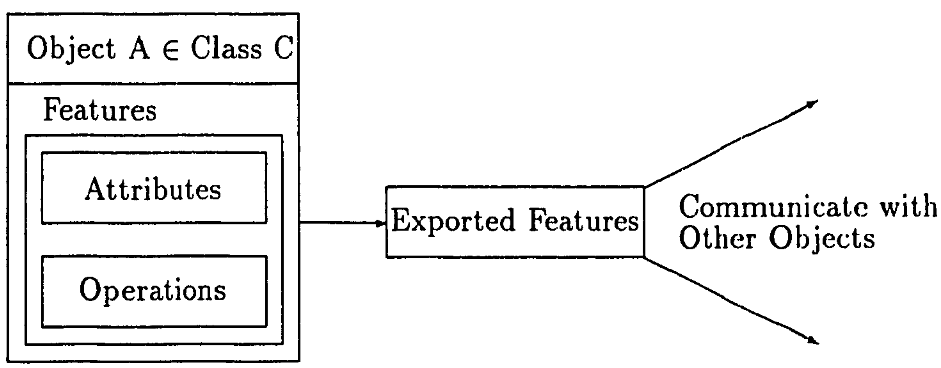


Figure 6.2: An Object in the Object-Oriented Design

6.2.1 Relationship Between Classes

One of the primary goals of object-oriented design is *reusability*. The three important concepts of object-oriented paradigm which promote reusability are *inheritance*, *polymorphism* and *dynamic binding* [Mey88b, Mey90, KoM90, Bud91]. The latter two concepts deal with implementation of objects and are thus related to run-time behavior of objects. Inheritance deals with the dependency relationships between classes which can be determined statically. Meyer [Mey90] states that inheritance can be determined at early stages of software development; however, he also comments that the inheritance structure representing the dependency relationships among the classes has to be reorganized during implementation in order to achieve effective reuse of components. Budd [Bud91] has given a few heuristics for obtaining inheritance between classes; however, no methodology seems to be known. Since efficiency is a concern of implementation, we do not address efficiency of the inheritance structure in this thesis; rather, we provide rules for deriving the inheritance structure from the specifications. The structure thus obtained may serve as the initial structure which might be reorganized for the sake of efficiency during implementation.

Inheritance

People differ in providing a precise definition for inheritance. However, the following concepts regarding inheritance have been agreed by several researchers in this area [Mey88b, KoM90, Bud91].

When a class A inherits another class B,

- A becomes a specialization of B; i.e., the features of A form a superset of the features of B. Accordingly, A is called the *specialized* class and B is called the *general* class [KoM90, Bud91]. As an example, *Rectangle* inherits *Polygon*; in this case, *Rectangle* is a specialization of *Polygon*.

- The feature set of A is strictly different from the feature set of B in order to be specialized [Bud91]. This might be obtained by adding new features to A in addition to the features inherited from B (for example, *Rectangle* inherits *Polygon*) or by renaming and/or redefining some of the inherited features in A [Mey88b, Mey90] (for example, *Square* inherits *Rectangle*). In the latter case, the renaming and redefining of features must be consistent with the other features in A.

- A should not re-export the features (without renaming or redefining) which are already exported by B [Mey90]; otherwise, confusions and inconsistencies might arise.

The inheritance relation is also called *is-a* relation [KoM90] and *sub-class* relation [Bud91].

Part-of Relation

Another important relationship between classes is *part-of* or *component-of* relationship. Sometimes, it is called *has-a* relationship [Bud91] or *client* relationship [Mey88b]. When a class A is *part-of* another class B,

- A is strictly a component of B; stated otherwise, B cannot survive without A. For example, *Wrist* is a *part-of Robot*. This also implies that at no time B and A are one and the same [Bud91]; otherwise, redundancy will occur, contrary to reusability.

- The features of B may exploit (make use of) the features of A. However, B cannot alter any feature of A. This implies that some features of B may be implemented using the features of A.

According to Meyer [Mey90], the two relationships inheritance and client are distinguished so that "Being a client means reusing the specifications and being a specialized class means having access to the implementation. A client class A, thus, communicates with the class B for which it is a client only through the exported features of B whereas a specialized class has complete access to the general class.

In addition to inheritance and part-of relationships, hierarchical object-oriented design methods such as HOOD [Gio90] define additional relationships between classes. These include *use, implemented-by, parent-child* and *senior-junior* hierarchical relationships. However, inheritance and part-of relationships are common to most of the existing object-oriented design methods and object-oriented programming languages such as HOOD [Gio90], Eiffel [Mey88b], C++ [Str89] and Smalltalk [McG87]. In this thesis, only inheritance and part-of relationships are addressed and we adhere to the definitions for these two relationships mentioned earlier. Issues such as *polymorphism* and *dynamic binding* arise at a more detailed design and implementation levels and so we do not discuss them in this context. Hereafter, by 'design', we mean an 'object-oriented design'.

6.3 Schema for Model-Oriented Specifications

As remarked earlier, we restrict to model-oriented specification languages that are traditionally used for functional decomposition techniques and are believed to be not well suited for the description of abstract objects. As we demonstrate in the next section, an object-oriented design can indeed be derived from model-oriented specifications.

The two major components of a model-based specification are *state space* definitions and specifications for *operations* affecting the state spaces. A state space consists of a set of global variables. In the case of VDM, there is only one state space and all the global variables are defined in it. In Z, more than one state space definition can exist and the global variables are distributed among them. The static relationships among the global variables can be asserted by *invariants*, which in the case of VDM, can be combined into a single logical formula and in case of Z, consists of individual logical assertions pertaining to the state spaces. The schema calculus mechanism in Z permits combining the state spaces and hence the local invariants can be combined. In either model, the specifications are consistent only if the invariants are respected by every operation.

An operation in a model-based specification is specified by two predicates, namely *pre-condition* and *post-condition*. The syntax of VDM distinguishes a pre-condition from a post-condition; however, in Z, we have to infer them from the predicate part of the schema. Pre-condition is a set of system constraints that are to be satisfied before the operation is invoked while the post-condition is another set of system constraints that must be satisfied after the operation successfully terminates. Both the pre- and post-conditions are defined over the global variables accessed in that operation and the parameters of the operation. The set of global variables accessed in that operation is made explicit indicating how the operation affects the state space. A schema of a model-oriented specification is given in Figure 6.3.

Hereafter we use the term 'specification' to refer to 'model-based specification' throughout this chapter.

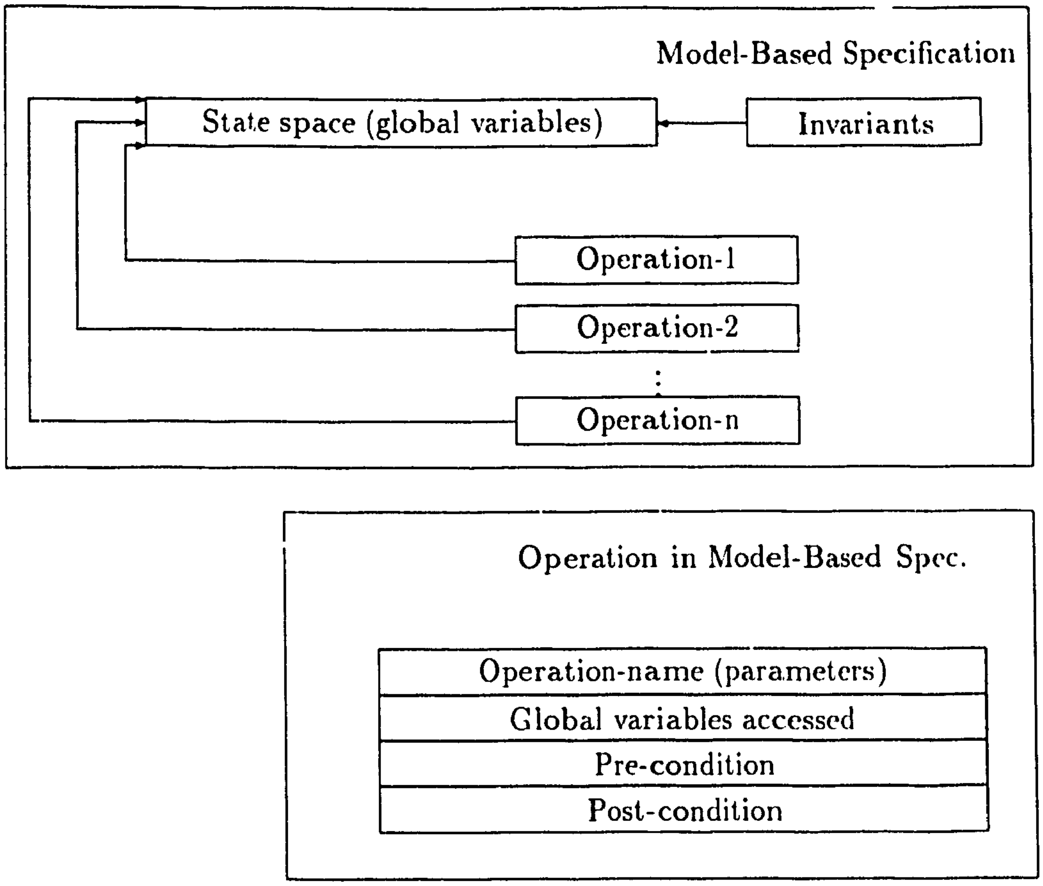


Figure 6.3: A Schema of a Model-Oriented Specification.

6.4 Transformation Process

The transformation from formal specifications to design is explained below informally; a more formal definition will be taken up later as a continuation of this work. There are four stages in this transformation process :

- identifying the classes in the design.
- identifying the attributes within each class.
- deriving the operations for each class.
- deriving the *inheritance* and *part-of* relationships between the classes.

A design contains more details than its corresponding formal specification, and consequently may require some information in addition to what is stated in a formal specification document. These additional information must be obtained from the application domain model through user interaction. This interaction may be automated with the help of a good domain model and a knowledge-base support for reasoning and retrieving the information. In our discussions below, we point out the particular stages where additional information may become necessary for the design.

6.4.1 Identifying the Classes

The state spaces in a formal specification contain the entities modeled within the specification language. As remarked earlier, the state space models system objects such as sets, maps and lists as well as domain objects. These models are abstract data types built from the primitive data types provided by the specification language. Hence each abstract data type in the state space that models an entity can be mapped into a unique class in the design.

For each simple type such as String and Nat, in the state space definition, a unique class is created in the design. For each composite type in the specification, a new class is created in the design corresponding to the composite type and a unique class is created for each component type of this composite type, making sure that redundant classes are not created.

Example 1 :

Consider a VDM state space specification in which a composite type called, 'Customer-record' is defined as follows :

```
Customer-Record     :: NAME : String
                       ACCOUNT-NO : Nat
                       ACCOUNT-TYPE : String
                       BALANCE : Nat0
```

In the design, a class corresponding to 'Customer-Record' will exist; in addition, classes corresponding to 'String', 'Nat' and 'Nat0' are also created.

In the above example, it is easy to see that 'Nat' is a subset of 'Nat0' and hence 'Nat' can inherit 'Nat0' and restrict its domain to only numbers greater than zero. Since at the specification level, data type dependencies are not explicitly addressed, it is up the designer to identify such dependencies in the design and resolve whether or not to keep them in the design. An analogous situation arises in renaming data types. Type renaming becomes vital for a meaningful understanding of the specifications. While transforming a renamed type, we create a new class for the renamed type and inherit the class corresponding to the type being renamed. The reason for inheriting will be made clear in a later section.

Having generated the classes from several data types in state spaces, we create a super class or root class corresponding to the global state space of the specification. The only state space in VDM is the global state space. Although VDM is revised to include several state spaces, at the time of writing this thesis, there is no publication reporting the extended feature. Hence we assume here that there is only one state space definition in VDM. The schema calculus in Z allows combining multiple state spaces to create a global state space. All other classes are made as components of the root class and hence there exists *part-of* relation between the root class and every other class derived earlier. The justification for creating a root class and making all other classes as components comes from the fact that every problem is initially viewed from top-down although it can be developed bottom-up at several intermittent stages.

6.4.2 Identifying the Attributes of Classes

Attributes of a class are the variables that are manipulated by the operations of the class. Some of these attributes may also be exported by that class. There exists a one-to-one correspondence between global variables of the specification and attributes of the classes. The transformation identifies how attributes of various classes can be directly derived from the global variables of the specification.

We first transform the global variables of the state space directly into the attributes of the root class. Since we have already identified all classes corresponding to the data types in the specification, it is straightforward to map these attributes to their respective classes. Variables corresponding to the fields of composite types are mapped to the attributes of the class corresponding to the composite type. Since the type corresponding to a field variable may itself be composite, this process is applied recursively until all the variables in the specification are mapped into the attributes of the classes.

Example 2 :

Consider the VDM specification fragment below :

BASE-COORD	:	Transformation
ROBOT-ARM	:	Manipulator
Manipulator	::	LINKS : Armtype-list JOINTS : Jointtype-list GRIPPER : Grippertype
Armtype	::	LINKID : ID–Rep GEOMETRY : Structure
Jointtype	=	Prisjoint | Revoljoint
Grippertype	:.	WRIST : Wristtype FINGER-GRIP-JOINTS : Fingertype → Jointtype
Wristtype	::	GEOMETRY : Structure
Fingertype	::	FINGERID : ID–Rep GEOMETRY : Structure

It is to be noted that, the line

Jointtype = Prisjoint | Revoljoint

should not be viewed as type renaming. Here, it refers to a type equivalence. In VDM, the operator '=' is overloaded in the sense we use the same operator for type renaming, type equivalence, assignment and comparison. So, in this case, Jointtype becomes a general type which can be either Prisjoint or Revoljoint depending on the context. The classes and attributes within each class obtained by the transformation are the following :

class *ROBOT* (* Super Class *)

attributes

Base-coord : *TRANSFORMATION*

Robot-arm : *MANIPULATOR*

class *MANIPULATOR*

attributes

  Links : *LIST[ARMTYPE]*

  Joints : *LIST[JOINTTYPE]*

  Gripper : *GRIPPERTYPE*

class *ARMTYPE*

attributes

  Linkid : *ID-REP*

  Geometry : *STRUCTURE*

class *JOINTTYPE*

attributes

class *PRISJOINT*

attributes

inherits *JOINTTYPE*

class *REVOLJOINT*

attributes

inherits *JOINTTYPE*

class *GRIPPERTYPE*

attributes

  Wrist : *WRISTTYPE*

  Finger-Grip-Joints : *MAP[FINGERTYPE,JOINTTYPE]*

class *WRISTTYPE*

attributes

  Geometry : *STRUCTURE*

class *FINGERTYPE*

attributes

  Fingerid : *ID-REP*

  Geometry : *STRUCTURE*

class *STRUCTURE*

attributes

class *ID-REP*

attributes

class *TRANSFORMATION*

attributes

6.4.3 Deriving the Operations of the Classes

As shown in Figure 6.3, an operation in a model-based specification has four major components, namely the operation header (includes name of the operation and the input and output parameters along with their types), the set of global variables accessed in that operation, the pre-condition and the post-condition. In VDM, these divisions are explicit whereas in Z we have to infer them from the structure of the specification. There are three steps in transforming operations in specifications to operations in the classes. In the first step we assign; in the second we redefine some of the operations, if necessary; and finally in the third, we clean up by removing redundancies.

<u>Step 1</u> :

1. If an operation Op in the specification accesses a set of global variables and these are mapped into attributes of the same class C in the design, then Op is transformed into an operation Op' of class C. The justification for this rule comes from the fact that the set of global variables accessed in Op determine the portion of the state space affected by Op. Consequently, their mapping into attributes of class C confirms that this portion of state space corresponds to an isolated object belonging to class C.

2. If an operation Op in the specification accesses a set of global variables and these are mapped into attributes of different classes $C_1, C_2, \ldots, C_n$, then Op is transformed into a set of operations $\{Op_i \mid Op_i \in C_i, 1 \leq i \leq n\}$. The justification is the same as before. This may seem redundant at this stage; however, in step 3, we eliminate such redundancies.

3. If a global variable is of composite type T, then due to the presence of subtypes in the composite type, the transformation creates operations in addition to those created in (1) and (2). Notice that a subtype itself may be composite;

in this case, the process is recursively applied until all the composite types are exhausted. Let the final set of classes to which Op is transformed be $\{C_j, 1 \leq j \leq k\}$ where k is the number of classes affected by Op. Operation Op_j belonging to class C_j is created (i.e., transformed from Op) if and only if

- C_j is the class assigned by the transformation to a subtype T_j of the composite type T.
- there exists an attribute v in C_j such that there is a variable $\overline{v}$ of type T_j associated with the composite type T and gets mapped to v.
- the variable $\overline{v}$ is affected either by the pre- or post-condition of Op.

For example, the operation 'Translate-link' in the specification given in Chapter 4 is transformed to the super class *ROBOT* in example 2 since this operation accesses the variable 'Robot-arm' in the super class. In addition, the same operation is also transformed to the classes *MANIPULATOR*, *ARMTYPE*, *JOINT-TYPE* and *STRUCTURE* since the variables 'Links', 'Joints' and 'Geometry' are all accessed by this operation. However, 'Translate-link' is not transformed to the class *GRIPPERTYPE* although it is one of the component types of the composite type *MANIPULATOR*. This is due to the fact that none of the variables corresponding to the attributes of the class *GRIPPERTYPE* (as given in the specifications) is affected by 'Translate-link'.

Step 2 :

If an operation Op in the specification is transformed to an operation Op_c in a class C, then Op_c may have to be redefined so as to remain meaningful in the context. We claim that the information for this redefinition is available in the pre- and post-conditions of Op. Depending on this information, Op_c may be redefined into one or more operations $Op_{c1}, Op_{c2}, \ldots, Op_{ck}$ in a class C where Op_{ci} implements one or more assertions stated in Op. As an example, the operation 'Translate-link' transformed to the class *STRUCTURE* is renamed to 'Translate-dist' since its purpose in this class is to translate a rigid solid through a finite distance.

Step 3 :

Some redefined operations that are redundant must be deleted from the classes and some operations may have to be merged. User interaction is necessary at this stage to validate elimination and merging based upon the semantics of the objects. Referring to the same example discussed in Steps 1 and 2, all the operations transformed to the class *STRUCTURE* are finally renamed into two operations 'Translate-dist' and 'Rotate-angle'; that is, every operation transformed to this class is ultimately implemented by one of these two operations.

Polymorphism is a consequence of such merging activities across several classes by finding a common object (code) required by several operations in several classes (not related by 'inheritance' and 'part-of'). This cannot be determined in the architectural design stage and consequently we believe that polymorphism is an implementation rather than a design issue. Since *dynamic-binding* is a consequence of *polymorphism*, we ignore *dynamic-binding* also in our discussion.

Notice that the transformation process assigns an operation, say $\overline{Op}$, in the specification to one or more operations across several classes; i.e., for an operation Op in a class C, there exists exactly one operation $\overline{Op}$ in the specification. We call Op, the *image* of $\overline{Op}$ and $\overline{Op}$, the pre-image of Op under the transformation.

6.4.4 Inheritance and Part-of

Having derived the classes and their components, it is necessary to derive the dependency relationships between every pair of classes. At the specification level, the relationships between abstract data types are explained in terms of the semantics of the specification language. The derivation of classes and their components is a syntactic process. The semantics of the specification language is used to derive the static communication between the classes.

First we obtain the part-of relationships and then we state the rules for deriving inheritance. The two reasons for doing in this order are – (i) part-of relation depends only on the state definition and hence it is easy to derive and (ii) the classes which have part-of relations do not have inheritance relations among themselves. Hence these pairs can be eliminated from consideration for inheritance.

The part-of relationship is directly derived from the state space definition of the specification. Classes corresponding to component types of a composite type are parts of the classes corresponding to the composite type. It is easy to observe that this rule obeys the definition of the part-of relationship mentioned earlier. Although it may suggest that the specification is written with the part-of relationships between the objects in mind, it is indeed the case that the part-of relation is explicit in an application domain; however, inheritance is not.

According to Budd [Bud91], there must exist a relationship of functionality between two classes which are related by inheritance. Consequently, we derive inheritance relationship between classes by analyzing the existing relationships among all pairs of operations, one from each class. The derivation process is divided into two major steps. In the first step, we derive the relationships between every pair of operations $<Op_i, Op_j>$, $Op_i \in C_i$ and $Op_j \in C_j$ where $\{C_i\}$ denote the set of classes in the derived design. We define the term *Op-inheritance* to denote the relationship between operations in two different classes. An operation Op_1 is said to Op-inherit another operation Op_2, if Op_1 accesses Op_2 and modifies internally (local to Op_1) the code of Op_2. In the second step of derivation process, the inheritance between pairs of classes will be derived.

Relationship between Operations

Let C_I be the collection of pairs of classes such that no pair belonging to C_I is related by the part-of relationship. Let $<C_1, C_2> \in C_I$. For $Op_1 \in C_1$ and $Op_2 \in C_2$, we derive the dependency relation as follows :

Let $\overline{Op_1}$ and $\overline{Op_2}$ denote the pre-images of Op_1 and Op_2 in the specifications. Let pre($\overline{Op}$) and post($\overline{Op}$) denote the pre-condition and post-condition of $\overline{Op}$. Any conjunct in the conjunctive normal form of a formula is called ***a subformula***. Two situations arise :

<u>Case 1</u> : $\overline{Op_1}$ and $\overline{Op_2}$ are different operations.

$$\begin{array}{lrcl} \text{Let} & \text{pre}(\overline{Op_1}) & = & A' \wedge X' \\ & \text{pre}(\overline{Op_2}) & = & B' \wedge X' \\ & \text{post}(\overline{Op_2}) & = & A \wedge X \\ & \text{post}(\overline{Op_2}) & = & B \wedge X \end{array}$$

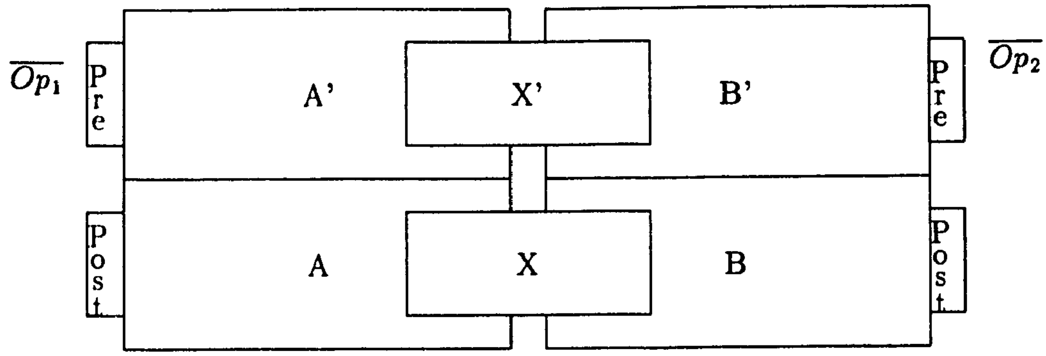


Figure 6.4: $\overline{Op_1}$ and $\overline{Op_2}$ are distinct.

If (A' $\Rightarrow$ B') $\wedge$ (A $\Rightarrow$ B) then Op_1 Op-inherits Op_2. The justification for this claim is the following : Overlapping subformulas, whether in pre- or in post-condition, indicate a functional overlap between the operations. The formula ((P $\Rightarrow$ Q) $\Rightarrow$ ((P $\wedge$ Y) $\Rightarrow$ (Q $\wedge$ Y)) is a tautology and consequently, P $\Rightarrow$ Q is a sufficient condition for the dependency relation to hold. This is the reason why we extracted subformulas from the pre- and the post-conditions.

Case 2 : $\overline{Op_1}$ and $\overline{Op_2}$ are one and the same in the specification; call this $\overline{Op}$.

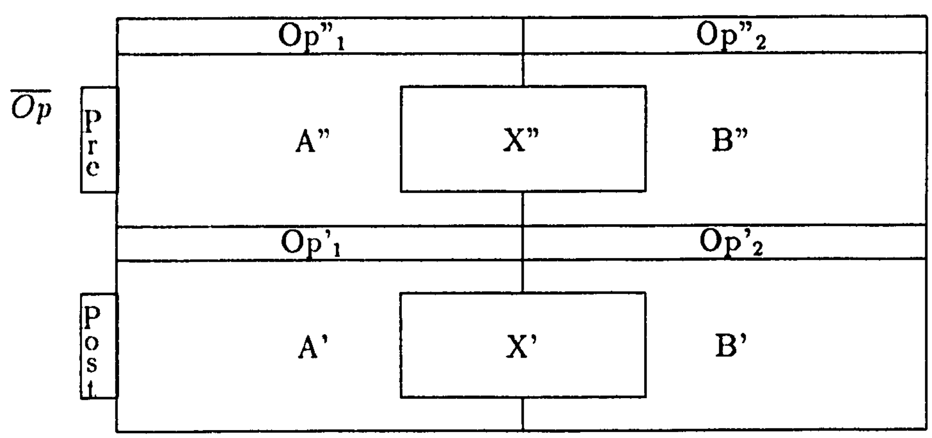


Figure 6.5: $\overline{Op_1}$ and $\overline{Op_2}$ are the same.

This situation arises when $\overline{Op}$ is transformed into two different operations in two different classes, possibly redefined or renamed as Op_1 and

Op_2. As explained in the previous section, the redefinition or renaming occurs when an operation in the design is due to only a portion of the pre-condition or the post-condition or both of its pre-image in the specification. Let Op''_1 and Op''_2 be the subformulas of pre($\overline{Op}$) such that

$$\begin{aligned} Op''_1 &= A'' \wedge X'' \\ Op''_2 &= B'' \wedge X'' \end{aligned}$$

Let Op'_1 and Op'_2 denote the subformulas of post($\overline{Op}$) such that

$$\begin{aligned} Op'_1 &= A' \wedge X' \\ Op'_2 &= B' \wedge X' \end{aligned}$$

The subformulas Op''_1 and Op'_1 together constitute the pre-image of Op_1; the subformulas Op''_2 and Op'_2 together constitute the pre-image of Op_2. If $(A'' \Rightarrow B'') \wedge (A' \Rightarrow B')$ then Op_1 Op-inherits Op_2. the validity of the implication depends on the correct interpretations of the subformulas and hence the application domain model. Consequently, the transformations must be supported by a knowledge-base or user interaction.

Specifications given in Chapters 3 and 4 address only two application domains within robotic assembly. Due to the independent nature of these two component domains, inheritance in the partial design does not arise. When all the subdomains for an assembly environment are specified, the overall design derived from the specifications will give rise to inheritance. For the sake of completeness in illustrating the full expressive power of the methodology, a portion of a library management system discussed in [AlP91b] is extracted and is given below :

Example 3 :

Consider the VDM state definition below :

LIB-SYSTEM	:	Library-System
Library-System	::	COLLECTION : Books–set USERS : Borrowers–set RESERVED : Queue
Borrowers	=	Faculty | Student

Let Op_1 be the operation 'FACULTY-BORROW' in the class 'Faculty' and Op_2 be the operation 'BORROW-BOOK' in the class 'Borrowers'. Assume that the pre-image of both the operations be only one in the specification and is named as 'BORROW'.

Let pre(BORROW) is TRUE and post(BORROW) be the following:

$(u \in \text{USERS (lib-system)}) \wedge$

$(\text{STATUS}(b) = \text{'loaned-out'}) \wedge$

$((u \in \text{Faculty} \Rightarrow \text{due-date'} = \text{current-date} + 30) \oplus$

$(u \in \text{Student} \Rightarrow \text{due-date'} = \text{current-date} + 15))$

Since the pre-condition is TRUE, we consider only the post-condition for deriving the relationship. Let Op_2', the pre-image of Op_2 be post(Op) itself and let Op_1', the portion of post(Op) which is the pre-image of Op_1 be the following :

$(u \in \text{Faculty}) \wedge (u \in \text{USERS (lib-system)}) \wedge$

$(\text{STATUS}(b) = \text{'loaned-out'}) \wedge$

$(\text{due-date'} = \text{current-date} + 30)$

Op_1' and Op_2' have a common sub-formula

$(u \in \text{USERS (lib-system)}) \wedge (\text{STATUS}(b) = \text{'loaned-out'})$

The rest of Op_1' (denoted as P) is

$(u \in \text{Faculty}) \wedge (\text{due-date'} = \text{current-date} + 30)$

and the rest of Op_2' (denoted as Q) is

$((u \in \text{Faculty} \Rightarrow \text{due-date'} = \text{current-date} + 30) \oplus$

$(u \in \text{Student} \Rightarrow \text{due-date'} = \text{current-date} + 15))$

Clearly $P \Rightarrow Q$ and hence Op_1 <u>Op-inherits</u> Op_2 i.e., FACULTY-BORROW <u>Op-inherits</u> BORROW-BOOK. This is meaningful since FACULTY-BORROW is a more specialized operation than the BORROW-BOOK operation.

Relationship among Classes

Having defined the dependency relationships between pairs of operations in all classes, we next derive the inheritance relationships among the classes.

For a pair of classes $<C_1, C_2> \in C_I$, C_1 inherits C_2 if for every operation $Op_2 \in C_2$, there exists an operation $Op_1 \in C_1$ such that Op_1 <u>Op-inherits</u> Op_2. However, if there exists at least one pair of operations $Op_1 \in C_1$ and $Op_2 \in C_2$ for which Op_1 <u>Op-inherits</u> Op_2 holds, and for the rest of the operations in C_2, no such claim can be made, we can create a new class C_3 with the following properties :

- every operation in C_3 is a copy $Op_2 \in C_2$ for which there exists an operation

$Op_1 \in C_1$ and Op_1 Op-inherits Op_2 holds.

- the attributes in C_3 are precisely those attributes in C_2 that are affected by the copied operations.
- no features are exported from C_3, meaning that C_3 is internally used by the classes C_1 and C_2.

Now, C_1 inherits C_3 and C_2 inherits C_3.

Identifying Exported Features

The specification describes only what a system does and not how it does. Hence, every entity in the specification (variable, data type, operation) is related to some other entity. By transforming these entities into the design, we carry over their respective relationships. Therefore we claim that every feature of a class obtained by the transformation is exported by that class; that is, for our consideration in this thesis all features are exported.

6.4.5 Invariants

Invariants stated in the specifications should be carried over to the design in order to assure the consistency and correctness of the design. As stated earlier, invariants assert the static relationships among certain global variables. As a consequence, every operation is to be checked to assure that the invariants are not violated. Although many object-oriented design methods do not include a separate section for invariants we feel that some means must be found to distribute the invariants to the classes in order to assert the validity of every operation in the design.

Since invariants are defined over global variables and the global variables are transformed into the attributes $A_1, A_2, \ldots, A_k$ belonging to various classes in the design, we can determine the dependency matrix between $C_2, \ldots, C_n$ and the rows are the global variables $A_1, A_2, \ldots, A_k$. The $(i,j)^{th}$ entry in the matrix is empty if the attribute corresponding to A_i is not in the class C_j. Assuming that the invariants are combined into a single formula expressed in Conjunctive Normal Form (CNF) we consider each subformula and examine the global variables in that formula. If the attributes corresponding to these variables belong to one column C_j, put this subformula in class C_j. However, when the attributes do not belong to any one particular column (class), determine the minimum set of columns such that the union of attributes in these classes match the variables in the subformula. Assign this subformula to the smallest super class which is related by inheritance or usage to the set of classes determined earlier.

The collection of operations within each class must not violate the formulas assigned to that class or its nearest super class. The lower level implementation must assure that the execution of every operation in a class respects the validity of the formulas in that class.

Chapter 7

Conclusion

The goal of integrating formal methods with the software development process is to ensure correctness of the tasks or processes at each stage of development. The software engineering community believes that such an integration will make the software development process cost-effective [IEE90a, IEE90b, IEE90c]. Formal methods are applicable to all stages of the software development process. Being formal, they enable the designer to reason about the processes at various stages in a software life cycle and hence study the behavior of the entities being specified.

Our aim in this thesis is to study the behavior of entities involved in a large complex software system. Leveson [Lev90] has mentioned that more work on formal methods is necessary (i) to develop tools for writing specifications (ii) to apply formal methods to more complex problems and (iii) to optimize the design process with the help of formally specified requirements. Only a few case studies in applying formal methods were reported in the literature [Hay88] during the tenure of this thesis; however, the problems attempted in these case studies were not as complex as the one chosen for this thesis. Froome and Monahan [FrM88] have also discussed the need for formal methods in developing software for safety-critical systems such as robotics.

One of the major problems in specifying a large complex software system is the *multiplicity of domains* involved in the project; i.e., the system involves the coordination of tasks from several distinct domains. It is required to study the behavior of entities involved in each domain separately by independently specifying them and then studying their combined behavior by properly aggregating these specifications.

Such an integration requires the interfaces to be specified and subsequently force the component specifications to be changed. Moreover, the choice of appropriate specification approach for specifying tasks in each domain is another important problem to be tackled.

The case study chosen in this thesis is the problem of performing automated assembly operations in a single static robot environment. It involves subtasks from three different application domains, namely solid modeling, robotics and assembly environment. Figure 7.1 shows a schematic view of an automated assembly cell and its components. In this figure, the boxes with asterisks indicate the contribution of this thesis.

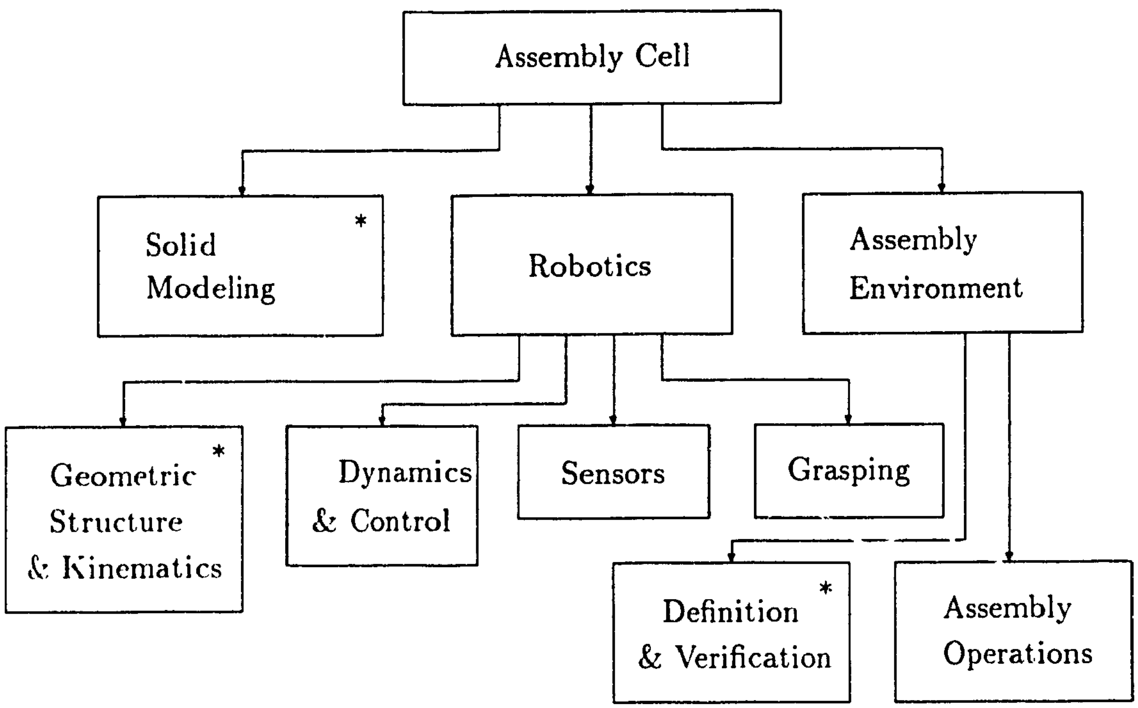


Figure 7.1: Automated Assembly Cell

In solid modeling, previous work concentrated on developing new representation techniques and algorithms for processing geometric information [ReV83]. However, no attempts were made to verify these algorithms. Such a verification is necessary because the algorithms may generate a solid which does not exist in real world. The specifications for regularized boolean operations given in Chapter 3 can be used for verifying a solid modeler.

Hoffmann and Hopcroft [HoH87] have developed a simulation system applicable to robotics. Even though this system can be used for offline verification of robotic systems, it is not sufficiently general to be applied to several applications. We claim that an abstract model of a robot must be developed in order to exhibit the behavior of several real life robots. Such a robot is called a *robotic agent*. There have been only a few attempts [Chr89] to formalize the notion of an integrated architecture for an intelligent robot. Our approach is different from these and can be extended to various problem domains in robotics. The specification for robot structure given in Chapter 4 can be instantiated by several robots in practice and the specification for kinematic operations can be the basis for a formal verification of the software implementing kinematics. Even though the specification for the structure of a robotic agent includes only prismatic and revolute joints, each having only one degree of freedom, joints with n degrees of freedom can also be modeled using this structure. This is accomplished by letting n joints, each with one degree of freedom, connecting links of negligible length. The operation of a controller for such joints which activates only one degree of freedom at a time can be specified using the specification for sequential activation of links given in Chapter 4 whereas the controller that activates more than one degree of freedom at a time corresponds to the concurrent model.

We claim that our contribution to automated assembly is unique for the following reasons : (1) There has been no attempt reported in literature so far to define the topology of assembly. (2) Verification of assembly process between two objects has been carried out only by a few researchers [PGL89, ThC88]. The method given by them is based on *group theory*. In this scheme, two objects can be assembled only if they belong to the same symmetry group and are dimensionally consistent. The basic requirement in using this approach is that the objects should possess rotational symmetry. Hence the approach is not general enough to be applicable to any solid. Moreover, group theory is used as a mathematical tool for the verification of the assembly process; no attempt was made to define the notion of assembly using group theoretic concepts. (3) In this thesis, the mathematical definition of shape operator is used to define feature of an object and subsequently the concept of assembly. Our approach does not impose any constraints on the nature of objects that are assembled. Paul Besl [Pau88] has described the computation of shape operators and their use in computer vision applications. Therefore, the definition and verification of assembly as given in Chapter 5 can be practically realized.

The method proposed in this thesis for the refinement of VDM specifications to object-oriented design is different from other approaches in the same field because it is independent of any existing object-oriented method and hence our contribution in this regard is significant.

7.1 Future Work

The work presented in this thesis is actually a subset of a much larger project. The ultimate goal is to develop an offline verification platform for an automated assembly environment using robots. The formal specifications to be developed for such a platform will become a geometric reasoning system independent of any lower level architecture and programming environment. The geometric reasoner can be used in environments where human interaction is hazardous or impossible; for example, the tasks performed by robots in space station belong to this category.

7.1.1 Solid Modeling

The specifications given in Chapter 3 can be extended to include objects with curved faces as well. To achieve this, it is necessary to specify curved edges and curved faces analogous to their planar counterparts. However, problems might arise in deciding the number of faces especially when a curved face does not fit into a regular primitive shape such as the lateral surface of a cylinder or a cone. We recommend using shape operators for defining surfaces and characterizing objects. Though this method is computationally expensive, Besl [Pau88] claims its applicability for vision applications.

7.1.2 Robotics

Robotics, by itself, is an independent discipline in Computer Science. The behavioral study of a general purpose robot is related to its kinematic and dynamic operations, sensor and control operations, path planning and collision avoidance in the robot environment. A complete set of formal specifications developed to include all these aspects will characterize an intelligent robotic agent. Such specifications will be quite useful in studying the behavior of autonomous systems.

Our contribution in this thesis addresses only kinematics. To develop an offline verification platform, the first stage is to study only the kinematic aspects. This will help modeling a robotic agent in a graphics based simulation system which is a model of the offline platform. However, if the aim is towards studying the behavior of an actual robot, then the specifications have to be extended to include all the other aspects mentioned earlier.

Grasping is an important component of every robotic application. In particular, automated assembly operations performed by robots require a great deal of work in grasping. Therefore, grasping is the the next component to be specified in developing the specification for automated assembly cell.

7.1.3 Assembly

In this thesis, we formally specified objects and their surface characteristics based on the notion of shape operators. These specifications are used in defining the topology of assembly process. Yet, much more work is to be done in the field of assembly. The actual assembly operations such as MOVE, PICK and PLACE are to be specified in order to complete the description of assembly. Specifications for assembly operations are to be developed concurrently with grasping in robotics and both of these will use the specification for kinematic operations.

The specifications for assemblv can be extended to include the formal description of tools such as screw driver and hammer that are used in the assembly process. These specifications enable one to study the behavior of the tools used in assembly environment and also to identify the necessary tools for a given application.

7.1.4 Design

Recently, several attempts have been made to refine formal specifications into design [Jon86, Gio90]. Techniques such as developing *executable specifications* and *rapid* *prototyping* enable the designer to get a quick implementation of the end product which captures most of the important functionalities of the software. However, there are pros and cons to these approaches as compared to the stepwise refinement of formal specifications. We follow the stepwise refinement approach because our aim at this stage is not to develop a cost-effective product but to study the total behavior of a large scale software. In this context, we proposed a new methodology to derive a design from a model-based specification. Further, the design is object-oriented and hence has the advantages over traditional functional design methods such as reusability and maintainability.

The methodology as proposed in the thesis can be easily implemented in a Prolog-style language with minimum user interaction. A complete automation of this method requires an application domain model to be built before deriving the design to be used during the derivation process. A knowledge-based approach is suitable for such purpose because a knowledge base is expandable. Therefore it is possible to reuse the application domain model for developing new software.

By introducing dependent types, *loose* VDM specifications can be transformed to a more detailed specification that lends itself to an error-free implementation; see [HDL90, AlP91a]. Data reification [Jon86] refers to the mapping from abstract data types to more concrete data types. After several applications of data reification and introduction of dependent types, we can assure that an object-oriented design derived from the original specification is free of any *junk object*. In some sense, this process may be called *design time testing*. A detailed design may be obtained by resorting to operation decomposition within the obtained object-oriented design. We have illustrated these principles in [AlP91a].

7.1.5 Refining VDM Specification

As stated in the introduction, the specification approach VDM has several limitations. For example, a specification in VDM cannot capture real-time and concurrency aspects which may be inherent in the application itself. In particular, tasks in robotics require these aspects to be captured in the lower level design specification. Though we claim that concurrency and real-time aspects are not part of behavior specification, design specification at a lower level must incorporate real-time and concurrent aspects. Therefore, we strongly recommend a continuation of this work towards mapping the behavior specification given in VDM or Z into a design specification which can address the real-time and concurrency aspects. Alagar and Ramanathan [AlR91] have demonstrated a functional approach for the specification of real-time and distributed systems. In their approach, events are the primitives and properties of events are expressed using functions. Since VDM, as well as purely functional formalisms are founded on denotational semantics, it may be possible to combine VDM specifications (sequential) and functional specifications [AlR91] that express concurrency and real-time into a single formal framework. Continued work in this research will be a challenge to the application of formal methods to software development.

Bibliography

[ABP88a] Alagar V.S., Bui T.D. and Periyasamy K. A reasoning system for solid modeling techniques applicable to robotics. In *Proceedings of the Symposium on Robot Control*, October 1988.

[ABP88b] Alagar V.S., Bui T.D. and Periyasamy K. Semantic CSG trees. In *Sensor Fusion : Spatial Reasoning and Scene Interpretation – Proceedings of the SPIE*, pages 101–112, November 1988.

[ABP90] Alagar V.S., Bui T.D. and Periyasamy K. Semantic CSG trees for finite element analysis. *Computer Aided-Design*, 22(4):194–198, May 1990.

[ABP91a] Alagar V.S., Bui T.D. and Periyasamy K. A formal framework for specifying robot kinematics. In *Proceedings of The European Robotics and Intelligent Systems Conference*, June 1991.

[ABP91b] Alagar V.S., Bui T.D. and Periyasamy K. Formal specifications for regularized operaions in solid modeling. revised and submitted to *Science of Computer Programming.*

[AlP91a] Alagar V.S. and Periyasamy K. Formal verification and testing of an object-oriented design. submitted for presentation in COMPSAC'91, Japan, 1991.

[AlP91b] Alagar V.S. and Periyasamy K. A methodology for deriving an object-oriented design from functional specifications. submitted for publication to *Software Engineering Journal* (under revision).

[AlR91] Alagar V.S. and Ramanathan G. Functional specification and proof of correctness for time dependent behavior of reactive systems. *Formal Aspects of Computing*, pages 1–31, January 1991.

[Ala91] Alagar V.S. Specification of software systems. Lecture Notes, Department of Computer Science, Concordia University, Montreal, Canada, 1991.

[And90] Andrews D. The Vienna Development Method. In Darrel Ince and Derek Andrews, editors, *The Software Life Cycle*, chapter 11. Butterworths, 1990.

[BAL91] Baldwin D.F., Abell T.E, Lui C.M., De Fazio T.L., and D.E. Whitney. An integrated computer aid for generating and evaluating assembly sequences for mechanical products. *IEEE Journal of Robotics and Automation*, 7(1):78–94, February 1991.

[Bar66] Barret N. *Elementary Differential Geometry.* Academic Press, 1966.

[BjJ82] Bjørner D. and Jones C.B. *Formal Specification and Software Development Method.* Printice Hall, 1982.

[BjJ87] Bjørner D. and Jones C.B., editors. *VDM'87 : VDM - A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Boo86] Booch G. Object-oriented development. *IEEE Transactions on Software Engineering*, 12:211–221, 1986.

[Bra89] Brady M. *Robotics Science.* The MIT Press, 1989.

[Bud91] Budd T. *An Introduction to Object-Oriented Programming.* Addison-Wesley Publishing Company, 1991.

[BuG80] Burstall R.M. and Goguen J. The semantics of clear, a specification language. In Dines Bjørner, editor, *Abstract Software Specifications, Proceedings 1979.* Springer-Verlag, LNCS 86, 1980.

[CDD90] Carrington D., Duke D., Duke R., King P., Rose G., and Smith G. Object-Z : An object-oriented extension to Z. In *Formal Description Techniques, II*, pages 281–296. North Holland, 1990.

[CHJ86] Cohen B., Harwood W.T., and Jackson M.I. *The Specification of Complex Systems.* Addison-Wesley Publishing Company, 1986.

[CRN90] Computing Research News. The News Journal of Computing Research Association, 1990.

[Chr89] Christiansen A.D. A framework for specifying robotic agents. Technical Report CMU-CS-89-155, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1989.

[Cox86] Cox B.J. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[Cra89] Craig J.J. *Introduction to Robotics.* Addison Wesley Publishing Company, 1989.

[FrMS88] Froome P. and Monahan B. The role of mathematically formal methods in the development and assessment of safety-critical systems. *Microprocessors and Microsystems (UK)*, 12(10):539,546, 1988.

[Gio90] Di Giovanni R. HOOD and Z for the development of complex software systems. In *VDM'90 : VDM and Z*, volume 428 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[Gri91] Gries D. Teaching calculation and discrimination : A more effective curriculum. *Communications of the ACM*, 34(3):44–55, 1991.

[GuH80] Guttag J.V. and Horning J.J. Formal specification as a design tool. In *Proceedings of the 7^{th} ACM Symposium on the Principles of Programming Languages*, 1980.

[HDL90] Hanna F.K., Deache N., and Longley M. Specification and verification using dependent types. *IEEE Transactions on Software Engineering*, 16(9):949–963, September 1990.

[Hay88] Hayes I. *Specification Case Studies.* Printice Hall, 1988.

[HoH87] Hoffmann C.M. and Hopcroft J.E. Simulation of physical systems from geometric models *IEEE Journal of Robotics and Automation*, RA-3(3):194-206, June, 1987.

[HoS88] Homem de Mello L.S. and Sanderson A.C. Automatic generation of mechanical assembly sequences. Technical Report CMU-RI-TR-88-19, Carnegie-Mellon University, The Robotics Institute, December 1988.

[HoS89] Homem de Mello L.S. and Sanderson A.C. A correct and complete algorithm for the generation of mechanical assembly sequences. In *IEEE International Conference on Robotics and Automation*, pages 58–61, 1989.

[HuL89] Huang Y.F. and Lee C.S.G. Precedence knowledge in feature mating operation assembly planning. In *IEEE International Conference on Robotics and Automation*, pages 216–221, 1989.

[HuL90] Huang Y.F. and Lee C.S.G. An automated assembly planning system. In *IEEE International Conference on Robotics and Automation*, pages 1594–1599, 1990.

[IEE90a] IEEE Computer, September 1990. Special Issue on Formal Methods.

[IEE90b] IEEE Software, September 1990. Special Issue on Formal Methods.

[IEE90c] IEEE Transactions on Software Engineering, September 1990. Special Issue on Formal Methods.

[Jon86] Jones C.B. *Systematic Software Development Using VDM.* Printice Hall, 1986.

[KoM90] Korson T. and McGregor J.D. Understanding object-oriented : A unifying paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[Lev90] Leveson G. N. Guest editor's introduction - formal methods in software engineering. *IEEE Transactions on Software Engineering*, 16(9):929–931, September 1990.

[LiP89] Liu Y. and Popplestone J. R. Planning for assembly from solid models. In *IEEE International Conference on Robotics and Automation*, pages 222–227, 1989.

[LiW77] Lieberman L.T. and Wesley M.A. Autopass : An automatic programming system for computer controlled assembly. *IBM Journal of Research and Development*, 21(4), July 1977.

[MeG87] Mevel A. and Gueguen T *Smalltalk-80*. Macmillan Education Ltd., 1987.

[Mey88a] Meyer B. Eiffel : A language and environment for software engineering. *The Journal of Systems and Software*, 8(3):199–246, 1988.

[Mey88b] Meyer B. *Object-Oriented Software Construction*. Printice Hall, 1988.

[Mey90] Meyer B. Lesson from the Design of the Eiffel Libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[PAB78] Popplestone J. R., Ambler A.P., and Bellos I.M. Rapt : A language for describing assemblies. *The Industrial Robot*, 5(3):131, 1978.

[PAB80] Popplestone J. R., Ambler A.P., and Bellos I.M. An interpreter for a language for describing assemblies. *Artificial Intelligence*, 14(1):79–107, January 1980.

[PAB90] Periyasamy K., Alagar V.S., and Bui T.D. Specifications for geometric primitives. Technical Report CSD-90-04, Department of Computer Science. Concordia University, Montreal, Canada, 1990.

[PGL89] Popplestone J. Robin, Grupen R.A., Liu Y., Dakin G.A., Oskard D., and Nair S. Planning for assembly with robot hands. In *Intelligent Control and Adaptive Systems - Proceedings of the SPIE*, pages 190–205, November 1989.

[PWL88] Popplestone J. Robin, Weiss R., and Liu Y. Using characteristic invariants to infer spatial relationships from old. In *IEEE International Conference on Robotics and Automation*, pages 1107–1112, 1988.

[Par65] Pars L.A. *A Treatise on Analytical Dynamics*. Heinemann Educational Ltd., London, 1965.

[Pau88] Paul Besl J. *Surfaces in Range Image Understanding*. Springer-Verlag, 1988.

[ReV83] Requicha A.A.G. and Voelcker H.B. Solid modeling : current status and research directions, *IEEE Computer Graphics and Applications*, 3(7):25 77, October, 1983.

[ReV85] Requicha A.A.G. and Voelcker H.B. Boolean operations in solid modeling : Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 3(1):30–44, January 1985.

[Req80] Requicha A.A.G. Representation of rigid solids : Theory, methods and systems. *Comupting Surveys*, 12(4):437, December 1980.

[Spi89] Spivey J.M. *The Z Notation - A Reference Manual*. Printice Hall, 1989.

[Str89] Stroustrup B. *The C++ Programming Language*. Addison-Wesley Pub lishing Company, 1989.

[Suf82] Sufrin B. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1(3), May 1982.

[ThC88] Thomas F. and Carme T. A group theoretic approach to the computation of symbolic part relations. *IEEE Journal of Robotics and Automation*, 4(3):622–634, December 1988.

[ThC89] Thomas F. and Carme T. Inferring feasible assemblies from spatial constraints. Technical Report IC-DT-1989.03, Institute of Cybernetics, Diagonal 647, 2 planta, 08028 Barcelona, Spain, June 1989.

[Til80] Tilove R.B. Set member classification : A unified approach to geometric intersection problems. *IEEE Transcations on Computers*, C29(10):874, October 1980.

[Til84] Tilove R.B. A null-object detection algorithm for constructive solid geometry. *Communications of the ACM*, 27(7):684, January 1984.

[Wol89] Wolter J.D. On the automatic generation of assembly plans. In *IEEE International Conference on Robotics and Automation*, pages 62–68, 1989.

Appendix A

Design of a Robotic Agent

class *ROBOT* (* Super Class *)

attributes

Base-coord : *TRANSFORMTION*

Robot-arm : *MANIPULATOR*

inherits

part-pf

operations

Translate-link

Rotate-link

Move-wrist-t

Move-wrist-or

Move-wrist-otr

Move-wrist-otrs

class *MANIPULATOR*

attributes

Links : *LIST[ARMTYPE]*

Joints : *LIST[JOINTTYPE]*

Gripper : *GRIPPERTYPE*

inherits

part-of *ROBOT*

operations

Translate-link

Rotate-link

Move-wrist-t

Move-wrist-or

Move-wrist-otr

Move-wrist-otrs

(* These operations implement the correspodning operations in the super class. *)

class *ARMTYPE*

attributes

Linkid : *ID-REP*

Geometry : *STRUCTURE*

inherits

part-of *MANIPULATOR*

initially assigned operations

Translate-link

Rotate-link

Move-wrist-t

Move-wrist-or

Move-wrist-otr

Move-wrist-otrs

final set of operations

Translate-the link

Rotate-the-link

class *JOINTTYPE*

attributes

inherits

part-of *MANIPULATOR, GRIPPERTYPE*

initially assigned operations

Translate-link

Rotate-link

Move-wrist-t

Move-wrist-or

Move-wrist-otr

Move-wrist-otrs

final set of operations

Check-Joint-type

Check-displacemnet

Check-rotation

class *PRISJOINT*

attributes

inherits *JOINTTYPE*

part-of

operations

class *REVOLJOINT*

attributes

inherits *JOINTTYPE*

part-of

operations

class *GRIPPERTYPE*

attributes

Wrist : *WRISTTYPE*

Finger-Grip-Joints : *MAP[FINGERTYPE,JOINTTYPE]*

inherits

part-of *MANIPULATOR*

operations

class *WRISTTYPE*

attributes

Geometry : *STRUCTURE*

inherits

part-of *GRIPPERTYPE*

operations

class *FINGERTYPE*

attributes

Fingerid : *ID-REP*

Geometry : *STRUCTURE*

inherits

part-of *GRIPPERTYPE*

operations

class *STRUCTURE*

attributes

inherits *SOLID*

part-of *ARMTYPE, WRISTTYPE, FINGERTYPE*

initially assigned operations

Translate-link

Rotate-link

Move-wrist-t

Move-wrist-or

Move-wrist-otr

Move-wrist-otrs

final set of operations

Translate-dist

Rotate-angle

Classes *TRANSFORMATION, SOLID, ID-REP* are assumed to be defined already.