THE INTELLIGENT MANUFACTURING WORKSTATION

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Subramanian Chandrasekar
June, 1999

OHIO UNIVERSITY
LIBRARY
I would like to express my sincere thanks to Prof. Robert P. Judd, my advisor for his invaluable guidance, patience and friendship. I would also like to thank the committee members, Dr. David A. Koonce, Dr. Bhavin V. Mehta and Dr. Charles M. Parks. In particular, I would like to thank Dr. David A. Koonce for the endless hours he made available for discussion and analysis. Thanks to Ann E. Shoemake for proof reading what must surely seem to be pages of mindless techno-bable and most importantly for being my friend when I needed it most. Thanks also to my friend Joseph Thomas for spiritual and menu advice, and among other things helping me deal with \TeX in a sane and non-violent manner.

I would like to dedicate this thesis to my mother Lakshmi Subramanian, my father R. Subramanian and my dear sister Radha for all their love and encouragement without which my life would not have much meaning.
# Contents

Acknowledgments iii

1 Preface 1

2 Introduction 4

  2.1 Background .................................................. 7
  2.2 The ISO 10303 standardization attempt ........................ 8
  2.3 Architectures .................................................. 9
    2.3.1 The CIM-OSA model ....................................... 10
    2.3.2 The OMA model ........................................... 11
    2.3.3 The IMDE model .......................................... 13
  2.4 Problem Statement ............................................ 16
  2.5 A Description of the problem .................................. 17
  2.6 Chapter Conclusions .......................................... 17

3 System Components 19

  3.1 PART ........................................................... 19
    3.1.1 Caveats .................................................. 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.3</td>
<td>Adding applications to the IMW</td>
<td>59</td>
</tr>
<tr>
<td>6.1.4</td>
<td>The Entity Access Layer</td>
<td>59</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Using the IMW</td>
<td>60</td>
</tr>
<tr>
<td>6.1.6</td>
<td>The Internals</td>
<td>61</td>
</tr>
<tr>
<td>6.2</td>
<td>PART</td>
<td>62</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Mapping internal keys to user-comprehensible keys</td>
<td>62</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Dispatching the requests</td>
<td>63</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Mapping Features to Operations</td>
<td>64</td>
</tr>
<tr>
<td>6.2.4</td>
<td>SQL queries</td>
<td>64</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Unit Systems</td>
<td>65</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Major TNR Routines</td>
<td>65</td>
</tr>
<tr>
<td>6.3</td>
<td>VNC</td>
<td>69</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Entity Interface Layer</td>
<td>69</td>
</tr>
<tr>
<td>6.3.2</td>
<td>VNC workspace</td>
<td>69</td>
</tr>
<tr>
<td>6.3.3</td>
<td>CLI Files</td>
<td>70</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Major TNR Routines</td>
<td>70</td>
</tr>
<tr>
<td>6.3.5</td>
<td>VNC Specific Layer</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>CA</td>
<td>73</td>
</tr>
<tr>
<td>6.4.1</td>
<td>CA Design Files</td>
<td>74</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Major TNR Routines</td>
<td>74</td>
</tr>
<tr>
<td>6.5</td>
<td>Chapter Conclusions</td>
<td>75</td>
</tr>
</tbody>
</table>
7 Conclusions

7.1 Research Goals ........................................... 77
7.2 Results ...................................................... 78
  7.2.1 Suitable Credit ........................................... 79
7.3 Future Work .................................................. 79

A UDMM schema .................................................. 85

B VNC Directory Structure .................................... 98

C PART Runtime table-space .................................. 99

D PART Environment table-space .............................. 101
List of Figures

2.1 The Object Management Architecture. ......................................... 12
2.2 The translation environment: TNR. .............................................. 15
3.1 A multiply described geometric-feature. ........................................ 25
3.2 Multiple geometric-features with common parameters. ..................... 26
3.3 Compound Operations producing multiple features. .......................... 28
3.4 PART's Tool-Assembly nomenclature. ........................................... 31
3.5 PART's class-instance relationship. ............................................... 32
3.6 A Typical VNC library ................................................................. 34
6.1 An Overview of the IMW system. .................................................. 54
6.2 Schematic of the IMW architecture. ............................................... 62
B.1 A Typical VNC target-model. ....................................................... 98
C.1 The Part Runtime table-space. ..................................................... 100
D.1 The Environment table-space. ..................................................... 102
Chapter 1

Preface

This thesis is a study of the problems associated with the successful and (reasonably) painless integration of complex software systems used in the manufacturing scenario. Increasing dependency on computer systems to effect many stages of the design and manufacturing processes has rendered a large number of these systems indispensable. This thesis involves the design and implementation of a prototype system called the Intelligent Manufacturing Workstation (IMW). The idea of a system that manages product data “behind the scenes” letting conventional software work without modification is very appealing as it causes minimal disruption to the end users of the software while providing significant value addition by doing away with duplication of effort or at the least reducing it significantly. The IMW is an attempt to allow rapid prototyping of products. This thesis incorporates a process-planner PART, a virtual visualization/simulation tool VNC and a cost estimation software Cost Advantage. These three tools have been chosen as candidate applications for the IMW as they are fairly complicated tools and they represent a resonable cross-section of the tools that are used in a typical manufacturing setup. A typical run on the process-planner is expected to yield several different ways of manufacturing a product or a set of alternative designs for the same product. The IMW effectively allows the user to
very quickly move all this data into VNC and verify that the machining operations do indeed work as intended and that correct product-geometries are manufactured. The next stage is to cost these varying methods of manufacture and decide on the most competitive design and process-plan.

Chapter 2 explores the background to computer integrated systems and concurrent engineering. A literature survey has been presented to cross-reference contemporary research on the subject. In particular, three architectures are studied to identify a possible basis for an integrated system such as the IMW. A concise problem statement is also presented with a set of design goals. This chapter concludes with some broad observations on the integration problem.

Chapter 3 discusses the applications that are a part of this prototype system. In order to make integration of applications seamless and as effective as possible it is vital that the integration system understands the internal data held by the system in sufficient detail or the standard interfaces available (if available) to manipulate this internal data. This chapter discusses certain details that were discovered as the result of extensive study of these applications with several test cases.

Chapters 4, 5 and 6 describe the actual implementation of the prototype system. The syntax of the TNR language has been designed with the explicit purpose of translating data between different domains. In essence, a TNR program is a translation map that converts data. The design goals and implementation of TNR is discussed in Chapter 4. Chapter 5 describes the Unified Data Meta Model (UDMM), a key aspect of the IMDE architecture, that was developed for this implementation. Chapter 6
actually describes all of the TNR programs that were written to implement the IMW. The final chapter, Chapter 7 summarizes the design goals of the IMW project, makes observations and draws a few conclusions from the implementation. A methodology has been developed with specific techniques to address problems that occur during the implementation of an integration suite such as the IMW. Some future ideas to expand and enhance the IMW have also been presented.

As the IMW has been successfully developed, and does in fact satisfy all of its intended design goals one may suggest that the problem of integrating complex softwares is a solvable one. A vital design goal was to generalize the system to degree that there is very little dependency of the architecture on the applications involved (Of course, one must note that the actual data transfer/translation depends on the application). It is a fervent hope that the IMW does in fact offer a framework for future integration efforts.
Chapter 2

Introduction

The recent years have seen considerable growth in the use of highly specialized software tools for design, prototyping, simulation, and manufacturing control. As businesses gear themselves to meet increased competition under the pressure of technological breakthroughs and the opening-up of international markets there has been a thrust to achieve higher throughput by reducing design lead time. One successful method of reducing this lead time is via the integration of various design processes with downstream manufacturing control and related business process into a unified engineering system. [CYU95]

Many advantages ensue from a transition to a concurrent system from a traditional and somewhat straight-jacketed serial system; the most significant of which is the relatively defect free manufacture of a product. An important factor in the defect free manufacture of a product is the resolution of problems not anticipated during the design phase such as tool and fixture crashes. In most serial systems, these problems are reported to the design or the process engineering department, as and when they occur on the production/assembly line.

Concurrent engineering tends to reduce the incidence of such problems, as manufacturing engineers work alongside with the process and product engineers to design
the process and the product simultaneously; thus a large number of the manufacturing related issues are resolved at product design time. Integration of various applications involves the integration of tools with differing domain views sharing common system-wide primary keys such as a product identification number. With this need to share resources and information across a manufacturing environment comes heterogeneity. This may be the result of one or a combination of the following factors [Vin97].

**Legacy systems** With most manufacturing systems, decisions to procure systems are made over extended periods of time. An unfortunate side-effect is that some of the systems may become too expensive or too critical to replace. In other cases, large sums of money spent make it economically infeasible to scrap these systems until the investment has paid off.

**Engineering tradeoffs** In most concurrent manufacturing environments there exist various software tools that create and manipulate various aspects of a product design/build cycle. With the wide proliferation of specialized software tools and their popularity among users one may infer that the building of a single monolithic tool to replace these tools would be extremely difficult if not totally impossible.

**Tool preference** Most of the engineers who are the end-users of these tools typically build up preferences to certain tools for a variety of reasons, such as their suitability or their relative superiority to tools available from other vendors. These preferences (especially in traditionally serial environments) seldom depend on
the tool's ability to interact with tools used in other departments downstream in the design process.

While heterogeneity in systems has its drawbacks, it does allow the best mix of technologies. If a coherent strategy is adhered to, this may allow the building of inter-operable and portable systems.

These tools tend to store and retrieve data from locally held databases which range from commercially available DBMS databases to flat files stored in some hierarchy of directories. In a such a data rich environment there exist the following broad types of data that need to be shared across various software tools [KJP96]:

**Static data** Two functionally similar software tools which share similar domain views may share data with an agreed database/file format. For instance two CAD tools may allow the importing or the exporting of data in various standard CAD data formats (IGES, PDES, DXF, IDL, CDIF, etc.).

**Dynamic model** This data usually refers to different portions of a dynamic model that may be executed. Moving this type of data is usually a more difficult task than that of static translation, though there exists a theoretical basis provided by various researchers on the subject.

**Model overlap data** This third kind of data is the overlap between tools with very dissimilar domain views and functionality. An Example of this kind of data would be simulation control code that is written to emulate unimplemented model control software. When the actual system is built by engineers it is vital
that design decisions made during the simulation stage be upheld in the final system. In such cases what is referred to as a “model transformation” is usually required.

In addition to heterogeneity there are problems associated with the semantics used by different tools operating in different domains as the very same label may be associated with completely different data held by two different tools, and incorrect transfer of information may ensue.

2.1 Background

The basis for a successful unified engineering design environment is an integrated architecture accompanied by a systematic and dogmatic adherence to an agreed methodology. While several attempts have been to made to design and implement such an architecture, it must be realized that this problem is extremely complicated and only particular solutions are available with generic guidelines. Furthermore, an increasing array of sophisticated technology available to, and used by, highly specialized personnel in the various stages of the design/build cycle in modern industrial facilities impedes the high degree of interaction necessary between different departments in a concurrent engineering environment.

Considerable work has been done in the area of product data management and some interesting perspectives have emerged as a result of academic research and industrial projects.
Urban et al. [USR+94] discuss an architecture to manage engineering design data. Their architecture propounds the concept of a Shared Data Manager (SDM) that has a knowledge of all the meta-data in the system. The SDM coordinates and allows individual tools to interact with other tools in the design environment. They also observe the heterogeneity in the data sources and suggest a tightly-coupled active\textsuperscript{1} database.

Heim [Hei94] discusses “the fundamental components, structures and relationships needed to create representations and support manufacturing system design synthesis with numerous models developed independently”. Heim goes on to describe a prototype implementation called ENVIronment for Interactive Simulation Integration Over Networks (ENVISION) and a conceptual architecture, Model Integration Architecture (MIA). Heim cites the availability of several competing software design/modeling tools that have individual merits and limitations as the primary need for a model integration approach as opposed to the more traditional aggregate refinement and decomposition.

2.2 The ISO 10303 standardization attempt

The ISO 10303 is an International Standard for the computer-interpretable representation and exchange of product data. This standard is split into several parts dealing

\textsuperscript{1}An active database differs from a \textit{passive} one in that it actively monitors constraints on the system and alerts users on violation of these global constraints, while passive databases tend merely to uphold the integrity of the database and respond to user initiated queries.
with various aspects of the standard. Brief descriptions of some of the relevant sections are listed below:

**Part 11** Product Data Exchange using STEP (PDES) Part 11 — The EXPRESS Language Reference Manual. EXPRESS is an object-oriented schema definition language that has been proposed with the view of defining the data exchange across tools. EXPRESS, in addition to typing data and defining inheritances, also may be used to communicate constraints [Int94a].

**Part 21** Product Data Representation and Exchange Part 21 — Clear Text Encoding of the Exchange Structure. The data stored as per an EXPRESS schema needs to be communicated across tools using the Part 21 scheme for encoding data as clear text. This uses a system of unique object Id’s assigned to each object which may be included in instances of other objects that reference this object [Int94b].

### 2.3 Architectures

Some architectures have emerged as feasible frameworks to implement an integrated product data management system that allows heterogeneous data sources to cooperate and work simultaneously on a product while examining it from different *domain views* or perspectives.
2.3.1 The CIM-OSA model

"The CIM-OSA or the open-systems architecture [...] defines an integrated methodology to support all phases of a CIM system life-cycle from requirements specification, through system design, implementation, operation and maintenance" [JV90a].

The purpose of the CIM-OSA is to efficiently use information technology to effect the transfer of information across the enterprise. The CIM-OSA model proposes what is referred to as the "integration cube" where the axes of the cube represent the instantiation, derivation and generation aspects of model creation.

The CIM-OSA model may be understood from the following broad views:

**Function view** The function view describes the requirements of the system (or a *domain*\(^2\)) in terms of structure, content, behavior and control and then specifies how these requirements need be implemented and finally describes the actual implementation of the system (or domain)[JV90a].

**Information view** Information being the "fundamental ingredient"[JV90b] for integration, this view deals mainly with aspects of the associated information technology. The CIM system is heterogeneous in nature; data is stored at distinct locations [CYU95]. Secondly, as multiple design processes manipulate this data, there is a need for constraint management and enforcement. The CIM-OSA model is centered around an object-entity relationship attribute (OERA) methodology.

---

\(^2\)A *domain* is defined as a "construct [...] used to define the part of the enterprise relevant for achieving a defined set of business objectives" [JV90a].
Resource and Organization views These views covers the integrating infrastructure that consists of the following system-wide services: "the information services administrating all information required by the various application processes; the business process services scheduling the provision of resources and dispatching the execution of enterprise; the front end services representing the various manufacturing resources to the business process services in a homogeneous fashion; and the communication services being responsible for system-wide homogeneous and reliable data communication" [JV90c].

2.3.2 The OMA model

The Object Management Group (OMG) has specified the Object Management Architecture (OMA) as a means of sharing information and resources in heterogeneous environments such as manufacturing environments. OMA emphasizes the growing popularity of Object-oriented Analysis (OOA) and Object-oriented Design (OOD) for a wide variety of applications including manufacturing [GZN94]. In a critique of Object-oriented approaches, Nof [Nof94] notes the "affinity" between manufacturing and object-orientation. A major advantage of OOD is the "one-to-one correspondence" [Nof94] afforded to entities in the program/specification with those (entities) in the real world. This architecture allows the seamless integration of "Objects" through a component called the Object Request Broker (ORB) as specified by OMA [Vin97]. Common Object Request Broker Architecture (CORBA) is a key component
Figure 2.1: The Object Management Architecture.

of the OMA and specifies a transparent mechanism for transferring messages and parameters across objects. CORBA is gaining a support among users in academia and industry. CORBA represents the capability to use object oriented technology to implement very complicated and scalable distributed systems by allowing applications to communicate with each other irrespective of where these objects reside. For a schematic of the OMA model please refer to Figure 2.1.

It supports services such as the naming service (for locating objects) and a trading service (to locate capabilities). Advantages stemming out of this technology include the ability to inter-operate without any consideration of the implementation details. CORBA also supports remote invocation of objects which allows the "starting" of an object in a remote address space and the option of "re-starting" the object on some
other machine if the object exits abnormally. While CORBA seems to be gaining popularity with its users, it is also haunted by the fact that most implementations of ORBs do not completely confirm with the standard. This is compounded by the fact that this architecture is fairly new and is still under revision.

### 2.3.3 The IMDE model

Design is asynchronous in nature and various concurrency issues impact any unified development effort. In addition, design sub-processes are expected to resolve local requirements which might be conflicting and, in cases, non-functional. Most production facilities recognize prototyping as a means of reducing technical risks involved in the development of a product. With rapid advances in virtual prototyping and virtual tolerance analysis, the technical viability of the product may be ascertained with a high degree of confidence. Since design life-cycle for certain systems may extend over a period of months and, in cases, years, it is vital that (costly) delays due to design changes be kept to the minimum. The IMDE architecture is proposed to enable rapid prototyping and product development as well as a means for “systematic integration of new design tools” [KJP96] in an extensible environment.

The major components of the architecture are briefly discussed below:

**The UDMM** “The Unified Data Meta-Model (UDMM) is a meta-model that contains all the types of entities, relationships among the entities, and constraints

---

3For additional information, please see the URL http://www.omg.org/news/begin.htm
that are commonly shared among the methods used for the manufacturing systems" [KJP96]. The UDMM specifies the format for all entities and constraints, if any, that control data corresponding to the entities. The global UDMM is never instantiated, nor is it ever stored. Since only the intersections or data common to multiple tools are of interest, the UDMM defines a global set of all these intersections.

The Intelligent Interface Each tool is intimately aware of its private data. However a major obstacle to seamless integration is the lack of knowledge of other tools' private data which may correspond to the private data held by a tool. The intelligent interfaces provide a layer of abstraction over the individual application processes. Since each intelligent interface understands completely the internal data structure of the tool, it is in a position to make requests for certain portions of data to other intelligent interfaces or satisfy similar requests made to it. Each intelligent interface has an intrinsic ability to "merge" incoming data from multiple sources (other intelligent interfaces in the system) and instantiate a model in a format acceptable to a particular tool. Any inconsistencies detected in the "merge" process are reported to the intelligent interfaces that are inconsistent (with each other) so that they may be resolved by human intervention. A key component of the intelligent interface is a translation environment (see Figure 2.2) that translates data from one format to another.
The Link Manager Since each intelligent interface is responsible for maintaining a clearly demarcated portion of the UDMM there is a need for a global process to maintain and resolve "links" in between the different intelligent interfaces. This is the responsibility of the link manager. An intelligent interface exports all the links to objects maintained by it to the the link manager which in turn exports links to objects to the environment. The links are also expected to track and notify the other intelligent interfaces of changes that could potentially affect them.

Message Board Any constraints, links or errors that are un-resolvable by the architecture are posted on a global message board providing the (human) designer with ultimate control over the domain and the model.
2.4 Problem Statement

The aim of this thesis is to integrate process-planning software as a partial implementation of the IMDE. This thesis will accomplish the following goals:

- Develop a working syntax for a language to be used for integration. This language (henceforth TNR, an abbreviation for "templates and rules" where templates and rules are the building blocks in this language) is being developed at the Ohio University's Center for Advanced Software Systems Integration (CASSI) and an initial, albeit stable, version of TNR will be used for the implementation of most of this thesis.

- Develop a translator to convert data from a process-planner and export it into the UDMM. The process-planner used is ICEM Technologies' PART (henceforth PART).

- A second part of the translation is to convert this data from the UDMM to a format acceptable by two packages. The first of these is a Cost Estimation Package called "Cost Advantage" (henceforth CA). The other tool is a prototyping tool "Virtual NC" (henceforth VNC) that allows the proving of NC programs on a model. This tool allows, beyond tool-path verification, the identification of collision and tool-crashes. Some rudimentary dimension checks may be performed on a virtually manufactured component.

\footnote{Further information regarding CASSI may be obtained at the URL http://webise.ent.ohiou.edu/~cassi/}
2.5 A Description of the problem

PART stores all of its internal data in an ORACLE database. Most of the extraction of data from PART's database is done using straight SQL queries. TNR supports SQL and the concept of bind variables. An EXPRESS-based toolkit with a C API (SDAI) is available from STEP Tools, Inc. This API is used by TNR and an opaque Express Query Language (henceforth EQL) is available from within TNR to manipulate an object-oriented database. The result of the queries, i.e. the data stored by PART, is to be stored in an EXPRESS database. This EXPRESS database will then be converted to a file conforming to the ISO 10303 Part 21 format, as per the UDMM schema, and made available to other tools in the manufacturing environment.

This data stored in an EXPRESS database will be pulled in by interfaces to VNC and CA (also to be implemented in TNR) using EQL queries and converted into formats appropriate for CA and VNC. VNC uses a custom format as does CA.

2.6 Chapter Conclusions

The problem of integrating diverse applications is non-trivial and many attempts have been made to try and solve it. The most common and, to a degree, successful, method has been the integrated design of a "super-tool" that satisfies all the requirements of a particular design aspect. There is no need for the conversion of formats as the tools have been written with a view of exchanging data and, in fact, the output from the preceding tool is the logical input to the succeeding one. The costs associated with
this kind of integration arise from the rather undesirable requirement of tying down all design and development efforts to a single vendor or some sort of consortium of vendors.

This method usually fails to solve the problem of enterprise-wide integration as very few vendors specialize in enterprise-wide integration suites and in cases this solution may prove to be prohibitively expensive or may be too generic to fit the specifications of a particular site. Another option is to design and frame an architecture that can allow the use of applications that have little or no knowledge of other applications in the system, with the architecture supporting all of the integration needs with specific services. These services would include:

- A model retrieval service where models may be imported into the domain-view of any particular tool.

- A model export service that allows certain models processed by the specific tool to be marked as ready for export.

- A model consistency verification service that allows a model created by a particular tool to be checked for consistency against the views of various other tools in the design environment.
Chapter 3

System Components

This Chapter discusses the proposed system. The major sections in this chapter examine each of the tools (PART, VNC, CA) and try to describe the local data held by the tools.

3.1 PART

PART stores its data in three logical table spaces named Runtime, Environment and Method. The three table-spaces are logical divisions of information, in terms of the content, as classified according to tool-usage. An attempt has been made to represent these table-spaces using the IDEF1X methodology [Com94]. This section briefly describes the process-planner and then explores the IDEF1X representations.

Diagrams of the relevant sections of this database are attached as appendices C and D. Before any further discussion of the IDEF1X representations the following limitations/caveats must be carefully noted.
3.1.1 Caveats

- PART disables all referential integrity constraints on the tables for reasons of efficiency. This means in there are no keys in a database containing over 300 individual tables. This particular scheme has necessitated that a sizeable number of assumptions be made. Wherever possible, such assumptions are documented on the IDEF1X diagram as well in the following text. It must also be appreciated that some aspects of the diagram especially the cardinality of certain relationships are educated guesses.

- PART uses a class-description table to describe a certain class. For instance, a MACHINE_TOOL_CLASS_DESCR.S table will point to a pair of tables via its DESCRIPTION_NAME and IMAGE_TABLE_NAME attributes. For instance, one of the tuples in this table may contain MILLING_MACHINES, and MILLING_MACHINE_IMAGES as values for the said attributes. These tables (MILLING_MACHINES and MILLING_MACHINE_IMAGES respectively) in turn contain instance-specific parameters pertaining to an instance of a milling-machine. Such a scheme implemented to (possibly) accommodate the addition of new classes of information is extremely hard to represent using the IDEF1X methodology. In this thesis a convention of showing a representative table on the top of the stack of possible tables has been adopted.

- In addition, some tables appear in multiple table-spaces; however these tables are not exact replicas of each other in terms of attributes or their values. A
notable example is the TOOL.ASSEMBLIES table. The table on the Runtime table-space is actually a loose-combination of the TOOL.ASSEMBLIES and the TOOL_SETS tables found the Environment table-spaces. Similar optimizations occur at various places.

For reasons of clarity, the word component is used in preference to part in the ensuing discussions to refer to a manufactured product.

3.1.2 The startpart tool

The first of these table-spaces RUNTIME contains information pertinent to the process-planner startpart. This tool allows the user to accomplish the following:

- Create a Product: A product may be created from ACIS file-format models. PART contains a built-in translator that accepts files in several standard formats and converts it into a PART model. Once the models for a blank and the component have been created and imported into PART, the user may select a combination of a component model, a blank model and a material to define a product. In addition, several similar products may be simultaneously processed as a product-group to optimize the usage of machine setups. Most of the data pertaining to product-groups does not lie in the intersection of the domain views of the tools in the systems.

- Recognize features on the product: The very first step towards the process-planning of a product is the recognition of features on the component model.
PART takes its best guess at recognizing features, and reports all the features it could recognize at the end of this process. In certain cases, features are incorrectly recognized or missed. For these cases, the user may correct these errors by manually selecting these features.

PART groups its features in an hierarchical fashion, however the UDMM does not support any structures to preserve this hierarchy. Each feature is assigned an *atomic-feature-id* which is unique for a certain product and model combination. Since the blank is of little interest to the other tools, features on the blank are not exported. Each feature is of a certain *atomic-feature-type* and has a specific *description* associated with them. Each particular atomic-feature-type has a certain set of parameters. These parameters are gleaned from the model and stored in various parameter tables.

- Decide on the setups needed to create the recognized features: A setup is defined as "a collection of features or operations in a single product, machined in a single clamping orientation"[par]. This process of deciding the *operations* needed to make each feature and grouping them into *setups* is first performed by PART based on certain rules which PART refers to as *methods*.

Further, PART also attempts to group these operations into *setup-groups* if multiple products are being processed in a product-group. The user can override most of these decisions and groupings.
• Decide on the machine to be used for making the component: At this stage the user must identify a specific *machine-tool* to process the component. All machine specific calculations are performed at this stage.

• Decide on the fixturing for the product: Fixturing decisions may be made at this point and the user may specify and construct fixtures for the given component.

• Decide on the specific methods for making the features: At this stage the rule-based process-planner tries to decide on specific tools for each of the operations.

• Plot the tool-paths for each of the operations: A typical CAM interface available at this stage allows the user to override any decisions that were taken by PART. Successful conclusion of this stage produces the APT file which may be run through various post-processors to generate correct NC code.

• Generate NC output: The APT files generated in the earlier stage may be run through a default post-processor. However due to observations made from several trial runs, it was decided that custom post-processors were needed to generate code for several machines available at Ohio University.
3.1.3 The PART Table-spaces

This section discusses briefly the three table-spaces that make up PART's database.

3.1.3.1 Runtime

This section explains in brief, some of the tables in the Runtime table-space and the relationships that exist in between them. Almost all of the data in the Runtime database is uniquely identified by a product. As a product is defined by two models (blank geometry and finished component geometry), the identifying key of the model identifies tables that are related to geometry. The geometry on a model is differentiated into two major constituents, viz. parameters and a description. The corresponding tables are named ATOMIC_FEATURE.DESCRIPTIONS and ATOMICFEATURES respectively.

The same geometric feature can be recognized as multiple features descriptions. Consider for instance, a slot that is defined by the location of the origin for the coordinate system, the orientation of the coordinate system and the parameters length, breadth and depth as illustrated in Figure 3.1.

The same slot could be represented in a multiple number of ways depending on the choice of the origin and the orientation of the coordinate system, with the length, breadth and depth parameters assuming different values. The feature-recognition module will recognize as many possible representations of a geometric feature as possible to afford the maximum number of options to the process-planner. Similarly, it is also possible that a large number of features share the parameters and differ only
in terms of the location of the feature. An example of this scenario would be holes drilled into the component that have a common diameter and depth, but located at different positions and orientations on the components. Figure 3.2 illustrates such cases.

PART separates a geometric-feature into a set of parameters and a description that defines the location and orientation of that particular feature. Each atomic-feature (in table ATOMIC_FEATURE) points to an ATOMIC_FEATURE_DESCRIPTION (via the AFD_ID). The atomic_feature_description identifies the set of parameters that
is applicable for that particular atomic feature. However to (possibly) accommodate the addition of new parameters to a feature, a type identifies the table (via the ATOMIC_TYPES table in the environment table-space) and a parameter id (AP_ID) identifies the tuple that contains data pertaining to this instance of the feature. In addition a parent-child hierarchy of features is also maintained using the COMPOUND_PARENTS table. In our first example, i.e. the case where the same feature is represented multiply there would exist two tuples that define the slot in the ATOMIC_FEATURES table which share a common AF_ID but have differing
AFD_ID s. The AFD_ID in turn would identify a set of parameters by means of the AP_ID (the table is identified the TYPE in the ATOMIC_TYPES table). In our second case, where multiple holes share the same common parameters, multiple features would point to the same atomic_feature_description. The COMPOUND_PARENTS hierarchy allows the identification of features which are created as a result of other features such as stepped-holes etc.

Setups and operations are identified by the product but do not identify each other as it is possible to have one of these without the other depending on the (PART) scenario\(^1\) is used. MMTS_INTERMEDIATE OPERATIONS is a table that stores information regarding the intermediate features that are obtained in the process of "making" a geometric feature.

Consider for instance, the machining of a stepped hole with a smaller (lower) hole with diameter \(d\) and length \(l\) and a larger (upper) hole with diameter \(D\) and length \(L\) as illustrated in Figure 3.3. These features may be produced in one of two ways, viz. drill a Hole with diameter \(D\) and length \(L\) and then drill the smaller hole with diameter \(d\) and length \(l\) or drill a hole of diameter \(d\) and length \(l + L\) and then for a depth of \(L\) remove the material to make the larger hole of Diameter \(D\) by end-milling or similar means. In the later case multiple features are produced as the result of multiple operations. The sequence of operations may look like the following:

- Drill inner hole of diameter \(d\) and length \(L + l\)

\(^1\)A scenario in PART identifies the order in which process-planning for a component is done.
Figure 3.3: Compound Operations producing multiple features.
• Mill a ring shaped sub-volume of outer diameter \( D \), inner diameter \( d \) and of length \( L \).

For convenience of notation let us refer to the features as \( f0 \) the flat surface, \( f1 \) (the lower hole) and \( f2 \) (the upper hole). Also, let use refer to the two operations as \( o1 \) (the drilling operation) and \( o2 \) (the milling operation). For this case PART will also create the intermediate feature \( i1 \) which has a diameter of \( d \) and a length of \( l + L \) which is produced as the result of \( o1 \). The table \texttt{MMTS\_OPERATIONS} contains operations performed in the manufacture of components as identified by the \texttt{PRODUCT\_ID} and an \texttt{OPERATION\_ID}. A \texttt{SEQUENCE\_NR} attribute correctly orders the operations within a setup. Now for each operation contained in the \texttt{MMTS\_OPERATIONS} table, there exists a less-worked-feature and a more-worked-feature which are stored in \texttt{MMTS\_LESS\_WORKED\_FEATURES} and \texttt{MMTS\_MORE\_WORKED\_FEATURES} respectively. The \texttt{FEATURE\_ID} identifies the atomic-feature (positive values for the key) or the intermediate feature stored in the \texttt{MMTS\_INTERMEDIATE\_FEATURES} table (the key is a negative number – i.e. in our case \( i1 \) will be identified by a negative number). A brief partial abstract of the data has been provided in Figure 3.3 to indicate what kind of data is contained in the \texttt{MMTS\_LESS\_WORKED\_FEATURES} (lwf) and \texttt{MMTS\_MORE\_WORKED\_FEATURES} (mwf).

As mentioned earlier, the \texttt{TOOL\_ASSEMBLIES} table in the runtime is a loose collection of data from various tables in the environment. This table is used to store the user suggested overrides for the default values for the various attributes
in this table; the defaults themselves taken from various tables in the environment.

This is the result of one of the many optimizations which involves a deliberate de-
normalization of the database.

3.1.3.2 Environment

The environment tables contain (relatively) static data for use by the process-planner.
The table-space is largely organized into several classes of information and speciali-
izations of these classes. The classes include machine-tools, machine-ends, tool-ends,
shanks and cutters. A specialization of any of these classes may be created by a user
of the startsapart tool. Experimentation using the tool suggests that a new class is
represented by:

1. An entry in the class-description table

2. A table whose name corresponds to the DESCRIPTION attribute in the class-
description table that contains the various parameters of this specialization.

Tool-assemblies in PART are represented by means of a set of relations described
in the ASSEMBLY_RELATIONS table. A discussion of the assembly-relations is
beyond the scope of this thesis. Assembly relations define the relations between
the various parts of a tool assembly, namely holders, adapters, cutting-tools and
optionally inserts.

Tool-Assemblies are stored in a rather involved fashion by PART to facilitate
the addition of more tools into the system and modification/removal of existing
tools. With reference to Figure 3.4, there are tables that define the various classes of machine-tools, machine-ends, tool-ends, shank-ends, cutters, fixture-ends and inserts.

Each of these class description tables declare the following fields\(^2\). For a schematic diagram of this class-instance hierarchy please refer to Figure 3.5

**DESCRIPTION_NAME** This field points to the oracle table-name that contains specifics pertaining to that instance.

**IMAGE_TABLE_NAME** This field points to the table that contains information pertaining to the images of the specific instance.

\(^2\)Due to a limitation of IDEFIX (all attributes are atomic and distinct in a domain) is forbidden to use the same attribute in two different tables. In Appendix D all class-descriptions these fields are prefixed by the initials of the table.
Figure 3.5: PART's class-instance relationship.
For instance, in the table CUTTER_CLASS_DESCRIPTIONS (table-space: Environment) contains a tuple that has BALL_END_MILLS as the DESCRIPTION_NAME and CUTTER_IMAGES as the IMAGE_TABLE_NAME. All parameters for a specific instance of the ball-end-mill class are stored in the table BALL_END_MILLS. Any images associated with this particular instance would be stored in the table CUTTER_IMAGES. These class tables are created from the system administration tool whenever a new class is added to the system.

The methods table-space contains information regarding the rules that need be used to do process-planning for the component and is of little interest to us. There is only one piece of information located in this table-space (operation name) that is used in this thesis and hence this table-space is not discussed any further.

3.2 Virtual NC

VNC is a prototyping tool that allows the simulation of NC programs on a virtual machine. The user creates a set of geometric models for the machine being simulated. Motion, both translational and rotational, is assigned to the model and limits are placed on travel over, and, around the various axes. An emulator for the NC controller is written in a special language[Den] called MIMIC to manipulate the machine-model to respond to NC instructions. At this point blanks and tool-sets may be created using a the built-in CAD modeling tool and NC programs may be run to virtually machine a component from the blank.
VNC may be used to prove programs generated as a result of process-planning done using PART. Once a geometric model has been built for a specific NC machine and kinematics have been assigned to the various axes the user may create a blank that has been processed by PART. The NC programs generated using PART to convert this blank model into the finished component can be loaded on this virtual machine.

VNC stores all of its data in a hierarchy of directories. On an SGI IRIX system a sample directory structure may look something as in Figure 3.6.

The VNC geometric models are maintained in a custom data format referred to as the v11 (version 11) format. In addition Deneb also specifies a version 1011 format for files generated by external programs. While the recommended method of transferring geometric data to VNC from external applications is via the version 1011 format, this
thesis uses the version 11 format. The difference in between the two formats lies in the level of error checking performed in both the cases. A version 11 format is assumed “to conform to Deneb requirements and virtually no error checking is performed to maximize system performance” [Den]. A version 1011 file is checked for duplicate vertices, convex polygons etc. However when VNC reads in a version 1011 file, it expects a certain amount of user input which make it difficult to make the transfer of data from PART to VNC seamless and as non-interactive as possible. While the non-interactive nature of this transfer might sound restrictive these transfers are expected to be routine. Furthermore this data is already available within the domain of the other tools and there should be no reason that the user be made to duplicate effort to re-enter or confirm this data.

This reasoning is used throughout this implementation and the attempt is always to make most operations as “automatic” as possible. The system attempts to try and avoid dependence on the users understanding of the other tools in the system.

Valid concerns may be raised as to the vulnerability of the created geometric models that are “pushed” into the VNC library. In response one must carefully note that it is highly unlikely that a deterministic TNR program would perform differently during operational runs once all programming errors have been eliminated via. a rigorous process of testing the suite. Also the issue of maintaining compatibility with succeeding versions of VNC is assured as long as Deneb robotics maintains backward compatibility with models created by earlier versions of VNC.
3.2.1 Tool Assemblies

The VNC tool assemblies are defined to be sub-types of devices. To build a tool that is usable by VNC as a valid tool the following steps need be followed.

Define Geometry The user needs to create geometric models for the various parts of the tool such as holder, adapter, shank and the cutting tool.

Build the Tool Device At this point the user uses the Build menu to construct a device using the geometry created earlier. The various parts are attached to each other.

Set Device Type Once a device has been built the user changes its type to a “Tool” device. At this point it is necessary to mark one of the parts on the device as the cutting part. VNC uses the geometry of the marked part to perform computations to determine material removal. VNC can also be watch for collisions between the various non-cutting portions of the tool assembly with other parts in the vicinity. This alone represents a very significant advantage over conventional tool-path verification programs that cannot check for collisions between non-functional parts of the system.

It is possible to use the CLI (Command Line Interface) [Den] for creating and assembling these tools. Such an option allows the architecture to use defined interfaces. This thesis attempts to create the tool-assemblies on the fly from data in the UDMM using the CLI interface.
3.2.2 Machines

VNC Machines are built similar to the tool-assemblies. However, the creation of the geometries of the various parts of machine-tools and putting them together is non-trivial. It is beyond the scope of this thesis to try and build machine-tools on demand. This implementation therefore maintains a list of manually created machines. These machine-tools are copied into the current library as per the data in the UDMM. The structure of the stored libraries is designed to specifically prevent namespace clashes.

3.2.3 Workcells

A VNC Workcell is a VNC machine with the MIMIC code loaded into the machine. It is possible to load tool-sets, blanks and (NC) machine-programs on this machine now and simulate the machine operations. In addition to the machine and the associated MIMIC it is possible to save other devices (such as the workpiece, tool-changers etc.) and machine programs as a single Workcell. This architecture therefore uses workcells as a means of representing a setup. CLI is used to create these setups.

3.3 Cost Advantage

Cognition's CA, "is a knowledge-based software system that provides expert-level design guidance and can analyze manufacturing alternatives, producibility, and predictive cost analysis"[RS].
**Cost Models** are built using the CA-Modeler tool. CA treats as costs being driven by the following cost-drivers:

1. Process
2. Material
3. Feature
4. Parts

In essence, a hierarchical tree of specializations is defined in the cost-model which is used by cost-advantage to generate cost-notes for specific components. CA is built on top of the Allegro CL 4.3 version of common lisp. It stores the cost-note internally as a lisp data-structure. The following is a map of a typical session for a run on the IMW. A cost-model has been developed for representing a hierarchy of specialization of the cost-drivers and appropriate cost calculations have been programmed into this model.

**Load Cost-Model** It is necessary to load a cost-model when creating a new cost-note to allow specializations from the hierarchy defined in the cost-model to be used. This cost-model allows a set of specializations for the material and the features to be created.

**Component/Assembly** CA divides products into components and assemblies, where an assembly is made up of components.
Material For a component, a suitable material type may be chosen along with major dimensions for the blank. This allows CA to consider material costs when performing costing for the component.

Process A process may be selected to specify costs involved with the process such as machine and operator rates.

Feature Much of the cost of a component is associated with the value-addition to a component by the “making” of features. A very complicated hierarchy of features have been modeled by the cost-model. Each of these features has its own set of parameters. Some of these parameters are computed from the values of other parameters. These features are modeled after features as recognized by the PART and the non-computed parameters mirror the parameters held by PART. Great care has been taken to preserve the same names in the models of features held by CA.

This thesis tries to create a cost-note using the cost-model created for IMW. Data read in from PART via the IMDE architecture is converted to CA features. Setups are entered into the cost-note to represent the setups. While a sizeable amount of data is transferred “automatically”, it must be noted that there are certain gaps in information which can be ultimately only filled in by a user of the tool. However the use of the IMDE definitely speeds up this migration of data from tools.
3.4 Chapter Conclusions

This chapter discussed briefly, the various major components of the system, i.e. PART, VNC and CA. These tools have been chosen to be the components of this partial implementation of the IMDE as they are very typical examples of real-world tools. PART stores its internal data in an ORACLE database. SQL queries are employed to extract data from this database and store data into the STEP format[Int94b] that is used to exchange data in this implementation. CA is built on top a commercial LISP implementation. Hence its data-files are lisp data-structures. A rigorous study of the format followed by the use of the various text-formating primitives available within TNR can produce compatible data files. VNC uses a hierarchy of directories and files to store its data files. Where specifications are available, they are used to create valid and conforming models of geometry(wherever possible). In other places, the CLI is employed as it is the recommended way of interacting with VNC.
Chapter 4

Translation environment

This chapter describes the design goals and syntax structure of TNR. While TNR is a language that is still under development, it must be understood that the syntax used in this implementation is fairly stable and only minor changes need be anticipated. One area where changes could be anticipated is in the functions and statements relating to EQL which is at its inception.

4.1 Design Goals

TNR was written to address the following design goals:

1. It is observed that a number of tools store information in custom formats which may or may not be easily read and changed. Vendor motivations range from information hiding to various efficiency related reasons. However it is of paramount importance for the effective functioning of an architecture such as the IMDE, or for that matter any product-data management system, that it (the system) is able to read and understand the state of the tool so that the necessary state-modifications may be instituted. It may be also noted that most tools do provide a programatic interface to access internal data.
TNR provides a plethora of input and output options. Its is possible to read/write files, issue SQL-queries, communicate over networks and connect to programs via (AF_UNIX) pipes in addition to store, retrieve, and query very complex object-oriented data structures.

2. Since the data in a manufacturing system may be very conveniently modeled using object-oriented schemas, it is necessary that there exist means to conveniently store and retrieve data in an object-oriented fashion. EQL is a query language that allows very complicated queries to be made on an EXPRESS based data-basing system that is built into TNR itself.

3. A significant amount of data handling that is done involves the creation of files in very tightly specified formats. Templates may be created using existing data files, and parameters may be added to instantiate new data files, while rules allow the decoding of formatted data-files to yield the parameters themselves. A large number of pattern-matching constructs are built into the syntax of TNR to allow rapid traversal of input data.

4. An important design goal was that the language should be sufficiently "high-level" while providing all the functionality of traditional languages like C and utilities such as Nawk(1) TNR takes care of low-level issues such as memory-allocation and provides for reliable operation of TNR programs.
4.2 Syntax Structure

TNR syntax may be grouped into the following broad categories:

**Variables** TNR supports two types of variables – scalars and vectors. TNR does not allow explicit typing of scalar. i.e. the same scalar may exist as a *real*, *integer*, or a *string* in its lifetime. TNR expects however that a scalar be convertible into an acceptable form for use in a particular expression. For instance, a value that is contained as a string when used in a numeric expression must be convertible to a numeric form. To attempt a numeric operation on a string that does not evaluate into a numeric value, will flag a runtime error.

Vectors are defined as sequences of scalars that may be grown or shrunk on demand. The elements of a vector are accessed in a manner similar to arrays in C using the square-brackets operator. It must also be noted that it is not mandatory that vectors be completely filled in, i.e. it is valid that “holes” exist in between the scalar contents of a vector.

Vectors, like scalars, are not typed. The various elements of the vector may dynamically change their type depending on the context of their use. The same vector may simultaneously contain a combination of string, real, integer, and uninitialized scalars as its members. An uninitialized scalar takes on the UN_INIT value, an evaluation of which is defined as a runtime error. TNR does support a function to test if a scalar has been initialized (value is not UN_INIT).
**Templates and Rules** Templates and Rules are the procedures that are used to implement "write" and "read" contexts in TNR. In a template context file-objects are treated as output devices while in a rule context file-objects are treated as input devices. Much of this distinction is historic and may eventually disappear with succeeding versions of TNR. Templates and rules allow a logical separation of the tasks of information retrieval and that of information insertion into tools in the IMW system.

It is currently a runtime error to issue template context functions such as `printf` and `print` in a rule, the reasoning being the current (open) file is being read from and any attempt to write to it leads to very peculiar conditions that are difficult to handle and debug. Likewise it is also a runtime error to attempt a read in a template context as the current file is being used as an output device. Sockets and pipes are treated as file objects.

Database objects, however, present a rather peculiar conundrum. While to disallow reads and writes into database objects depending on the context may seem to be the logical thing to do, such measures greatly deteriorate the quality of the queries that may be issued. Hence both `select` as well as `insert` type of queries are allowed in templates and rules.

Some justification can be offered for this design decision as database-objects do exhibit a high degree of distinction from file-objects, in that, they are operated on by means of complex query languages.
Statements TNR supports traditional C[KR88] like statements. The if ... else ...
for(;;) ... , do { ... } while(), while() { ...} and the
switch () { case:...default: } statements behave similar to the respec-
tive C constructs except for the following differences.

- The for statement does not support the comma operator. While the
  comma operator is convenient syntactically, its absence does not abstract
  any functionality from TNR.

- The switch statement incorporates two changes over the traditional C
  version. The first distinction is that the case labels may use expressions
  which are evaluated and compared to the discriminating expression. A
  second modification is the addition of the continue statement to the
  cases which asks TNR to continue checking from the next case onwards for
  a match.

Text Manipulation TNR supports a rich array of text manipulation functions.
Some of these functions have been adapted from Nawk(1) and C and have
been greatly enhanced. The functions printf and scanf have been im-
plemented with almost their complete functionality (some format specifications
like %p do note make any sense in TNR and have been omitted from this
implementation). TNR also borrows the split function from Nawk(1) and
greatly enhances it. An inverse of this function, merge has also been provided.
TNR assumes a default field separator (ASCII \034, 'FS'), however this default
may be overridden locally in the particular function-call or may be modified to take on a new value. It is possible to build comprehensive parsers with suitable combinations of split, splitd, sub, gsub and merge.

**Numerical Functions** TNR supports atan2, cos, exp, int, log, pow, sin, sqrt, and tan as built-in functions. These functions have their usual meanings and the arguments are automatically converted to numeric values with type checking before the actual calculation.

**System** TNR supports the following system-related functions: fork, pipe, kill and env. Fork behaves as system(3S) library function. It also may be noted that kill can only deliver the **SIGKILL** signal to the said process.

**Database** TNR allows the opening of standard relational databases that support SQL. The TNR function db takes two parameters. The first parameter is the type of RDBMS and the second parameter is an initialization string. Currently TNR supports ORACLE, however additions may be made trivially.

TNR allows a STEP object-oriented database to be created/opened and queried using the custom query language EQL. EQL is a very powerful query language that is used to query the STEP database sub-system that is embedded into TNR.
4.3 Chapter Conclusions

Over a considerable period of testing and refinement TNR has proved itself to be a very powerful and versatile tool for the writing of translators. TNR provides most of the desired functionality available in several languages in one easy to use package. The reliability and the relatively high-level design of the TNR syntax relieves a programmer of much of the tedium associated with this kind of translation. TNR bears a strong resemblance to C which is a great aid to learning to use the language. While TNR does compromise on efficiency with a preference to functionality, this tradeoff is well justified as TNR is not expected to be used to implement time-critical code.

EQL initially designed for simplifying operations on STEP files, has also proven itself to be a very rich compliment to TNR to store internal data. TNR variables (scalars and vectors) are expected to be used for storing simplistic data. Any data that requires a higher degree of structure is expected to be stored and manipulated using EQL from within TNR programs.
Chapter 5

Unified Data Meta Model

The UDMM is an understanding of all the different types of data that could potentially exist in the IMW system. The current version of the UDMM is defined using EXPRESS, an object-oriented schema representation language. For a listing of the UDMM defined using the EXPRESS language please refer to Appendix A. This chapter discusses the various classes of information that exist in the IMW. All information in the UDMM may be categorized into the following three types.

1. Independent Entities: These entities model the information pertaining to objects that exist in the real world and identifiable by some particular key. An effort has been made in the IMW to provide access to these entities as they may be “visible” to other applications.

2. Dependent Entities: These are entities which have no uniqueness characteristic associated with them, i.e. multiple instance of these entities may exist in the same local-store which contain identical information.

The following sections of this chapter list and briefly describe the various entities that occur within the UDMM.
5.1 Independent Entities

WorkstationClass This represents an instance of a workstation.

MachineProgram This is an independent entity that represents a NC machine program that may be used on a certain workstation.

OperatorClass This is used to represent operator skill levels and a corresponding hourly rate.

TooledWorkstation This represents a workstation that has been completely tooled and is ready for some set of operations.

WorkCellClass This represents a grouping of workstations that may be used to produce a certain class of products.

PortableToolClass This represents miscellaneous portable tools (such as screwdrivers, ratchets, hex-keys etc.)

ToolAssembly This represents a tool-assembly. A tool-assembly may be made of one or more tool-elements and/or sub-assemblies. An Example of this would be a milling tool-assembly made of a holder (tool-element), an adapter (tool-element) and a cutter-body with inserts (taken together a sub-assembly). This is a fairly comprehensive structure that allows trees of assemblies to put together. In addition, location information may be embedded into the tool-assemblies to locate the various tool-elements and sub-assemblies with respect to each other.
Trivial parts, such as nuts, bolts etc., may be represented by specifying a number of occurrences (without the location of each individual item).

**ToolElement** This represents a "branch" in the tool-assembly "tree".

**PartAssembly** Part-assemblies are structured similar to tool-assemblies and can accommodate the same degree of complexity. In addition part-assemblies also contain the information relating to the geometry and manufacture of that particular part.

**PartElement** A PartElement represents an atomic component of the part-assembly that is produced as the result of a sequence of operations.

**Feature** A feature is the representation of a geometric (possibly machined) feature.

**Operation** An operation is a machining or assembly operation that is performed on a part or a part-assembly to produce a feature or a sub-assembly.

**Setup** A setup is a grouping of all the various operations that would be done in a single setting.

### 5.2 Dependent Entities

**FeatureDescription** This is a description of a geometric feature. (one may note that a feature maybe multiply described see the Runtime section on Page 24)
**FeatureParameter** This is a set of parameters that pertain to a particular feature.

(It is possible that several features share a common set of parameters)

**MachinedFeatureLink** This entity represents a grouping of operations that produce one or many features.

**Parameter** This is the base class for all Parameters.

**IntegerParameter** A specialization of Parameter to accommodate Integer data.

**TextParameter** A specialization of Parameter to accommodate Text data.

**RealParameter** A specialization of Parameter to accommodate Real data.

**CurrencyParameter** This is the base class for all Monetary data.

**USDollar** A specialization of CurrencyParameter to accommodate monetary data in USD.

**Speed** This is the base class for all speeds.

**RevPerSec** A specialization of Speed to handle speeds in rev. per sec.

**MeterPerSec** A specialization of Speed to handle speeds in meter per sec.

**Feed** This is the base class for all feeds.

**MeterPerSec** A specialization of Feed to handle feed rates in meter per sec.

**MeterPerRev** A specialization of Feed to handle feed rates in meter per rev.
Model  A container class that represents a file on some disk system.

LocationType  A SELECT type that may be either a LocationName or a LocationPosition.

LocationName  An entity that represents a named location (for instance, a tag point in VNC may be named as “tool_change_point”).

LocationPosition  An entity that represents a location in 3-D space in terms of its location and orientation.

PartialOrder  An entity that allows the ordering of a SequentialType.

SequentialType  A SELECT type that defines all the ordered types in the IMW system.

5.3 Chapter Conclusions

Considerable effort has been taken to ensure that the system is adaptable to changing environments with no or minimal impact on the architecture. While the implementation of merge is beyond the scope of this thesis, every attempt has been made to make the UDMM context-insensitive so that the merging of entities need make no special assumptions. While additions may be anticipated the UDMM to accommodate more specializations and further classes, this version of the UDMM does contain a reasonable understanding of the manufacturing domain and provides the data-structures necessary to model the data contained within.
Chapter 6

System Description

The last two chapters described the TNR translation-environment and the UDMM. This chapter describes the actual implementation of the translators that effect the transfer of information from within PART to a local-store and from the (current) local-store into the domains of the applications VNC and CA.

Since the scope of the thesis is limited to one-directional transfer of information from PART to CA and PART to VNC, they (the translators) have been designed for unidirectional transfer. However, these translators are comprehensive examples that deal with both the conversion of information from the UDMM to local formats and vice-versa.

These translators augmented by the understanding gained into the specific tools (PART, VNC, and CA), make the implementation of bidirectional translators for each of the tools fairly straightforward. During the design of the translators a special emphasis has been made to keep the translation as generic as possible; new translators can be built by extending or modifying existing code.
6.1 System Overview

This section provides an overview of the entire prototype system built for the thesis.

A schematic of the IMW is given in Figure 6.1.

![Diagram of IMW system](image)

Figure 6.1: An Overview of the IMW system.

6.1.1 The User Interface

This layer provides a user-interface to the IMW and allows the user make certain requests of the system. The user-interface consists of two major components that share a clean boundary. The Graphical User Interface (henceforth GUI) is written in Java and it communicates to a TNR counterpart that executes the requests. A TNR program may in addition to answering requests, ask for confirmations, request for additional data from the user and make status reports to the User. The GUI handles
these requests from the TNR programs asynchronously. Communication between the two components occurs through the following means:

1. Command line Parameters: The GUI passes a sizable amount of information to the TNR interface as command line parameters that describe the specific "service" that is requested by the user. There are four parameters that may be supplied to the main() template in the main.tnr file.

   **Request** This a delimited request that describes the nature of the request, the application that is expected to transact the request and the applicable UDMM Entity. In this implementation, there occur the following major classifications of these requests:

   **List** These rules return a list of available keys (within the application) to the user. As noted earlier, these keys are the counter parts of internally used keys which are not user comprehensible.

   **Clear** These are used to clear portions of the current local-store. However due to certain limitations in EQL, it is currently only possible to clear an entire local-store.

   **Delete** This request allows the user to delete a Model used by an application. Currently, only VNC and CA support the delete command.

   **Export** Currently this is only implemented for PART and this retrieves an entity into the current local-store. The GUI (i.e. the user) supplies a key which is converted into the appropriate internal key (a primary key
to some table in the PART database. In the future similar mechanisms may be implemented for VNC and CA as well.

**Import** Currently implemented for VNC and CA, this converts the data in the current local-store to valid models that exist in the domain of the respective application.

More service categories may be added as the IMW is expanded as the result of future work.

**Key(s)** This is an optional parameter that contains a delimited set of user-supplied keys. This allows for a request to be simultaneously (and somewhat efficiently) executed on a large number of entities.

**Step File** This optional parameter names local-store (step-file) to be used for this particular request. It must be noted that all “write” transactions append to the specified step-file. A special request exists for explicitly clearing the current local-store.

**Target Model** This optional parameter names the target application-model for use by the application. The name of the target-model is modified by the application interface to adhere to a standard convention. This convention is implemented in the `mk_model_name` template of the particular application. A target model may mean different things to different applications. For instance, VNC would interpret the target-model as the name of a VNC library while CA would treat the name as the name of a Cost-Note.
2. Standard Output of TNR: TNR uses the standard output to communicate information to the GUI. All messages are prefixed by special designators that indicate the nature of the information being output-ted to GUI.

3. A Named UNIX pipe: Some of the messages sent to the GUI from TNR may be requests for confirmation or for further information necessary for the TNR routines to complete the particular request. The GUI may choose to "listen" to the results of a request made of the TNR programs. Any request made of the GUI by the TNR programs is forwarded to the user by means of appropriate dialog boxes and the user-response communicated to TNR by means of the pipe. Synchronization mechanisms built into Java (Monitors) help ensure that critical sections of data are protected against modification by multiple threads of control.

Particular care has been taken to organize this code to allow independent reorganization of both the TNR programs and GUI as long as the implementations adhere to the specified interface.

The GUI also keeps track of the current local-store (the step-file argument of main.tnr) and (if applicable) the current target-model. The GUI allows the user to create, select or remove this local-store and the target-model.
6.1.2 GUI Implementation issues

This section details the implementation structure of the GUI. It is assumed that the reader understands the Java programming language. The GUI consists of two major Frames. The first frame (code resides in the IMW object) provides access to all of the major functions available within the IMW. The options available within this frame are:

1. Export Data from the PART application
2. Launch The PART application
3. Import Data from the current local-store into the VNC application.
4. Launch the VNC application
5. Import Data from the current local-store into the CA application.
6. Launch the CA application

These are available as Menu Options on this frame. The other Menu available within this frame is the Settings Menu which executes a command that brings up a configuration file in an editor. This configuration file essentially defines the settings for the IMW system. It is necessary to source this configuration file and restart the IMW system after settings have been modified to incorporate the changes.
6.1.3 Adding applications to the IMW

The addition of a new application to the GUI requires that the Java code be modified as follows:

1. Add the MenuItem in the imw object.

2. Implement a derived class of the AppDialog class\(^1\).

3. Add the action Handler in the imw object to invoke the show method of the appropriate AppDialog object.

4. Add the appropriate TNR templates/rules that actually effect the import/export of data for that application.

6.1.4 The Entity Access Layer

This layer contains all the templates and rules that access the data pertaining to independent entities as stored in the local databases of the applications. It is seen that most entities are internally referenced by keys (for instance, id numbers) that do not make much sense to the user. Hence a user comprehensible key is provided to and expected from the GUI. This layer obtains the internal keys from the application that correspond to the user supplied key and then dispatches it to the appropriate template or rule.

\(^1\)This is not truly a Dialog object even though it may appear so. There are certain bugs and limitations in the version 1.0.2 of the java Runtime that necessitate this “hack”.
6.1.5 Using the IMW

The GUI has been designed to reflect the intended purpose of the IMW, i.e. the export of data from an application. This system maybe likened to the clipboard feature available with several popular Word-processors. The initial Frame provides as a set of Menu options, different tasks that may be performed using the IMW. The launches of the various applications is fairly straightforward and except for VNC the launch application option starts the application and returns to the IMW. The PART export option brings up a Application interface dialog box. In this box it is possible to set, create or clear the current local-store. A set of radio-buttons will update a list of keys available for export from PART. Once a key (or keys) is selected the user may export all relevant data from PART to the current local-store. Once this is done, the user may dismiss this dialog-box to bring up either the VNC or the CA import dialog-box. It is possible to set a target model name. Now selecting the radio-buttons would list the appropriate keys available within the local-store for the said entity type. Once a key has been selected the user may choose to import all the data available within the current local-store into the domain of the application. It should be noted that there exist certain cases where confirmations are needed to effect this transfer. In such cases the confirmations would be presented to the user and the responses considered. It is also possible to delete the target models and make new (empty) models.
6.1.6 The Internals

The Internals: This layer comprises of several templates and rules that actually export data from or import data into the application. This code is further (loosely) organized into various layers that range from UDMM specific to application specific. The code has been physically separated into different files, each of which pertain to a specific application.

The philosophy used while exporting a independent Entity is that all of its sub-entities will be exported automatically. In addition, before any independent entity is retrieved into the current local-store, the system inspects the current local-store to see if the requested entity has already been retrieved by an earlier request. If an instance matching the requested entity is found, it is returned, otherwise the entity is retrieved and returned to the calling template. This algorithm ensures that only one instance of an independent entity exists in the current local-store. While importing an entity into the domain of an application, the IMW system does not have to implement special mechanisms to prevent redundancy of data. The Figure 6.2 captures this philosophy. An application that is importing data into its local domain may need to maintain certain libraries to completely transfer data as certain entity conversions are not feasible (for instance, the WorkstationClass entity from between PART to VNC).
6.2 PART

This section describes important aspects of the PART translator. The code for the translator resides in the file `part.tnr`. A discussion on some of the issues involved with the implementation of this translator are presented herein.

6.2.1 Mapping internal keys to user-comprehensible keys

Internal keys usable by an application are seldom user comprehensible. There are significant risks associated with exporting internal keys to the user or worser still, into
the local store. The attempt in the IMW has always been to export information within the intersection of domain views versus information that is specific to the application. Internal keys are decidedly a part of the private data held an application. Any export of these keys makes the current local store dependent on the structure of the data stored inside an application and susceptible to changes within the application. However, independent entities, by definition, are distinguishable by means of system wide identifiers. The IMW maps internal keys to these user comprehensible identifiers and presents or accepts only these identifiers. Since every export and listing operation is atomic (in a manner of speaking), any changes to internal keys in between these requests does not affect the transfer of data from inside PART.

6.2.2 Dispatching the requests

An attempt has been made to develop a generalized solution for the dispatch of import requests. This implementation uses the dynamic function call operator in TNR to call an appropriate template. This mechanism would facilitate the easy addition of more rules to export further classes of information, than is handled by this implementation. A deliberate naming convention and interface specification for the rules that accomplish the actual export has been defined to allow further expansion without making changes to the existing code thereby achieving good data-abstraction.
6.2.3 Mapping Features to Operations

There exists a mapping in between Operations performed and the features create as a result of these operations. However, this relation is a “many to many” relationship with a set of operations contributing to produce one or more features. This mapping comes in handy in applications such as CA where it is of significant benefit to display cumulative costs for like operations.

6.2.4 SQL queries

A very deliberate attempt has also been made to isolate all the SQL queries used to interrogate the database. It has been assumed that Technomatics will try to retain as much of the structure of the existing PART database as possible. This is a safe assumption as it does not preclude changes which will in all probability inevitably be made to the structure of the database to accommodate features into newer versions of PART. However, this assumption does provide us with a means of localizing and containing the scope of these changes to an extent that they may be made with just an understanding of the new database and variations from the existing version.

All the global queries are initialized from a template init_part_global_var(). As long as new queries are written to replace the queries in the file to retrieve the same data from the database in the same order no changes are needed, even to this lowest layer. A more realistic scenario would involve making minor changes to the rules in this layer while the major changes would reside in the init_part_global_var() template.
6.2.5 Unit Systems

This layer also includes the code for translating the dimensioned attributes stored in PART, from the (internal) units system, to the SI system, which is the standard for the data transferred in between the various interfaces. PART uses a UNITS_SYSTEM_ID attribute to identify the units used in that particular dimension. Queries that return values which have dimensions associated with them are expected to return the UNITS_SYSTEM_ID. However even within a single unit-system, there can exist differing units. A user supplied vector exists for each of the queries that returns one or more columns that have a unit associated with them. This vector, indexed by the UNITS_SYSTEM_ID, contains a coded string that specifies a set of conversion factors to be applied on the individual columns. All units are converted in the PART_to_UDMM() rule. The conversion strings may specify a mnemonic conversion factor (for instance, mm, in, sq.in etc.) or may indicate that a column is not to be converted by specifying a "*". For instance a typical set of strings may look like the following:

\[
\begin{align*}
\text{cf_example[1]} & = "*:mm: sq.mm: sec: USD:*" \\
\text{cf_example[2]} & = "*:in: sq.in: sec: USD:*"
\end{align*}
\]

6.2.6 Major TNR Routines

The following sub-sections discuss the various major routines that make up the interface to PART.
**PART.list** This is the rule that is called by the interface to list all the keys available within PART for a particular independent entity. The independent entity is a parameter as is the step-file name (which is included for reasons of symmetry)

**PART.export** This is high level rule is called by the interface to execute a export request. This resolves the rule name that is to be called by using the dollar operator and executes the appropriate rule for the particular entity.

**PART.put.p21.PartElement** A PartElement is the most significant entity that may be exported from PART. This entity references all of the independent entities contained in PART’s domain view. Optimizations discussed in 3.1.3.1 mean that there is no clear distinction made between the storage of a geometric feature and one of its many descriptions. This problem is exacerbated by the fact that multiple features share common parameters and the there is no direct reference into these set of parameters. An assumption has been made to generate a temporary index for feature-parameters that no AP_ID exceeds 10000. This assumption allows for significant simplification of the code.

**ToolAssembly** A ToolAssembly is identified by means of the name of the cutting-tool used in this tool-assembly. However, the same cutting-tool may be used in multiple tool-assemblies. In such cases the different tool-assemblies are identified by appending an internal key to the cutting-tool name. One may note that this is not an export of the internal key as this has been primarily done to create a unique name for the tool-assembly.
PART.put.p21.Operation This rule inserts and Operation entity into the current local-store. Since the name of the Operation is incorrectly stored in PART, the IMW queries the methods table-space to retrieve a Rule String and parses it to obtain the operation-name.

PART.mk.sequences This rule recursively queries the most-worked-features and the less-worked-features table (please refer to Figure 3.3 and the associated text for a detailed explanation) to chase down all the features that are manufactured as the result of a sequence of operations.

PART.put.p21.MFL.set This rule calls the PART.mk.sequences rule to insert MachinedFeatureLink entities that link the operations with the features that they yield.

PART.map.AF.to.CF Due to certain optimizations in the PART database it is very difficult to construct a single SQL query to retrieve a CF.ID given an AF.ID. It is necessary to have the CF.ID to identify the appropriate feature-description that is being machined.

PART.mk.fp.query This rule queries the internal database and obtains the SQL query, the names, the data-types and the feature-type for feature parameters.

PART.put.p21.ToolAssembly This rule exports a ToolAssembly from within PART. This extracts (and inserts) the ToolElement (or sub-type) entities and creates a ToolAssembly entity with the correct locations of the tool-elements.
**PART.mk.tool.type**  This helper routine maps a PART cutter-class to a EXPRESS enumeration.

**ToolElement**  This includes the rules PART.put.p21.MillHolder, PART.put.p21.MillAdapter and PART.put.p21.CuttingToolElement. A ToolElement is the super-class of many specialized tool-elements. It should be noted that the sub-classes that are not contained by PART have not been handled in this implementation of the IMW. These routines insert the appropriate specialization of the ToolElement into the current local-store.

**PART.put.p21.Model**  This rule inserts a Model entity that corresponds to the geometries that are held by PART. This entity serves to identify the model on a particular host computer.

**PART.put.p21.WorkstationClass**  This rule inserts an instance of a Workstation-Class as understood by PART. The optional variable `include_compat` decides if the compatible tools are to be exported with the WorkstationClass (Only a very small sub-set of all of the Compatible tools are used for manufacture of a product).

**PART.to_UDMM**  This rule implements the conversion algorithm for converting values that have units associated with them to the SI system. For a more detailed explanation of this algorithm please see Section 6.2.5.
6.3 VNC

This section details the VNC translator that reads a STEP database and create a hierarchy of directories and files usable by VNC and allow a setup to be visualized in the VNC environment. The code for the translator resides in the file tnr/vnc_utils.tnr.

6.3.1 Entity Interface Layer

The VNC interface provides access to "push" VNC Models into the VNC domain. The VNC domain principally contains knowledge pertaining to the UDMM independent entities PartElement, Model, WorkstationClass and ToolAssembly. Among these independent entities the Model and WorkstationClass mainly serve to identify appropriate VNC models.

The information regarding the ToolAssembly is used to construct ToolAssemblies on the fly by this interface. A PartElement may be imported into the VNC domain to essentially push some or all of the above mentioned entities to completely represent a PartElement.

6.3.2 VNC workspace

The VNC library consists of a hierarchy of directories. VNC, on startup, read a path initialization file (.vncpthfig) and initializes paths to certain directories. An important directory that it looks for is the CONFIG directory. This directory contains configuration files that specify where VNC should look for various components of a
library. VNC allows a variation in the library hierarchy as long as the entries in the configuration file correctly point to the specific directories. The IMW standardizes the hierarchy of directories (see Appendix B to avoid name clashes between different exported entities). By default all exported entities append to the current VNC library. A naming convention is defined in the \texttt{VNC.mk\_library\_name} template.

### 6.3.3 CLI Files

The result of many importing many of the entities into the UDMM is a CLI file in addition to the part-geometries. In order to construct the actual Workcells etc. it is necessary that these CLI files be executed using VNC. At the end of an import request the system checks to see if there exist any CLI files that haven't been built as yet. If such files exist they are run through VNC and then either removed or renamed (depending on a environmental variable setting).

### 6.3.4 Major TNR Routines

The following sub-sections discuss the various major routines that make up the interface to VNC.

\texttt{VNC\_listModel} This rule is called by the interface to list all of the Models that are currently available within the domain of VNC. This is used by the VNC Launch manager to allow the user to launch VNC with a selected target-model.
VNC_list This rule invokes the UDMM_list rule to list all of the keys that are available within the current local-store that could be imported into the current target-model.

VNC_delete This template removes a target-model from the VNC domain after requesting for confirmation from the user.

VNC_create This template creates a blank VNC library in the VNC domain. If the library exists it is removed (after confirmation by the user).

VNC_import This is high level template is called by the interface to execute a import request. This resolves the rule name that is to be called by using the dollar operator and executes the appropriate template for the particular entity.

VNC_launch This template is called by the interface to ask TNR to select and launch VNC with a specific CONFIG file. Since each manufactured setup is represented as a VNC Workcell, it is convenient to be able to start that particular setup and execute it within VNC.

VNC_put_model_WorkstationClass This copies an tape-archive file from a library that corresponds to the requested workstation and inserts it into the current target-model.

VNC_put_model_Model Conversion of ACIS models into VNC models is beyond the scope of this thesis. Hence it becomes necessary to recreate these (blank)
models in VNC. This implementation retrieves a Blank model from a user specified location and inserts it into the current target-model.

**VNC\_put\_model\_MillHolder** This creates the geometry for a MillHolder by calling the appropriate low-level geometry routines.

**VNC\_put\_model\_MillAdapter** This creates the geometry for a MillAdapter by calling the appropriate low-level geometry routines.

**VNC\_put\_model\_CuttingToolElement** This creates the geometries for the CuttingToolElement by calling the appropriate low-level geometry routines.

**VNC\_put\_model\_ToolAssembly** ToolAssemblies are created in two stages: the geometric models necessary to build the tool-assembly are created in the v11 format and a CLI file containing the commands necessary to put the tool-assembly together is written out. The geometries for various parts of the ToolAssembly are created by calling the appropriate template and the locations held with the ToolAssembly is used to create the CLI file needed to assemble the ToolAssembly from the constituent geometries.

**VNC\_put\_model\_Setup** Setups are modeled in VNC using Workcells. After calling other templates to insert various components of the setup (such as the workstation, tool-assemblies etc.), this template writes out a CONFIG file that corresponds to this setup.
write CLI This template prefixes the cli file with an execution priority. Since it is not possible to append to a file from within TNR, it becomes necessary to concatenate all CLI files and execute them simultaneously. In order to maintain some sort of order of execution (for instance build all the tool-assemblies before they are mounted onto a workstation) this prefix is set by the template that writes out the CLI file.

build CLI This template checks a directory to see if there exist any unbuilt CLI files. If such files exist they are piped through VNC to build them.

VNC V11 geometries The templates V11.hdr, cone, fillet, hemisphere, taper create geometries for VNC objects. These take a set of applicable parameters and calculate the coordinates of the points and the polygons.

6.3.5 VNC Specific Layer

Many helper templates are used to construct geometry conforming with the VNC version 11 format. These templates reside in the file `tnr/vnc.tnr` are documented therein.

6.4 CA

This section details the CA translator that tries to read a STEP database and generate a CA datafile for the cost analysis to be performed.
6.4.1 CA Design Files

Design Commands are the preferred way to import data into CA. For a list of the Design Commands and usage please see [RS]. At the end of an import request the created design file is run through the ca-batch program to create the desired model.

6.4.2 Major TNR Routines

The following sub-sections discuss the various major routines that make up the interface to CA.

CA_list This rule invokes the UDMM_list rule to list all of the keys that are available within the current local-store that could be imported into the current target-model.

CA_listModel This rule is called by the interface to list all of the Models that are currently available within the domain of CA.

CA_launch This template is invoked by the launch manager to start CA with an appropriate cost-model.

CA_create This template creates an empty cost-model. If the model exists it is deleted before the creation of the new model.

CA_delete This template removes a CA cost-model after user confirmation.

CA_import This high level template is invoked by the GUI to dispatch the appropriate import routines.
The Cost-note does not allow for white-space to be used for indentation purposes. Hence a scheme of characters has been employed to indent the various level of details in the Cost-Note. The CA interface uses the Design-File commands available in CA to create a design file and uses the `ca-batch` command to create the model. The interface uses and internal step-database `tnr/ca.fpstp` that stores information about the feature parameters.

### 6.5 Chapter Conclusions

This chapter examined in detail each of the interfaces. It began by introducing the system architecture in general. As TNR is a language that is still under development one must realize that its syntax is subject to change. However such changes and their impact to the IMW system can be expected to be minimal. The interfaces have been designed in a consistent fashion and they respond to similar requests with similar responses. Considerable effort has been taken to ensure that the system is adaptable to changing environments with no or minimal impact on the architecture. A structure has been defined for the implementation of an interface for an application. It is worthwhile noting that even though the implementation of `merge` is beyond the scope of this thesis, it has been considered in the design of the various components of the IMW.
Chapter 7

Conclusions

The lack of communicating and cooperating manufacturing software tools presents a major hurdle to seamless computer integrated manufacture and concurrent engineering. In many cases a difficult choice needs to be made between using tools suitable for a specific engineering problem versus the use of cooperating tools. Existing practices and preferences usually dominate this decision and the solution tends to prefer the use of "good" tools versus tools that communicate. An interesting observation is that most of these tools provide some means or mechanism to extract or insert information. An architecture such as the IMDE presents itself as a very strong candidate solution for this problem of the lack of interaction in between existing tools.

IMDE works with existing software and attempts to translate data in between tools operating in different domains. The subsequent portion of this chapter discusses the aims, observations and the results of the research. A brief section discusses further work that may be attempted as a logical continuation of this thesis.
7.1 Research Goals

This research primarily makes the following efforts:

1. Definition of a standard meta-model to express the communicable data in the system. A meta-model expresses the understanding of data in the manufacturing scenario. This data-model is never instantiated, but is used to communicate the data contained in the domain of an application to the domain of another.

2. Gain understanding into the data held by existing tools via various methods of inquiry. A clear understanding is necessary of the applications and their local data to determine those sections of this data that might lie in the intersection of the application domain with other domains.

3. Define a methodology to translate data in between the tools. The process of translation is very sensitive to the type of data being translated. However, it would of considerable advantage if a high-level methodology that establishes a convention for effecting such translations could be developed.

4. Implement a mechanism to integrate the tools into a concurrent, cooperating system.
7.2 Results

With these efforts in mind the following results may be considered.

1. A meta-model was defined using the EXPRESS schema. This model, while by no means complete, is a good starting point. This model expresses entities that are most likely to exist in the intersection of the domain views of the tools in this system. It is expected that some new entities may be added to this model and some of the existing entities undergo additions to accommodate the new domain-view intersections created with the addition of more tools. This model has been built on a set of basic entities which are expected to be reused in the aforementioned changes.

2. In this thesis an understanding was gained into PART, VNC, and CA by assiduous experimentation. VNC geometric models were built as per specification from Deneb. However many of the other file-formats for both VNC and CA were gleaned by making guesses on the meaning of various entries in the data-file. PART was explored using the SQL queries and piecing together the information obtained. This had to be constantly checked and challenged by actual use of PART.

3. A methodology for effecting translation has been defined as a syntax for the TNR language. TNR and the contained EXPRESS based database provides a very good framework for expressing a translation map in between the tools.
4. The actual mechanisms were implemented as TNR programs that extract information to the UDMM and then insert this data into tools. A layered approach makes scalability of the architecture a central design goal. This mechanism illustrates a possible methodology for future implementations.

7.2.1 Suitable Credit

No successful research is conducted in a vacuum, and this research has built on a number of concurrent efforts at the CASSI. In addition to the references listed in the bibliography section of the thesis, this thesis has drawn heavily from an (yet) unpublished PhD. Dissertations[Hua]. In this dissertation, the author develops a comprehensive mechanism for the IMDE that includes the actual implementation of TNR and EQL.

7.3 Future Work

Obvious candidates for a continuation of this research includes the building of a full-scale system with bi-directional translators. The author is concerned with the “fragility” of the interface due to the fact that the understanding of the tools is based on the reverse engineering of tools. While these interfaces have been tested extensively, a “production” design should rely on specifications obtained from the vendors of a tool. The acquisition of such specifications touches on several copyright and propriety laws and is beyond the scope of the thesis.
However, the thesis does prove the validity of the IMDE for the purpose of providing effective integration. Integration of existing prototypes [Joh97] to this architecture using the philosophies developed in the course of this thesis is another possible extension. Such an extension would bring some yet un-envisaged problems to light and may prove to be a very fruitful exercise.
Bibliography


Lizhong Huang. “EQL and TNR: Two languages for advanced software system integration.” Ohio University, Ph.D. Dissertation to be published.


Appendix A

UDMM schema

This appendix is the UDMM schema in express. Please refer to [Int94a] for details on the EXPRESS language.

SCHEMA UDMM;

(*
 *  
 *  SCCS control data
 *  This file:  udmn.exp
 *  Version:  5.8
 *  Last Update:  98/02/10
 *)

(*

Change Log

2/5/98 -- chandru
Added the Sequence type that sequences Operations, Setups etc.

11/20/97 -- chandru
Added FeatureDesc, FeatureParams, MachinedFeatureLink modified PartElement, Feature to deal with multiple-descriptions of a geometric-feature..

10/18/97 -- chandru & nie
Added CurrencyParam and child USDollar, Speed, Feed and children Decision to use the SI units-system for all units in the UDMM Derived a hierarchy ToolElement to accomodate different types of cutting and non-cutting tool-elements with specialized parameters. Added SpindleSpeed & FeedRate to Operation.
10/11/97 -- chandru
a. added ToolType in the ToolElement ENTITY and associated enumeration.

9/26/97 -- chandru
a. changed the Orientation attribute of the LocationPosition to be unit vectors rather than angles.
b. added unique rules to Model Entity
c. removed Direction attribute from Feature (see a).

9/9/97 -- chandru
a. corrected omission of Features from PartElement and removed uniqueness constraint on Feature.

6/25/97 -- chandru
a. rationale behind assigning attributes as OPTIONAL for sets OPTIONAL refers to the knowledge of the existence of values for the set while the number of values is determined by the number of members in the set so
  OPTIONAL SET[0:?] OF X may mean () -- no values exist of X for this entity
  $ -- this entity does not yet know of the existence of any values

6/22/97 -- chandru
Major changes to the UDMM structure over the version 4.1 of the udmm.exp
a. Naming rules
  1. All sets have pluralized names
  2. attributes do not use names used by entities
b. WorkstationClass now replaces both WorkBench and MachineCenter
c. New Entity OperatorClass.
d. ToolAllocation now replaces FixtureAllocation
e. OffsetData is replaced by parameter
f. ToolAssembly, ToolNode, ToolType modified

CONSTANT
  DollarSymbol : STRING := 'USD';
END_CONSTANT;
ENTITY Parameter
ABSTRACT SUPERTYPE OF 
(ONEOF (IntegerParameter, RealParameter, TextParameter));
  Name : STRING;
END_ENTITY;

ENTITY IntegerParameter
SUBTYPE OF (Parameter);
  Value : INTEGER;
END_ENTITY;

ENTITY RealParameter
SUBTYPE OF (Parameter);
  Value : REAL;
END_ENTITY;

ENTITY TextParameter
SUBTYPE OF (Parameter);
  Value : STRING;
END_ENTITY;

ENTITY CurrencyParameter
ABSTRACT SUPERTYPE SUBTYPE OF (RealParameter);
(*
  No attributes
  To workaround sdai's problems with TYPES
*)
END_ENTITY;

ENTITY USDollar
SUBTYPE OF (CurrencyParameter);
(*
  This doesn't work with the implementation
  but hopefully will in the future
*)
DERIVE
  Symbol : STRING :=DollarSymbol;
END_ENTITY;

ENTITY Speed
ABSTRACT SUPERTYPE OF (ONEOF(RevPerSec, MeterPerSec));
  Value : REAL;
END_ENTITY;
ENTITY Feed
ABSTRACT SUPERTYPE OF (ONEOF(MeterPerSec, MeterPerRev));
    Value : REAL;
END_ENTITY;

(* revolution per second *)
ENTITY RevPerSec
SUBTYPE OF (Speed);
(*
   * No attributes, to preserve hierarchy
   * To workaround sdai's problems with TYPES
   *)
END_ENTITY;

(* meter per second *)
ENTITY MeterPerSec
SUBTYPE OF (Speed, Feed);
(*
   * No attributes, to preserve hierarchy
   * To workaround sdai's problems with TYPES
   *)
END_ENTITY;

(* (radial feed in) meter per revolution *)
ENTITY MeterPerRev
SUBTYPE OF (Feed);
(*
   * No attributes, to preserve hierarchy
   * To workaround sdai's problems with TYPES
   *)
END_ENTITY;

ENTITY Model;
    FileFormatType : STRING;
    FileName : STRING;
    FileDir : STRING;
    FileVersion : STRING;
    Platform : OPTIONAL STRING;
    Host : OPTIONAL STRING;
UNIQUE
    FileName, FileDir, Host;
END_ENTITY;
TYPE
LocationType = SELECT(LocationName, LocationPosition);
END_TYPE;

ENTITY LocationName;
  Location : STRING;
END_ENTITY;

ENTITY LocationPosition;
  Position : ARRAY[0:2] OF REAL;
  Orientation : ARRAY[0:2] OF ARRAY[0:2] OF REAL;
END_ENTITY;

TYPE
SequentialType = SELECT(Setup, Operation, Task, WorkInstruction);
END_TYPE;

ENTITY Sequence;
  Items : LIST[0:?] OF SequentialType;
END_ENTITY;

ENTITY PartialOrder;
  Less : SequentialType;
  More : SequentialType;
END_ENTITY;

ENTITY WorkstationClass;
  WCType : STRING;
  Description : OPTIONAL STRING;
  CompatibleTools : OPTIONAL SET[0:?] OF ToolAssembly;
  CompatibleFixtures : OPTIONAL SET[0:?] OF ToolAssembly;
  BurdenRate : OPTIONAL REAL;
  Models : OPTIONAL SET[0:?] OF Model;
  Parameters : SET[0:?] OF Parameter;
UNIQUE
  WCType;
END_ENTITY;

ENTITY MachineProgram;
ENTITY WorkInstruction;
    Name : STRING;
    Tasks : SET [1:?] OF Task;
    WorkCell : WorkCellClass;
END_ENTITY;

ENTITY Task;
    Name : STRING;
    Description : STRING;
    Operator : OPTIONAL OperatorClass;
    NumOperators : OPTIONAL INTEGER;
    Time : OPTIONAL REAL;
END_ENTITY;

ENTITY OperatorClass;
    Name : STRING;
    BurdenRate : OPTIONAL REAL;
    Description : OPTIONAL STRING;
UNIQUE
    Name;
END_ENTITY;

ENTITY TooledWorkstation;
    Name : STRING;
    Description : OPTIONAL STRING;
    PortableTools : OPTIONAL SET [0:?] OF ConfiguredPortableTool;
    PositionOffsets : OPTIONAL SET [1:?] OF PositionOffset;
    ToolOffsets : OPTIONAL SET [1:?] OF PositionOffset;
    ToolAllocations : SET [0:?] OF ToolAllocation;
    FixtureAllocations : SET [0:?] OF ToolAllocation;
    WorkStation : WorkstationClass;
END_ENTITY;
UNIQUE
  Name;
END_ENTITY;

ENTITY WorkCellClass;
  Name : STRING;
  Description : OPTIONAL STRING;
  Workstations : OPTIONAL SET[1:?] of WorkstationClass;
END_ENTITY;

UNIQUE
  Name;
END_ENTITY;

ENTITY ToolAllocation;
  Tool : ToolAssembly;
  Location : LocationType;
END_ENTITY;

ENTITY ConfiguredPortableTool;
  PortableTool : PortableToolClass;
  Elements : OPTIONAL SET [0:?] OF ToolElement;
END_ENTITY;

ENTITY PortableToolClass;
  Name : STRING;
  Description : OPTIONAL STRING;
  BurdenRate : OPTIONAL REAL;
UNIQUE
  Name;
END_ENTITY;

ENTITY PositionOffset;
  Name : STRING;
  Data : SET[1:?] OF RealParameter;
END_ENTITY;

TYPE
  ToolElementType = SELECT(ToolAssembly, ToolElement);
END_TYPE;

ENTITY ToolAssembly;
ENTITY ToolNode;
    Element : ToolElementType;
    NumRequired : INTEGER;
    Locations : OPTIONAL SET [1:NumRequired] OF LocationType;
END_ENTITY;

ENTITY ToolElement;
    Name : STRING;
    Description : OPTIONAL STRING;
    Models : OPTIONAL SET [0:?] OF Model;
    Cost : OPTIONAL REAL;
    CatalogNumber : OPTIONAL STRING;
    Parameters : OPTIONAL SET [0:?] OF Parameter;
END_ENTITY;

(* specializations of the ToolElement ENTITY
   *  CuttingToolElement, HoldingToolElement
   *)

ENTITY CuttingToolElement
SUBTYPE OF (ToolElement);
    Material : OPTIONAL STRING;
    SpindleDirection : DirectionType;
    InsertElem : OPTIONAL CuttingToolElement;
    NumInserts : INTEGER;
END_ENTITY;

(* using the UniGraphics system *)
TYPE
    DirectionType = ENUMERATION OF (clw, cclw, none);
END_TYPE;
TYPE
    OrientationType = ENUMERATION OF (left, right);
END_TYPE;

TYPE
  TurningToolType = ENUMERATION OF (standard, button, boring_bar);
END_TYPE;

TYPE
  GroovingToolType = ENUMERATION OF (standard, ring_type_joint);
END_TYPE;

TYPE
  ThreadingToolType = ENUMERATION OF (standard, buttress, acme);
END_TYPE;

ENTITY LatheTool
  SUBTYPE OF (CuttingToolElement);
  Length : REAL;
  Orientation : OrientationType;
  Width : REAL;
END_ENTITY;

ENTITY TurningTool
  SUBTYPE OF (LatheTool);
  BoringRadius : OPTIONAL REAL;
  HeelAngle : OPTIONAL REAL;
  LeadAngle : OPTIOANL REAL;
  NoseRadius : REAL;
  OrientAngle : REAL;
  Tooltype : TurningToolType;
END_ENTITY;

ENTITY GroovingTool
  SUBTYPE OF (LatheTool);
  HeelAngle : OPTIONAL REAL;
  InsertAngle : REAL;
  InsertWidth : REAL;
  LeadAngle : REAL;
  LeftRadius : OPTIONAL REAL;
  MaxDepth : REAL;
  NoseWidth : OPTIONAL REAL;
  RightRadius : OPTIONAL REAL;
  Tooltype : GroovingToolType;
END_ENTITY;
ENTITY ThreadingTool
SUBTYPE OF (LatheTool);
  HeelAngle : REAL;
  IncludedAngle : OPTIONAL REAL;
  InsertWidth : REAL;
  MaxDepth : REAL;
  NoseRadius : OPTIONAL REAL;
  NoseWidth : OPTIONAL REAL;
  ReliefAngle : REAL;
  TipOffset : REAL;
  Tooltype : ThreadingToolType;
END_ENTITY;

TYPE
  CircularToolType = ENUMERATION OF (milling_tool, drilling_tool,
                                           reaming_tool, tapping_tool);
END_TYPE;

ENTITY CircularTool
SUBTYPE OF (CuttingTooIElement);
  ToolType : CircularToolType;
  Length : REAL;
  Diameter : REAL;
  FluteLength : REAL;
  NumberOfFlutes : INTEGER;
  (* tool-tip specification *)
    LowerRadius : REAL;
    TaperAngle : OPTIONAL REAL;
  (* in case of drilling-tools we convert the point angle to tip angle *)
    TipAngle : REAL;
  (* extensions to the UG system *)
  (* tool-shank specification *)
    ShankDiameter : OPTIONAL REAL;
    ShankLength : OPTIONAL REAL;
    MiddleDiameter : OPTIONAL REAL;
END_ENTITY;

ENTITY HoldingToolElement
SUBTYPE OF (ToolElement);
  (* allows other 'non-standard' holding elements to put in the UDMM
   * rather than insert them as ToolElements to differentiate them
   * from cutting-tools

  Length : REAL;
  Diameter : REAL;
  FluteLength : REAL;
  NumberOfFlutes : INTEGER;
  LowerRadius : REAL;
  TaperAngle : OPTIONAL REAL;
  TipAngle : REAL;
  ShankDiameter : OPTIONAL REAL;
  ShankLength : OPTIONAL REAL;
  MiddleDiameter : OPTIONAL REAL;
END_ENTITY;
* No attributes, to preserve hierarchy
* To workaround sdai's problems with TYPEs
*)

END_ENTITY;

ENTITY MillHolder
SUBTYPE OF (HoldingToolElement);
   HolderDiameter : REAL;
   HolderLength : REAL;
   MachineEndClass : OPTIONAL STRING;
END_ENTITY;

ENTITY MillAdapter
SUBTYPE OF (HoldingToolElement);
   AdapterDiameter : REAL;
   AdapterLength : REAL;
   AdapterShankDiameter : OPTIONAL REAL;
   AdapterShankLength : OPTIONAL REAL;
END_ENTITY;

(* *)

TYPE
   PartElementType = SELECT(PartAssembly, PartElement);
END_TYPE;

ENTITY PartAssembly;
   Name : STRING;
   Description : OPTIONAL STRING;
   Components : SET [1:?] OF PartNode;
UNIQUE
   Name;
END_ENTITY;

ENTITY PartNode;
   Elements : SET [0:?] OF PartElementType;
   NumRequired : INTEGER;
   Locations : OPTIONAL SET [1:NumRequired] OF LocationType;
END_ENTITY;

ENTITY PartElement;
   Name : STRING;
   Models : OPTIONAL SET[0:?] OF Model;
Material : OPTIONAL STRING;
Cost : OPTIONAL REAL;
Parameters : SET [0:?] OF Parameter;
SetupSeq : OPTIONAL SET [0:?] OF Sequence;
OperationSeq : OPTIONAL SET [0:?] OF Sequence;
Features : OPTIONAL SET [0:?] OF Feature;
OpnFeatureMap : SET [0:?] of MachinedFeatureLink;

UNIQUE
Name;
END_ENTITY;

ENTITY Feature;
  FType : STRING;
  Descriptions : SET [1:?] Of FeatureDesc;
UNIQUE
  Descriptions;
END_ENTITY;

ENTITY FeatureDesc;
  Location : LocationType;
  Data : FeatureParams;
  (* pointer to containing feature for efficiency should be an inverse *)
  DescribedFeature : Feature;
END_ENTITY;

ENTITY FeatureParams;
  Params : SET [1:?] of Parameter;
END_ENTITY;

ENTITY MachinedFeatureLink;
  Features : LIST [1:?] OF FeatureDesc;
  Operations : LIST [1:?] OF Operation;
END_ENTITY;

ENTITY Operation;
  Name : STRING;
  Time : OPTIONAL REAL;
  ToolLifeUsage : OPTIONAL REAL;
  Tool : ToolElement;
  SpindleSpeed : OPTIONAL Speed;
  FeedRate : OPTIONAL Feed;
  theToolAssembly : OPTIONAL ToolAssembly;
END_ENTITY;
ENTITY Setup;
  Name
  Operations
  Location
  Programs
    Instruction
    Workstation
    TooledWorkstation
END_ENTITY;

END_SCHEMA;
Appendix B

VNC Directory Structure

This Appendix shows the directory tree structure for a typical target-model. In this example structure, it is assumed that a WorkstationClass called $T30$ and a PartElement called $demo\_part$ have been imported into the target-model example.

Figure B.1: A Typical VNC target-model.
Appendix C

PART Runtime table-space

This appendix contains an IDEF1X diagram of the PART Runtime table-space.
Figure C.1: The Part Runtime table-space.
Appendix D

PART Environment table-space

This appendix contains an IDEF1X diagram of the PART Environment table-space.
Figure D.1: The Environment table-space.