INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0500
The design of a software environment architecture based on executable process descriptions

Sarkar, Soumitra, Ph.D.
The Ohio State University, 1989
THE DESIGN OF A SOFTWARE ENVIRONMENT ARCHITECTURE BASED ON EXECUTABLE PROCESS DESCRIPTIONS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Soumitra Sarkar

The Ohio State University

1989

Dissertation Committee

Prof. Bruce Weide
Prof. Jayashree Ramanathan
Prof. Dik Lee

Adviser's Approval

Bruce W. Weide

Adviser

Department of Computer and Information Science
ACKNOWLEDGEMENTS

First of all, I would like to thank my adviser, Jay Ramanathan, for introducing me to the field of software environments. It was out of her many original ideas and unique insights, that the fundamental concepts behind this dissertation took shape. I would also like to thank the other members of my committee, Bruce Weide and Dik Lee, for their suggestions regarding the contents of this thesis, and Bruce for helping me out as my official adviser during the last one year.

I would like to thank past members of the TRIAD group – Walt McKnight, Chin Li, Steve Damion, Jim Kiper, Bob Vermilyer, Ron Hartung, Bill Hochstettler, and Vasudevan Venugopal – for all that I learnt from the many interesting and spirited discussions that we had over the years during the weekly meetings. Special thanks are due to Venu, for the useful discussions which helped clarify many of the issues addressed in this dissertation. I would also like to thank Venkat Ashok at Universal Energy Systems for sharing his thoughts on the design and implementation of the commercial version of the TRIAD system.

I would like to thank Karsten Schwan for the many valuable lessons about computer science research that I learnt, while working on his ISSOS project. I would also like to thank other members of the ISSOS group – Rajiv Ramnath,
Sridhar Vasudevan, Dave Ogle, and Anil Gidwani, for a very enjoyable year of system design and implementation.

I would like to thank B. Chandrasekaran for introducing me to the field of artificial intelligence, and for giving me the opportunity to work with him. I would also like to thank Prasad Vishnubhotla and Neelam Soundararajan for patiently answering my questions about parallel programming.

I would like to thank my friends, both in and outside the department, for the good times that we had throughout my stay in Columbus. I would particularly like to thank Amit Bagchi for his help when the times were rough.

I would like to thank my parents for their moral support, and for withstanding my many years of absence from home while I worked on this dissertation. I would particularly like to thank my father for being a great source of inspiration to me.

Finally, I would like to thank my wife, Anita - first, for helping me with my system implementation efforts, and secondly, for the many years of emotional support during the good times and the bad. I could have survived without the former, but not the latter.
VITA

March, 1956 ............... Born, Calcutta, India.

July, 1978 ............... B. Tech, Electrical Engineering,
                      Indian Institute of Technology, Kharagpur, India.

July, 1980 ............... M. Tech, Computer Science,
                      Indian Institute of Technology, Madras, India.

September, 1980 ......... Software Engineer, Softek Pvt. Ltd.,
                      New Delhi, India.

PUBLICATIONS

   International Software Process Workshop - Representing and Enacting the
   Software Process, May 1988, coauthored with Venkat Ashok, Jayashree Ramanathan,
   and Vasudevan Venugopal.

2. “Providing Customized Assistance for Software Lifecycle Approaches”, IEEE
   with Jayashree Ramanathan.

   Computers in Engineering Conference, August 1988, coauthored with Venkat
   Ashok, Jayashree Ramanathan, and Vasudevan Venugopal.

FIELDS OF STUDY

Major Field: Software Engineering.

Minor Fields: Artificial Intelligence, Operating Systems.
# Table of Contents

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Problems of Large-scale Software Development</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Existing Approaches to Constructing IPSEs</td>
<td>6</td>
</tr>
<tr>
<td>1.2.1 The Hardwired Approach</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 The Toolkit Approach</td>
<td>8</td>
</tr>
<tr>
<td>1.2.3 Intermediate Approaches</td>
<td>13</td>
</tr>
<tr>
<td>1.3 A Process-Based IPSE Architecture</td>
<td>14</td>
</tr>
<tr>
<td>1.3.1 The Structure of a Process Program</td>
<td>17</td>
</tr>
<tr>
<td>1.3.2 Execution of a Process Program</td>
<td>19</td>
</tr>
<tr>
<td>1.3.3 Important Characteristics of the Language</td>
<td>21</td>
</tr>
<tr>
<td>1.3.4 The Advantages of a Process-Based Architecture</td>
<td>23</td>
</tr>
<tr>
<td>1.4 The Scope and Organization of the Dissertation</td>
<td>32</td>
</tr>
<tr>
<td>II Related Research</td>
<td>34</td>
</tr>
<tr>
<td>2.1 Customizable Environments</td>
<td>35</td>
</tr>
<tr>
<td>2.2 Data Modeling</td>
<td>37</td>
</tr>
<tr>
<td>2.2.1 Traditional Data Models</td>
<td>38</td>
</tr>
<tr>
<td>2.2.2 Semantic Data Models</td>
<td>41</td>
</tr>
<tr>
<td>2.2.3 Object-oriented Data Models</td>
<td>45</td>
</tr>
<tr>
<td>2.2.4 Modeling Versions and Revisions</td>
<td>47</td>
</tr>
<tr>
<td>2.2.5 Long Transactions</td>
<td>49</td>
</tr>
<tr>
<td>2.2.6 Use of Data Models in Prototype Software Environments</td>
<td>51</td>
</tr>
<tr>
<td>2.3 Activity Modeling</td>
<td>52</td>
</tr>
<tr>
<td>2.3.1 Event Modeling in Databases</td>
<td>54</td>
</tr>
<tr>
<td>2.3.2 Knowledge Based Approaches to Activity Modeling</td>
<td>55</td>
</tr>
</tbody>
</table>
2.3.3 Modeling Software Development Activities 57
2.4 Tool Integration ................................................................. 62
2.5 User Role Modeling and Access Control ............................. 65

III Language Features for the Definition and Manipulation of Database Objects 68
3.1 Type Definition ................................................................................ 70
  3.1.1 Scalar Types and Type Constructors ................................. 71
  3.1.2 Object Definition ............................................................... 77
  3.1.3 Representing Relationships between Objects .................. 79
3.2 Object Manipulation Operations ............................................... 90
  3.2.1 Object Creation ............................................................... 91
  3.2.2 Object Deletion ............................................................... 93
  3.2.3 Accessing Objects and their Attributes ...................... 94
  3.2.4 Object Modification ...................................................... 103
  3.2.5 Other Language Issues ................................................... 105
3.3 Transactions and Concurrency Control ................................. 108
  3.3.1 A Concurrency Control Mechanism for Interactive Design Transactions 109
  3.3.2 A Long Transaction Mechanism ................................. 111
  3.3.3 Long Transactions, Constraints and Side Effects ......... 118
  3.3.4 Restrictions on Checkout, Checkin and Update Operations .... 119
  3.3.5 The Relationship Between Long and Short Transactions ...... 123
  3.3.6 Mixing Long and Short Transactions .............................. 126
3.4 Database Triggers ................................................................. 128
  3.4.1 Defining Trigger Procedures ............................................ 129
  3.4.2 Using Triggers for Constraint Checking ...................... 135
  3.4.3 Using Triggers for Information Propagation .............. 143
  3.4.4 Object Deletion and Immediate Triggers ..................... 147
  3.4.5 Trigger Chains and Transactions ............................... 149
  3.4.6 Information Propagation and Attributes Derived by Triggers 151
3.5 The User Interface ................................................................. 157
  3.5.1 The Object Editor ......................................................... 158
  3.5.2 Triggers and the Object Editor ..................................... 160

IV Representations of External Tools and User Roles 163
4.1 The Representation of External Tools ....................................... 164
  4.1.1 Defining Tools as a Special Class of Objects .............. 165
## VI Implementation of the Virtual Machine on a Relational Database System

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>An Overview of the runtime system</td>
<td>269</td>
</tr>
<tr>
<td>6.1.1</td>
<td>The Information Manipulated at Runtime</td>
<td>271</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The components of a Process Program</td>
<td>272</td>
</tr>
<tr>
<td>6.1.3</td>
<td>An Alternative Implementation</td>
<td>274</td>
</tr>
<tr>
<td>6.2</td>
<td>Object Storage and Manipulation</td>
<td>276</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Mapping Objects into Relations</td>
<td>276</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Mapping Object Model Operations to Relational Operations</td>
<td>278</td>
</tr>
<tr>
<td>6.2.3</td>
<td>The Implementation of Long Transactions</td>
<td>284</td>
</tr>
<tr>
<td>6.2.4</td>
<td>The Implementation of Triggers</td>
<td>289</td>
</tr>
<tr>
<td>6.2.5</td>
<td>An Example of a Runtime Routine</td>
<td>298</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Maintaining Type Information</td>
<td>300</td>
</tr>
<tr>
<td>6.3</td>
<td>Activity Manipulation Operations</td>
<td>301</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Creation, Deletion, and the Manipulation of Attributes of Activities</td>
<td>302</td>
</tr>
<tr>
<td>6.3.2</td>
<td>The Translation of Scripts into Base Language Procedures</td>
<td>304</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Operations for Controlling Script Execution</td>
<td>306</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Operations for Sending and Receiving Messages</td>
<td>310</td>
</tr>
</tbody>
</table>

## VII Summary and Conclusions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Summary of Goals and Results</td>
<td>317</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Contributions of this Research</td>
<td>319</td>
</tr>
<tr>
<td>7.2</td>
<td>Justification of the Programming Language Paradigm</td>
<td>320</td>
</tr>
<tr>
<td>7.3</td>
<td>Evaluation of Results, and Future Research Issues</td>
<td>323</td>
</tr>
<tr>
<td>7.3.1</td>
<td>The Process Based Approach to IPSE Construction</td>
<td>324</td>
</tr>
<tr>
<td>7.3.2</td>
<td>The Data Model</td>
<td>327</td>
</tr>
<tr>
<td>7.3.3</td>
<td>The Activity Model</td>
<td>334</td>
</tr>
<tr>
<td>7.3.4</td>
<td>The Tool Model</td>
<td>337</td>
</tr>
<tr>
<td>7.3.5</td>
<td>The User Model</td>
<td>339</td>
</tr>
<tr>
<td>7.4</td>
<td>Conclusions</td>
<td>341</td>
</tr>
</tbody>
</table>

## BIBLIOGRAPHY

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>342</td>
</tr>
</tbody>
</table>
List of Figures

1.1  The hardwired, toolkit and process-based approaches  .......... 16
1.2  The structure of a process program  .................. 19
1.3  Tailoring an IPSE with a process program  .............. 24
1.4  The virtual machine  ................................ 30

3.1  Object Definition Example  ................................ 78
3.2  Examples of 1:1 and 1:N Relationships  ................. 80
3.3  Example of Inverse Relationship  ...................... 81
3.4  Example of Union Type  ................................ 84
3.5  A Binary Relationship with Attributes  ................. 87
3.6  A Ternary Relationship  ................................ 88
3.7  Object Deletion  ........................................ 94
3.8  Subobject Deletion  ...................................... 95
3.9  Change_record Object  .................................. 97
3.10 Examples of Complex Access Expressions ............... 97
3.11 Long Transactions and Referential Integrity .............. 120
3.12 Setting Links to Objects .................................................. 121
3.13 Object Dependencies and the Checkin Operation .................. 124
3.14 Prohibiting Changes to a Module With a Trigger .................. 137
3.15 A Modified Change Policy .................................................. 138
3.16 Preventing Duplicate Copies in the Baseline ......................... 140
3.17 Preventing Checkin of Incomplete Objects ......................... 141
3.18 Error handling ............................................................... 143
3.19 Recording Changes at Checkin Time .................................. 145
3.20 Triggered Delete .............................................................. 148
3.21 Flagging Modules Affected by Change ............................... 155

4.1 Definition of a Compiler Tool .............................................. 167
4.2 Definition of the Envelope for Compiling a Module ................ 168
4.3 Updating the Symbol Table Object ....................................... 171
4.4 Alternative Version of the Compile, module Envelope ................ 172
4.5 Tool Invocations and Nested Long Transactions .................... 175
4.6 The Definition of a Programmer Role ................................... 181
4.7 The Definition of a Programming Manager Role ..................... 183
4.8 The Definition of a Programming Team ................................ 185
4.9 The Multiple Semi-Public Database Architecture .................... 195

5.1 An Activity Hierarchy ....................................................... 206
5.2 Language Definition of the Activity Hierarchy ........................................207
5.3 Possible State Transitions of an Activity ................................................212
5.4 Examples of Message Send and Receive ..................................................217
5.5 Example of a Select Statement .................................................................222
5.6 Object Types for Documenting the System Design .....................................224
5.7 Message Flow Among Subactivities of System Design Phase .......................225
5.8 Definition of Attributes of System Design Phase Activity ..........................230
5.8 continued. Script for Controlling System Design .......................................231
5.8 continued. Script for Controlling Review ..................................................232
5.8 continued. Script for Controlling Iteration ...............................................233
5.9 Activity Script for System Design .............................................................235
5.10 Activity Script for Subsystem Design ......................................................236
5.11 Role Types of Designers ............................................................................238
5.12 Allowing General User Commands At Any Synchronization Point ..............245
5.13 Defining Subtasks of an Activity .............................................................248
5.14 Activities and Message Flows in a Change Request Procedure ...................251
5.15 A Hierarchy of Experimental Databases in Infuse ....................................258
5.16 Data Objects for Representing Partitions ...................................................261
5.17 The Script of the “make_partition” Activity .............................................262
5.18 The Script of the “change_module” Activity .............................................263
5.19 Running Test Cases Prior to Checkin ........................................................266
CHAPTER I

Introduction

The goal of this dissertation is to describe the design of an architecture for building software environments that can provide customized assistance for the software development processes that are followed in very large projects. In this dissertation, such environments are referred to as Integrated Project Support Environments, or IPSEs. The architecture is based on the concept that an IPSE should maintain and execute an explicit representation of a software process to provide project-specific support for policies which control the various activities performed during software development¹. In addressing that goal, this dissertation presents a process programming language as a vehicle for building executable representations of the software process. It also presents the design of a runtime system that constitutes a virtual machine for executing process programs.

The following hypotheses drove this research:

¹The term software development encompasses all the activities that are typically carried out in developing software, such as design, coding, testing, configuration control, maintenance, project management, etc.
• In large-scale software development projects, the identification of well-defined processes, and adherence to the policies of the process, in particular those that deal with coordination and communication issues, is essential for ensuring success.

• An IPSE needs to provide explicit support for defining and enforcing such policies. However environment architectures based on loose collections of tools cannot satisfy this goal.

• An IPSE architecture that supports the definition of the policies of a software development process, and its subsequent interpretation or execution, needs to be developed.

• A conventional programming language paradigm is adequate for describing policies that deal with multiuser coordination and communication.

• A runtime system for executing programs in that language is implementable.

The sections that follow further elaborate the first three points by justifying why current software environment architectures are inadequate for large projects. The description of the alternative architecture, the design of a programming language that can be used to describe software development processes, and the feasibility of the implementation of a runtime system for the language, are described in subsequent chapters.
The rest of the chapter is organized as follows: Section 1.1 characterizes the nature of large-scale software projects, distinguishing them from individual or small team software development efforts. Section 1.2 describes current approaches to building IPSEs for supporting large projects. Section 1.3 motivates the need for a new approach to building IPSEs, based on the merits and demerits of existing approaches. That section also outlines the proposed process programming approach.

1.1 The Problems of Large-scale Software Development

Very large software development projects pose a set of problems that are quite different in nature from those faced during one-person, or small team projects. In an individual software development effort, the emphasis is on coding (programming-in-the-small [36]). In a project involving a small team of programmers, the emphasis is on defining module interfaces (programming-in-the-large [36, 133]) and in managing the parallel versions and revisions of code being developed by the various team members. As the number of programmers increases from one to many, there arises the need to coordinate and control their interactions, and for reasonably small teams (less than 10 programmers, say), simple facilities to support parallel development of versions and concurrency control of revisions, usually suffice. However, as the project effort and team size grows an order of magnitude larger (with a team size of more than 100, say), project management issues dominate and systems that support these aspects of software development need to be constructed.
Large-scale software development is a highly complex activity involving numerous people, who play diverse roles in a project. In an effort to introduce some degree of discipline into the process, the software development life cycle (from the inception of a concept to the termination of an installed system) has been separated into distinct phases, to control complexity and measure progress. Each phase is further divided into tasks and subtasks. A particular specification of the granularity of the various phases, and the ordering and degree of concurrency between the phases and their components, constitutes a software life cycle model. Textbook life cycle models (e.g. waterfall [117], spiral [19], etc.), Department of Defence standards (DOD-STD-2167a [96]), or software life cycle standards [124] provide very general guidelines for identifying the activities in a project.

When an organization undertakes a large project, it typically refines a general life cycle model to create a more specific model of the software development process to be followed in the project. Such a software process model defines in greater detail the various tasks to be performed, such as the steps that constitute a task, the objects that are produced and consumed by a task, and the proper flow of data between tasks being carried out in parallel. The roles of the project members who will perform the tasks and the tools that will support various tasks are identified. Procedures and techniques are defined to coordinate and control the activities of project members; examples include management oriented monitoring and reporting procedures, configuration control procedures etc. In large scale soft-
ware development, the need to create such a detailed software process model, and to adhere to the policies (rules and regulations) dictated by such a model, is of paramount importance.

While software environments have largely addressed the problem of individual and small team software development efforts, the special problems of large-scale software development have not been adequately addressed at all (see [103] for a detailed comparison of various classes of software environments). Integrated programming environments such as Gandalf [94], CPS [131] and Interlisp [132] provide considerable support for individual programming. Environments such as the UNIX² Programmer’s Workbench [40] (containing SCCS [112], or RCS [135]), Ada programming support environments such as ALS [7], and Apollo’s Domain software engineering environment DSEE [79], provide basic support for coordination with version control facilities. CMU’s Smile system [69] provides coordination facilities with its experimental databases. These mechanisms allow the individual developer a great deal of freedom for performing most software development activities, and coordination is enforced only while extracting from or depositing information to the version control database. This is clearly not good enough for very large scale software development.

In large projects, strict policies have to be enforced to control the activities that are concurrently performed by various project teams, and to facilitate proper

---

²UNIX is a trademark of AT & T Bell Laboratories.
communication and flow of data between these activities, in accordance with an underlying software process model. Software development environments that exist today do not adequately support enforcement of such policies. As a result, organizations are forced to use existing environments that are only adequate for small team projects, and to use manual procedures to ensure adherence to those policies for which automated support does not exist. The goal of this dissertation is to address the problem of building IPSEs that can provide support for enforcing such policies. In the sections that follow, existing approaches to building IPSEs are described, along with their pros and cons. That discussion is used to motivate the process programming approach that is proposed.

1.2 Existing Approaches to Constructing IPSEs

There are two broad approaches to building software development environments that represent two ends of a spectrum: the hardwired approach and the toolkit approach. In the hardwired approach, an environment is built from scratch with custom-built tools that are integrated around a specially constructed database and a uniform user interface. In the toolkit approach, a collection of existing tools are put together around the standard file system and user interface of an operating system (OS), to provide support for various activities in the project. Sections 1.2.1 and 1.2.2 describe these two approaches, while section 1.2.3 describes other approaches that lie between these extremes.
1.2.1 The Hardwired Approach

In the hardwired approach, an IPSE is built from scratch. A number of special purpose tools are constructed and integrated around a common database representation. One example of such an environment is the RSL/REVS [3] system for specifying and analyzing requirements for real-time systems. Another example is the Infuse environment [70], which supports a set of policies for coordinating simultaneous modifications to a number of modules by members of a maintenance team. Environments built using this approach are typically very well integrated. The tools are custom designed to effectively share information with each other and policies are enforced to control the use of the system’s capabilities.

However, such environments are also typically quite inflexible because the policies supported by the system cannot be tailored without modifying the implementation of the system. For example, the change management policies supported by Infuse are hardwired in its implementation. This approach has worked quite successfully with programming environments (e.g. Interlisp), but it will probably not be so widely acceptable for IPSEs that support large-scale software development. The syntax and semantics of a programming language are very well defined. Where different programming environments may differ is in the user interface, or in the nature of incremental semantic checking (which is actually tailorable in language-oriented programming environments such as Gandalf and CPS). In case of large scale software development, there is no consensus at all as to what is the best life
cycle process to be followed for a certain kind of project, and any environment which supports very specific hardwired policies will probably be unacceptable to many organizations.

There is yet another major distinction between tightly integrated programming environments and tightly integrated IPSEs that makes the latter very difficult to build. IPSEs that aim to support all phases of software development are much broader in scope and are much more difficult to build from scratch. That explains why the few hardwired environments that exist restrict their scope by supporting only one individual phase of software development. A custom-built environment which provides very detailed (but hardwired) process support for all phases of software development would probably be useless beyond the organization whose policies it is based on, and the cost of building such an environment would probably be prohibitive in view of its limited applicability.

1.2.2 The Toolkit Approach

Due to its ease of implementation, the toolkit approach is by far the most common technique that has been adopted for building environments for large software development projects. In the toolkit approach, a collection of tools (some already in existence, others developed specially for a project) are put together to constitute an IPSE. Some of the tools perform small, well-defined tasks (e.g., a compiler), while others are often complete environments for individual phases (e.g., in a toolkit
IPSE, RSL/REVS could be used as a requirements specification “tool”).

The major advantage of this approach is the ease of IPSE construction. One has to decide on an appropriate toolset and the OS provides the rest: a file system based information repository and the user interface for invoking tools. The UNIX programmers workbench is an example where the OS provides some additional tool integration facilities in the form of pipes, a language for writing shell scripts [74], etc. The philosophy behind the approach is to use the OS and its file system as the integration mechanism that controls how tools are used by project members to manipulate project data. It is the limitations of a general purpose OS and file system to support these capabilities that lead to the major disadvantages of this approach:

- There is no way of enforcing correct tool usage due to the loosely coupled nature of the IPSE.

- The IPSE does not enforce or support a coherent software development process.

- The information repository is typically file-based. File-based systems provide limited data modeling, access control and concurrency control facilities.

- The degree of integration of the tools is limited by the fact that the tools are usually built in isolation and are not designed to share information with each other. Additionally, when the underlying file system provides very little
structure (e.g. UNIX), the task of sharing complex data structures across
tools is made more difficult.

These points are elaborated in some detail in the following sections.

Loosely Coupled Architectures

While tools provide automated support for various aspects of software development, there is typically no means to ensure that the tools are used properly, or are used at all. For example, Cheatham describes a project management assistant tool [25] which provides support for various activities of software development. However, his system cannot ensure that the tool is actually used to perform all project related activities. While the OS provides an interface for invoking executable programs (tools), it does not have any knowledge of what is proper tool usage. To limit the usage of tools to a fixed set, to restrict their usage within a given context, or to constrain their order of use requires an additional layer of software on top of the OS that incorporates such knowledge. Tightly integrated environments such as Gandalf and Infuse provide their own user interface, and force the user to use only the tools integrated within the system, according to the rules hardwired in the environment. Such a tight degree of control is impossible to achieve in a pure toolkit environment.
The Absence of Integrated Process Support

In an IPSE constructed using the toolkit approach, the overall life cycle process supported by the IPSE is essentially the sum of the policies embedded in all the tools it is comprised of. While some of the constituent tools may be quite independent of specific policies (e.g. a compiler), others may be full fledged environments that support well-defined policies for a particular phase. For example, one could imagine a toolkit IPSE where RSL/REVS is used for requirements specification, Inscape/Infuse is used for development and maintenance, and a configuration control system [16] is used to support configuration management policies. The degree of process support is thus directly related to the nature of the available tools. Since an isolated tool, or even an environment for a particular phase of software development is rarely designed with other tools or environments in mind, the result of putting such pieces together cannot result in any coherent process support. Since the policies are hardwired, they cannot be tailored to suit special needs of an organization.

There are also certain aspects of software development processes related to overall project control and coordination that are typically not supported by any specific tool. Examples of such processes include ensuring the proper sequencing of activities, the proper flow of data from one phase to another through appropriate reviews and checkpoints, enforcing the creation of a plan for a phase before that phase begins, and later enforcing adherence to that plan. This causes gaps in
the overall process support provided by the IPSE. In toolkit environments, such policies are enforced manually, a task that requires extra manpower and especially competent management to execute successfully. The very approach of trying to utilize the capabilities of existing tools, which by themselves are not tailorable, leads to this absence of integrated and total process support in toolkit IPSEs.

The Shortcomings of File Based Data Repositories

Tools that comprise a pure toolkit IPSE typically share information using files. File-based systems have inherent weaknesses that make them less than ideal as information bases on top of which IPSEs should be built [92]. A file can only have a fixed number of attributes; definition and manipulation of application-specific attributes is not supported by a typical file system. While files can sometimes be organized in hierarchies using directory structures, more general non-hierarchical relationships between them cannot be represented.

The granularity of information in a file is too coarse, and any additional complex structure (beyond simple records) that is imposed on the contents of a file has to be interpreted by the tools that manipulate the file. The maintenance of the integrity of the information is also distributed among the logic of those tools. File-based systems have limited provisions for concurrency control and no facilities for handling non-trivial queries. Access control is an additional issue that is extremely important in a large project. There is a growing realization that file systems should
be replaced in this role by more general database management systems (DBMS) that can maintain richer models of information as centralized schemas, provide integrity management and concurrency control, and incorporate improved access control facilities.

**Tool Integration**

The major difference between the hardwired and toolkit approaches is that in the former, the tools are designed to work with each other, while in the latter they are not. In the hardwired approach, the formats of the output data of tools are made compatible with the input data formats of the tools they will be coupled with. In the toolkit approach, the tools themselves are designed in isolation and any coupling of tools requires intermediate data translation to make the inputs and outputs compatible.

**1.2.3 Intermediate Approaches**

Some environments have attempted to overcome the limitations of file systems by building their own data management capabilities in the form of program support libraries (e.g. [34]), specialized storage structures (e.g., ALS [7]), and support for version management (e.g., RCS [134], SCCS [112]). Some environments are beginning to address the issue of integrated process support within a toolkit model. For example, the ISTAR system [42] supports a very general contract model of software
development, apart from integrating external tools into the system. Hardwired support for the contract model is provided in the form of commands to create, delete and revoke contracts, and in the automatic creation of contract databases in response to such commands. While the external tools can be replaced and new tools added, it is not possible to tailor the system to support more detailed and refined versions of the contract model itself (without modifying the implementation), since it is hardwired in ISTAR. However, the model itself is so general that the likelihood of it being inapplicable to many organizations, or the danger of an organization quickly outgrowing the support provided by the system, is relatively low. The approach seems to be a true compromise between the two ends of the spectrum described in Sections 1.2.1 and 1.2.2.

1.3 A Process-Based IPSE Architecture

The architecture proposed in this dissertation addresses the issue of providing support for a specific software development process, since the lack of such support is an important problem of large-scale software development. The architecture addresses the major disadvantages of the hardwired and toolkit approaches that have limited their effectiveness. The proposal is based on the realization that:

1. Support for all aspects of a detailed software development process cannot be adequately provided by distributing that support over isolated tools.
2. Tailorability of the software process is an important issue since the capability to experiment with different processes is an essential requirement.

The key principle behind the proposed approach is the following: *The representation of a software development process (and thereby, the policies dictated by the process) should be explicitly maintained and executed by an IPSE, to provide project specific process support.* In this dissertation, the executable representation is referred to as a *process program*, and the resulting IPSE architecture is described as *process-based*. The essential features of this process-based architecture are listed below:

- At IPSE *construction* time, a process program is developed to tailor the system according to the software process to be followed in a project.

- At *use time*, the IPSE executes the program to provide customized process support for a project.

The process program is the core of the IPSE. It embodies the policies of a software development process, and its execution controls the manner in which the project members apply tools to manipulate objects, as they perform the various activities of the project. The IPSE architecture provides a virtual machine on top of the OS that provides process and data management capabilities. A process program uses those capabilities to tailor the IPSE for a specific project. It should be

---

3In [130], Taylor et al. argue that the functionalities provided by an environment architecture should be very similar to that provided by an OS, but at a different level of abstraction.
Figure 1.1: The hardwired, toolkit and process-based approaches noted that in a toolkit approach, there is no separate entity, besides the individual tools, that can be identified as the IPSE. In a hardwired approach, the individual functionalities of the IPSE (which could be thought of as tools), cannot be separated from the glue that puts them together and enforces policies of their use. It is the former that leads to the lack of integrated process support of toolkit IPSEs, and the latter that leads to the lack of tailorability of hardwired IPSEs. In providing a facility for process programming, and thereby separating process descriptions from individual tools, the proposed architecture addresses the above disadvantages. Figure 1.1 highlights the fundamental differences among these architectures. The sections that follow describe the nature of process programs, the virtual machine that executes them, and the advantages of this approach.
1.3.1 The Structure of a Process Program

Since the primary goal of a process program is to model a software development process, it is important to investigate the nature of software process specifications to identify the desirable features of a language for process programming. As described in section 1.1, the primary purpose of a software process model is to identify the different activities that are carried out during software development, as well as the relationships between these activities (ordering, concurrency, etc.). It is also necessary to model the objects that are produced and consumed by these activities. Tools are typically used to support some of these activities and the functionalities of the tools, as well as their relationships with the data objects, needs to be identified. In a large project, different project members play diverse roles as they perform various activities. Those roles need to be defined. The policies of a software process dictate how the activities performed by project members are to be coordinated, as they apply tools and produce data objects that are the results of those activities. A process program must be able to represent such relationships between the project related activities, tools, data objects, and the project members, to effectively model these policies.

The process programming language proposed in this dissertation directly addresses the above representational requirements. The language provides activity, data, tool and user modeling features to represent the various components of a software process model. The different features are unified under one language.
framework to allow representation of the relationships among these components that are embedded in the policies. The salient features of the language are listed below.

- The language provides *data modeling* features to model the data objects produced during software development. The internal structures of objects, and their relationships with other objects, appear as schema (data type) definitions.

- A tool used in the project is represented internally as an executable object, which acts as an operator whose operands are data objects.

- Every project member plays one or more *roles* in the project. Each role is represented as a *role object*, that defines role specific views of data objects, access to tools, and permissible operations on data objects and role objects.

- A project related task is represented as an *activity object*. An activity object\(^4\) is always associated with a role that represents the real life project member who performs the task. It contains imperative code that defines the steps that constitute a task, the subtasks that need to be performed to complete the task, and the allowable concurrency and sequencing constraints and data flow among these subtasks.

\(^4\)Referred to as an “activity” where it would be unambiguous.
Program ...

Types ------ contains definitions of data objects
... Tools ------ definitions of tool objects
... Roles ------ definitions of user roles
... Activities ------ activity descriptions
... Init ------ system initialization code
... end ...

Figure 1.2: The structure of a process program

It is a strongly typed language in which typed objects model data, tool interfaces and user roles, and activity objects model the “flow of control” of the software process.

1.3.2 Execution of a Process Program

Figure 1.2 shows the high level organization of a process program. The execution of the program is controlled by the code in the activity objects. Every task being performed by a project member is represented internally by an activity. The code inside an activity controls the manner in which the user performs the task according to the policies embedded in its definition. It is typically designed to hand over control to the user to perform creative aspects of the task, and to retain

---

*A project member is also an IPSE user. In the dissertation, the terms project member and IPSE user (or simply, user) are used synonymously.*
control only when it needs to perform clerical tasks in response to the actions of the user. An activity can create subactivities, initiate their execution, exchange messages with other activity objects, manipulate data and role objects, and invoke tools. Though an activity executes as an independent process, it can interact with other activities using a message passing paradigm; this is the basis of support for multi-user coordination. Even when it hands over control, an activity can limit the scope of what the user can do (the tools that can be invoked, the data objects that can be manipulated), thereby enforcing task (and role)-specific access control policies.

Activity objects are hierarchically organized to reflect the task structure of a project. At IPSE installation time, the "init" part of the program is executed (see Figure 1.2). This typically instantiates a "root" activity object, associates it with a key (project) management role, and initiates its execution. This root activity interacts with the real life project manager, and instantiates subactivities (which might represent top level phases of a waterfall-like model, for example) to be performed under the control of next level managers. As various tasks (e.g., coding a module) are completed, the internal activity objects are deleted and as new tasks need to be performed (e.g., fixing a bug in a module), new activity objects are instantiated; this is the pattern of execution of the process program. Execution lasts as long as the IPSE is required in the project. For example, if the maintenance phase of the project is to be supported by the IPSE, the process program will exe-
cute as long as the software product is supported by the development organization.
Otherwise, execution could stop after product delivery/installation.

1.3.3 Important Characteristics of the Language

Two important aspects of the language are highlighted at this point. The first aspect is the imperative nature of the language, and the reasons for making that choice are outlined. The other issue that is discussed is the importance of providing language features to support concurrency and persistence.

On the Imperative Nature of the Language

The process programming language is primarily imperative in nature. It is important to highlight the scope of what is intended to be represented in process programs, to justify this choice. Software development involves a large number of creative tasks such as design, coding, debugging, etc. Software life cycle models and more detailed models of software development based on them, do not address the issues of how to perform individual creative activities; they simply provide guidelines on how to control the overall process. As individual project members perform creative tasks in isolation, they are typically unaware of the actions of others. Policies are simply rules and regulations that attempt to control the potential conflicts that may occur among individuals working in parallel. Their basic purpose is to define procedures that make the global state of a large project more
visible to the people who make key decisions, and to constrain the actions of individuals based on these key decisions. The assumption is that an imperative language can be used to model such policies, and Chapter V provides numerous examples to justify this assumption.

A number of software engineering methods (e.g. Jackson[65], Structured Design [126]) have been proposed to provide some intellectual control over creative tasks. Such methods themselves are defined procedurally as a set of steps that provide guidelines on how to decompose a task, when to make key decisions, and when to evaluate the effects of those decisions. Since the methods are procedural in nature, they too can be represented using the imperative language being proposed.

Many artificial intelligence (AI) systems have attempted to automate expert tasks such as diagnosis, design, planning, etc. Special purpose knowledge representation languages (e.g. CSRL [24], DSPL [23]) provide features to model the knowledge required to perform such tasks, and control paradigms that use the knowledge structures to make inferences. The process programming language proposed here does not address such issues, and therefore does not need to support the specialized control paradigms of AI languages.

Support for Concurrency and Persistence

Whatever the control paradigm supported by a process programming language (imperative, declarative, functional, rule-based), it must provide features to sup-
port concurrency and persistence to be a viable medium for building models of software development processes.

Since a large project always involves multiple project members, any model of the software development process should be able to represent the parallelism inherent in the process. The proposed language provides features for spawning parallel activities dynamically, and for controlling the synchronization and flow of data among activities executing in parallel. Examples in Chapter V illustrate the adequacy of these constructs for modeling software processes.

The notion of persistence has to be supported by the language. Since the duration of execution of a process program always extends beyond a login session, the data objects and the state of the activity objects manipulated by a process program have to be saved in stable storage. The data modeling features of the language are based on an underlying model of objects and attributes, and data management features of persistence, concurrency control, etc. are provided by the language's runtime system. Chapter III describes the data modeling and manipulation features of the language using examples from software development.

1.3.4 The Advantages of a Process-Based Architecture

The notion of a process program distinguishes this architecture from existing approaches. A process program facilitates tailorability of policies, thereby addressing a major disadvantage of the hardwired approach. It describes how the capabilities
Figure 1.3: Tailoring an IPSE with a process program of the tools are to be combined together and therefore can act as a flexible and powerful integration mechanism. It thus improves upon the basic techniques of the toolkit approach. These points are elaborated next.

Tailorability of Software Process Descriptions

A process programming language provides appropriate abstractions for modeling the policies of a software development process. A process program also allows the separation of the policies from the implementation of the rest of the IPSE's features. These attributes of the process-based approach make it easy to understand and modify the policies supported by an IPSE, as the needs of the organization change from one project to another. The architecture therefore addresses the tailorability issue of the hardwired approach.

Figure 1.3 illustrates the environment generation process. A process program
is developed according to the requirements of the project. It is then compiled and linked with the runtime system to produce a customized IPSE. In the process-based approach, the IPSE has two components, a variant part that consists of the process program, and a fixed part that is a virtual machine for executing the process program. The virtual machine constitutes the runtime system for the language and it provides generic activity, data and tool management facilities that support the proposed language abstractions. Figure 1.4 illustrates the components of the virtual machine and Chapter VI describes the design of the virtual machine on a relational backend.

In real-life projects, software development process models are never completely built up front. Decisions about the techniques to use in later phases of the project (e.g., how rigorous the testing process should be) often depend on the results of the previous phases (e.g., the result of initial systems analysis that forecasts the size of the system). This gives rise to a very difficult problem that must be addressed: the ability to modify (and enhance) a process program during its execution. A very sophisticated programming environment for the language will be required to provide such dynamic tailorability. The design of such a programming environment is beyond the scope of this dissertation, but Chapter VII discusses some related research issues.
Improvements Over the Pure Toolkit Approach

The process-based approach retains the most significant advantage of the toolkit approach: the ability to build an IPSE by putting together the functionalities provided by existing tools. However, the architecture addresses many of the disadvantages of a pure toolkit approach that were outlined in Section 1.2.2.

The architecture provides its own interface and forces all access to tools and data objects to occur through the system. A process program incorporates knowledge of whom to give access to tools (according to role definitions) and when to give such access (as dictated by the activities that may be performed at a given time). Only tools that are represented within the process program may be accessed from within the IPSE, and the data management facilities ensure that objects produced by external tools cannot be introduced into the project database without IPSE control.

A process program can model those aspects of a software development process that are typically not supported by any existing tool. All tasks as well as the relationships between these tasks are modeled in the program. This can lead to a degree of integrated process support that cannot be achieved by a collection of isolated tools which only support certain aspects of the process.

The system provides a general purpose object base, a database management facility for storing and manipulating the objects produced in a project. The data modeling component of the language is based on an underlying model of objects
and attributes which can represent the structure of complex objects as well as arbitrary (hierarchical and non-hierarchical) relationships among them. The data model also supports triggers and long transactions. Triggers are procedures attached to objects and attributes; they are activated by data manipulation operations and typically perform constraint management functions. Long transactions provide control over concurrent accesses to data objects. The schemas that describe the structure of the data objects and their relationships to each other appear as data type definitions in the program. Data manipulation features are provided in the language to allow triggers and activity objects to manipulate objects and relationships. Chapter III describes the data model and the language features. Chapter VI describes the implementation of the object base manager.

The data model provides more powerful information modeling capabilities than file based systems. The database also provides more control over the access to data objects. In many file based environments (e.g., Smile), a user can always manipulate the files directly by going outside the system. In this architecture, all data objects are maintained in a centralized database, providing a greater degree of security. The trigger facility allows the database manager to maintain the integrity of the information. File systems cannot support such features and any constraint management has to be carried out by each tool that manipulates the data. Finally, long transactions provide a concurrency control facility that goes beyond features provided by any file system.
The process based architecture encourages reduction of the scope and size of individual tools, thereby easing the task of integrating them. Existing toolkit IPSEs are often composed of "monolithic tools", which are full-fledged hardwired environments that support a particular phase of the life cycle. Such monolithic tools typically perform their own data and process management functions and are very difficult to integrate with other tools because they provide limited access to their internal components and data representations. With a process programming facility, data and process management functions can be pulled out of individual tools. The tools can provide specialized functions that lie beyond the scope of a software process description, and the IPSE can enforce policies of tool usage, combine tool fragments to provide more powerful tools, and provide its data management facilities to enable the sharing of information between tools. This should encourage the construction of tool fragments with well-defined functionalities and interfaces, which are easier to combine with each other in flexible and useful ways (see [60] for a more detailed argument justifying this approach).

The object base is yet another feature of the architecture that can be exploited to support better techniques for tool integration. When tools exchange data through files, any complex structure imposed on a flat file has to be implemented by the tools themselves. The IDL [91] system provides facilities to define the structure of the inputs and outputs of tools, and automatically defines routines to access and manipulate these structures. Garlan has proposed a view-oriented
approach to tool integration [52], in which a centralized database maintains the information required by all the tools, and each tool defines its data requirements as views of this database. Conventional DBMSs have always supported the definition of database views, and Venugopal [139] is investigating the kinds of view definition capabilities that need to be introduced in the object base of this architecture, to enable better information sharing among tools. These issues are described further in Section 2.4.

The Virtual Machine

The runtime system for the process programming language constitutes a virtual machine that consists of the following generic facilities: an object base manager, a process manager, facilities to interface to external tools, a user interface, and an object editor that provides interactive object manipulation facilities. Figure 1.4 illustrates the components of the virtual machine and the interfaces between the IPSE and the external entities that it manages.

The object base manager is implemented on top of a relational DBMS. It provides generic object manipulation and long transaction facilities by using the data manipulation operations and short transaction capabilities of the backend database. The object base manager also controls the activation of triggers in response to the manipulation of objects. A generic object editor is provided for interactive manipulation and browsing of objects.
Customized IPSE

**PROCESS PROGRAM**
- object structure and relationships
- interfaces to tools
- user role definitions
- activity definitions

**VIRTUAL MACHINE**
- process manager
- user interface
- object manager
- tool interface (envelope)
- tool interface

**Tools**
- input file
- output file

**Figure 1.4: The virtual machine**
The process manager supports the activity management functions of the language. It provides runtime support for the creation and deletion of activity objects, and the message passing and synchronization primitives that are used to control concurrent activities. It maintains the status of activity objects, and in conjunction with the user interface, uses that information to control the activities that a project member may perform.

The runtime system provides functions that allows the process program to interact with external entities such as files and executable programs. This mechanism is used by the process program to integrate external tools. There are two broad categories of tools that an IPSE could integrate:

- Existing tools that are unaware of the object format in the database, and are designed to work with files.

- Tools that describe their data requirements as views of the objects in the object base, and can directly manipulate them using data management functions.

Figure 1.4 shows two external tools that are integrated by the IPSE. Tool T1 is of the first category. To integrate such a tool, the IPSE has to translate object formats to files that T1 can understand, and go through the reverse process to represent the tool's outputs in the database. Such translation code is part of T1's tool (invocation) interface definition. Tool T2 is of the second category and
1.4 The Scope and Organization of the Dissertation

The remainder of the dissertation describes the data, tool, user role and activity modeling features of the language. Design and implementation of the virtual machine is also described. The main focus of this dissertation is on the activity and data modeling components of the language. Simplifying assumptions are made about the kinds of tools that can be integrated, and the nature of role based access control that will be supported, in order to present a complete proposal.

Descriptions of database triggers, tool interfaces, and the executable parts of activities does involve the use of general purpose programming language features such as control constructs and procedure definitions. The special notations for data, tool, user and activity description and manipulation need to be embedded in a general purpose language. The embedding itself, and the general purpose programming language requirements of a process programming language is not a focus of this dissertation. Ada [54] has been used as the general purpose language in the examples presented.

Chapter II discusses related research on customizable environments, data models for software engineering and manufacturing, work on activity modeling, various approaches to tool integration, and some proposals on modeling role based access control.
Chapter III presents the data model for the object base and the language features for defining and manipulating objects. This chapter also describes the capabilities of the generic object editor.

Chapter IV describes how interfaces are defined for a restricted class of tools, and briefly outlines the more general approach being investigated in [139]. Chapter IV also describes the minimal assumptions that have been made about role based control of access to objects and tools, and how relationships between roles are used to represent group hierarchies.

Chapter V presents the activity modeling features of the language. Examples of life cycle processes and policies are presented, to illustrate how the relationships among data objects, tools and users are represented in activity descriptions.

Chapter VI describes the design of the runtime system for the language. The design is presented in two parts. The first part describes the design (and implementation) of the object base manager on top of a commercial relational database. Design of the activity manager is presented as a layer on top of the object base.

Chapter VII discusses the difficulties of applying the process based approach, and the problems of implementing the virtual machine using state of the art DBMSs. Finally, open research issues such as the use of more declarative notations for process programming, the design of appropriate programming environments to support dynamic changes to executing process programs, and the need for further data modeling research, are discussed.
CHAPTER II

Related Research

This chapter surveys related research in a number of areas that are relevant to the work described in this dissertation. Since IPSE tailorable is a major issue, various existing approaches to providing environment generation and customization facilities are described in Section 2.1. Persistence is another important focus of this dissertation and at the heart of that issue is the need to select an appropriate data model. Section 2.2 surveys different classes of data models, as well as database programming languages based on those models. Section 2.3 describes various approaches that have been proposed for activity modeling. The survey covers long term event modeling in databases, knowledge-based approaches to activity modeling for project management, and existing and proposed approaches for activity modeling in software environments. Section 2.4 references work on tool integration. Section 2.5 describes research on specifying access control policies for software development.
2.1 Customizable Environments

Environment generation facilities are most commonly available for structure (syntax-directed) editor based programming environments. For example, the Gandalf system [94] provides the ALOEGEN tool to build a customized structure editor based on a grammar description for a language. The degree and the type of incremental semantic checking that the editor is to perform can be described using action routines (written in a language called ARL [4]). Extended commands can be defined to integrate external (existing) tools, or to build new tools that are integrated around a centralized parse tree representation of the program.

Various other structure editor based environments also provide similar capabilities. In the synthesizer generator [110], attribute grammars [78] are used to specify the nature of semantic checking that should be performed by the resulting CPS editor. Kaiser has proposed extensions to attribute grammars for defining not only static semantics, but also dynamic and environmental semantics [67].

The original TRIAD software environment used grammar based techniques to support the generation of customized environments that support software engineering methods. The tuner tool of TRIAD [83] provided syntax description capabilities to define templates for structuring information produced during software development. Attributes and action routines were used to specify how the environment should support a user in following the steps of a method. Those initial
ideas of customized method support have evolved into the general process-based architecture described in this dissertation.

Two prototype software environments, Genesis [107] and Marvel [68], provide rule-based formalisms to describe software development tasks and project policies. Since these systems deal more directly with activity modeling issues, the description of these environments is presented in section 2.3, along with other approaches for activity modeling.

Environment extensibility is one of the key goals of the Arcadia project [129]. The approach to environment extensibility is also based on the process programming. The Arcadia project mirrors most closely the goals and the architecture described in this dissertation, and is discussed in Section 2.3.

Despite differences in the formalism used to describe the desired characteristics of the resulting environment, all the above systems have a common component. A virtual machine, often referred to as an environment kernel, provides a fixed set of capabilities based on an underlying model. The customization language provides abstractions to manipulate the facilities of the kernel. For example, in Gandalf, the model is based on abstract syntax trees, and the kernel (the runtime library) provides tree manipulation operations. The customization language consists of a syntax description language for defining “data structures”, and ARL for defining the semantics of structure manipulation (performed with the syntax-directed editor). The process based architecture proposed here is based on the same fun-
damental principles. However, since the focus is no longer on programming but on supporting all phases of software development, the abstraction mechanisms and the virtual machine characteristics are quite different from architectures of tree based programming environments.

2.2 Data Modeling

Traditionally, software environments have used file systems as information repositories. Section 1.2.2 describes some of the shortcomings of file based information repositories, from the standpoint of their limited information modeling, integrity management and concurrency control capabilities (also see [92]). The need to move to a more general database management facility is being increasingly felt by environment builders.

A database (often referred to as an information system) that maintains information about a domain can be viewed as a model of that domain. Information systems are thus conceptual models (see [22]) and a conceptual modeling language (CML) is required to construct such a model. Traditionally, languages for describing information systems have concentrated on the data to be stored, and have been based on a data model – a collection of rules for specifying the structure of data and the operations to be performed on the data [136]. The problems with traditional models are described in the next section. Section 2.2.2 describes research on newer data models that try to overcome some of these problems. Most
of that research has concentrated on the data modeling requirements of manufacturing applications and little attention has been paid to the specific needs of the software engineering community. The data model described in Chapter III reflects an attempt to adapt the various research proposals to fit the special requirements of IPSE object bases.

2.2.1 Traditional Data Models

The term "traditional data model" is used to refer to hierarchical, network and relational data models [138]. Kent [72, 73] has elaborated in great detail the limitations of these systems for building conceptual models of the real world. A lot of his criticism is valid for the domain of software engineering.

Traditional data models are record based. A concept is modeled as a record type which has a fixed number of fields (representing attributes of the concept), each field containing values of a fixed type and length. Kent refers to these properties as "horizontal" and "vertical" homogeneity, respectively. Such a rigid format of data is unsuitable for modeling information produced during software development. For example, a subprogram (a concept that subsumes procedures and functions) may or may not have a return value. Horizontal homogeneity makes it difficult to model the concept of a subprogram using one record type. Also, a certain field may not always have one unique type. For example, while an "if-statement" in a programming language typically consists of a "conditional" and a "statement", the
"statement" itself may be an "if-statement", a "while-statement", an "assignment" and so on. The fact that a "statement" could be a union of several types cannot be represented within the constraints of vertical homogeneity. The requirement that records must be in first normal form [138] prevents a field from having a variable number of values. Thus, if a document can have one or more authors, the fact cannot be modeled using one record type.

The language constructs provided by traditional data models are more appropriate for modeling the way in which the data is to be stored and accessed than for modeling the actual concepts underlying them. The sparsity of basic constructs means that they must be able to model very diverse concepts in the real world, thereby resulting in their having very weak semantics. For example, the relational model provides the record construct to model both entities and relationships in the real world. The notion of a foreign key can be used to represent relationships between different records. However, relational DBMSs do not treat foreign keys as special fields, for ensuring that arithmetic operations are not performed on them, or for maintaining referential integrity [33] to ensure that for every non-null foreign key there exists a target record.

Hierarchical and network models provide the concept of a link to represent relationships. The hierarchical model limits all relationships to be of the parent-child type. The network model is more general, but restrictions on the nature of a DBTG set do not allow the direct representation of N:M relationships (see [26]}
and [72] for more detailed discussions of this point).

In record based models, a single record does not always correspond to a single entity in the real world. Additionally, a record is typically identified by a combination of fields that always have a unique value (a key) and in such models, an entity being modeled by a record is the same thing as its key. This causes a problem when a particular entity may not have a valid value for the key field (e.g., a person doesn’t have a social security number), the key field needs to be changed (e.g., a document is reclassified, giving it a new document id), or there are several equally valid key fields. Relationships are represented between the keys of entities and this is the root of the dangling reference (lack of referential integrity) problem.

Finally, data definition and manipulation commands based on these models are typically embedded in a general purpose host programming language (e.g., SQL statements embedded in PL/I or Cobol [64]). The types provided in these languages typically do not match the structures maintained by the database, giving rise to unnatural programming paradigms. Experimental languages like Rigel [115], Plain [140] and Pascal/R [120] have attempted to integrate relational structures and operations into traditional programming languages. While these languages are cleaner than PLI/SQL, they still suffer from the limitations of record based models.
2.2.2 Semantic Data Models

A large number of other data models, which overcome many of the limitations of the traditional data models, have been proposed in the literature. These models are often collectively referred to as *semantic data models*, since they concentrate more on representing the actual semantics of the domain than on how the data is to be stored. Some CMLs have been incorporated into programming languages (often referred to as database programming languages) that provide uniform type structures (data definition) and manipulation capabilities based on the underlying data modeling primitives. Excellent surveys on semantic data models and database programming languages can be found in [20], [63] and [6].

Early Semantic Data Models

Three early proposals that attempted to address some of the limitations of traditional data models were Abrial's binary model [1], Chen's Entity-Relationship (ER) model [26], and Codd’s RM/T model [30].

Abrial’s binary model is a minimalistic model. The model provides primitives to describe fundamental entities (that have no further attributes), as well as relationships among these entities. More complex concepts are represented as collections of such fundamental entities and relationships. The binary data model is based on the same philosophy as the semantic network model for knowledge representation [111]. It was one of the first models to stress the importance of maintaining a
one-to-one correspondence between real world entities and data model objects, distinguishing objects from their keys, and representing relationships between entities and not their keys.

The ER model provides separate primitives to model entities and relationships between them. This leads to cleaner database design, by preventing some of the database design anomalies that can appear in record based models where the same record may represent one or more entities and/or relationships. Both entities and relationships can have attributes. Both binary and higher order relationships, as well as 1:1, 1:N and N:M relationships can be represented uniformly (in contrast to the network model).

Codd's RM/T model is an extension to the basic relational model. The RM/T model provides explicit support for relationships by introducing the notion of a surrogate that uniquely identifies a record. It also provides primitives for modeling events in terms of objects. That feature is described in the next section.

"Advanced" Semantic Data Models

A number of semantic data models have been strongly influenced by knowledge representation research in AI (e.g., frame languages [49]), as well as the languages Simula [32] and Smalltalk [53]. These languages provide primitives to model objects as collections of attributes, describe classes of objects that are organized into subclass hierarchies, and permit inheritance along class hierarchies. Four CMLs
that are based on these fundamental concepts, and which also provide fully inte-
grated database programming language features, are Adaplex [123] (based on the
functional data model Daplex [121]), Dial [55] (based on SDM [56]), Galileo [2] and
Taxis [90]. Representative features of these CMLs are described in this section.
The discussion is based on the material in [20].

The one-to-one correspondence between real life entities and objects in the
model is emphasized in all CMLs. As in the three models described in the previous
section, relationships can be stated between objects rather than their keys because
the two need not be viewed as synonymous. This is a fundamental principle that
is supported by all semantic data models.

Most relationships are expressed as attributes (or properties) of objects. An
attribute of an object can be viewed as a function that maps the object to some
value and therefore, represents a unidirectional binary relationship. An object is a
collection of attributes, together with their values. This is referred to as property
aggregation.

Objects are grouped into classes. There are two notions behind a class defini-
tion, extension and intension. The extensional notion treats a class as a collection
of instances. The intension of a class is like a type definition that defines the
attributes that a prototypical member of the class should have. Every object is
required to be the member of at least one class. Many CMLs allow an object to
belong to more than one class. This allows the representation of real life situations
where, for example, the same person can be a student and an instructor (thereby belonging to two classes).

Classes are organized into subclass hierarchies using IS-A relationships. For example, a “student” class could be related to the the “person” class with an IS-A relationship, to represent the concept that a student “is a” person. The manner in which an IS-A relationship is interpreted depends on whether the intensional or extensional notion of a class is being considered.

In the intensional interpretation, a class inherits all the attributes of its superclass. This concept is called property inheritance and it leads to concise descriptions of the entities in a domain. The Taxis CML uses inheritance to organize all basic concepts in the language including objects and transactions.

The extensional interpretation of IS-A expresses a set-subset relationship between elements of a class and its subclass. Every instance of a class is a member of its superclass (e.g., every student is also a person). This is an existence constraint that is automatically enforced by the system.

There are other special features that are not supported by all CMLs. Taxis treats classes themselves as objects, thereby allowing them to have attributes that describe the class as a whole, rather than any particular member of the class. For example, a “professor” class can have an attribute that describes the “average-age” of professors. Many CMLs allow multi-valued attributes, where the value of an attribute is a set or a sequence. Some CMLs allow attributes to be defined
as procedures, so that their values are computed on demand, rather than being stored explicitly. The latter feature gives these models an object-oriented flavor (see Section 2.2.3).

Besides modeling the static aspects of a domain (the object structures and relationships), CMLs also provide primitives to model the dynamics of an enterprise as part of the conceptual model. All the CMLs referenced above provide primitive data manipulation features to perform insertion, deletion and querying of objects. Both standard as well as non-standard control structures (e.g., to iterate through a group of objects that is the result of a query) are provided. All languages support the notion of a transaction that is guaranteed to leave the system in a consistent state.

2.2.3 Object-oriented Data Models

Object-oriented (OO) data models represent another class of models that is related to semantic data modeling. Object-oriented data models are based on the concept of abstract objects in programming languages. They focus more on the behavioral aspects of objects, in contrast to semantic data models that focus on structural aspects. Objects in an OO data model encapsulate complex procedures and functions that mirror the semantics of the application. Some OO models do not support a general purpose query language, forcing all interaction to be through specially defined functions.
The Gemstone database system [105] supports a model that essentially mirrors the type structure and message-passing control flow of Smalltalk. The users can write programs in Smalltalk, and the mapping of Smalltalk objects to persistent objects in the database is handled in a (relatively) transparent fashion by the DBMS. Vbase [97] and Orion [10] are two other systems that support object-oriented styles of programming and manage the mapping between in-memory data structures and the persistent store.

Triggers provide a restricted form of object-orientedness, by providing procedural definitions of actions to take in response to primitive operations such as modification and deletion of attributes/objects. Triggers were first proposed for relational databases [43] and some limited functionality was implemented in IBM's Query-By-Example system [147].

The Postgres system [127] allows the fields of a relation to have values that are database queries (written in the query language QUEL [128]). When such "QUEL-valued" attributes are accessed, the associated query is executed to yield complex derived information. The system also allows the definition of triggers.

The Cactis database [61] allows the definition of derived attributes as arbitrary computable functions. The functions are stated in a form similar to semantic functions of attributed grammars, and the system is geared towards efficient incremental evaluation of these functions, using physical data organization techniques tailored for manipulating attributed graphs.
Dayal et al. (the HIPAC database project [47]) have also proposed a complex database architecture that can efficiently support triggers in a real-time environment. The system will support a complex trigger definition scheme, which allows a user to define the triggering condition, a (real time) deadline on it, and whether the trigger should execute as part of the transaction that caused it to fire, or as a separate transaction.

2.2.4 Modeling Versions and Revisions

The representation of versions and revisions of an entity is a very important modeling requirement for IPSE object bases. In a software development project, a module of code can have a number of distinct versions that represent different branches of development (e.g., a "fast" version, a "compact" version, etc.). Each version in turn can have a number of revisions, which represent modifications to correct bugs or enhance functionality.

RCS and SCCS provide "data models" to represent such version structures, where the fundamental unit is a file of text. These systems disallow modifications to any revision of a version except the latest one, and also provide operations to merge different branches of development in a semi-automated fashion. A specific version/revision of a file is addressed by appending a sequence of numbers (e.g., 1.3.5) to the file name.

Data models for supporting the representation of versioned objects in databases
have been proposed by Dittrich and Lorie [39], Batory and Kim [13] and Katz et al. [71]. Version modeling features have also been proposed in the Damokles [38], Orion [27] and Iris [14] DBMSs. The fundamental unit that can be versioned is any database object (a collection of attributes), a feature that makes these proposals more general than the RCS and SCCS models. Basic version/revision relationships are supported by all the proposals.

Katz's versioning model [71] supports the description of hierarchies of versioned objects. For example, the configuration of a software system could be described in terms of components that are modules of code. Not only may each module be versioned, but one could imagine the existence of different versions of the configuration descriptions themselves. RCS or SCCS do not support hierarchies of versions. In those systems, versions of configuration descriptions would appear separately from the versions of the modules and any relationship between the two would have to be expressed using symbolic names that are not recognized by the system.

Orion models a versioned object as a generic object with a collection of versions. Just as a class may be viewed as a collection of instances, each versioned instance is treated as a collection of versions. The generic object contains attributes that do not vary from one version to another (e.g., its name), and system attributes (e.g., the identifier of the default version). The versions themselves may have additional attributes. This distinction between versioned and generic objects is made in all
the above proposals.

While a general capability for defining objects and relationships can be used to represent version/revision structures, the incorporation of specialized versioning primitives allows the DBMS to enforce additional constraints (e.g., to disallow updates to revisions that are not the latest), and to apply efficient implementation techniques (e.g., delta storage [135] of revisions). In effect, tools like RCS and SCCS become an integral part of the DBMS, on top of which other tools can be built.

2.2.5 Long Transactions

Both traditional and semantic data models support the notion of a transaction as an atomic unit of work. Transactions execute concurrently with other transactions, and either complete successfully by committing, or are aborted due to a deadlock or a database error. A transaction that is in the middle of execution is in an inconsistent state, and the transaction manager of a DBMS ensures that

- no other transaction sees data that is in an inconsistent state, and
- the database is never left in an inconsistent state; i.e. a transaction either commits successfully, or else all its effects are undone.

Concurrency control and transaction management schemes supported by conventional databases are based on the short transaction model, where a typical transaction is not expected to execute for more than a few seconds. The policy of
undoing all the work of one of the transactions involved in a deadlock is based on
the assumption that the amount of work done by a transaction is not significant,
and can easily be repeated. These are not valid assumptions for engineering design
databases.

Kim et al. [75] have defined the notion of a long transaction, based on public
and private databases, that is more suitable for design applications. In the long
transaction model, stable objects (e.g., chapters of a design document) reside in
a public database, where they can be viewed by everybody. If a designer wants
to modify a stable object, (s)he performs a check-out operation to create a local
copy of the object in a private database. Modifications to the private copy may
be performed over long periods of time (say days or weeks), during which the
object is in an inconsistent state and is not visible to others. When changes are
finalized, the object is returned to the public database with a check-in operation.
The check-out check-in interval corresponds to a long transaction.

When more than one object is involved, the check-out of all the objects followed
by one check-in operation to submit all the objects back to the public database,
corresponds to a long transaction. In such a scenario, an individual check-out
operation may fail because the object has already been checked out by another
long transaction. Such an operation could also lead to a deadlock. In the short
transaction model, when an operation causes contention the transaction is made
to wait, and if a deadlock occurs, one of the transactions is aborted. In the
long transaction model, waiting (for days or weeks) for an object to be released is not a viable option. Also, since a lot of work is typically performed in one long transaction, undoing all of it due to a deadlock is not permissible. The long transaction model of Kim et al. proposes that feedback be given to the user, who then resolves the contention or deadlock manually.

2.2.6 Use of Data Models in Prototype Software Environments

Some software environments have attempted to use database mechanisms for representing and managing software project information. The structure of project information is represented using an existing or specialized data model. The data itself is stored in general purpose (often relational) databases or file systems, with a layer of software providing the mapping between the data model and the backend repository.

Linton [81] has tried to exploit the power of relational queries to provide interesting views of programs stored in relations. The inadequacies of relational DBMSs for representing the structure of programs at a fine level of granularity are discussed at great length in his dissertation. The Project Master Database (PMDB) project [102] used a derivative of the ER model to represent project data. Penedo [101] describes the implementation of this model using the Ingres relational database. The paper also discusses the limitations of the ER model for software engineering applications, as well as the problems of implementing the model using
Ingres. The Genesis [107] project also uses a similar model.

The data model for the Damokles database system proposal [38] is an extension of the ER model that is specially tailored for software project information. The model supports the definition of complex objects that consist of hierarchies of subobjects. It also provides primitives to define versioned objects. The Damokles proposal also includes support for long transactions and the specification of arbitrary consistency constraints.

The OROS (Object, Relationship and Operation System) data model [114] has been proposed as the basis of the object base for the Arcadia software environment. The model supports the definition of three fundamental entity types – objects, relationships and operations. Unlike object-oriented models where each operation belongs to an object, or ER-like models where objects and relationships are primary and operations are secondary, the OROS model gives equal importance to each fundamental entity type, allowing each to be defined independently of the other. An entity of type operation can be used to define software tools that are integrated within the environment.

2.3 Activity Modeling

An activity model is a representation of instantaneous and long term events, and the relationships between them. Instantaneous events are atomic operations which cause changes in the state of a system, but where the component steps of the oper-
ation are not represented in the model. In contrast, long term events are typically composite events, whose component steps, which may themselves be atomic or composite, are explicitly represented in the model. Relationships between events are primarily causal (event A occurs only if event B takes place) or temporal (event A overlaps with event B). Such relationships may be expressed in very high level declarative terms (e.g., in a constraint language), a rule based language (where a precondition, expressed in terms of a global state, describes when the event can take place), or by using imperative programming language constructs for sequential and parallel control flow. The problem of event modeling has been addressed in databases and artificial intelligence, often with very different goals in mind. A detailed model of software development, from the inception of a concept to the delivery of a complete system, could be viewed as a model of the project defined as a highly complex long term event.

Section 2.3.1 describes research in modeling short and long term events in databases. Section 2.3.2 describes knowledge based approaches to event modeling for reasoning about activities, automation of (robot) planning, and assistance for project management. Section 2.3.3 describes various research proposals for modeling the software process. Some are simply notations for specifying software processes, while others are executable languages.
2.3.1 Event Modeling in Databases

The RM/T and SDM data models provide primitives to represent short term events and the relationships between them. In RM/T, events are special objects that have an associated time of occurrence. Special semantic relations such as “must be followed by” and “may be followed by one of” can be defined between these event types, and the system enforces these constraints when the event objects are manipulated by a user.

The Taxis CML provides language primitives for modeling long term events. For example, in a library database, the borrowing of a book can be modeled as one long term event. When a book has been borrowed, it may be renewed, an overdue notice may be sent, or the book may be returned (signalling the end of the “borrow book” event). Taxis provides a formalism called scripts [28], based on Zisman’s extension to Petri nets [146], for modeling long term events. Each component (short term event) of a script is represented as a transition of a Petri net. When the transition fires, a chunk of associated code gets executed. A transition fires when tokens from preceding nodes are available (standard Petri net semantics), and additional preconditions attached to the transition evaluate to true. Preconditions may be based on values of database objects, temporal predicates, as well as “wait” conditions that are satisfied when messages arrive from another script. Different scripts may execute in parallel (e.g., corresponding to books borrowed by different users), representing concurrent activities in the domain being modeled. The state
of a long term event, represented as tokens of the Petri net, reside in the database. Persistence of the state of an event is essential since the duration of such an event may extend across many login sessions of the user of the database.

While the RM/T model enforces temporal constraints among events, it does not automatically cause any events to occur. The script mechanism represents causal/temporal constraints "procedurally" and also performs activities automatically in response to appropriate triggering conditions (such as sending out overdue notices for books). The latter reflects more closely the goals of software process modeling and at least one proposal has been based on the same approach (see Section 2.3.3).

2.3.2 Knowledge Based Approaches to Activity Modeling

AI systems that automate the task of planning (see [111]) are based on models of some (restricted) world. Such models consist of two categories: process knowledge and state knowledge. State knowledge is represented as predicates defined on objects. Process knowledge describes operations that take the world from one state to another. Operations have preconditions, expressed as states that enable the operation to fire, and postconditions, that describe the effects of operations. A planning task is typically described as a goal (state of the world) to be achieved, and the operation definitions are used to select a particular order in which to perform the operations to satisfy the final goal. The general process of discovery
is called state-space search, and in this specific context, the technique is known as goal based planning.

Since the goal based planning process composes a set of operations at runtime to perform a task, the approach is more flexible than Petri net based models where the order of tasks is effectively hardwired into the model. Initial work on task modeling in office automation systems used the Petri net approach. More recently, researchers have adopted the goal based planning approach (e.g., [31]), based on the belief that even simple office tasks contain too many exceptions and special conditions that cannot be adequately handled by a procedural model. Huff et al. [62] describe the use of goal based planning techniques in providing automated support for software development processes.

In the Callisto system, Sathi et al. [119] used the frame based knowledge representation language SRL, to model project related activities. Frames are used to represent activity knowledge in a form very similar to Petri nets, and the very flexible inheritance capabilities are used to create what is essentially a Petri net interpreter. The system can be used to perform simulations of an actual project, to schedule events according to deadlines, and to reschedule events when slips occur. The system architecture is geared towards supporting management decision making; it cannot be used to actually control the activities of project members.
2.3.3 Modeling Software Development Activities

Approaches for modeling software development processes have been based on one of two goals. Some researchers have concentrated on building speciation oriented models. The principle goal is to provide modeling primitives that allow software process descriptions to be stated precisely, leading to a better understanding by both the designers of the process and the people who will use it. Others have concentrated on building executable models of software development activities which can be used to provide automated support for the process. Initial research on software process modeling focused on specification languages for representing and reasoning about the process (see [142]). More recent proposals have begun to address the problem of building executable models (see [41, 137]).

The next two sections cover specification-oriented and executable process modeling languages that have been recently proposed, but which are not yet at the implementation stage. The final section covers some prototype implementations that attempt to provide activity support in more limited ways.

Specification Oriented Modeling

Rombach has developed a graphical scheme [113] for representing the activities of software development, their hierarchical decomposition into subactivities, and the flow of data between these activities. After the process description is used to identify the various data elements, the structure of the data itself is specified. Williams'
behavioral model [143] formally specifies an activity in terms of its preconditions (which must be true for the activity to begin execution), and its postcondition (which holds after the activity has completed). An activity can be decomposed in terms of its subactivities. Activities communicate by exchanging messages; the type and number of messages is specified within the model.

Building Executable Process Models

Hoffnagle et al. [60] were the first to propose that separation of process descriptions from monolithic tools would result in smaller tools, greater visibility of the process, and greater flexibility in experimentation with processes. They viewed a software environment as an interpreter for process descriptions, but did not make any concrete proposals for a process modeling language except for suggesting that they should be represented as “life cycle control rules”. The idea that a programming language could be used to build executable models of software development processes (process programs) was first proposed by Osterweil [99].

The Arcadia environment project is also investigating the problem of designing an IPSE architecture based on a virtual machine for executing process programs. Their initial proposal for a process programming language, APPL/A [58], is a superset of Ada. The language provides facilities to represent relations among various software objects, and how the consistency of these objects can be verified and maintained. Runtime support for this capability would provide a functionality
similar to Make [48].

Ould et al. [100] have used a formalism called role-activity diagrams (RADs), very similar to Taxis scripts (see Section 2.3.1), to represent concurrent activities of software development. Each individual software development task performed by a project member is represented by an RAD. The control flow of RADs is represented using Petri net nodes and transitions. RADs can dynamically spawn other RADs and exchange messages with them.

The Software Designer's Associate (SDA) project [77] is a proposal for a design environment that is also based on the concept of executable process descriptions. One important characteristic of the design process is the occurrence of backtracking, which is used to undo current decisions and return to previous decision points in order to reevaluate and revise them. The project is experimenting with two separate formalisms for process modeling that can be used to represent such iterative backtracking. One proposal is based on an extension to attribute grammars. The other formalism is an extension to Prolog.

The Alvey Information System Factory study is a very ambitious proposal that examines the requirements and possible architectures of automated Information System Factories (ISFs). An ISF is a next generation IPSE that supports the definition of not just software, but complete information systems consisting of hardware and software. The initial high level design of the ISF architecture [95] is based on a virtual machine that can be tailored for different software develop-
ment projects with process programs. The architectural design does not describe any specific features of a process programming language, but the need to support concurrency and persistence is heavily stressed.

**Activity Modeling in Existing Prototype Software Environments**

Genesis [106] is a prototype software environment that uses production rules [111] to represent the controls that must be enforced on the activities of project members, and activities that may be automated in response to the actions of users. A production rule has a precondition, stated in terms of user actions and roles, and an action part that is executed if the precondition evaluates to true. For example, a production rule could represent the policy that if a module has already been tested and the current user is a programmer, (s)he cannot modify it. Another rule might state that if a module is modified, a predetermined suite of test cases should be executed and the results saved in a special object. The system uses a forward chaining (data driven) [111] control paradigm to cycle through the rules, in response to user actions.

The Marvel system [68] also uses production rules to represent activity knowledge. As in Genesis, a rule has a precondition and a postcondition that are defined in terms of attributes of objects in the object base. The action part itself consists of an activity. An activity is an integral software development task, such as "compile module" or "change component", that is performed by invoking a tool. More
complex activities such as "fix bug", which might require many smaller tasks and involve many users, are not handled. After the activity is complete, the postcondition becomes true. Marvel uses both forward and backward chaining to execute its rules. An activity cannot be performed unless its precondition is true. Once an activity is complete the preconditions of some other rules may become true, which in turn might be executed by the system. This is similar to the forward chaining control of Genesis. When the user wants to perform an activity whose precondition is not satisfied, the system uses backward chaining to explore if some other activity could be automatically performed to enable the precondition of the requested activity to become true. This is a form of goal driven search that is used in AI planning systems (see Section 2.3.2). Backward chaining can be used to simulate the behavior of Make.

The Toolpack project [98] explicitly addressed the problem of controlling tool invocations. In Toolpack, a structure called the tool dependency graph is used to represent relationships between tools and object types. For example, a relationship between object types "Fortran source code" and "object module" could express the fact that a Fortran compiler (tool) is used to transform the former to the latter. The interpreter for the tool dependency graph provides the same functionality as Make, as well as the backward chaining control of Marvel. The relational extensions of APPL/A effectively model tool dependencies too.
2.4 Tool Integration

In a software environment, tools provide distinct functionalities. Tool integration research deals with the design of mechanisms that enable different tools to exchange data effectively and thereby, work cooperatively with each other. There are two basic models of tool integration, the sequential (or file based) model and the concurrent (or database) model. Garlan [52] compares these models in detail. Garlan's view-oriented model of tool integration is an extension of the database model. This section briefly surveys the different tool integration models.

In the sequential model, tools communicate through well defined interfaces. The output of one tool becomes the input of another tool, often through some transformation. Examples of the sequential model are mechanisms such as Unix pipes [74], and IDL readers and writers [91]. The sequential model offers a number of advantages. It is relatively easy to introduce new tools in the system. When tools have relatively simple input/output formats (e.g., text streams), they can be reused in different contexts, and be composed with each other to provide useful functionality. However, in this model, transformations have to be performed on the data being exchanged across the tools. Tool communication is primarily in one direction. Additionally, tools typically execute in sequence, though Unix pipes are an exception.

In the concurrent model, tools communicate through shared objects in a com-
mon database, where data produced by one tool is immediately available to an­other. The database manager can provide support for data retrieval and ma­nipulation, integrity management, constraint management, and through a trigger mechanism – it can also invoke tools automatically in response to changes in the “state” of the database. This simplifies the construction of each tool, in contrast to the sequential model where such data management facilities have to be replicated in each individual tool. In the sequential model, tools often build and maintain duplicate representations of data (e.g., a compiler and a pretty printer may both build parse tree representations of a program). In the database model, common data is shared through the database, and common functionality is provide by ex­actly one tool. This model has been used by all structure oriented programming environments (e.g., Gandalf, CPS), and also in Balzer’s proposal for an operating system built around an active database [8].

In the concurrent model, there is no support for data encapsulation for individ­ual tools since all tools have access to all database objects. The model encourages the construction of tools that are tailored to a particular database structure, and which have close dependencies with other tools. This makes it difficult to reuse or modify individual tools. To alleviate these problems, Garlan has proposed a view-oriented model of tool integration [52], where each tool defines its interface to the common object base, as a view definition. The concept of views was first proposed in the ANSI-SPARC 3-schema architecture [138], where external schemas
(views) of different database applications are manually merged together to form a conceptual schema for an enterprise.

Garlan's system is based on an in-memory object base, tailored for a programming environment. His scheme also restricts the view definition capabilities of individual tools in order to permit automatic merging of views. Extensions of this model to the more global context of large-scale software development and its implementation on a persistent object base, are being investigated by Venugopal in [139].

The view-oriented approach is useful for creating new tools. However, the technique cannot be used to integrate existing tools into an environment that provides a structured object base. Given the fact that a large number of useful file based tools already exist, their incorporation into any environment can always be beneficial, even if the degree of cooperation between such tools cannot be as close as in the view-oriented approach. Traditionally, existing tools have been integrated into environments with specialized object bases, with the use of translators. The tool is invoked through the environment's interface, and the environment provides up and down translation facilities between object structures and flat files (see figure 1.4). In the TRIAD project, such translators have been referred to as “view extractors” (down translators) and “output distributors” (up translators). The same concept has been referred to as “tool envelopes” in the ISTAR and Marvel systems.
2.5 User Role Modeling and Access Control

All file systems provide the distinction between the owner of a file and its other users. Others such as Unix provide the notion of a group. Commercial DBMSs provide features to define access rights (e.g., read, write) for different database units. In a relational database such as SQL/DS, such an unit could be a relation. These primitives cannot be used to model hierarchical group structures and role-based access control.

Dittrich [37] has proposed an extension to conventional database schemes for describing hierarchical group structures. The structure of the organization using a shared database is modeled as a tree. Intermediate nodes represent groups while leaf nodes represent individuals. Corresponding to each leaf node, there exists a private database whose contents may be read and modified only by the individual represented by the node. Corresponding to an intermediate node, there exists a "semi-public" database that is readable by all the groups and individuals that are descendants of the node. The contents can be modified only by the administrators of the group. The proposal does not address the modeling of different organizational roles, and the need to restrict access to different object types in the database on a role-by-role basis.

The most extensive support for modeling the access to different types of data objects by different project members, on the basis of their roles in a project, have
been provided by Minsky's Darwin system [85]. Darwin's focus is on providing unified and closed control over the way a software system evolves. The control is unified because the same primitives are used to represent restrictions on the activities of the modules in the system (typically handled by scope rules), as well as the meta activities of the builders as they modify the modules. The process of establishing controls is also represented within the same model, giving rise to closed control.

Every activity within the system is carried out in the context of a frame. A frame contains a set of privileges, and an actor, which is an agent that can operate from the frame. The actor inherits all the privileges of the frame, and the frame interpreter uses that description to control the activities of the actor. When the actor is a module, the corresponding frame, called a module frame, contains program privileges. These control the use of program resources such as variables, procedures, etc. by the module. When the actor is a programmer, the corresponding frame, called a builder frame, contains meta privileges that authorize the use of meta operations such as the creation and manipulation of frames and the distribution of privileges among frames, thereby controlling how the system can be changed.

The Darwin system is based on the send-receive transport model of Minsky [84]. The schematic send-receive model of Sandhu [118] is a variation of Minsky's model. Both protection models provide features for specifying complex policies regarding
object access as well as the transport of privileges. Such fine degree of control cannot be achieved with conventional protection models such as the access-matrix model [35] or the take-grant model [66].
CHAPTER III

Language Features for the Definition and Manipulation of Database Objects

This chapter describes the data model for the object base of the IPSE architecture. The object base maintains the structure and relationships between primarily (but not restricted to) large-grained objects that are traditionally maintained using file and directory structures or special purpose library systems (e.g., [34]). The object management system allows the definition and manipulation of different types of objects and relationships, maintains the integrity of the object base, and controls concurrent access by different project members and tools.

The model is based on the use of typed objects for representing the structure and relationships among entities in a project-specific database, traditionally referred to as a schema. These typed objects are defined/used in the type definition component of a process program (Figure 1.2). The basic modeling primitives have been adapted from existing object-oriented and semantic data model proposals. Generalization and version modeling primitives have not yet been incorporated.
The syntax and semantics of the type definition language, based on data model rules for defining database structure, is described first. The notion of an extension is also introduced. Operations are described for the creation of objects (and inserting them into their extensions), and the deletion, modification and associative access of objects in an extension. Next, a long transaction mechanism is introduced as an additional concurrency control mechanism for a design environment. The relationship between long and conventional transactions is discussed. The data model also supports the definition of trigger procedures that can be attached to database objects, to maintain integrity constraints and to propagate the effects of updates to the database. The syntax of trigger definition and the semantics of trigger activation is presented.

The operations on the object base are integrated into a general purpose language that is used for defining triggers (Section 3.4), tool envelopes (Chapter IV), and executable components of activities (Chapter V). The integration of the data manipulation features into a specific programming language has not been addressed in detail. The syntax of Ada is used in the examples whenever general purpose programming language features are required. The fundamental concepts for the design of an integrated database programming language have been adapted primarily from Adaplex (which is also embedded in Ada), and to a lesser extent from Dial and Taxis. However, the incorporation of long transactions and triggers, and language features for modeling user roles and activities, represents significant ex-
tensions to the basic database programming language in directions not explored in
the above languages.

The rest of the chapter is organized as follows. Section 3.1 describes the data
model rules for defining object structure and relationships. Section 3.2 outlines the
data manipulation operations and their embedding in a general purpose program­
moving language. The notion of long (design) transactions is discussed in Section 3.3.
Section 3.4 describes triggers definition, and the semantics of trigger propagation.
Section 3.5 outlines the features of a generic object editor tool, which is part of
the user interface of the IPSE. This section also describes how the object editor,
in conjunction with trigger procedures, provides a facility which can replace many
special purpose editor tools that are constructed in conventional environments.

3.1 Type Definition

An object type corresponds to a prototypical real world entity whose instances are
stored in the database. An object is persistent, in the sense that the lifetime of an
object can extend beyond the program (e.g. a tool) that creates it. An object is an
aggregation of attributes. In an instance of an object, those attributes hold values
that represent the state of the object. Each attribute is typed, the type defining
a domain that restricts the range of values that it may contain. Primitive scalar
types as well as the definition of structured types are supported for describing the
types of attributes. Variables in the process program\textsuperscript{1} can also be of these types, in addition to the types supported by the underlying general purpose programming language. The next section describes the scalar types and type constructors for building structured types. Section 3.1.2 describes the syntax and semantics of object definitions.

3.1.1 Scalar Types and Type Constructors

The language provides the basic types integer, real, boolean, enumerated scalars, string, text and binary, and the type constructors set, sequence, union and object, for building structured types. This section describes the scalar types, sets and sequences, and operations on these types. The following section covers unions and object definition.

Scalar Types

The scalar types (i.e. types with no further components) that are defined in the language are - integer, real, boolean, enumerated scalars, text and binary. The declaration and usage of the first four types is identical to that of Ada.

An attribute of type text represents a string of (printable) characters with no specified upper bound on its length. The type binary is similar to the text type, except that the characters in the string are not constrained to be printable.

\textsuperscript{1}Such variables occur in trigger procedures, tool envelopes, and in the executable components of activity definitions.
Text and binary attributes are treated as scalar types because no operations are provided in the data model for manipulating the internal structure (if any) of such values. An attribute of type text corresponds to a conventional text file, while a binary attribute corresponds to byte stream files such as object code or executable images.

Text and binary attributes allow existing applications to be integrated into the IPSE. An entity that is conventionally represented as a file can now be represented as an object. The information in the original file can be represented as a text or binary attribute which is still processable by file based tools. Other attributes of the object can represent relationships with other information in the object base. Variables of type text and binary are not permitted because there is no predefined upper limit on the size of such data. Procedures are provided to convert "values" of text and binary attributes into files, where they can be manipulated using the file management operations of the underlying language. Operations are also provided to convert existing files into text or binary attributes.

The procedure convert_text_to_file writes the "value" of a text attribute into a named file. In the procedure call

convert_text_to_file (expression, source_file);

expression should evaluate to a text attribute whose value is written onto the given file. The format of such expressions is described in Section 3.2.3. The variable source_file should be of (the predefined Ada) type OUT_FILE. There is
an analogous function called `convert_binary_to_file`.

The functions `convert_to_text` and `convert_to_binary` are used to save the contents of files into text and binary valued attributes respectively. The following function call

```plaintext
expression := convert_to_text (source_file);
```

assigns the contents of the file to the text valued attribute defined by expression. The variable `source_file` should be of the type IN_FILE.

The option of writing text and binary attributes into files prior to their manipulation is more general than the provision of a few general operations in the model for manipulating them directly in the database (e.g., using cursors as in [57]). This is particularly true of tools like linkers and loaders that often deal with files of records rather than unstructured binary files.

**Strings**

A string type, denoted by the predefined name `string`, is a sequence of printable characters. All strings are of variable length, with a default maximum length of 255 characters. The maximum length can also be specified in the declaration, as in

```plaintext
type line is string(80);
```

String constants are enclosed within "s. The concatenation operator "&" and standard functions `length` and `substr` are provided.
Sets

A set is an unordered collection of unique items. The allowable item types of a set are integer, real, string, union types, and object types. Text and binary values cannot be aggregated into sets because there is no efficient way to check for duplicate values. Secondly, the operation for removing elements from sets is based on a comparison of values, giving rise to the same problem. Powersets and sets of sequences are not allowed. In the declaration

```
type mod_list is set of string (20);
```

mod_list is defined to be a set whose elements are 20 character long strings. Since a set represents a collection of values in the object base, there is no predefined maximum limit on the size of a set. However, a size constraint on a set can be defined to limit the maximum number of elements it can have. The following declaration limits the number of elements of an instance of mod_list to at most 20

```
type mod_list is set (20) of string (20);
```

A set constant is represented as a list of values, enclosed within "{" and "}". For example, \{"stack", "queue"\} represents a set of two strings, that could be an instance of the type mod_list. {} denotes the null set.

The operations set_insert, set_remove and set_delete are provided for manipulating sets. Each procedure takes two parameters, an expression that evaluates to a set address, and the value of a set of items to be inserted or removed from that set. For example, if S represents the set \{"stack", "queue"\}, then after performing
the following operations -

```
set_insert (S, {"dequeue"}); set_remove (S, {"queue"});
```

S has the value {"stack", "dequeue"}. If an element is already present in a set, attempting to insert it into the set has no effect. The set_delete procedure takes a set as a parameter and removes all its elements.

```
set_delete (S)
```

is equivalent to

```
set_remove (S, S)
```

The member function is provided for checking set membership. The predicate

```
member ("stack", {"queue", "stack", ... })
```

evaluates to the boolean value TRUE. The size function returns the size of a set.

Sequences

Sequences are ordered collections of values\(^2\) in which duplicates are allowed. The item types which can occur in sequences include the types that can form set elements, and also the types text and binary. Since sequences allow duplicates, and the operation to remove elements from sequences does not depend on a comparison of values, text and binary element types are permitted. The declaration

```
type proc_list is seq of string (20)
```

defines proc_list to be an ordered list of strings. As in sets, a size constraint can

\(^2\)The term collection is used to refer to both sets and sequences.
be defined to limit the maximum number of elements in a sequence. There is no predefined upper limit on the size of a sequence.

A constant sequence is represented as a list of values delimited by "<" and ">". <“push”, “pop”, “append”> is a possible instance of proc_list. <> represents an empty sequence.

The operations seq_insert, seq_remove, seq_delete and seq_append are provided for manipulating sequences. The insert procedure takes three parameters, an expression that evaluates to an address of a sequence to be manipulated, the value of the sequence to be inserted, and the position at which the insertion is to take place. The first element in a sequence has the position 1. After insertion, the first inserted item assumes the position specified. The remove operation is used to remove a number of items from a sequence. Its parameters are the address of the sequence to be manipulated, the number of items to be removed, and the position from which items should be removed. If S denotes the sequence <“push”, “pop”, “append”>, then after the operations

```
seq_insert (S, <‘top’>, 3); seq_remove (S, 1, 4);
```

S has the value - <“push”, “pop”, “top”>. The seq_delete procedure takes a sequence as parameter and removes all its elements. It is equivalent to

```
seq_remove (S, 1, size (S))
```

The seq_append procedure takes a target sequence as its first parameter, and the sequence to be appended to the target sequence as its second parameter. If S
and $T$ are sequences, then

$$\text{seq_append}(S, T);$$

appends $T$ to the end of $S$, and is equivalent to

$$\text{seq_insert}(S, T, \text{size}(S)+1)$$

$\text{Size}$ and $\text{member}$ functions with semantics identical to the ones for sets are also provided. An indexing operator is provided to access a given element of a sequence. The operator is similar in functionality to one dimensional array access in conventional programming languages. $S[i]$ accesses the $i$th element of $S$.

### 3.1.2 Object Definition

An object is defined as an aggregation of attributes. In an instance of the object, the values of its attributes define the properties of the object, as well as its relationships with other objects. Figure 3.1 illustrates the definition of a project_member object. An attribute may have an initial (default) value (e.g., hrs_per_week) that is assigned to it when the object is created. As illustrated by the set valued attribute skills, an object (unlike a tuple in a relational database) is not constrained to be in first normal form.

When an object is created, all single-valued attributes are created with initial values, or with a null value if no initial value is specified. However, if a nonnull constraint is defined for an attribute, then a value must be provided for the attribute at object creation time, and it can never have a null value. The keyword nonnull
type project_member is object
attributes
    name : string (20);
    employee_id : string (10);
    hrs_per_week : integer range 20..40 := 40, nonull;
    skills : set of string;
        -- e.g., 'programmer', 'tester', 'technical writer', etc.
    ...
    key {employee_id};
end object;

Figure 3.1: Object Definition Example

defines this constraint. Sets and sequences have implicit nonull constraints and are created as empty collections. Null values cannot be inserted into collections.

The keyword key defines a key constraint on the employee_id attribute. A key constraint on a set of attributes restricts every instance of the object type to have a unique value for that combination of attributes. In contrast, non-key attributes (such as hrs_per_week) can have identical values in different objects. A key attribute must also satisfy the nonull constraint.

Object Instantiation

An object type represents a template. An instance of an object is created with a blank template. An object instance is automatically assigned a unique surrogate id that cannot be manipulated by a user.

Corresponding to every object type, there exists a class (or the extension of the
type) which represents all the instances of that type. The class name and the type name are identical. The distinction is made by the compiler in a context-specific manner. Insertion of instances in the appropriate class is automatically carried out by the system (see section 3.2.1).

3.1.3 Representing Relationships between Objects

When the type of an attribute of object type A is another object type B, it represents a relationship type between A and B. The name of the attribute represents the name of the relationship type. In an instance of A, a non-null value of the attribute is a surrogate id of an instance of B, that represents a one way link between the two objects. Attributes (and variables) whose domains are objects are therefore pointers to object instances\(^3\). The link is one way because it can be accessed using the attribute of A's instance, but not from the instance of B. Attributes whose domains are objects represent unidirectional, 1:1, binary relationships.

In Figure 3.2, the type module has a 1:1 relationship with object type module\_design, represented by the attribute named design. A module type also participates in hierarchical relationships (described later) with module\_specification and module\_body objects. 1:N relationships can be represented by sets or sequences. The change\_history attribute defined in the module\_body and module\_specification objects represents a relationship between one instance of a module, and zero or

\(^3\)The object model does not allow one object to be nested within another, as in the NF\(^2\) model [50].
type nametype is string (20);

type module is object
  attributes
    name : nametype;
    implementor : string (10);
    design : module.design;
    interface : module_specification, subobject;
    implementation : module_body, subobject;
    -- further structure not described
    ......
    key {name};
  end object;


type module_specification is object
  attributes
    name : nametype;  -- Same as the name of the module object
    change_history : seq of change_record;
    specs : text;
    ......
  end object;


type module_body is object
  attributes
    name : nametype;  -- Same as the name of the module object
    change_history : seq of change_record;
    code : text;
    ......
  end object;

Figure 3.2: Examples of 1:1 and 1:N Relationships
**Inverse Relationships**

Inverse relationships are defined by declaring an attribute of an object to be the inverse of a relationship in which the object type is the target. In Figure 3.3, the implementation attribute of the module.design object defines the inverse of the design relationship above. The value of the implementation attribute of a module.design object is that particular module object, which participates in the design relationship with that module.design object. Attributes defined to be inverses are **virtual**, because they are not explicitly stored in the system. The "value" of an inverse attribute (when accessed) is derived by the system by constructing an

```plaintext
type module.design is object
  attributes
    name : nametype;
    design_overview : text;
    interface_descr : text;
    pseudo_code : text;
    implementation : module,
      derivation = inverse module.design;

end object;
```

Figure 3.3: Example of Inverse Relationship

more instances of a change_record object. In this case, the relationships are represented by a sequence of pointers to change_record objects. Change_history is defined to be a sequence because the order of changes must be preserved.
appropriate query based on the definition of the inverse. The type of an inverse attribute is typically a set. It is possible to define an upper bound on this set, or to define the type to be single valued (which implicitly defines the upper bound to be 1). The previous example constrains a module design to have only one implementation, a meaningful constraint in the absence of versions. The concept of ordered sequences, nonull constraints and initial values are however not meaningful for inverses. Since inverse attributes are virtual, only read operations are permitted on them.

Another example of a virtual attribute is given below, where a set valued attribute is defined to be the same as a given class.

\[
\text{mset : set of module, derivation = module;}
\]

Any access of the attribute mset translates into an access of the module class. Such a derivation is only permitted for set valued attributes. More complex derivations of attributes, specified as arbitrary queries are allowed in certain database languages but have not been investigated further in this proposal.

An attribute that represents an inverse relationship is a derived attribute (besides being virtual), because its value is computed by the system. Another class of derived attributes is described in Section 3.4. Those attributes are not virtual (i.e. there exist corresponding stored values in the object base), but their values are derived by the execution of trigger procedures.
"Part-of" Relationships

Certain relationships can be defined to have the part-of semantics. Part-of relationships have predefined semantics which affect the outcome of some database operations.

When two objects participate in a part-of relationship, the target object can be thought of as a subobject (or child) of the source. Unlike arbitrary relationships, instances of part-of relationships are constrained to form directed-acyclic graph (DAG) structures between the objects they relate, to reflect the semantics that an object cannot be a subobject of one of its parts. The imposition of a pure tree structure is too constraining for software engineering applications, where shared subobjects can occur. For example, two documents (or two versions of a document) can share a common chapter, or a module can be thought of as a subobject of a subsystem and of a library (object).

A subobject’s existence depends on its parent. Therefore, when an object is deleted, all its subobjects are deleted too, subject to other constraints on deletion being satisfied. However, if an object is shared by more than one parent, deletion of one of the parent objects does not result in an automatic deletion of the shared child.

The keyword subobject is used to designate relationships with part-of semantics. An object and all its subobjects can be treated as a compound object. Some commands have variations that apply to compound objects and these are introduced
type subsystem_design is object
  attributes
    subsystem_name : nametype;
    major_design_decisions : text;
    design_alternatives : text;
    subsystem_interface : interface descr, subobject;
    ....
    components : seq of uniontype {subsystem_design, module_design}, subobject;
end object;

Figure 3.4: Example of Union Type

in later sections. Figure 3.2 illustrates the definition of hierarchical relationships between a module and its specification and implementation.

Union Type

In modeling the products of software development, it is often necessary to represent the fact that an object type can take part in a relationship with any one of a set of target object types. The union construct is used to declare the type of an attribute to be the discriminated union of object types. In Figure 3.4, a subsystem design is decomposed into a number of components, each of which may be a subsystem or a module. The example also illustrates the definition of a recursive relationship.

Since a union type declaration is restricted to a discriminated union of references to objects rather than entire objects, the space allocated for a union type is constant, and insecurities caused by overlapping locations (as in variant records
of Pascal [141]) cannot occur. It is possible that the actual type of the value of a union type attribute may change over time. For example, a component of a subsystem may initially be designed to be a module, but a later iteration may modify it to be a subsystem. Thus, the language must support dynamic type checking.

The union type facility provides the same capabilities as "choice" production rules of grammars (see [83]). The following is a grammar representation of the above object declaration, using standard BNF notation.

```
subsystem_design ::= subsystem_name ... components
components ::= subsystem_design | module_design
```

The corresponding "database" is a syntax tree, and tree manipulation functions in syntax editor based systems (e.g., ALOE) also check for correct refinement of metanodes at runtime.

In data models that support IS-A relationships, special extensional semantics of class hierarchies can be used to model discriminated unions. The standard semantics of extensions automatically defines an instance of a class to be a member of all its superclasses. Some data models also allow subclasses to be constrained as nonoverlapping. If the supertype of those subclasses is not explicitly instantiatable (i.e. all instantiations of the supertype are only automatically created by the system as a result of instantiations of a subtype), then the superclass represents the discriminated union of its subtypes. If design_class is defined to be such a superclass of subsystem_design and module_design, then the following declaration
components : design_class, subobject;

has the semantics of a discriminated union. It is also essential that the corresponding database programming language allow all attributes of an instance of a subclass to be accessed from a variable whose domain is the superclass, and check the validity of access at execution time.

The partitioned class declaration of Galileo [2] allows a non-overlapping membership constraint to be defined on members of its subclasses. However, Galileo does not allow all subclass attributes to be accessed through a variable whose domain is the superclass to avoid dynamic type checking, and thus cannot support union semantics. Taxis [93] does support this feature. One other problem with this approach to modeling unions is that whenever a union of a number of types is to be defined, a named superclass has to be declared. The union declaration may be viewed as a facility for declaring an unnamed superclass with the semantics of partitioned subclasses.

Representing Complex Relationships

While 1:N binary relationships and inverses can be represented naturally in the object model, more complex relationships cannot be directly represented using attributes. Such complex relationships include higher order relationships (involving more than two object types), binary M:N relationships, and relationships that have attributes of their own. They have to be modeled as separate object types.
type compilation is object
  attributes
    source : uniontype {module, module_body,
                        module_specification};
    object : object.code;
    error.report : errors;
    datecompiled : string (10);
    compiled_by : project_member;
    compiler_version : string;
    compiler_options : set_of string;
  end object;

end object;

end object;

Figure 3.5: A Binary Relationship with Attributes
type upward_compatible_except is object
  attributes
    old_system : system;
    new_system : system;
    differences : seq of non_compatible_feature;
  .......
end object;

Figure 3.6: A Ternary Relationship

One example of a binary relationship with attributes is illustrated in Figure 3.5. When a module_body, a module_specification, or both (represented by the module object) are compiled, object code is produced, and the compilation object represents the relationship between the appropriate source object being compiled and the object code produced. The relationship has a number of attributes such as the date of compilation, the person who performed the compilation, information about the compiler version and options used, etc., that cannot be meaningfully associated with either the module_body or the object_code objects. Compilation is a relationship that relates two objects, represented by the source and object attributes. The other attributes represent properties of the relationship. The same technique can be used to represent more general M:N relationships.

Figure 3.6 illustrates a ternary relationship, representing the list of features that prevent new_system from being upward compatible with old_system. In Fig-

*More complex compilation units are not considered.*
ure 3.5, if the information about the derivation of object code from source code is complex and is maintained in a separate object (of type "derivation_info", say), then compilation too will become a ternary relationship.

The ER model has separate primitives for representing entities and relationships and represents all relationships using the same primitive. In contrast, many semantic and object-oriented data models typically do not distinguish between entities and relationships and model both as objects, in accordance with a principle known as semantic relativism\(^5\) (see [22]). Some object-oriented models (e.g., [145]) introduce the notion of attributes of attributes to handle properties of relationships. However, that is not general enough to handle higher order and M:N relationships and has not been adopted in many systems.

One advantage of semantic relativism is the ease of schema integration. A concept may be modeled as a relationship in one view and an entity in another view, and their integration into one conceptual schema does not pose a problem as it would in the ER model. That is an important issue for view based tool integration [139]. A single primitive for entities and relationships also allows relationships to participate in other relationships, a feature that the standard ER model cannot support. However, it does lead to a certain degree of nonuniformity in accessing objects via the relationships they take part in, depending on the nature and complexity of the relationship.

\(^5\)Even the relational model adheres to this principle.
Referential Integrity

One form of referential integrity [33] is automatically maintained by the object management system. If an instance of an object is the target of a relationship, then the object cannot be deleted. Any attempt to delete such an object raises an exception.

Referential integrity is only maintained for “permanent” relationships represented as attribute values. Variables (in triggers, tool envelopes and the executable components of activities) that contain references to objects are temporary and cannot prevent the target object from being deleted. Therefore, variables can contain dangling references while attributes cannot.

Key attributes of objects can also be used to represent relationships via foreign keys (as in the relational model). However, referential integrity for such relationships is not supported. The key definition facility is provided for constraining certain properties of an object to have unique values in the extension, and use of the feature for maintaining relationships is not explicitly supported by the model.

3.2 Object Manipulation Operations

This section describes the basic object manipulation operations. Operations are provided for creating and deleting objects, for accessing objects through their relationships to other objects, and for performing predicate based search for objects in a collection. Language constructs for iterating through members of collections
are described.

3.2.1 Object Creation

The `create_object` operation is used to instantiate a typed object and to optionally assign values to one or more attributes of the object. The first parameter to the create function is a type name, and the remaining parameters are values to be assigned to all attributes with nonull constraints at creation time. Since the attributes of an object have no particular order, the "named component" notation of Ada is used to initialize their values. The operation returns a reference to the newly created object. The following statement creates a new instance of a module object, assigns a value to the name attribute, and assigns the reference to the newly created object to variable M of type module.

```plaintext
M := create_object (module, name => 'stack');
```

All attributes that have nonull constraints must be assigned values at the time of object creation. Other attributes may optionally be assigned values during creation.

For every object type T, the extension of that type can be represented as a variable whose type is `set of T`. One side effect of object instantiation is the automatic insertion of a reference to the object, in the extension of the object's type. The delete operation removes the reference from the extension. The extension can only be manipulated by these two operations; direct manipulation with `set.insert`
and set_remove operations is not allowed. The extension of a type is referenced by the same name as its corresponding type. The disambiguation is performed by the compiler based on the context of use.

A copy_object function is provided for creating a copy of an existing object. The first parameter to the function is a reference to the object to be copied. A new object is instantiated and all attributes of the source object (except a class of attributes described in Section 3.4.6) are copied to the new object. Optional parameters to the function specify values that are to be explicitly assigned to attributes of the new objects, overriding the values that are copied from the source. The operation returns a reference to the newly created object. The following statement creates a copy of the "stack" module created above (referenced by variable M), and assigns to the variable M1 a reference to the new copy.

M1 := copy_object (M, name => 'queue');

The name attribute is assigned a new value because it is part of a key, and there cannot be two instances of a module with the same value of name. In the absence of key constraints, two objects can have exactly the same values for all their attributes. However, they will still be distinguishable because of their distinct surrogate ids.

The above copy operation creates a new module object whose interface and implementation attributes still refer to the specification and body of M. Typically, this is not the semantics desired. A nocopy option can be specified for an attribute
(see Page 156) to prevent its automatic copy.

The copy_object operation creates new copies of its argument, and of all its subobjects. A different set of create and copy operations are provided for use in the context of long transactions. These are described in Section 3.3.2.

3.2.2 Object Deletion

The "delete_object" operation is used to delete an instance of an object and (as a side effect) to remove the reference to the object from its extension. The parameter to the operation is a reference to the object to be deleted. The following statement deletes the module object that was created above.

    delete_object (M);

If the parameter to the delete_object operation is an attribute, its value is first made null prior to the deletion; otherwise, referential integrity will be violated. If M is a variable referencing a module, the statement

    delete_object (M.design);

deletes the design object reference by the attribute, and as a side effect, the attribute will contain a null value.

Figure 3.7 outlines the operations performed prior to the deletion of an object. Because the delete operation is invoked for all attributes that represent part-of relationships, the deletion of an object causes a cascaded deletion of all its subobjects. Any subobject of the deleted object that has only one parent is deleted.
foreach attribute A in the object to be deleted do
    if the domain of A is an object, but not a subobject then
        A := null;
        Update internal count for number of incoming links of target;
    else if the domain of A is a subobject then
        temp := A; A := null;
        Update count of incoming links of target;
        delete (temp);
    end if;

Figure 3.7: Object Deletion

However, if such a subobject also takes part in other non-part-of relationships, its
tried deletion raises an exception because of the violation of referential in­
tegrity, and the entire chain of deletes is aborted. Any subobject that participates
in more than one part-of relationship, but in no non-part-of reationships, is not
deleted. As a result, none of its subobjects get deleted either. No exception is
raised in this case. Figure 3.8 illustrates this with an example.

A different set of delete operations are provided for use in the context of long
transactions. These are described in Section 3.3.2.

3.2.3 Accessing Objects and their Attributes

Access expressions based on “dot” notation can be used to access objects and
their attributes. Contents of collections, including extensions of types, can also be
accessed using a predicate based search operation. These are covered in the next
two sections.
Case 2
If 01 is deleted then 03 is also deleted. If 01 is deleted, referential integrity is violated for 03.

Figure 3.8: Subobject Deletion
Expressions

Given a reference to an object, attributes of the object can be accessed using access expressions based on dot notation. For example, if MD is a variable of type module\_design (figure 3.3), the following expressions yield the value of the name and pseudo\_code attributes of module\_design

\[ \ldots \text{MD\_name} \ldots ; \ldots \text{MD\_pseudo\_code} \ldots \]

If an attribute represents a relationship, the target object's attributes can also be accessed. The reference

\[ \ldots \text{MD\_implementation\_name} \ldots \]

yields the name of the module that is the implementation of the the design object referenced by MD. Accessing objects via attributes is equivalent in functionality to performing relational equi-joins using foreign keys.

Multivalued (set or sequence) attributes complicate the semantics of access expressions. A reference to a multivalued attribute in an access expression yields the contents of the collection that the attribute represents. If the elements of the collection are references to objects, attributes of those objects can also be accessed. Consider the module\_design, module and change\_record objects of Figures 3.2, 3.3, and 3.9 respectively.

Figure 3.10 illustrates some valid access expressions and their values, based on the objects defined above. In an expression of the form a\_b\_c\ldots, the presence of a sequence valued attribute makes the value of the entire expression an ordered
type change_record is object
attributes
  change_made_by : string;
  date_of_change : string;
  lines_changed : set of integer;
  reason : text;
......
end object;

Figure 3.9: Change_record Object

...M.implementation.change_history ...
  -- a sequence of references to change_records.
...M.implementation.change_history.date_of_change ...
  -- a sequence of dates (strings).
...M.implementation.change_history[3].date_of_change ...
  -- the date_of_change attribute of the third change.
...M.implementation.change_history.lines_changed ...
  -- the set union of all the lines_changed attributes
  -- of the change_record objects in change_history.

Figure 3.10: Examples of Complex Access Expressions
sequence, except when a set is also present. In the latter case, the value of the expression is a set. In Figure 3.10, the value of the second expression is a sequence because change.history is an ordered collection of references to change.record objects. The value of the last expression is a set, because the value of each lines_changed attribute is an unordered collection. Expressions of such form that evaluate to sets can contain duplicates. When the result of such an expression is inserted into a set attribute or variable, duplicates are automatically eliminated.

Avoiding Invalid Expressions During Execution

The validity of an expression of the form a.b.c can be statically type checked in all cases except where union types are involved. In addition, due to the presence of null values and variables containing dangling references, access expressions may raise exceptions at execution time. A number of system defined functions are provided to test the presence of such conditions.

Null Values Any attempt to evaluate the value of a variable or attribute that has a null value raises an exception. If the variable M contains a reference to a module, and the lines_of_code attribute of module M has a null value, then execution of the statement

```
loc := M.lines_of_code;
```

will raise an exception.
The *is_defined* function can be used to check if a value is defined, without actually evaluating it and raising an exception in the process. The *is_defined* function takes an access expression as a parameter and returns the value TRUE if the expression can be evaluated without encountering any null values. The first example above can be made more robust by coding it as

```plaintext
if is_defined (M.lines.of.code) then
  loc := M.lines.of.code;
```

The above expression evaluation will still raise an exception if M points to a non-existent object. This is addressed next.

**Non-existent references** The *exists* function takes one parameter, a variable whose domain is an object. It returns the value TRUE if the parameter does not have a null value, and if the object referenced by it exists in the database.

**Type Errors** Consider the subsystem_design object of Figure 3.4, which has a components attribute defined as follows.

```plaintext
components : seq of uniontype {subsystem_design, module_design}, subobject;
```

If the variable SD is of type subsystem_design, then the statement

```plaintext
name := SD.components[1].subsystem_name
```

will raise an exception if components[1] is not a reference to a subsystem_design
object, but refers to a module\_design object instead. The *type\_of* function can be
used to check the type of an object (reference) at runtime. The *type\_of* function
returns the type name of its argument.

The above statement will also fail if SD is undefined, or if the sequence attribute,
components, is empty. The example below illustrates how a runtime exception can
be avoided by using the *type\_of*, is\_defined and size functions together with the
short circuit control forms of Ada.

\[
\text{if exists (SD) and then} \\
\text{size (SD.components) } \geq 1 \text{ and then} \\
\text{type\_of (SD.components[i]) = subsystem\_design then} \\
\text{loc := SD.name.components[i];}
\]

The type name \text{subsystem\_design} is used above as an enumerated scalar constant to
compare it to the value returned by the *type\_of* function. This should be contrasted
with its use as a type identifier in object declarations, and as a class identifier in
the next section.

**Predicate Based Search**

Contents of collections (sets and sequences, including extensions of objects) can
be selectively accessed based on predicate expressions. An expression called a
\textit{collection constructor} (similar to the set constructor of Adaplex [123]) is used to
construct collections from the elements that satisfy the predicate.
The collection constructor is an expression of the following form

\[(\text{identifier in expression where predicate})\]

The value of expression must be a collection, but cannot be another constructor. A valid expression must therefore be the name of an extension of an object, or an access expression. Evaluation of a constructor yields the collection of all values in the expression that satisfy the predicate. The value of a constructor expression is ordered only if the expression itself is ordered.

The following constructor yields a set of references to all module objects that have been implemented by "sarkar"

\[(m \text{ in module where } m.\text{implementor} = \text{'sarkar'})\]

The identifier "module" refers to the extension of the module type. The result of the above expression is a set because all extensions are sets.

The following expression yields a sequence of change_records that represent changes made on the date "Mar-3-1989", to the implementation of the module referenced by the variable M.

\[(\text{ch in M.\text{implementation}.\text{change\_history where ch.\text{date\_of\_change} = '\text{'Mar-3-1989'}}})\]

The original ordering of change_record objects is preserved in the result.

Since the constructor expression represents a collection, it can appear in the beginning of a dotted expression. In the next example, the above constructor is
extended to return the sequence of names of the project members who made the changes.

\[(\text{ch in M.implementation.change.history where ch.date_of_change = 'Mar-3-1989').change_made_by}\]

All the runtime errors that can occur during expression evaluation (as described in Section 3.2.3) can also occur when evaluating the predicate of a constructor. The is_defined and type_of functions, together with the short circuit control forms can be used to avoid such errors. For example, the following constructor expression will raise an exception if for a particular instance of a module, the implementor field has a null value.

\[(m in module where m.implementor = 'sarkar')\]

This is avoided by redefining the constructor as

\[(m in module where is_defined (m.implementor)
and then m.implementor = 'sarkar')\]

This will generate a set of references to only those modules whose implementor field is not null, and whose value is the string "sarkar". This can be made the default interpretation of a constructor predicate\(^6\), whereby the two constructors above have the same semantics. By having such default semantics, predicate definitions can be made simpler.

Type violations can also occur in predicates. Let SD be a reference to a sub-

\(^6\)That is equivalent to the "exclusive" option of the Adaplex set constructor.
system.desig n object. The constructor expression

\[(\text{sdc in SD.components where} \text{sdc.subsystem.name} = 'parser')\]

will raise an exception if any element of the components attribute references a module instead of a subsystem.desig n object. The following variation of the above constructor will avoid a runtime error

\[(\text{sdc in SD.components where} \text{type.of (sdc.components) = subsystem.desig n and then} \text{sdc.components.subsystem.name} = 'parser')\]

Expressions that yield attribute values and collections, standard logical and comparison operators, the member function, and boolean expressions can be used to construct predicates. Similar features are provided in other database languages such as Dial, Adap lex, and Taxis. The allowable complexity of predicates, and the issue of "completeness" of query languages is not explored in this dissertation, since that is not a critical issue in the design of an IPSE object base. Query language issues for (structurally) similar object models have been explored in Orion [11].

3.2.4 Object Modification

Single-valued attributes of objects are modified using assignment statements. Multivalued attributes are modified using the insert, remove and delete procedures described in Section 3.1.1. When an access expression appears as the target of
an assignment to a collection (i.e. as the first parameter of an insert, remove, or delete operator), it must be possible to evaluate it as a valid address of a collection to which an assignment can be made. Since access expressions can contain sets and sequences, such address evaluation is not always possible. Use of such an expression is an error.

For example, the expression

\[ M\.change\.history\.date\.of\.change \]

cannot be evaluated as an address and therefore, cannot be used as the target of an assignment. The value of the expression is only a virtual sequence and does not correspond to a sequence valued attribute which can be directly modified. The expression

\[ M\.implementation\.change\.history[i] \]

cannot appear as a valid target of an assignment statement either. The [] operator can only be used to access the items in a sequence. All modifications to collections have to be made using the collection manipulation operations.

If M is a reference to an instance of module, then

\[ M\.name := \textquote{stack}; \]

assigns a value to the name attribute. If CR1 and CR2 are two references to change_record objects, the following statement appends them to the change_history attribute of M's implementation.

\[ seq\.append (M\.implementation\.change\.history, \langle CR1\rangle); \]
Due to the semantics of part-of relationships, even assignments can have side effects. For example, the assignment

```
SD.subsystem_interface := SI1;
```

results in the deletion of the previous interface_descrn object related to the subsystem_design SD, as well as any of its subobjects, in accordance with the subobject deletion semantics.

The reserved constant `null_value` denotes the absence of a value in an attribute of a database object. The null value can be explicitly assigned to an attribute in an assignment statement if it does not have a nonull constraint. For example

```
SD.subsystem_interface := null_value;
```

removes the link to the existing interface_descrn object, an action which deletes the object as a side effect. Inserting a `null_value` into a set or a sequence is not permitted.

### 3.2.5 Other Language Issues

A special iteration statement is provided for accessing the elements of a collection individually. This, as well as exception handling features are discussed in this section.
Iterating Through Collections

The iteration statement has the following syntax

```plaintext
foreach loop_parameter in collection
    loop sequence_of_statements end;
```

The collection can be the extension of a type, a collection constructor, or any access expression that yields a collection. The semantics of the iteration statement is as follows: the loop_parameter is successively bound to each element of the collection and can be used to access the value of that element. The loop_parameter itself cannot appear as the target of an assignment statement. The following is an example of a foreach statement that iterates through the elements of the change_history of a module

```plaintext
foreach ch in M.implementation.change_history
    loop ... end loop
```

The iteration statement provides a functionality similar to cursors in relational database interfaces to programming languages, and is provided in all database programming languages in some form.

If a collection is a sequence, the foreach statement iterates through the collection in the order maintained by the sequence. Since an operator is available to access a particular element of a sequence, the iteration statement of the underlying programming language can also be used to loop through sequences as shown below

```plaintext
for i in 1..size (M.implementation.change_history)
```
Exception Handling

The Ada exception handling mechanism is used to deal with exceptions that arise during database manipulation. Four predefined exceptions related to object access are raised by the system – CONSTRAINT_ERROR, INTEGRITY_ERROR, ACCESS_ERROR and LOCK_CONFLICT.

The CONSTRAINT_ERROR exception is raised when any implicit or inherent constraints of the data model are violated. These include the violation of referential integrity, an attempt to create a cyclic part-of link, evaluation of a reference to a non-existent object, evaluation of an expression that yields a null value, etc.

The INTEGRITY_ERROR exception is raised by triggers, when they detect the violation of some domain specific integrity constraint. The exception is also raised when declaratively specified constraints such as size limits on collections and nonnull constraints are violated. This is discussed further in Section 3.4.

The ACCESS_ERROR exception is raised when an attempt is made to access an object in a mode for which the user does not have the appropriate authority. Access control issues are discussed in Chapter IV.

The LOCK_CONFLICT exception is raised when an attempt to checkout an object fails, or when an attempt is made to update a checked out object using
short transactions. This is further described in the following section.

3.3 Transactions and Concurrency Control

Traditional DBMSs support the concept of a transaction as an atomic unit of work. A transaction is a sequence of database operations. If the transaction commits successfully, the effect of all the operations are made permanent in the database but if it aborts, all of the database updates performed by the transaction are cancelled. A transaction is also the unit of concurrency control. More than one transaction can concurrently read an object, but only one transaction can update an object. Objects that are updated by one transaction are not visible to other transactions.

In databases that use a 2-phase locking protocol [138], a transaction that wants to read or update an object has to wait if the corresponding object is locked in a conflicting mode by another transaction. The waiting transaction can continue after the transaction holding a lock on the record either commits successfully or aborts. Circular waits lead to deadlock, which results in one of the transactions being rolled back, undoing all its changes and releasing the locks that it held in the database, so that other transactions waiting to access those objects can continue. When an optimistic concurrency control policy [138] based on time stamps is used, transactions do not wait for each other but depending on the patterns of access, deadlocks can be more frequent.
In the protocols described above, it is typically assumed that transactions do not execute for very long periods of time, and do not perform a large number of database operations. Therefore, waiting and undoing of work are acceptable. In design-oriented domains such as software development, these assumptions are not valid. The next section motivates the need for an additional transaction mechanism that can be used when objects need to be locked for extended periods of time that transcend login sessions. The mechanism is based on the locking of objects, but does not cause waiting or rollbacks due to deadlock. It is based on the proposal of Haskin and Lorie [57], and has been extended to incorporate the concepts of links, referential integrity, and triggers. The relationship between conventional and interactive design transactions is also discussed.

### 3.3.1 A Concurrency Control Mechanism for Interactive Design Transactions

In design-oriented domains such as software development, a project member often accesses an object (e.g., the design of a module, the source code of a module, a chapter of a user manual) for long periods of time (days or weeks), while performing updates to it. However, it is essential that other project members be able to read the original version. It is unreasonable to assume for example, that read access to a design document is prohibited for an extended period because a change is being made to it. A concurrency control protocol that prohibits all forms of read access to an object undergoing change is not feasible in such a domain.
In such a mode of work, updates to an object should be logically viewed as occurring in the *private workspace* of the individual performing the modifications. The private workspace contains a temporary copy that no other project member has read or write access to. The original object (referred to as the *baseline* object), is locked to prevent simultaneous modifications on the object. However, certain modes of parallel read access are permitted. When the updates to the private copy are completed, the modified version replaces the baseline copy. Project libraries typically implement mechanisms on top of file systems to support such a mode of work. An object base for software development must provide built-in mechanisms for such support.

Read operations that can occur in parallel with updates are not protected from changes (including deletions) that are made to an object. An alternative mode where a project member can lock an object for the purpose of reading its contents must be also be supported. In such a mode, simultaneous reads can occur but updates are prohibited. File systems typically do not support setting such locks.

Read and write locks can create conflicts that are similar to the ones that occur with conventional locking schemes. The notion of waiting for a lock to be released is not acceptable in a domain where a lock on an object can be held for very long periods of time. Waiting also leads to deadlocks, and the only method of recovery from deadlocks involves undoing the effects of one set of changes performed. Since

---

7If the object is versioned, a new revision is created instead. The problem of representing multiple versions and revisions of an object is not addressed.
such changes may represent many days of work, automatic rollback of all changes is not acceptable. That is one other reason for avoiding a wait mode on lock conflicts. The next section presents a transaction mechanism that addresses these requirements.

3.3.2 A Long Transaction Mechanism

The long transaction mechanism is based on the notion of a public database, and multiple private databases. Objects reside in the public database, where they are accessible to all project members\(^8\). The unit of concurrency control is an entire object. To lock an object for reading or updating for an extended period of time, it must be *checked out* in an appropriate mode. The check out operation is (logically) equivalent to transferring a copy of the object into a private workspace.

When the operations have been performed, the object is *checked in* back to the public database. The first check out of an object in the context of an activity (see Chapter V) begins a long transaction, which is terminated by the last check in operation. At any moment, all the objects that have been checked out by a user constitute the user's *checkout domain*\(^9\).

---

\(^8\)Subject to access control policies discussed in Chapter IV.

\(^9\)In the mode of interaction described in section 3.5 a user may explicitly check out objects. Additionally, user actions may activate triggers and executable components of activities, all of which may check out or check in objects on behalf of the user. *All* such objects constitute the user's checkout domain.
Basic operations

The following operations are provided for interactive design transactions.

Checkout_r (objectlist):- Objectlist is a list of references to objects. The checkout_r operation checks out each object in read mode, setting a read lock on the object in the process. An object that has been checked out in read mode can be simultaneously accessed by others in read mode. However, simultaneous access for updates is not allowed. If the object is already checked out for update, then a LOCK.CONFLICT exception is raised. Automatic waiting does not take place.

Checkout_w (objectlist):- The checkout_w operation checks out each object in the list in update mode. The operation results in the creation of a local copy of the baseline object in a (logically) private workspace. A write lock is put on the baseline object, preventing simultaneous read or update mode checkouts, and modification by short transactions. The LOCK.CONFLICT exception is raised if the object is already checked out in read or write mode and cannot be successfully checked out.

All references to an object by a database user (or by tools, triggers and activities that execute on the user's behalf) resolve to the local copy of the object if the object is a member of the user's checkout domain. References to the object by other users resolve to the baseline copy. The system always searches the private database before looking for the object in the public database. The private copy of an object has the same surrogate id as the baseline object, and the two are
distinguished by a separate field, internal to the system. The automatic resolution of references ensures that all changes to an object take place in the local copy if it exists.

If X (a variable or an attribute) is a reference to an object, the value of X always resolves to the correct copy, as mentioned above. If such a reference is preceded by the “#” symbol, then the reference resolves to the baseline copy, even if a private copy of the object exists in the user's checkout domain. Thus, the expression #X could be used to force all accesses to be made from the baseline copy. This feature is useful in certain kinds of trigger procedures, as described in Section 3.4.

**Data Access and Update without checkout** :- All objects can be read (subject to access control policies) at any time, irrespective of the mode in which they may be checked out. This prevents objects from being "locked out" from read access for extended periods of time, hampering the work of individuals who may just need to refer to a latest document, irrespective of the degree of instability of that document. Access without checkout is not secure, because the object may be modified, or even deleted at any time.

Objects can also be updated without checking them out. That amounts to using the conventional (short) transaction mechanism of the object base. The relationship between long and short transactions is described in Section 3.3.6.

**Checkin (objectlist)** :- The checkin operation completes the transaction for each object in the list. If the object was checked out in read mode, the read lock is
released. If the object was checked out in write mode, the changes made in the local copy replace the baseline copy.

When the keyword all is used as a parameter to the checkin operation, all objects in the user's checkout domain are checked in. However, it is not essential that all checked out objects should always be checked in together. The flexibility offered by the ability to check in individual objects is necessary in the domain. However, it violates the 2-phase locking protocol since new locks may be acquired after some (old) locks have been released.

There are some restrictions on which checked out objects can be checked in individually. These restrictions, and the rationale for imposing them, are discussed in later sections.

Update (objectlist) - This is similar to the checkin operation, except that no locks are released. If an object is checked out in read mode, the update operation has no effect. If an object is checked out in write mode, updating that object results in the baseline copy being made identical to the private copy. The write lock is not released however. If O is an object checked out for write, update (O) is equivalent to the following sequence of commands

\[
\text{atomic checkin (O)}; \text{ checkout (O)}; \text{ end atomic}
\]

where the two actions are \textit{atomic} in the sense that no other user can checkout the object O in between the checkin and the checkout operations above. The keyword all can be used as a parameter, to update all checked out objects.
The update operation is useful in many situations that occur in large projects. For example, while a module is under development a periodic review may be held on a regular basis. An update operation can be used to update the baseline at the time of such a review, so that the reviewers may be able to see the up-to-date changes, without the developer giving up control of the objects.

**Undo** - This command only applies to *all* checked out objects. For each object that is checked out in read mode, the read lock is released. For objects checked out in write mode, the local copy is deleted and the write lock on the baseline copy is released. Thus, none of the work that was done on the private copies is reflected in the baseline.

**Checkout.r*, checkout.w*, checkin** - These are generalized versions of the above operations that take the same parameters (a list of objects). The operation is applied to each object and transitively to *all the subobjects* of that object. If the variable SD refers to a subsystem design object, the operation

```plaintext
cHECKOUT.W* ( SD);
```

checks out SD and all its subobjects in write mode. These operations allow an object and its subobjects to be treated as one complex object (see Section 3.1.3).

**Object Creation in the Context of Long Transactions**

The create object operation described in Sections 3.2.1 creates an object in the public database. In the context of long transactions, this is not always the creation
semantics desired. Since a baseline object is created, the undo operation is not sufficient for deleting it. An explicit delete_object operation will have to be invoked to undo the creation\textsuperscript{10}. This violates the general principle of allowing all operations to be performed in a "temporary" mode within long transactions, leaving open the option of either committing or aborting the changes made at a later time.

An alternative creation operation called \textit{create_object_local} is introduced. The parameters to \textit{create_object_local} are the same as its variant, but the semantics are different. When an object is created with the \textit{create_object_local} operation, only the private copy is instantiated, and that copy is automatically checked out in write mode for the user. To make the creation permanent, the object must be explicitly checked in. If a \textit{create_object_local} is followed by an undo operation, the private copy is deleted and the net effect is as if the creation never took place.

The \textit{copy_object_local} operation has the same semantics, since it is equivalent to a create operation followed by a set of modifications.

\textbf{Object Deletion in the Context of Long transactions}

The delete_object operation introduced in Section 3.2.2 can only be used to delete baseline objects which are not checked out. None of the subobjects of an object being deleted must be checked out either. Even if the object being deleted is checked out for update by the user who initiates the deleting transaction, a

\textsuperscript{10}That itself is not foolproof since between the \textit{create_object} and \textit{delete_object} operations, some other user may check out the object in a conflicting mode.
LOCK_CONFLICT exception is raised.

The delete_object_local operation is used to delete the private copy of an object. If the object is not already checked out, a CONSTRAINT_ERROR exception is raised. Otherwise, the private copy of the object is deleted, but the deletion is not made permanent until a checkin (all) command is issued. All subobjects of the object are deleted too, according to the semantics of deletion, and if any of the subobjects are not checked out, they are explicitly checked out before deletion of the private copy. If the deletion is followed by an undo operation, the local record of the deletion is removed and there is no net effect on the baseline since the original object and all its subobjects still remain in the baseline.

Since references to objects resolve to the private copy, a deleted object cannot be selectively checked in by the user who performed the delete. Consider the semantics of the following statements:

```
delete_object (0);  -- 0 is a variable
checkin (0);
```

After the deletion is performed, object O no longer exists in the user's checkout domain and the checkin operation will raise a CONSTRAINT_ERROR exception. Deletions can only be committed by performing a checkin (all) operation. This non-uniformity is a consequence of allowing selective checkins and automatic resolution of references.

A solution to the above problem, and a useful feature in general, is provided by
the operation `delete_object.immediate`, which takes a list of objects as a parameter. Each object in the list must be checked out for update by the user. The private copy is deleted and checked in immediately. The treatment of subobjects is similar to that of the `delete_object.local` command.

### 3.3.3 Long Transactions, Constraints and Side Effects

In traditional DBMSs, a short transaction is typically considered to be the *unit of consistency* during which temporary violations of integrity constraints can occur. In the mechanism proposed here, certain constraints are checked immediately, while others are checked when a long transaction is committed with a checkin operation\(^\text{11}\).

Declaratively specified constraints are checked immediately. Violations of key, range, size (of collections) and nonnull constraints are reported immediately, even if they occur in checked out objects. Because design transactions are typically very long, reporting simple constraint violations weeks after they occur is not particularly beneficial to the user. The trigger mechanism provides the flexibility of defining more complex constraints, that may be either be checked as soon as the object base is updated, or when the changes made during a long transaction are committed.

\(^{11}\)This checking of constraints is controlled by two classes of triggers. To keep the proposal simple, a third category of triggers which check certain constraints after a *short transaction commit* is not proposed, though that is the central concern of conventional integrity management proposals.
The policy of not committing changes to attributes (including links) of a checked out object until checkin time impacts the enforcement of referential integrity. Consider the two cases in Figure 3.11. In case 1, object A is checked out for write by user 1, while object B is not checked out. Even if user 1 removes the link from A to B, user 2 will not be able to delete object B until user 1 commits the changes to A. Until A is checked in, object B in the baseline still participates in the relationship and any attempt to delete it will violate referential integrity. In case 2, objects A and B are both checked out for write by the same user. If the link between A and B is broken, the private copy of B can be deleted by user 1 because it has no incoming links, even though the link between A and B still exists in the baseline.

3.3.4 Restrictions on Checkout, Checkin and Update Operations

While the long transaction mechanism is fundamentally useful, delaying changes to the database, combined with the flexibility of checking in individual objects can have certain undesirable side effects. These side effects, and the restrictions on the operations which remedy them are described below.

The first restriction is on setting links to objects by modifying attributes of checked out objects. An object that is not checked out can be read by anybody with appropriate access rights. However, an object that is to be the target of a link must be checked out in read mode. Since the target object is not modified,
Figure 3.11: Long Transactions and Referential Integrity
the option of not locking it for read could have been provided. However, in the presence of multiple users, that can have undesirable consequences at checkin time, as illustrated below.

In Figure 3.12, the private copy of object A sets a link to object B, which is not checked out. The change is not reflected in the baseline, because the changes to A are not committed. Another user checks out B for write, deletes it, and
commits the changes. When the user who set the link from A to B tries to commit the changes made, a CONSTRAINT.ERROR exception is raised because the link from A to B cannot be committed. Additionally, any access to object B through the private copy of object A (all accesses to A by the user access the local copy of A) will also raise an exception since the link would be a dangling reference. Forcing the target of a link to be checked out for read, prior to creating the link solves part of the problem. It is still possible for object B to be individually checked in, without checking in object A, leading to the same problem as before. That is addressed by the next restriction on individual checkin and update operations.

An object cannot be checked in alone, if there exist other objects still not checked in that may possibly affect or be affected by the commit action. In case 1 of figure 3.13, since object A depends on object B to exist when it is checked in, object B cannot be checked in alone. The operation checkin (B) will raise a CONSTRAINT.ERROR exception. The operation checkin (A, B) will execute successfully.

Another example of a dependency between checked out objects is shown in case 2 of Figure 3.13. Object A exists in the baseline and as a private copy. Object B has been created using the create.object.local operation and only a private copy exists. Object A cannot be checked in alone because the link to B cannot be committed in the absence of a baseline copy of B. Object B cannot be checked in alone either, because of the link between A and B in the private copy. The
operations checkin (A) or checkin (B) will raise exceptions, but checkin (A, B) will succeed.

3.3.5 The Relationship Between Long and Short Transactions

While the long transaction mechanism is introduced to overcome the limitations of short transactions for design oriented interactive domains, the mechanism itself is implemented using conventional (short) transactions. Also, while the long transaction mechanism is useful for interactive transactions which span login sessions, conventional database transaction mechanisms are more appropriate for software tools that need to access objects for short periods of time in batch mode (e.g., compilers). Therefore, both mechanisms need to be supported by the IPSE object base. These issues are discussed in the following sections.

The Implementation of Long Transactions

The checkout operation sets object level locks, and makes a private copy of an object if it is checked out in write mode. The locking and copying process itself is implemented as one conventional transaction, but once the operation is successful, the changes are usually committed to the database (see Section 3.4.5 for exceptions to this). While a lock that controls a short transaction is owned by the transaction, a lock used to checkout an object is owned by the activity under whose control the user performed the checkout operation (see Chapter V).
Case 1

Private DB

Baseline

B cannot be checked in individually

Case 2

Private DB

Baseline

Either A or B cannot be checked in individually

Figure 3.13: Object Dependencies and the Checkin Operation
All database programming languages as well as commercial DBMSs provide constructs or operations to delimit (short) transactions. In the current proposal of this language, checkout and checkin operations are provided to control long transactions, and `atomic` and `end atomic` constructs are provided to control conventional transactions. If database operations are not enclosed within `atomic` and `end atomic` pairs, each operation is treated as a short transaction. The sequence of statements

```plaintext
checkout (01); checkout (02); checkout (03);
```

is usually executed as:

```plaintext
atomic checkout (01); end atomic;
atomic checkout (02); end atomic;
atomic checkout (03); end atomic;
```

If the above statements are programmed as:

```plaintext
atomic
checkout (01); checkout (02); checkout (03);
end atomic
```

the short transaction is committed only after all the three objects have been checked out successfully. Increasing the number of operations within a short transaction increases the chances of deadlock and rollback and should not be used unless necessary. Since the long transaction mechanism provides facilities to undo changes, enclosing checkout operations within an atomic unit is unnecessary to
facilitate recovery.

3.3.6 Mixing Long and Short Transactions

While checkout and checkin operations are provided for interactive design transactions, conventional database transactions can also be used. As mentioned in the previous section, each database operation is treated as a conventional transaction, unless a number of such operations are declared to be atomic. If a sequence of operations declared to be atomic fails because of a deadlock, it is automatically retried.

Since long and short transactions can be mixed, certain restrictions have to be set on their interaction. Short transactions can read and update objects that have not been checked out, with the usual semantics of waiting on lock conflicts, and rollback on deadlock or database error. Irrespective of the mode of checkout of an object, a short transaction can still access attributes of the object. However, any attempt to update an object checked out in update mode raises a LOCK_CONFLICT exception unless the write lock is owned by the user who initiated the short transaction. In the latter case, only the private copy is updated.

If a short transaction sets a link by modifying an attribute of a baseline object, the target need not be checked out in read mode. Even if the target is checked out for read, or if it is checked out for update by the initiator of the short transaction, the operation will succeed. However, if the target is checked out in update mode by
another user, a LOCK_CONFLICT exception is raised, because it violates the restriction on setting links to checked out objects that was discussed in Section 3.3.4. If the setting of an object-valued attribute by a short transaction is viewed as the sequence of operations

```plaintext
checkout (0); ...modify attributes...; checkin (0);
```
then the above constraints on setting links can be seen to be consistent with the restrictions on similar operations in the context of long transactions, the goal being to protect a checked out object from side effects (involving referential integrity) which could prevent its deletion. The semantics of object creation and deletion have already been presented.

The original proposal on design transactions by Haskin and Lorie [57] allowed long and short transactions to be mixed. Database objects could be accessed using System-R operations which used the semantics of conventional transactions. A special checkout operation was provided to create a private copy of an object. The checkout operation set a special lock on the object (to prevent simultaneous conflicting checkouts) and also set conventional read (S) locks on all components (tuples) of the object, to permit simultaneous read operations using System-R operations.

While the mechanism of setting S locks on all components of an object does allow simultaneous access by conventional transactions, any attempt to update such a component would result in an indefinite suspension of the short transaction
because of the possibly long duration of the checkout. This issue is not explicitly addressed in the paper, though the option of using lower level non-waiting versions of System-R routines for acquiring locks is suggested. In the alternative mechanism proposed above, short transactions still wait for acquiring conventional locks, but if the object has been checked out, an exception is raised and no waiting takes place.

3.4 Database Triggers

A simple form of database triggers ([47], [44], [147], [127]) is provided in the model for maintaining database integrity, and for propagating the side effects of update operations on the object base. The notion of triggers is based on the concept of procedural attachment, and is similar to daemons of frame systems [49] and action routines of Gandalf [4]12. Similar facilities can be simulated on object-oriented databases that allow the definition of get and put methods (e.g., VBase [97]).

The syntax and semantics of trigger procedures is described in the following section, to illustrate the use of triggers for constraint checking and propagation. Section 3.4.5 describes how the short transaction mechanism is used to implement triggers semantics.

12A non-trigger-oriented approach to building active databases is described in [61].
3.4.1 Defining Trigger Procedures

A trigger is a body of code, that is logically associated with an attribute of an object, or with an entire object. A trigger is never explicitly invoked by the user of the object base. Every trigger has an associated precondition, and the trigger is automatically activated by the object manager whenever the precondition is satisfied.

The precondition of a trigger is a "primitive" update operation that can be performed on the object base\textsuperscript{13}. Since a trigger is activated by an update operation, triggers cannot be attached to inverse attributes. The body of a trigger can contain statements that manipulate the object base, invoke tools and interact with user.

The fundamental operations that can appear as trigger preconditions are variations of the operations create, delete, modify, insert and remove\textsuperscript{14}. For example, when a trigger attached to an attribute has a certain form of a modify precondition, the trigger is executed when that attribute is modified. The different forms of the modify precondition determine whether the trigger is executed as soon as the modification is made or only when the modification to the attribute is being checked in. The former are referred to as immediate triggers and the latter as

\textsuperscript{13}Various trigger proposals vary in the degree of complexity of the precondition, which, in its most general form, can be an arbitrarily complex query. The form of preconditions defined above represents the other end of the spectrum.

\textsuperscript{14}In the proposal in [109], a "get" precondition was also included. Due to the difficulty of efficient implementation of such triggers, particularly in conjunction with constructor expressions, that feature is not included in this description.
delayed triggers. The precondition of a trigger, as well as the attribute or object it is attached to, determine the number and type of formal parameters it should have. The object base designer defines the names of the formal parameters; these are bound to specific values by the object manager when the trigger is invoked.

The different preconditions of immediate triggers are create, delete, modify.pre, modify.post, set.insert.pre, set.insert.post, set.remove.pre, set.remove.post, seq.insert.pre, seq.insert.post, seq.remove.pre, and seq.remove.post. The preconditions of delayed triggers are create.checkin, delete.checkin and modify.checkin. These preconditions are described below.

Preconditions of Immediate Triggers

Immediate triggers are executed as soon as an update is made to the object base. The preconditions and the formal parameters of immediate triggers are described below.

Create, delete: These are preconditions of immediate triggers that are associated with an entire object. A trigger with a create (delete) condition is executed as soon as the object is created (deleted). A trigger with a create precondition is executed after the object is created. A trigger with a delete precondition is executed before the object is deleted. Immediate create and delete triggers do not have any parameters. These triggers are invoked in response to all variations of the create and delete operations, respectively.
Modify pre, modify post: These are preconditions of immediate triggers that can be attached to a single-valued attribute of any type. A modify pre trigger is executed before the modification is made (i.e., the object base still contains the old value), while the modify post trigger is executed after the attribute has been updated.

A modify pre trigger is often defined to check if the value that is going to replace the existing value of an attribute satisfies some constraint (e.g., the value should always increase). The existing value can always be accessed from the object base, but the new value must be passed to the trigger as a parameter. Therefore, modify pre triggers have one formal parameter whose type is the same as the type of the attribute. The exception to this is text and binary attributes. Instances of text and binary values can only exist as attributes of objects and local variables and formal parameters of that type are not permitted in the language. Therefore, a modify pre trigger associated with a text or binary attribute does not have any formal parameter.

A modify post trigger does not have any parameters. The attribute in the object base contains the newly replaced value and the primary function of such triggers is to propagate side-effects of a modification. Note that there is no restriction on what a trigger procedure can do, and therefore, it is also possible for a modify pre trigger to propagate side effects of a modification.

The modify pre trigger for an attribute is called before any declarative con-
straints are checked. The modify_post trigger is called after any system-defined side effects have been computed. For example, if an attribute represents a part-of relationship, modifying its value is equivalent to removing the old link, the side effect of which could result in the deletion of that target subobject according to the semantics of subobject deletion. That deletion is carried out by the system before any modify_post trigger associated with the attribute is executed.

The invocation rule for modify_pre triggers is violated at object creation time. In the example below

```vql
create_object (module, name => 'stack');
```

the name attribute has to be initialized since it has a nonull constraint. The object cannot be created without this attribute having a value. Therefore, even if the name attribute had a modify_pre trigger, invoking it would be inconsistent with its semantics because the attribute has already been updated in the object base. Therefore, when an object is created, only modify_post triggers are executed for attributes with nonull constraints.

**Seq_insert_pre, seq_insert_post**: These are equivalent to the modify preconditions, except that they are preconditions of triggers attached to sequence attributes.

A seq_insert_pre trigger is executed prior to the insertion of a single value into a sequence. The seq_insert_post trigger is called after the insertion is complete and the object base has been updated. If M is a reference to a module_body, and the sequence attribute change_history has a seq_insert_pre trigger, the execution of the
procedure

    seq_insert (M.change_history, <CH1, CH2>, 2);

will result in the trigger procedure being invoked twice.

A seq_insert_pre trigger has two formal parameters, the value being inserted and
the position at which the insertion is being made. When the trigger is attached
to a sequence whose elements are of type text or binary, the value parameter is
absent. Seq_insert_post triggers do not have any formal parameters.

Set_insert_pre, set_insert_post :- These are preconditions of triggers associated
with sets. Their semantics are the same as for sequences, except that set_insert_pre
triggers do not need a parameter that indicates the position of insertion.

Seq_remove_pre, seq_remove_post :- These are equivalent to the insert condi­
tions, except that triggers with such conditions are executed when elements are
removed from a sequence. A seq_remove_pre trigger has two formal parameters,
the value of the element being removed, and the position from which it is being
removed. If the element type is text or binary, then the value parameter is absent.
Seq_remove_post triggers have no formal parameters.

Set_remove_pre, set_remove_post :- These are similar to seq_remove_pre and
seq_remove_post, except for the absence of the value parameter.
Preconditions of Delayed Triggers

Delayed triggers are executed only when the private copy of an object is checked in. Since triggers are only associated with update operations, objects that have been checked out in read-only mode cannot cause any triggers to be executed at checkin time. Also, baseline objects that are directly created, deleted or modified using short transactions do not cause delayed triggers to be invoked. For objects that have been checked out for update, delayed triggers are invoked only for those attributes that have been actually updated during the long transaction\(^\text{15}\).

The following preconditions determine when delayed triggers are executed.

**Modify.checkin**: This condition is associated with delayed triggers attached to *any* attribute (i.e. no distinction is made between single and multi-valued attributes). Modify.checkin triggers have no formal parameters. Such a trigger, if associated with a collection, is invoked if any insertion or removal of elements has been performed during the long transaction. Irrespective of the number of such insertions and deletions, the trigger is invoked only once. The “old” value of the attribute exists in the baseline object, while the “new” value that will replace it is in the private copy.

**Create.checkin, delete.checkin**: Create.checkin (delete.checkin) triggers are invoked when an object that was created (deleted) as part of a long transaction

\(^\text{15}\)Implementation of such a facility requires that a *log* of operations on objects be maintained by the object manager. Implementation details are discussed in Chapter VI.
is checked in. These triggers do not have any parameters. Note that since a delete_object_immediate operation is equivalent to a delete followed by a checkin, both immediate and delayed delete triggers are invoked in succession.

**System Defined Variables in Triggers**

The system defined variable, `self` is automatically defined for every trigger. In an executing trigger, it contains a reference to the object instance whose modification caused the trigger to be invoked. Self always resolves to the correct copy (private or baseline) of the referenced object. The variable `#self` always resolves to the baseline copy. This form of reference is useful in modify_checkin triggers, where `self` can be used to refer to the "new" object, and `#self` to refer to the "old" object. Note that `self` is undefined in delete_checkin triggers because the private copy has been deleted, while `#self` is undefined in create_checkin triggers because prior to a checkin operation, a baseline copy does not exist. Such erroneous use can be statically detected. Other predefined variables are introduced in the examples.

**3.4.2 Using Triggers for Constraint Checking**

A number of examples are presented to illustrate how triggers may be used for checking constraints that cannot be expressed declaratively.
Constraint Checking with Immediate Triggers

In Figure 3.14, an is_frozen attribute and a trigger have been added to the module object, and an inverse link and a trigger have been added to the module_body object (see Figure 3.2). The purpose of the triggers is to prevent frozen modules from being modified. The is_frozen attribute is set by a manager with appropriate authority. If set to "yes", any attempt to change the code attribute of module_body is aborted, even if the person attempting to make the change has update access to the module. A similar trigger should also be associated with module_specification.specs.

The trigger attached to the is_frozen attribute ensures that the value of the attribute is either "yes" or "no". The trigger associated with the code attribute (which represents the source code) prohibits any modification to the implementation if the module is frozen. The safeguard provided by this trigger is successful only if programmers with write access to a module.body are unable to modify the is_frozen attribute of module. That is ensured with an appropriate definition of role specific user views that is discussed in Chapter IV. The modify_pre trigger attached to module_body.code has no formal parameter since the attribute is of type text. Note that triggers do not (need to) have names because they cannot be explicitly invoked.

A stronger policy about freezing different revisions of modules might specify that once a revision is frozen, it can never be modified. Such a restriction can
type module is object
attributes
...
  is_frozen : string (3);
  interface : module_specification, subobject;
  implementation : module_body, subobject;
......
triggers
  trigger (new_value : string (3))
    on is_frozen:modify.pre is
    begin
      if new_value /= 'yes' and new_value /= 'no' then
        abort ('Value of is_frozen attribute must be yes or no', INTEGRITY_ERROR);
      end if;
    end;
end object;

type module_body is object
attributes
  parent_module : module,
  derivation = inverse module.implementation;
  change_history : seq of change_record;
  code : text;
......
triggers
  trigger on code:modify.pre is
  begin
    if self.parent_module.is_frozen = 'yes' then
      abort ('Module frozen. Implementation cannot be changed', INTEGRITY_ERROR);
    end if;
  end;
e nd object;
end object;

Figure 3.14: Prohibiting Changes to a Module With a Trigger
... trigger (new_value : string (3))
  on is_frozen.modify.pre is
  begin
    if new_value /= "yes" and is_frozen = "yes" then
      abort ("Module already frozen", INTEGRITY.ERROR);
    else if new_value /= "yes" and new_value /= "no" then
      .......
    end if;
  end;
...

Figure 3.15: A Modified Change Policy

be represented by a different version of the modify.pre trigger attached to the
is_frozen attribute, shown in Figure 3.15. If the value of the is_frozen attribute
is already "yes", any attempt to change its value is immediately aborted. Thus,
a module once frozen can never be "thawed". This is the standard semantics of
systems that maintain revisions of modules.

Scope rules for triggers allow them to access attributes of the object type to
which they are attached. In Figure 3.15, the reference to is_frozen is equivalent to
self.is_frozen. The abort procedure provides a mechanism for a trigger to report
the nature of the error that caused an operation to be aborted. This is further
described in the section on error handling.
Constraint Checking with Delayed Triggers

Delayed triggers are executed at checkin time. For every attribute that has an associated modify_checkin trigger, and which was actually modified during the long transaction, the trigger is invoked at checkin time. If any trigger invokes the abort procedure, the checkin operation fails. In particular, when executing a checkin (all), an INTEGRITY_ERROR exception raised by a trigger during the checkin of any object causes the checkin of all the objects that belonged to the user's checkout domain to be undone. However, the checkin process could be continued for the remaining objects to uncover more constraint errors. That is an implementation issue.

The example in Figure 3.16 illustrates the use of a delayed trigger to check for key constraint violations at checkin time. When making modifications to the implementation of a module which is part of a subsystem, a programmer typically wants to test the initial modifications by linking the changed module to the versions of the remaining modules in the subsystem that existed at the time the modification request was made. Since other programmers may be simultaneously changing other module implementations, there are two options available to the programmer for creating an environment in which to test local changes:

- Check out the module to be modified in update mode, and checkout the remaining modules in read mode to prevent modification.
type module.body is object
  attributes
    name : nametype;
  ... 
  triggers
    trigger on create_checkin is
    begin
      if size ((m in module.body where m.name = self.name)) > 1
        then
          abort ('Duplicate module body '-' & self.name,
                     INTEGRITY_ERROR);
        end if;
      end;
    end object;

Figure 3.16: Preventing Duplicate Copies in the Baseline

- Check out the module to be modified in update mode and create local copies
  of all the other modules in the subsystem.

Option 1 is impractical because it would prevent simultaneous changes to other
parts of the system. Option 2 is typically used, but one must ensure that these local
copies are deleted after the changes have been tested with the previous baseline.
In particular, one must prevent local copies from being checked in to the base-
line, resulting in more than one copy of what is essentially the same module.body.
This is achieved by the trigger which effectively implements a delayed key con-
straint on module.body.name. Note that if module.body.name had a declarative
key constraint instead, the copy_object operation would raise an exception.
type module_body is object
  attributes
    reason_for_change : text;
    .......
  triggers
    trigger on code.modify.checkin is
    begin
      if not isDefined (reason_for_change) then
        abort ("Reason for changing module" & self.name & "'not filled in'", INTEGRITY_ERROR);
      else .......
      end if;
    end;
end object;

Figure 3.17: Preventing Checkin of Incomplete Objects

The declarative key constraint even disallows duplicate private copies. Therefore, if one user has created a module with a certain name but not yet checked it in, that would still prevent another user from creating a (private copy of a) module with the same name, since a declarative key constraint exists on the name attribute (Figure 3.2). This constraint cannot be enforced by triggers\textsuperscript{16}, and therefore, has been provided as the semantics of the declarative constraint.

Another example illustrating the usefulness of delayed constraint checking is shown in Figure 3.17. The module_body object of Figure 3.16 has been extended to include a “reason_for_change” attribute, in which a programmer is expected to

\textsuperscript{16}An immediate create trigger will only “see” the user’s local copy and will be unable to account for local copies of other users.
fill in a description of the change made. The value of the attribute is initially null when the implementation is checked out for update. A delayed trigger ensures that the value of the attribute is not null at checkin time, if the code of the module has been changed during the long transaction. A similar trigger should also be defined for the module specification object.

Error Handling

The abort procedure is predefined by the system. The parameter to abort is a message that describes the error which caused the operation to fail, and the exception that is to be raised. The abort procedure inserts information about the location of the error (the surrogate id of the object, the name of the attribute if applicable), and the error message in an error_record object. Then, it raises the desired exception. The system-defined function “get_error_record” returns a reference to the current error record. It is typically examined in exception handlers. The predefined object type error_record is shown in Figure 3.18.

The predefined function “remove_error_record” removes the error_record created as a result of an abort. It should be executed by an exception handler that continues execution after dealing with an INTEGRITY_ERROR exception. Otherwise, subsequent errors will accumulate in the same error_record object.
Figure 3.18: Error handling

3.4.3 Using Triggers for Information Propagation

The generality of trigger procedures can be exploited not only to check for constraints, but also to automatically make changes to the database. Update operations performed by a trigger can be viewed as actions that maintain the consistency of the object base in response to the original update. Examples of such actions are outlined below.

An object can be created (deleted) in response to other creation (deletion) operations. For example, if the compilation object of Figure 3.2 is deleted (by a compiler, or a "garbage collector"), a trigger could automatically delete the corresponding object_code object, maintaining the constraint that an object_code
object can only exist in the database as long as it is related to the source code object by a compilation relationship.

If a module specification is updated, all other modules that use some resource provided by the module could be flagged (by setting some attribute), signalling the condition that they may need to be recompiled.

Updates to objects could activate triggers, that record history information in a global object that is specific to a task assigned to a project member. Such a feature could be used to implement a “distributed” history manager, where the logic of what history information to record is embedded in the triggers associated with an object.

In general, triggers, in conjunction with the generic editor tool described in Section 3.5, can be used to implement special purpose interactive tools, where the triggers implement the semantics of the tool while the editor provides browsing and object manipulation operations.

Section 3.4.3 illustrates triggered updates with two examples. Section 3.4.6 motivates the need for a special class of attributes called derived attributes. The relationships between trigger chains, short transactions and long transactions is described in Section 3.4.5.

Examples of Triggered Updates

The example in Figure 3.19 extends the example of figure 3.17. If the imple-
type module_body is object
attributes
  name: nametype;
  parent_module : module,
      derivation = inverse module.implementation;
  change_history : seq of change_record;
  code : text;
  reason_for_change : text;

triggers
  trigger on code.modify_checkin is
    ch_rec : change_record; -- local variable
    begin
      if not is defined (reason_for_change) then
        abort ('Reason for changing module'
          & self.name & 'not filled in', INTEGRITY_ERROR);
      else
        ch_rec := create_object (change_record);
        -- No long transaction is involved in object creation.
        ch_rec.change_made_by := ...;
        ...
        ch_rec.reason := self.reason_for_change;
        seq_append (change_history, <ch_rec>);
        self.reason_for_change := null_value;
      end if;
    end;
end object;

Figure 3.19: Recording Changes at Checkin Time
mentation has been modified and the reason_for_change attribute has a non null value, the modify_checkin trigger creates a new change_record object, fills out various attributes of change_record including the "reason" attribute, and appends it to the change_history for the module_body. It also assigns a null value to the reason_for_change attribute of the module object to prepare it for the next modification session. A similar trigger is defined for module_specification.specs.

Note that the trigger that is invoked at checkin time updates the object being checked in and also creates a new change_record object using the conventional transaction mechanism. A trigger can also use the "local" variation of the create and delete operations to manipulate private objects. If a checkin (all) operation is in progress, any private objects created or deleted by a delayed trigger are also checked in. However, if only selected objects are being checked in, private objects created/deleted by delayed triggers, but not committed, become part of the user's checkout domain.

The example of Figure 3.20 illustrates a triggered deletion. When an instance of a compilation object (see figure 3.5) is deleted, any associated object_code object should also be deleted. It is possible that the compiler (or the user, using the interactive interface) that deletes the compilation object correctly deletes the object_code object also, in which case the trigger does not perform any actions. The trigger acts as a safeguard in case the tool or the user does not correctly maintain the integrity of the database.
The trigger ensures that the object_code is deleted in the same manner as the compilation object. If the compilation object is deleted by a tool using the delete_object operation, the object_code is also deleted in a similar manner so that when the short transaction is committed, both deletions become permanent. If the delete_object_local operation is used instead, the trigger ensures that both the compilation and object_code objects are deleted but not checked in. In case the changes are aborted, both objects will remain in the baseline. The same is true if the delete_object_immediate operation is used. Current_operation is a predefined system variable of type enumerated scalar, whose elements represent the primitive update operations.

3.4.4 Object Deletion and Immediate Triggers

The capability of triggers to propagate side effects, along with the object base semantics of referential integrity and cascaded deletion, can cause complex interactions. These interactions are illustrated by examples. Figure 3.7 on page 94 outlines the operations performed when an object is to be deleted. These operations are performed after the system has verified that the deletion of the object will not violate referential integrity.

When deleting an object, for each attribute that represents a non-part-of relationship, the link is removed. If it is a part-of relationship, the link is removed and the subobject is deleted. Immediate triggers that are invoked as a result can
type compilation is object
  attributes
    source : uniontype {module, module_body,
                        module_specification};
    object : object.code;
    .......
  triggers
    trigger on delete is
    begin
      if is_defined (self.object) then
        if current_operation = DELETE_OBJECT then
          delete_object (self.object);
        else if current_operation = DELETE_OBJECT_IMMEDIATE then
          delete_object_immediate (self.object);
        else if current_operation = DELETE_OBJECT_LOCAL then
          delete_object_local (self.object);
        end if;
      end if;
      exception
        when LOCK_CONFLICT =>
          abort ("'object_code checked out in conflicting mode'",
                 INTEGRITY_ERROR);
    end;
end object;

Figure 3.20: Triggered Delete
set links to the (ancestor) object being deleted (because the object still exists), as a result of which the object cannot be deleted because referential integrity will be violated. This is not a technique for aborting deletions; raising an exception is a much cleaner method. It is just an example of complex interactions that can occur between the semantics of triggers and the implicit semantics of database operations, that a designer of trigger procedures should be aware of.

3.4.5 Trigger Chains and Transactions

Since triggers can invoke database update operations, they can cause more immediate triggers to be invoked, leading to a chain of trigger activations (see Figure 3.21). If a trigger (say T1) updates an attribute which has an associated immediate trigger (say T2), T2 is invoked immediately, and control returns to T1 only when the execution of T2 is complete. Trigger chains are thus like procedure invocations and should be contrasted with the flow of control of rule based systems (e.g., [51]), and formalisms like action equations [67].

A chain of trigger activations initiated by a database update executes as one short transaction because that is the unit of consistency for a trigger chain. Triggers are viewed as performing constraint checking or consistency maintainence operations. Enclosing the trigger chain within a conventional transaction that encompasses the original database update ensures that either all the consistency checks and side effect propagation operations succeed along with the original op-
eration, or else everything is rolled back (if the rollback occurs due to a deadlock, it is automatically retried). The atomicity provided by a 2-phase locking protocol also ensures that triggers activated by other users do not see intermediate results of derived computations.

Because of these semantics, explicit constructs for controlling the scope of short transactions (atomic ... end atomic), should not be permitted in trigger procedures unless a nested transaction facility can be implemented. The decision to enforce the default semantics was motivated at least in part by the difficulty of implementing the more flexible alternative, on an existing database that does not support nested transactions. Active DBMSs like HIPAC [47] provide greater flexibility to the database designer by permitting a trigger to execute as a (short) transaction that is separate from the transaction that caused it to be invoked.

Forcing a trigger chain to execute as one short transaction causes long transaction operations, and even entire tool invocations, to execute without committing if they are invoked from within a trigger. Therefore, a tool envelope (see Chapter IV) that has explicit statements to control the scope of short transactions, cannot be properly executed if invoked from a trigger. One solution is to ignore directives in tool envelopes to control short transactions, if the tool invocation occurs from a trigger (or in the more general case, if the statement that invokes a tool is a member of a group of statements declared to be atomic). As before, the availability of a nested transaction mechanism would allow implementation of
a more flexible scheme. Forcing tool invocations and checkout/checkin operations to execute as one conventional transaction partly counters the benefits of the long transaction mechanism. Therefore, tool invocations and long transaction operations should be embedded in triggers only if absolutely necessary for maintaining database consistency. Experience with writing triggers has shown that typical consistency maintaining operations are quite short.

3.4.6 Information Propagation and Attributes Derived by Triggers

In the example of Figure 3.19, the modifications to the change_history attribute by a trigger procedure are local to the object in which the trigger is embedded, and there is no possibility of the trigger being unable to make the change due to a lock conflict. However, in the example of Figure 3.20, such a conflict can occur if the object_code and compilation objects are checked out by different users, an erroneous situation which causes the trigger to abort the deletion of the compilation relationship because the consistency maintaining actions cannot be carried out\(^ {17} \).

Aborting an operation because a consistency maintaining side effect cannot be carried out is too drastic a step in many cases. There are many scenarios in which values computed by triggers can be usefully propagated to other objects, but not permitting the original operation to complete successfully if the propagation cannot be carried out (due to a lock conflict) is detrimental to progress. Consider

\(^ {17} \text{Such conflicting modes of checkout can be prevented by stricter access control.} \)
the example where a "uses" attribute and its inverse relate modules that provide resources with the modules that use them. If a provider of a resource is modified, a trigger could detect the condition and set flags in all the affected modules. However, since the unit of a long transaction is an entire object, it is quite possible that some of the flags cannot be set by a trigger because the affected modules are checked out by other project members. While the trigger will not get into a wait state (as per the semantics of the checkout operations), it will be unable to perform its task successfully on all affected modules. The problem is caused by the long duration of design transactions, and the solution to the problem works around the restrictions imposed by the long transaction mechanism.

A special class of derived attributes whose values are computed by triggers can be defined in any object. They are referred to as *trigger-derived* attributes. Trigger-derived attributes have two special properties:

- Triggers updating these attributes are not restricted by the long transaction mechanism.

- They cannot be modified by a database user.

The first property is motivated by the need to avoid conflicts caused by long transaction locks. Even if the object to which the derived attribute belongs is checked out by a different user, the update is permitted by the system. The second property is motivated by the semantics of trigger-derived attributes. If a
derived attribute is viewed as a value computed by the system, it is not meaningful to permit a user to override that value.

The example of Figure 3.21 adds a few more attributes to the module object to illustrate the use of trigger-derived attributes. The modify_checkin trigger associated with the specs attribute of module_specification is similar to the trigger of Figure 3.19 because it records the change history of the specification. However, it also sets the specs_modified attribute of the (parent) module object. A trigger attached to that attribute sets the is_dirty attribute of all modules that use resources of the changed module. The declaration - derivation = trigger, defines is_dirty to be a derived attribute whose value is computed by a trigger procedure. Since the trigger does not update non-derived attributes of other objects, it will never fail because of a lock conflict.

The example has been presented in a language independent fashion. Different languages and their environments represent such information and maintain consistency in different ways. In C, a compilation unit (file) can be thought of as a module body, and interfaces are "h" files that are included by source files. The "include" macro represents uses relationships between interfaces and bodies. In Ada, packages and tasks correspond to modules since they have specification and body components. However, uses relationships are represented using "with" clauses in library units (which correspond to compilation units) which may correspond to arbitrary combinations of specifications and bodies of packages and tasks,
as well as subprograms. In environments that incorporate module interconnection language facilities, "provides" and "requires" relationships between modules are represented using language constructs. In each case, different sets of typed objects would have to be defined to represent information about these relationships.

In the simplified example above, it is assumed that the uses relationships between modules are assigned by the designers of the system using the generic object editor tool. Preprocessing of source code (e.g., examining `#include` statements in C code, or "with" clauses in Ada library units) can be performed to check conformance with the use clauses.

The example reflects a very simple minded policy of flagging any module that uses an interface, irrespective of the kind of change made (this is the policy specified for Ada [54]). With a more detailed specification of module interfaces (e.g., as used by the AdaPIC toolset [144]), a more complex analysis of the changes made can be used to pinpoint more effectively the actual modules affected by the change. The trigger itself may contain the analysis code, or it may invoke an external analysis tool.

The example also provides one rationale for representing a module entity with three object types. With a smaller granularity of objects, simultaneous checkout of each of the three objects is possible. In particular, to create instances of the uses relationship, only module objects need to be checked out for read, allowing modifications to `module_specification` and `module_body` to continue in parallel.
type module is object
  attributes
  ..... 
  uses : set of module;
  used_by : set of module,
    derivation = inverse module.uses;
  specs_modified : string (3) := "no",
    derivation = trigger;
  is_dirty : string (3) := "no",
    derivation = trigger;
  triggers
    trigger (value : string (3)) on specs_modified.modify.pre is
    begin
      if value = "yes" then
        foreach mod in self.used_by loop
          mod.is_dirty := "yes";
        end loop;
      end if;
    end;
  end object;

type module_specification is object
  attributes
  parent.module : module,
    derivation = inverse module.interface;
  ..... 
  triggers
    trigger on specs.modify_checkin is
      ch_rec : change.record; -- local variable
      begin
        if not is_defined (reason_for_change) then
          -- code described in previous examples
        parent.module.specs_modified := "yes";
        end if;
      end;
    end object;

Figure 3.21: Flagging Modules Affected by Change
Additional Semantics of Trigger-Derived Attributes

Changes to trigger-derived attributes are committed to the database as soon as the trigger chain commits the short transaction. Since the change is not restricted by long transaction locks, any change to a trigger-derived attribute is reflected in both the baseline and private copies of an object. The rationale for this is the following. In the above example, suppose a change to module M1 results in module M2 being flagged, where M2 is already checked out for update by another user. If the is_dirty attribute is only changed in the private copy of M2, and the user issues an undo command, the value of is_dirty will never be reflected in the baseline copy even though M2 is still “dirty”. Additionally, a database query issued by any user should reflect the fact that module M2 has been flagged as dirty.

When a copy_object operation is invoked (see section 3.2.1), not every attribute is copied from the original to the target object. By default, all non-derived attributes are copied, while derived attributes are not copied as a result of a copy_object command. This default interpretation can be overridden by using the keywords copy and nocopy to specify whether an attribute’s value should be copied into a new object as a result of a copy_object command.

In the previous example, if a copy is made of a “dirty” module, the is_dirty attribute is assigned the initial value “no”, signalling that the copy is no longer dirty. This semantics is based on the assumption that the creator of the copy examines the interfaces of the modules that provide the resources used by a module.
before creating a copy, and therefore, does not need to be warned.

In the example of Figure 3.19, the non-derived attribute change_history is computed by a trigger. If a copy is made of a module_specification or module_body object, the copy inherits the change history of the original object because of the default interpretation of the copy option. To alter this semantics, the definition of change_history should be changed to:

```plaintext
change_history : seq of change_record, nocopy;
```

3.5 The User Interface

The user interface of the IPSE (see Section 1.3.4) is the front end for all interactions with the process program driven system. The system provides two classes of commands, a set of generic commands available in all instantiations of the system, and a set of project specific commands that vary from one system to another\(^\text{18}\).

The generic commands provide capabilities to query the object base about type information such as the different types of objects available, as well as instance information such as the objects in one's checkout domain. Commands are available to checkout, checkin and update groups of objects, undo changes made in long transactions, and create and delete objects, using the object base operations outlined in the previous sections. If a database operation results in an error, commands are

\(^{18}\)An analogy could be drawn with the language independent and language dependent commands of syntax editor based programming environments such as Gandalf.
available to view the error record object (see Section 3.4.2). Generic commands to manipulate user role objects and to query activity objects are described in chapters IV and V respectively.

In the process program, variables are used to store values and references to objects. At the level of the user interface, the facility for defining synonyms is provided. A synonym is a user defined name that is bound to a value (integer or string) or a reference to an object. Synonyms are particularly useful as operands to commands such as tool invocations, where parameters that are references to objects have to be passed (see Chapter IV).

Project specific commands provide access to external tools that are available in the context of an activity, and also include customized commands for performing different tasks that vary with the progress of an activity. This is discussed in Chapter V.

A generic object editor is provided to browse through the network of database objects and to manipulate attributes of objects. This is the subject of the next section.

3.5.1 The Object Editor

The object editor provides access to individual objects and provides both navigational and editing capabilities. Each object has a corresponding display view. Currently, only a form based view for individual objects, and network oriented
iconic views [108] for different segments of the object base are available. However, it is possible to develop more general unparsing schemes (e.g., as in Aloegen in the Gandalf system [94]) for describing flexible display views\(^{19}\).

The form view of an object (see [83]) displays an object as a form, showing the attributes that the user can view or update as fields in the form. Each attribute is displayed as a tag that describes the purpose of the attribute, followed by the value of the attribute, unless it is of type text, binary, or a reference to an object.

For text attributes, a text editor can be invoked to edit the contents of the attribute. For attributes that represent relationships, commands are available to delete existing relationships, create new ones, and traverse links that represent existing relationships and visit the target objects. The editor therefore provides rudimentary hypertext like capabilities, and the underlying data model is powerful enough to support more sophisticated user interfaces. The form based editing and browsing interface could be compared to syntax tree based interfaces, because the structure of a form and its relationships to other forms is derived from the underlying database schema.

All objects must be explicitly checked out for update before they are modified by the object editor. This ensures that modifications to objects made interactively by users only occur as long transactions. Each modification to an attribute is implemented as a short transaction that commits the change to the private copy.

\(^{19}\)Even the specification of form views requires a primitive display language. That language is not described here.
3.5.2 Triggers and the Object Editor

The object manipulation capabilities provided by the editor, in combination with the semantics of trigger procedures and the facility for defining role specific views, can be used to tailor the editor to provide the capabilities of special purpose tools.

For example, [7] describes how a special purpose tool can generate procedure and formal parameter declarations from interface descriptions, and present them to the programmer to start coding. A special purpose editor tool that prevents modification to the system generated declarations, but allows the rest of the procedure to be defined, is required for the scheme to succeed. Such a capability can be easily replicated by defining two attributes in a “subroutine” object

```
procedure_header : text;
procedure_body : text;
```

and defining a user view that prohibits programmers (i.e. those associated with the programmer role) from updating the header attribute.

System models, module interfaces, requirements and related test cases, designs, etc. can all be viewed as a network of related objects. The browsing facility provided by the object editor is useful in examining any object and its related objects, and has the same advantages as hypertext systems.

The editing capabilities based on the underlying data model simplify the task of generating related chunks of information. Consider the following two attributes of a “review” object
Individual reviewers can create a review object, insert comments about a module in the appropriate attribute, and create a link between the review and the module being reviewed in parallel with other reviewers, using the object editor and interface capabilities.

When a user updates an attribute, immediate triggers are invoked by the system. The actions of those triggers can be viewed as the incremental execution of a tool whose logic is distributed among triggers. For example, as a user modifies and commits various objects such as modules, test cases and documentation, triggers could record the modifications in an activity specific "history" object. Such triggers can be viewed as a distributed implementation of a history manager tool (e.g., as in DSEE [79]). Even in structure oriented programming environments, incremental semantic checking code can be equivalently viewed as distributed implementations of language analysis tools. For example, action routines or semantic functions that incrementally check for type compatibility provide the same functionality as a semantic analyzer, except that the former works in an incremental mode and the latter in batch mode.

In earlier versions of the TRIAD system [83], object templates (forms) were used to organize designs according to an underlying method (e.g., Jackson), and attributes and embedded triggers were used to ensure that different parts of the
design were created in accordance with the order specified by the method. That is a form of process (method) support at the level of design. It is technically possible to provide all forms of process support as envisaged in this dissertation, using specialized attributes to maintain states of activities, and triggers to enforce constraints on the progress of those activities. Chapter V presents more high level primitives that make the description of project activities easier to specify, understand and maintain.
CHAPTER IV

Representations of External Tools and User Roles

This chapter describes the representations of external tools and user roles, that are interpreted by the virtual machine to control the invocations of tools and manipulation of objects in the public database, by project members. Though neither issue is the major concern of this dissertation, they are important constituents of the architecture and of direct relevance to activity modeling. Some simple assumptions that have been made about the nature of tool and user models are presented, references are given to ongoing work on tool integration, and possible extensions to user role modeling are discussed.

The modeling of relationships between external tools, the definition of views of the object base required by a tool, and the design of the tool manager component of the virtual machine to control tool invocation and interaction, is being addressed in a separate dissertation [139]. For the purpose of presenting the process programming approach, minimal assumptions are made about the tool model.
and these are presented in the following section.

Role based access control is also an important aspect of project control. As mentioned in the survey in Section II, protection models like Minsky's send-receive transport model [84] and Sandhu's SSR model [118] need to be incorporated into any environment architecture for large scale software development because fine-grained control over access to objects and tools, and control over the transport of privileges is an important aspect of project control. Issues in integrating and implementing such sophisticated protection models are beyond the scope of this dissertation. A simplified scheme for describing role based views and access patterns is presented in Section 4.2. Possible extensions to this scheme are discussed in Section 4.2.4.

4.1 The Representation of External Tools

There are two classes of tools that will be integrated within an IPSE — tools that have already been developed to work with conventional files, and newly constructed tools that use database access functions to directly manipulate information in the object base. This distinction is illustrated in Figure 1.4 on page 30.

The facilities of a sophisticated data model can be best exploited by the latter category of tools, which can interact with each other by sharing information in the database at any desired level of granularity. Such tools define a view of the object base using a view definition facility (see [139]). Access to components of views
is automatically translated into statements that access the underlying database objects.

There are however a large class of tools which have already been developed to share (large granularity) information through files, and since these tools provide many useful functions, they too need to be integrated into an IPSE. These are the only category of tools that are considered in this chapter. The technique of creating tool envelopes [12], [42] is typically used to integrate file based tools. The technique was first used by Kiper (see [76]), who decomposed an envelope into view extractor and output distributor functions. Envelopes are described with an example in the next section. Section 4.1.2 discusses the problem of nested long transactions in the context of tool invocations. Section 4.1.3 discusses possible extensions to the representation of external tools to incorporate incremental make-like facilities as in Marvel [68] and Odin [29].

4.1.1 Defining Tools as a Special Class of Objects

External tools integrated into the IPSE are represented internally using tool objects, a special category of objects that represent executable programs whose purpose is to provide automated support for various software development activities. This is a departure from PCTE+ and CAIS+ where all executable programs (tools as well as the products of software development such as load modules and shell scripts) are represented uniformly within the object management system. While that is
sufficient for a general purpose object management system, a process-based IPSE needs to maintain explicit representations of tools since one of its goals is to control and automate tool invocations in supporting the process of software development.

A tool object is illustrated in Figure 4.1. There is one such tool object for every tool integrated by the IPSE\(^1\). It contains the location of the executable code (in some directory), and the version of the tool, information which is useful for recording tool derivation histories. The tool manager\(^2\) uses information in the tool definition to control its invocation. The compiler tool (as represented in the process program) provides three separate operations for compiling module specifications, module bodies, or both together. Compilation of more complex aggregate compilations units (e.g. as in Ada) has been ignored. The file based tool itself may expect a file with any valid combination of module specifications and bodies, and the logic for mapping internal objects to files is encoded in the definition of the envelopes.

An envelope has to have complete knowledge of the functionality and side effects of a tool to be able to effectively provide an interface between an executing process program and the tool. It has to be aware of the options accepted by a tool, the

\(^1\)Unlike data, user role and activity objects which represent instantiabale types, tool objects should be viewed as instantiations of a generic tool object type, where the instantiation is performed by the IPSE designer and is specially interpreted by the tool manager.

\(^2\)The tool manager is the component of the virtual machine that controls tool invocation and cooperation. In the simplified model presented here, a tool manager is a trivial component of the virtual machine. The design of the tool manager in the context of composite and cooperating toolsets is presented in [139].
tool compiler is
  class file-based;
  version "1.2.1";
  location "/usr/bin/...";
  operations
    envelope compile_module (m : module, opt : options,
      c : out compilation, e : out errors) is
    begin
      -- Contains code to extract the module specifications
      -- and body from the object base, output it to a file,
      -- invoke the (file-based) compiler, and redistribute
      -- its output to the object base.
    end compile_module;
    envelope compile_body (mb : module.body, opt : options,
      c : compilation, e : out errors) is
    begin
      ..... end compile_body;
    envelope compile_specs (ms : module.specification,
      opt : options, c : out compilation, e : out errors) is
    begin
      ..... end compilespecs;
  end compiler;

Figure 4.1: Definition of a Compiler Tool
envelope compile_module (m : module, opt : options, 
  c : out compilation, e : out errors) is
source_file_name, options_string : string (20);
source_file, symtab_file : OUT_FILE; -- Predefined
st, stvar : symbol_table;
begin
  file_name := m.name & ".xxx";
  -- appropriate extension appended
  create (source_file, source_file_name); -- Ada file create
  checkout_r (m, m.specification, m.implementation);
  convert_to_file (m.specification, source_file);
  convert_to_file (m.implementation, source_file);
  -- output symbol table file
  create (symtab_file, "... name expected by compiler ...'');
  st := (stvar in symbol_table); -- set of one element
  loop -- Checkout symbol table for read with busy wait
    begin
      checkout_r (st); exit;
      exception when checkout_error => null;
    end
  end loop;
  convert_text_to_file (st.contents, symtab_file);
  -- Generate options_string from the opt object

  -- Invoke external compiler tool with inline parameters
  ret_code := system_call (self.location & " " &
                     source_file_name & " " & options_string);
  if ret_code /= 0 then raise TOOL_ERROR;
  -- Otherwise, check if there were compilation errors.
  -- Also, create a new compilation object and assign values to
  -- the attributes. Finally, update the new symbol table in
  -- the object base.
  ......
end compile_module;

Figure 4.2: Definition of the Envelope for Compiling a Module
files accessed, and the format of all the files produced in response to the various options. The envelope for compiling a module is partially defined in Figure 4.2. A lot of detail has been omitted to simplify the presentation.

The compile_module envelope creates a set of files that are expected by a conventional compiler. In the source file, it outputs the text of the module specification and implementation, after checking out the database objects in read mode. It is assumed that for separate compilations, a global symbol table that contains information about all modules in the system, is used by the compiler to check for interface errors. The symbol table itself is stored as a binary attribute of an object of type symbol_table. This symbol table is also checked out for read before outputting the actual contents onto a file that the (file-based) compiler can use. An object of type options is passed to the envelope with various compile time options as values of attributes. These attributes are extracted and converted into a string. Finally, the command string for invoking the compiler is created and the system_call function is invoked to start execution of the tool.

After the execution of the tool is over, the reverse process is carried out. Output files from the compilation process are inserted into the object base as attributes of objects. For example, an instance of the compilation object (see Figure 3.5) is created, and appropriate relationships between compilation, module, and object_code objects are represented as attribute values. The relevant code is straight

---

3The symbol table object is the only instance of its type. This is characteristic of engineering databases where there are often many types and few instances per type.
forward and is omitted. The protocol used to control concurrent access and update of various objects is more involved and is discussed next.

Before a module is compiled, it is checked out in read mode. If the checkout operation fails because of a conflicting lock, a LOCK\_CONFLICT exception is raised and control returns to the invoker of the tool (envelope). Since a module may be checked out for an indefinitely long time, the option of waiting for it to be released is not considered. However, checkout of the symbol table is treated differently. Unlike module objects, symbol table objects are only checked out by various invocations of the compiler\(^4\) for short periods of time. Before a compilation is begun, the global symbol table is checked out in read mode, and its contents .attribute (containing the information used by the compiler) is written into a local file. After compilation is over, the compiler updates this file. The envelope reflects this change in the object base by replacing the contents attribute. Prior to doing that however, it has to check the object out in update mode. The code for doing that also uses busy waiting (see Figure 4.3). Note that if the read lock is not released prior to the checkout\_w operations, there can be a deadlock if two or more compile\_module envelopes execute the above code.

The example illustrates the limitations of the long transaction mechanism for handling short lived transactions. Since automatic waiting is inhibited, busy waiting has to be used. That is computationally expensive and can also lead to starva-

\(^4\)As discussed in Section 4.2, with sophisticated access control schemes, check out of the symbol table by any user or tool except the compiler can be prevented.
... code of compile\_module envelope, continued ...

checkin (st); -- Release read lock

loop
  begin
    checkout\_w (st); exit;
    exception when checkout\_error => null;
  end;
end loop;
st\_contents := convert\_to\_binary (st\_contents, symtab\_file);
......
checkin (all); -- release all locks

Figure 4.3: Updating the Symbol Table Object

tion. A better alternative is to use a mix of long and short transaction for different objects.

In the envelope redefined in Figure 4.4, the modules being compiled are checked out to prevent simultaneous checkout by a user (in case the modules have not already been checked out by the invoker of the compiler), but the symbol table object is accessed using the conventional transaction mechanism with wait semantics.

Though an envelope can be used to integrate an existing tool, the degree of integration cannot be as great as is possible if two tools were directly designed to cooperate with each other using the shared object base. Integrating such existing tools can also result in work being duplicated by the IPSE and the tool. For example, the Ada compiler already has extensive library management facilities built into it because of the programming-in-the-large features of Ada. Facilities
envelope compile_module (m : module, opt : compile_options, 
  c : out compilation, e : out errors) is
  -- Declarations same as before ...
begin
  -- Modules checked out and written to source file as before ...
  -- output symbol table file
  create (symtab_file, "... name expected by compiler ...'");
  st := (stvar in symbol_table);  -- set of one element
  convert_text_to_file (st.contents, symtab_file);
  -- Contents output to file using conventional read lock,
  -- which is then released because each operation is atomic
  ...... ret_code := system_call (self.location &
  source_file_name & options_string);
  if ret_code /= 0 then raise TOOLERROR;
  ...... st.contents := convert_to_binary (st.contents, symtab_file);
  -- If the contents are being simultaneously modified by another
  -- invocation of the compiler, the above operation waits
  ...... end compile_module;

Figure 4.4: Alternative Version of the Compile_module Envelope
for preventing conflicting access to the library are already built into the compiler and mechanisms for concurrency control as presented above are partly redundant.

The envelope-based approach can be used to extend the capabilities of an existing tool. In the above example, since the symbol table is kept under the control of the object management system (an alternative would have been to leave it in some standard directory where the compiler can maintain it), the access control schemes that can be used to protect the symbol table can be much more strict than those supported by the underlying file system.

In Kiper's dissertation [76], it has also been demonstrated how existing tools like the C compiler and Unix debugger can be made more incremental, when integrated with an IPSE that represents information at a finer level of granularity. It has also been realized that it is not reasonable to expect all tool builders to conform to the standards set by one object management system (e.g., CAIS+, PCTE+ in the context of tools written in Ada) and the need to integrate "common off-the-shelf" tools (the COTS paradigm) makes the envelope approach important.

4.1.2 Nested Long Transactions

The user interface of the IPSE allows the user to checkout objects for the purpose of browsing or editing. When a user invokes a tool, it (or in this case, its envelope) too can perform checkout operations to lock objects that may not have been explicitly checked out by the user (e.g., module and symbol.table objects in the first version
of the envelope). This gives rise to a nested transaction model and issues related to that are presented below.

If an object is already checked out for update by the user, and a tool also checks it out prior to manipulation, then at checkin time, the object is not committed to the public database. The checkout by the tool creates a new private database owned by the tool. If the tool creates new objects (using the create_object_local operation), those objects become part of the tool's private database too. A checkin operation invoked by the tool transfers objects from the private database of the tool back to that of the user. If the creation of a private database (initiated implicitly by the first checkout_w operation) is seen as the beginning of a long transaction, then the analogy with conventional nested transactions [87] can be carried over.

If the user has no private database (i.e., he or she has not checked out any objects yet), and invokes a tool with baseline objects as operands, then the tool becomes the parent transaction and a checkin operation by the tool results in all the objects being transferred back into the baseline. This is illustrated with the examples in Figure 4.5, assuming that the compile_module envelope only uses long transaction operations to control concurrent access to objects.

In case 1, the user has not checked out the module object that is to be compiled. When the compiler is invoked, the relevant objects are checked out, and after compilation is over and the checkin (all) command is issued, all the objects are checked back in. In the second case, the user has already checked out the
Figure 4.5: Tool Invocations and Nested Long Transactions
module that is to be compiled. When the compile_module envelope executes the checkin (all) command, all objects created by the tool become part of the user's private database.

With the first definition of the compile_module envelope, this protocol causes a problem. In case 2, if the symbol_table object is not checked in until the user (the parent transaction) issues a checkin command, it may remain locked for an excessively long time and no other compilations can proceed. This problem is solved by the second definition of the compile_module envelope, where short transactions are used to manipulate "global" objects like the symbol table, and long transactions are used to manipulate and create "private" objects like modules, object code, etc. In this case, after a module has been compiled, the module and object code objects still remain in the private database, but the symbol table never becomes part of the private database. Similarly, if a tool envelope uses the create_object operation instead of create_object_local, then the newly created objects are immediately created in the public database, bypassing the private database mechanism.

4.1.3 Representing Relationships Between Tools

A tool object contains information about how a tool is to be invoked. The Marvel [68] and Odin [29] systems use the representation of relationships between tools to control tool invocations in more "intelligent" ways. While that issue is not addressed in this dissertation, the framework that has been proposed above, i
conjunction with the tool manager being designed in [139], can be easily extended to incorporate the features of the above two systems.

In Odin, tools are related to each other using a structure called the tool dependency graph. A tool dependency graph describes for each tool the input object types that it uses as operands, and the output object types that it produces. For example, the compiler tool takes an object of type module as input, and produces objects of type object_code, compilation and symbol_table (other possibilities include cross references, object code with monitoring information, etc.). A linker tool takes object_code objects as input and produces executable objects. A relationship between the linker and the compiler tools exists because both share the common type object_code. The Odin system allows the user to request the derivation of an object of a certain type. The system applies the appropriate tool (using the tool dependency graph) to produce the desired object. If some objects required as input to a tool do not exist, the system first applies tools that produce those objects. This is a form of backward (or goal-directed) reasoning that is performed by the system.

The Marvel system represents relationships between tools using preconditions and postconditions, expressed as well formed formulae in terms of database objects and types. If a tool is invoked (e.g., the linker, to produce an executable object), and the precondition for execution of the tool is not satisfied (say, some object_code does not exist), the system attempts to apply appropriate tools whose postconditions
satisfy the precondition (e.g., the compiler's postcondition states that it produces object code, and therefore, it is invoked to satisfy the precondition of the linker).

Either formalism can be adopted with simple extensions to the tool object definition. The backward reasoning mechanism will have to be implemented as part of the tool manager which controls the invocation of tools. These issues are being addressed in [139].

Both Osterweil [99] and Kaiser [68] describe the facilities provided by such a backward reasoning tool invocation system as a form of process support, because the user is provided with assistance in applying tools. The granularity of the process being supported is at the level of a single user applying tools. This should be contrasted with the much larger granularity of process support that is addressed by the activity model described in Chapter V.

4.2 User Roles and Access Control

This section presents rudimentary features for modeling the different roles that are played by the members of a project team, and their use in controlling access to objects in the public database. Each role is a typed object, which has a set of associated privileges that determine policies regarding access to tools, the allowable views of data objects of different types, and the allowable operations on data and role classes. These are role (type) specific policies represented as an access control scheme. A separate class of objects are provided to aggregate individual roles into
group hierarchies. An access control list based mechanism is proposed to control access to individual objects. The relationship between the access control scheme and rights in an access control list is described. The limitations of the formalism, and possible extensions to support more sophisticated access control policies are suggested.

4.2.1 Role Objects

In typical operating systems (e.g., Unix), a user is identified by a user id, but no further distinction is made in terms of the different roles that an user typically plays, manipulating different classes of objects (files) and using different tools (executables) in playing a role. For example, a programmer typically reads the specification and design of a module, creates and modifies a body of code, and uses compiler, linker and debugger tools to perform a programming task. In contrast, a tester might read the specifications, test cases, and the test plan, use the compiler and linker as well as a suite of testing tools (e.g., instrumentors, test data comparators), and create test incident reports.

While it is possible that the same person plays the role of a programmer during the coding phase, and the tester during the integration testing phase, it is clear that the objects and tools that one requires access to depend on the role that is being played. One uniform access control mechanism based simply on the notion of a user and an account cannot be used to specify access control policies at a finer
level of granularity.

A role object consists of a set of attributes and a set generic privileges that an instance of the role acquires. These attributes can be modified using the object editor. The domains of these attributes are the same as that of data objects and additionally, can also be other role types. However, additional qualifications of attributes, declaring them to be subobjects, derived, nonull, etc. are not allowed.

A role is a typed object, which can be instantiated in a manner similar to data objects. When a role object is instantiated, it is bound to a particular user id (a string) that represents the person who can play the role. That person then inherits all the privileges specified in the role. This notion of a subject type (distinguished from object types) is used in many protection schemes (e.g., [85], [118]).

Figure 4.6 defines the role object for a programmer. The proposed formalism allows the IPSE designer to specify for a role the tools that can be invoked, the views of data and user role objects that can be manipulated using the object editor, and the data and role objects that can be created using the operations provided by the user interface. The semantics of these definitions are described in the following sections.

**Controlling Tool Invocation**

The *can invoke* declaration takes as an operand a set of tool operation (envelope) names. The keyword *all* is a shorthand for all the operations of a tool. Any
Figure 4.6: The Definition of a Programmer Role

instance of a role (a user) can invoke the specified tools while performing the activity corresponding to the role (see Chapter V).

In conventional operating systems, a group of users can be given execute access to an executable object that represent a tool. In contrast, the above declaration represents a generic policy regarding tool access by programmers, that can be specified at type definition time and does not require explicit manipulation of the access control list of executable objects at runtime.

As mentioned in Section 4.1.1, tools are treated as a special class of executable objects since controlling tool invocation is a principal function of an IPSE. It is believed that the ability to specify policies about tool usage that are applicable to
all instances of a role is sufficient, and the manipulation of access control lists of individual tools to provide a finer granularity of control (e.g., to give access to a tool to only a subset of all instances of a role type) is unnecessary.

Object and Role Creation

The creation of an object of a certain type is also controlled using role specific policies. The instantiation of an object has the side effect of inserting a reference to that object in the class corresponding to the object’s type. Therefore, the right to create an object of a given type, is equivalent to the right to insert elements in the corresponding class.

As with tool invocation, it is believed that the policy of allowing the manipulation of a class can be specified in a role specific manner. It is possible to associate an access control list with an object class (explicitly represented as a class object), using which, rights to invoke the insert operation of a class can be given (at runtime) to only a subset of all instances of a role. However, that degree of flexibility is considered to be unnecessary and the ability to assign role specific rights to create objects is specially provided to the IPSE designer.

The ability to create role objects themselves is also treated in a manner similar to object creation. Only managers will typically have the right to create roles of subordinates, who constitute the team that the manager supervises. Figure 4.7 defines the programming_manager role. The manager has access to a different
type programming_manager is role
  can invoke {CPM.PERT.all, ...};
  -- tools that the programming manager can invoke
  can create_object {task_assignment, pert.network, ...};
  -- Allowable manipulation of object classes
  can create role {programmer, programming.team, ...};
  view of module is
    (can read all,
     can update {is.frozen});
    ...

end role;

Figure 4.7: The Definition of a Programming Manager Role

set of tools, has a different view of the module object, and is allowed to create a
different set of objects, including programmer role objects.

Views of Objects

Each role has a definition of the views of different object types that the role player
is allowed to manipulate. These views are defined in terms of projections of the
attributes of an object type. Even if a role player has modify access to an object
of a certain type, he or she is only allowed to view and update the attributes that
are specified in the view definition for that type. These views are specified with
respect to the object editor, which is the generic tool provided by the system to
manipulate objects.

In Figure 4.6, the programmer is given read and update access to various at-
tributes of a module. In Figure 4.7, the programming manager is given read access to all the attributes of the module object (specified by the keyword `all`), and is allowed to update the `is_frozen` attribute by which the module is frozen, preventing any further modification (see example in Figure 3.14 of Chapter III). Since the `is_frozen` attribute is not part of the programmer's view of the module, he or she cannot bypass the policy of not being able to modify frozen modules that is enforced by the trigger.

External tools integrated into the system may specify a different view of an objects, and therefore, access to those attributes can only take place through the tool. In the tool definition scheme presented here, no explicit view is defined for a tool; it is implicit in the procedural code of the envelope. For example, while the user may be given the access rights to modify a symbol table⁵, only the compiler tool's envelope has the authority to modify the contents attribute (the actual symbol table file), and the user cannot update that attribute using the object editor. In Section 4.2.4, a more general scheme which allows all access to an object to be controlled by a tool is discussed.

### 4.2.2 Representing Groups and Their Hierarchical Organization

*Group* objects that aggregate individual role objects of various types can be de-

---

⁵In the current system, that would be necessary because tools inherit access rights from the user roles that invoke them.
type programming_team is group
    attributes
      programmers : set_of programmer, subgroup,
                     derivation = programmer;
end group;

Figure 4.8: The Definition of a Programming Team

fined. Group objects can also aggregate other group objects, to model complex organizational hierarchies.

Figure 4.8 defines the programming_team group. A group object contains a set of attributes, some of which are of a special category that represent members of the group. Attributes that are defined to be subgroups can contain references to other role or group objects, and are constrained to form DAG structures, in a manner similar to subobjects of the data model. In Figure 4.8 the derived attribute programmers represents all instances of the programmer role. The keyword subgroup defines group membership. If a subgroup attribute of a group object (say G) references another role or group object, then G is defined to be a direct ancestor of the latter object. If G is a direct ancestor of the group G1, which in turn is a direct ancestor of the role object R1, then G is an indirect ancestor of R1.

Typically, the programming_team group will have one instance, that is created by the programming_manager before instantiating any programmers. The group object will then represent all programmers, and can be used in access control lists
to give all members of a group access to individual objects.

While derived attributes as in the above example can be used to represent teams, non-derived attributes can also be used to represent aggregations that are controlled explicitly. For example, while all programmers may have the privileges described in the programmer role, certain programmers may be more experienced than others, and may be given access rights to more sensitive objects. Two separate group objects which represent programmers with different levels of experience can then be defined, membership in the group can be explicitly controlled by the programming manager, and these separate group objects can appear in different access control lists to control access to objects based on levels of experience.

A group object called public is predefined by the system. This object represents the root of the group structure and is the (direct or indirect) ancestor of all role and group instances, even if they have no other ancestors. Membership in the public group is implicit.

Operations on Group and Role Objects

Group and role objects are like data objects, because they are typed objects with attributes that have instances. The user interface provides create_role and delete_role operations, and the object editor can be used to view and manipulate attributes according to the constraints defined in role specific views.

Like data objects, different versions of create_role and delete_role operations
are provided in the context of long transactions. Even triggers may be associated with attributes of roles and groups. However, in the examples that have been constructed, no particular need has been found for such triggers, or even for very general attributes (except for the kind of derived attribute that defines group membership in the programming team group). However, these features are still provided because no extra mechanism beyond what already exists for data objects is required in the virtual machine.

When a role object is instantiated, it has to be associated with a particular user id that corresponds to a valid account in the computer system in which the IPSE executes. Every role object type has a system defined attribute $user which must be assigned a string value at creation time. The declaration of this predefined attribute is

$\text{user} : \text{str in g  (10), nonull;}

Every invocation of the create_role operation must contain an assignment to this attribute since it has a nonull constraint. The string value must be a valid account. This constraint is checked by the system.

### 4.2.3 Controlling Access With Access Control Lists

The proposed protection scheme uses the mechanism of access control lists (ACLs) to control access to individual objects. For simplicity, it is assumed that ACLs are only associated with individual data and role objects. Tools are treated as a
special class of executable objects and the right to execute a tool is given to all
instances of a role, making it unnecessary to associate ACLs with tool objects.
The same is true of rights to create data objects and roles, making it unnecessary
to associate ACLs with data and role classes.

The basic operations that can be performed on data and role instances are
delete, update and read\(^6\). The control_acl operation is provided to modify the
access control list itself. For each object, there is one ACL corresponding to each
operation, whose declaration is of the type

\[
\text{set of uniontype \{}\text{programmer, programming manager, ...}\}\]

An ACL for a given object and operation is a set of references to role and group
instances for which that operation is permitted. The type definition implicitly
includes all the role and group types since there is no restriction on the granting
of access to particular roles\(^7\). A role player may perform a given operation on an
object, if a reference to the role instance, or a reference to a group that is a direct
or an indirect ancestor of the role instance, appears in the corresponding ACL. If
the predefined group, public, is present in an ACL, all role players can invoke the
operation on the object corresponding to the ACL.

Though read and update operation are defined for individual attributes, rights
are specified for an object as a whole. To read or update an attribute, the role

\(^6\)Object execution has been ignored. This is discussed in Section 4.2.4.

\(^7\)This could be an additional policy specifiable in a role or group declaration, as discussed in
Section 4.2.4.
player must have the appropriate right for the object. The role specific view determines the validity of the attribute access. A modify right on an object implies the right to read, but object deletion is treated separately.

The role type superuser is predefined by the system. At environment initialization time, one instance of this role is created and assigned to the IPSE designer (the person who compiles the process program). The superuser role type has the privilege to manipulate all tools, object classes and instances, and may perform any operation on an object, bypassing the ACL mechanism. Only one instance of the superuser type may be instantiated (a built-in constraint) and this instance is bound to the predefined variable rootuser.

Variations of this scheme have been proposed in PCTE+ and CAIS+. Neither of these schemes use the notion of a user role with associated privileges; a generic user object is provided to represent individual users. However, both proposals use the notion of group objects to aggregate user objects into different groups. Thus, group structures in those systems are much more likely to be DARGS than in this scheme, since the leaf objects represent instances of a generic user object type which will often belong to more than one ancestor group. This has implications on the efficiency of checking the validity of an operation.
Operations on Access Control Lists

Two operations are provided to manipulate access control lists, add_access, and remove_access. Each operation takes three parameters, a reference to the data or role object whose access control list is to be manipulated, a reference to a group or role instance for which the access right is being manipulated, and a set of operations for which access is being given. The operation is specified as one of the keywords - delete, read, update or control_acl. The operations add_access* and remove_access* are similar to the above, except that they give access to on an object and all its subobjects.

To be able to manipulate access control lists of an object, the invoker of the operation (or an indirect ancestor) must have the control_acl right for the object. A role player with the control_acl right has complete control over the granting or revocation of rights for operations on that object.

Object Ownership

The creator of a role or data object automatically becomes the owner of that object. The owner of an object is typically the role player who initiated the creation of the object, though this can be overridden by triggers and activities (see page 192). Every object and role type has a predefined attribute called $owner which is defined as the union of the role types which have permission to create instances of the class. This is determined (by the compiler) from the can create
declarations in the role objects. For example, if only programmers and programming.managers can create module.body objects (the latter will not be allowed to write or modify code however, because of the the restriction specified in the role specific view of module.body) then the $owner attribute of the module.body object has the implicit type declaration

$$owner : \text{uniontype} \{\text{programmer}, \text{programming.manager},$$
$$\text{superuser}\};$$

The role type superuser can also own objects.

$owner$ is single valued because every object has a single owner. Its type is always a role object. The role player who is the owner of an object inherits all access rights (including control.access) to the object. The distribution of rights to other role players is controlled by the owner (or by a process on behalf of the owner) unless the control.acl right is granted to another role player. Though the owner of an object may remove rights for himself or herself (to prevent accidental modification of an object, say), he or she may always restore it, even if the control.acl right has been revoked. The control.acl right is implicitly available to the owner even if it is not explicitly present in the ACL.

The change.owner operation is provided to transfer ownership of an object to another role player. Only the owner of an object (or the superuser) may invoke this operation. The original owner does not lose any of the rights that he or she
possessed (though they may subsequently be revoked), and the new owner inherits all rights to the object.

Privileges of Executing Programs

Two classes of executable code are distinguished. The first category of programs consists of triggers and procedural components of activities that enforce various policies. The second category consists of the envelopes of external tools. Each executes with distinct privileges.

All triggers and procedural components of activity definitions execute with superuser rights. This facilitates the enforcement of policies regarding the propagation of information by triggers to maintain data integrity, and the control of invocation of tools by the procedural components of activities.

In the example of Figure 3.21 in page 155, a trigger updates the “is_dirty” attribute of modules that use the resources provided by module. If the trigger executed with the rights of the user role that initiated it, the user would have to have modify rights for each of the affected modules for the operation to succeed. Chapter V presents examples where a role player associated with an activity does not have direct access to a tool; all invocations of the tool occur through the activity. Such policies would not be representable without giving triggers and activities superuser privileges. Note that while superuser privileges allow a user or a program to violate access control restrictions, integrity constraints represented
by database triggers cannot be violated.

Though triggers and activities operate with superuser privileges, the initiating role player is still made the owner of the created object by default. Therefore, when a create_object operation is invoked, the system automatically assigns a reference to the current role instance as the value of the $owner attribute. Triggers and activities can explicitly override this default by assigning a reference to any role (including rootuser) to $owner. $owner is an internal attribute and it cannot appear in the view of a role type; thus it cannot be assigned a value from the create operation available at the user interface.

Invoked tools execute with the same privileges as the user. However, since tools and users typically have distinct views, access to certain attributes can only be made through tool invocations. This is often not good enough. For example, for the compiler to be able to create and delete compilation and object_code objects, the right to delete these objects have to be assigned to every programmer who may invoke the compiler (this would typically be done by assigning the right to the programming_team group). However this allows members of the programming team to arbitrarily create and delete objects of type object_code and compilation. By preventing access to the attributes that represent relationships between these objects by the definition of appropriate views, some degree of control is still possible. However, this solution is less than satisfactory. These issues are discussed in Section 4.2.4.
Public and Private Databases Revisited

The access control mechanism that has been described controls operations on objects in the public database. The long transaction mechanism offers a separate level of protection related to concurrent access. Long transactions do interact with the protection mechanism in an indirect manner. The access control list of an object cannot be modified if the object is checked out by any user other than the role player who initiates the modification. The restriction is motivated by the need to preserve the integrity of a long transaction. If an object has been checked out for update and significantly modified, removing the update rights of the person making the modification in the middle of the long transaction would result in a significant loss of work.

Many proposals for CAD-CAM databases have extended the simple public-private database architecture of Haskin et al. [57] to include many more levels of databases. Objects in the public database are visible to everybody, private databases are only accessible to individual users, and semi-public databases are accessible to well defined groups such as programming teams, testing teams, etc. The semi-public databases are hierarchically organized to reflect organization hierarchies (e.g., see [37] and Figure 4.9). An object that is to be modified has to be checked into the private database of an individual, but all objects in semi-public databases which are ancestors of the private database can be read by an individual. Objects can be directly moved from one database to another only if they have a
parent child relationship.

While the proposal in this dissertation adopts the public-private database approach for concurrency control, controlling the access of objects in the public database by an access control list mechanism allows one to deal with more general protection requirements than is possible in the multiple semi-public database scheme. Note that the role and group object hierarchies of the model proposed here can be used to implement the semi-public database scheme; it is indeed the implementation mechanism suggested by Dittrich in [37], though general access control lists are not allowed.
The multiple semi-public scheme assumes that access to shared objects can always be compartmentalized along group boundaries. This need not necessarily be true in a large project environment. For example, once a module is undergoing integration testing, the programming team will typically not have any access to it. However, if a bug is found during the test, the configuration manager (the current owner of the module) will give modify access to the original developer to correct the bugs. Using the access control list, it is possible to give modify access to the module to one programmer, without allowing other members of the programming team to have any access to it. This is not possible in the less general scheme of [37].

In a hierarchically organized semi-public database scheme, there is no direct relationship between the semi-public databases for the testing group and an individual programmer, and the module would first have to be moved from the testing database to the programming group database (under the control of the appropriate group administrators), before it can be checked out by an individual programmer (see Figure 4.9). This gives read access to the module to each member of the programming group.

While the ACL based mechanism is more general, it significantly complicates the query evaluation process. A query of the form

\[(m \text{ in module where lines.of.code } \geq 200)\]

that appears in a tool returns the set of modules whose lines.of.code has a value
greater than 199, and which the role player (the invoker of the tool) has read access to. The above query is appropriately modified by the compiler to check for the presence of the given role instance, or any of its ancestor groups in the ACL for module objects. The implementation mechanism suggested by Dittrich is also based on query modification, but the query processing overhead is less since each object belongs to precisely one private or semi-public database, and so the ACL for an object is always a singleton instead of a set.

Since triggers and activities work with superuser privileges, the previous query would not be restricted by access rights if it appeared in the body of a trigger or an activity. In fact, the trigger of Figure 3.16 in page 140 that checks for a delayed key constraint violation, may not perform correctly if it did not execute with superuser privileges.

4.2.4 Shortcomings of the Model

There are a number of major shortcomings of the above access control model that make it inappropriate for enforcing strict access control policies of large projects. These are outlined below.

One problem with the model is that any role player with control_acl rights has complete control over the granting of access rights to other roles. There is no control over the transport of privileges. This is an issue that has been addressed in the send-receive access control model by Minsky [85], and in the schematic send-
receive (SSR) model of Sandhu [118]. In the SSR model, a filter function maps pairs of role types to an access right on an object type. A role player can grant access to a certain operation of an object of a given type to another role player, only if the corresponding types appear in the definition of the filter function. Thus, seemingly meaningless operations such as the granting of write access to a module by a programmer to a manual_writer, can be prevented by type specific policies that constrain the source and destination role types of the transport of a privilege.

The proposed model does not control the granting of rights on a per operation basis. Therefore, anyone with control_acl right can grant rights to all operations. Finer granularity of protection is necessary whereby, rights to each operation as well as the right to grant others access to those operations need to be individually controlled. Once that is available, there is no reason to grant all rights to the creator of an object. The SSR model not only specifies the types of objects that a role may instantiate, but also defines the rights that are automatically assigned to the creator.

The send-recieve model includes a demand operation, using which, instances of a role can demand rights to all objects of a specified type. This is similar to access control schemes in database management systems (e.g., SQL [64]), where access rights to read or update objects are assigned at the level of a whole class. This makes it unnecessary to manipulate ACLs of specific instances of objects in that class. In the proposed model, the same functionality can be provided by
designing a create trigger, that automatically adds a group object that represents all instances of a role class (e.g., programing_team) to the access control list of an object. However, what is effectively an access control policy gets buried in the definition of a group object and a trigger definition.

Both PCTE+ and CAIS+ allow user objects, as well as executable objects to appear in access control lists. An executing process is allowed to perform an operation on an object if the ACL contains a reference to the user object that initiated the process or one of its ancestors (group objects) 8, or it contains the static executable object or one of its ancestors (representing tool groups). The latter feature allows certain objects to be accessible only through specific tools or tool groups. One could also include type specific policies in tool definitions, to specify object types that may only be created through tools. Even triggers and activities can be given specific rights that need not be inherited from a user role, as an alternative to giving them unlimited superuser capabilities. These issues have not been explored.

One major shortcoming of the data and protection model is the fact that it completely ignores the issue of execution. It is assumed that as far as the object management system is concerned, all objects are static, and the execution of load modules occurs under the control of the underlying operating system. The CAIS+ and PCTE+ proposals are much more complete in this respect since they allow

---

8This is similar to the semantics of access control in the proposed model.
the representations of static executable objects, as well as executing processes, to coexist in the system. In this system, load modules are executed using a specially defined tool envelope, that expects as its operand an object of a predefined type. This is illustrated with examples in Chapter V.
This chapter presents language features for modeling project related activities, the relationships between them, and the policies for controlling the sequencing, concurrency and coordination among project activities, that constitutes a particular software life cycle model. Language constructs for message based parallel programming have been adapted for modeling project activities. The syntax of activity definition, the semantics of activity execution by which external activities are controlled, and the relationship between the data, user, tool and activity models, is the subject of this chapter.

Section 5.1 describes the overall structure of the activity model, and how its execution enforces policies for controlling and coordinating project activities. Section 5.2 describes the syntax for defining activity types and the relationships between them. Section 5.3 describes the language constructs that are used to define the executable component of an activity. The use of these language features is illustrated in subsequent sections with a number of examples.
5.1 The Function of an Executable Activity Model

Section 1.1 described the importance of a detailed software life cycle model that defines each activity in a project in terms of the objects that may be manipulated, the tools that can be invoked, the allowable sequencing of the steps that constitute the activity, and the definition of procedures that check the validity of a step and prepare the context for subsequent steps. The model specifies the relationships and flow of data between individual activities, the decomposition of an activity into multiple subactivities that are assigned to project teams or team members, and the constraints on the allowable overlap and sequencing between activities.

The activity modeling primitives defined in this chapter are used to represent such a life cycle model. Every separately identifiable activity in a project is internally represented in the process program as a typed executable object, analogous to the notion of a process in parallel programming languages. An activity contains attributes that are used to represent its internal state as well as its relationships with other objects. Hierarchical relationships between activities define how an activity is statically decomposed into subactivities. Executable code associated with the instance of an activity contains embedded policies regarding how the activity should be performed by a project member, and the important event that should be signalled to other activities.

Every activity type is associated with a user role type, defining the constraint
that the activity can only be performed by a certain kind of role. When an activity is instantiated, it has to be associated with a role instance. Before using the facilities of the IPSE, a project member must first select an activity that has been assigned to a role instance that is associated with his or her user id. The role player works in the context of an activity, exercising the privileges defined in the role (see Section 4.2) regarding object creation, tool invocation, and the allowable views of data and role objects. The role player can also manipulate all the objects accessible to the role instance. This is the only mechanism for playing a certain role. If the activity is suspended or terminated (see Section 5.3.1), the privileges of the associated role become unavailable and the project member has to switch to another activity, if one has been assigned.

An activity typically provides a number of entry points at which it is ready to receive messages, either from the assigned role player, or from other activities. The role player sends messages to an activity to signal the initiation or termination of a step. Activities send messages to each other to coordinate their execution. In response to a message, an activity may manipulate database objects, invoke tools, create, initiate, or delete subactivities, present new entry points representing new steps, etc. All policies about performing the activity that are defined in the life cycle model are embedded in this executable code. The language features for representing the static structure and relationships between activities, and for programming the executable component, are described in Sections 5.2 and 5.3
respectively.

5.2 Defining the Structure of, and Relationships Between, Prototypical Activities

A prototypical activity is defined as a typed object with attributes. The attributes represent the state of an executing activity as well as its relationships with other activities. All attributes need to be persistent because activities often execute for long durations that can span multiple login sessions. The domain of an attribute can be the same as those permitted for data objects (see Chapter III), and additionally, can also contain references to other role and activity objects. Additional qualifications of an attribute, defining it to be a subobject, derived, having a nonull constraint, or representing a subgroup, etc., are only meaningful in the context of data or role objects and cannot be specified.

A subset of the attributes may be defined to be formal parameters of the activity. These must be initialized when an activity is instantiated\(^1\). No other attributes may be explicitly initialized at creation time. This is further described in Section 5.3.1.

All attributes are manipulated by the executable component of the activity; direct manipulation with the object editor, as in the case of data and user role objects, is only allowed in a limited form. A subset of the attributes may be defined to be visible to the role player associated with the activity. This is declared

\(^1\)Such attributes essentially have nonull constraints.
with the keyword *public*. Public attributes of an activity instance can be viewed and any links can be traversed, using the object editor. However, no modification operations may be performed.

### 5.2.1 Relationships Between Activities

The primary relationship between activities is *hierarchical*. An attribute whose domain is an activity type represents a 1:1 binary relationship type between the two activities. The keyword *subactivity* is used to define the relationship to be hierarchical. Unlike data and role objects, subactivity relationships must be *strictly* hierarchical. Attributes that reference subactivities are public by default and are always visible to the role player.

Activity instantiation is tied to the activity hierarchy. An activity can be created by (the executable component of) another activity only if the newly created activity can be a valid subactivity of the creator, as determined by the declaration of an appropriate attribute. When a new activity is created, it must be hierarchically related to the creating activity, which becomes its parent. This is ensured by enforcing all activity creation to occur through the executable component, even though it may only take place in response to a user command. Also, a parent activity cannot terminate until all its children have completed. Given this semantics of activity execution, DAG structures are not meaningful and are not permitted.

Figure 5.1 illustrates a hierarchy of activity types that describes the design phase.
of a project. Figure 5.2 shows the definition of some of those activities and their relationships. The attributes parent, reqs, and specs are the formal parameters of the system_design_phase activity type.

Non-hierarchical relationships can also be represented with attributes. These represent additional relationships between activities which might need to coordinate with each other.
type system_design_phase (parent : design_phase;
  reqs : requirements; specs : specifications) is activity
role system_design_manager;
attributes
  sd_activity : design_system, subactivity;
  pdr : prelim_design_review, subactivity;
  sd_revision : revise_system_design, subactivity;
  rd_review : redesign_review, subactivity;
....
end activity;

type design_planning is activity
role system_planning_manager;
attributes
  init_plan : initial_project_planning, subactivity;
  pd_schedule : schedule_detailed_design, subactivity;
...
end activity;

Figure 5.2: Language Definition of the Activity Hierarchy
5.3 The Executable Component of an Activity

The executable part of an activity is defined as a unit of sequential code, that executes (logically) as a separate process in parallel with other activities. This executable component is referred to as the activity *script*. After an activity has been created, its formal parameter attributes have been initialized, and it has been hierarchically linked to its parent, its execution is initiated by the parent activity. The functions that can be performed by the activity script are described below.

Data and Role Manipulation – All the operations on data and role objects described in Chapters III and IV can be invoked in the activity script. One function of the script is to perform clerical activities, and execute management procedures on behalf of the user. Since this typically involves creating, deleting, or manipulating data and role objects, the above capabilities are essential.

Tool Invocation – All tools can be invoked by the script. Automatic tool invocation on behalf of the user is also an integral part of enforcing management procedures.

Activity Manipulation – Operations are provided to create and delete activities, initiate, suspend, resume, and terminate their execution, send and receive messages from scripts of other activities, and to receive messages from the user. The script uses these operations to encode policies about the initiation and sequencing of activities, and their synchronization with other activities.
All direct interaction between a role player and an activity instance occurs through messages that the script is ready to receive from the user. This object-oriented mode of user interaction with activities should be contrasted with the navigational (hypertext-oriented) mode of data and role object manipulation using the object editor.

After an activity is created and its execution initiated, it may go through a number of states which determine how other activities and the role player may interact with it. The activity manipulation operations, and the state changes they cause, are described in Section 5.3.1.

5.3.1 Activity Manipulation Operations

There are three distinct categories of activity manipulation operations – operations that create or delete instances of activities, operations that control activity execution, and message passing operations for synchronizing execution with other activities and the user. These are described in the following sections.

Activity Creation and Deletion

Activity creation and deletion is very similar to the corresponding operations for data and role objects, but variations of these operations that execute as long transactions are not provided. Subactivity creation and deletion occur under the control of the activity script and can only be invoked as part of a conventional
transaction.

The `create_activity` function is provided for creating instances of an activity type. The first parameter to the function is the name of an activity type. Additional arguments are the names of attributes that are defined as formal parameters of the activity, and the reference to the role instance that is going to be responsible for the activity. The system defined attribute called `$role` contains the reference to the role instance that performs the activity. No other attributes may be assigned values at creation time. As in data and role objects, a class that represents all instances of an activity type is associated with the type. When an activity is instantiated, a reference to the newly created instance is automatically inserted in the class.

The `create_activity` function returns a reference to the new activity instance, which must be assigned to an attribute which represents a subactivity. This rule (which can be statically checked) enforces activities to be hierarchically related. An activity which has been instantiated, but which has not yet begun execution, is said to be in an `instantiated` state.

An activity can only be deleted by its parent activity. The `delete_activity` procedure takes as its argument a reference to an activity. The state of the given activity must either be instantiated, or terminated (see next section), i.e., its script should not be in the middle of execution or be temporarily suspended. The activity is deleted, and any subactivities of the activity are also deleted. All of the subac-
tivities must also be in a valid state. Since subactivity relationships are strictly hierarchical, the complications of subobject deletion caused by DAG structures (see Section 3.2.2) do not occur.

**Operations to Control Activity Execution**

The operations *initiate, suspend_work, resume_work, and abort* are used to control the execution of activity programs. Each operation takes a variable number of arguments, each of which is a reference to an activity. The activity must be a subactivity of the activity whose script executes the operation.

An argument to the initiate operation must be an instantiated activity that has not yet begun execution. The initiate operation begins execution of the script associated with that activity, changing its state to *executing*.

When an executing activity executes a synchronous send or an accept statement (see next section), it goes into a *waiting* state. An activity that is in an executing or waiting state is said to be *live*.

The suspend_work operation temporarily halts a live activity, changing its state to *suspended*. The resume operation has the reverse effect. If an activity is suspended, the role player cannot attach to the activity and exercise the privileges defined by the role instance. The mechanism is provided to allow the parent activity to (temporarily) freeze all work that can be performed in the context of a subactivity.
The abort operation terminates the execution of a subactivity. The state of the activity changes to terminated. Even in the case of normal completion of execution of a script, the activity goes into the terminated state. When an activity is aborted, all its subactivities also get aborted. When an activity is explicitly aborted, the long transaction owned by that activity is aborted with an implicit undo statement.

If the activity that is the argument of any of the above operations is not in the state expected (e.g., the argument to an initiate operation is already live), a CONSTRAINT_ERROR exception is raised. The function activity_state takes as its argument a reference to an activity and returns one of the constants INSTANTIATED, EXECUTING, SUSPENDED, WAITING or TERMINATED, which describes the current state of the activity. Figure 5.3 shows the different states that an activity goes through, from creation to deletion.
Controlling Script Execution During User Interaction  While a role player
*directly* interacts with an activity by sending messages to it, other activities such
as tool invocation and manipulation of objects are also performed in the context
of an executing activity. As described in Section 5.1, a project member must
first select an assigned activity before manipulating the tools and objects that the
corresponding role instance has access to. If the activity is not live (e.g., it is in
a suspended, terminated, or instantiated state), it cannot be selected as a context
of work.

A more complex situation occurs if the state of an activity is changed to sus­
pended or terminated, while a project member is working within the context of
the activity, editing an object, or invoking a tool. In the case of object editing, the
current attribute is allowed to be modified (it may be a text field, with the mod­
ification command being issued after a long interval to update the private copy)
before the activity is suspended. In the case of object browsing, the suspended
state is detected during the next browsing command. In the case of tool invoca­
tion, the activity is suspended after control returns from the tool. In the case of
an interactive tool, the activity will be successfully suspended only after the user
quits the tool. Thus, unlike operating system level processes, responses to suspend
or terminate operations cannot always be made effective immediately.

The above conditions address the situation where an activity has to change its
state while a user is working in the context of that activity. A different situation
occurs when a parent activity issues a suspend_work or abort statement for a child script, that is in the middle of execution. Since scripts can invoke tools, including interactive ones, the question of when the suspend or abort operation takes place has to be addressed. As in the above case, immediate action is not taken. To simplify the implementation, it is assumed that an executing script can be only suspended or aborted at points where the script gives up control. This includes synchronous send, accept and accept_cmd statements. Explicit tool invocations by the script are not included in this category.

Note that since suspend_work or abort statements cannot be executed immediately, they too need to be included in the category of statements whose execution should result in the script giving up control. When a script executes one of these statements, it does not continue until the states of the child scripts have been changed, which in turn cannot be effected until the child scripts reach execution points at which they can give up control. Note that due to this restriction, suspend_work and abort statements are now added to the above three statements at which point a script can be aborted or suspended.

Messages and Entry Points

Both waiting and non-waiting modes of message transmission, and an operation to receive messages from activities, are provided for activities to communicate with each other and synchronize their execution. The semantics of these operations
are outlined in this section. The next section describes a variation of the message receive statement that is used to define points of interaction with the role player assigned to the activity. Section 5.3.3 describes the non-deterministic select statement for selectively receiving messages from one of multiple sources, which could be other activities, or the user.

Messages are sent to, and received at entry points. An entry point has a name and a set of formal parameters. A message is sent to an entry point of a given activity, with a list of actual parameters which must match the formal parameters. The allowable types for the formal parameters are those allowed for attributes of activities, except text and binary types for which variables are not allowed. All formal arguments are "in" parameters. The reason for this restriction in the context of asynchronous message transmission is described below.

The send statement is used to send messages in asynchronous mode. It takes as its argument the name of an entry point of a given activity. The entry point is qualified using "dot" notation, and is followed by a list of actual parameters using positional or named component notation. The values of the arguments are sent to the target activity, where it will be received by an accept statement that defines the given entry point (see below). Execution of the sending script does not wait for the message to be received.

To represent a synchronous send, the entry is invoked as a procedure, as in Ada. The execution of the sender is suspended until the receiving activity accepts
the message, and the activity goes into a waiting state. In both forms of the send operation, if the target activity is neither live nor suspended, a TASKING.ERROR exception is raised.

The waiting state is distinguished from the suspended state described above. When an executing activity goes into a waiting state after executing a synchronizing statement, the role player can continue to perform data and role manipulation and invoke tools in the context of the activity. Depending on the statement on which the activity is waiting, the user may or may not be able to directly send messages to the activity. However, if an activity is explicitly suspended, even the role-based privileges are not available to the user. The state of an activity can change from waiting to executing due to a message being sent or received. However, an activity in the suspended state can be made live only by the execution of a resume.work statement.

The accept statement receives a message from an activity. The parameter to an accept statement is an entry point and a list of formal parameters. If a message has already been sent to the entry point, the values are bound to the formal argument and execution of the receiving activity continues. If the message was sent in synchronous mode, the (previously waiting) sender is also allowed to continue. If no message exists for that entry point, the activity goes into a waiting state until such a message arrives.

The scope of the formal parameters of an accept statement is defined by the
— Message send by instance of system_design_phase activity
to a design_planning activity, referenced by des_plan
send des_plan.first_phase_complete (sd);

— Message receive in instance of design_planning
accept first_phase_complete (sd : system_design) do ... end ;

Figure 5.4: Examples of Message Send and Receive
do ... end pair (see Figure 5.4). If a message is sent in synchronous mode, the
sender is allowed to continue only after the entire accept statement (i.e., the code
upto the end statement) has been executed. This is similar to an Ada rendezvous
[54]. If the message was sent in an asynchronous mode, the formal parameters
are bound to the values in the message, but the execution of the accept statement
does not affect the execution of the sender. Because of the asynchronous mode of
message transmission, the formal parameters of entry points cannot have “out”,
or “in out” parameters.

Figure 5.4 illustrates the use of the message passing commands. In the first
statement, an instance of the system_design_phase activity (see Figure 5.2) sends a
message to a design_planning activity to signal completion of the first phase of the
system design activity. The attribute des_plan is a reference to the design_planning
activity that has been initialized by the parent activity (system_design_and_planning
in Figure 5.1) at the time of activity creation. The entry point first_phase_complete

is defined in the design_planning activity type\(^2\). The data sent is a reference to an instance of a system_design object type, that has been produced by (the role players that performed) that activity and its subactivities.

The accept statement defines the entry point first_phase_complete, along with its formal parameter. It accepts a message that is sent to the entry point, irrespective of the activity that sends it. If the identity of the sender is desired, it has to be passed as a parameter.

### 5.3.2 User Messages

The accept_cmd statement provides entry points for a role player to interact with an executing activity. A user sends messages to an activity at these entry points, to typically signal the beginning or the end of a step of the activity. Any action that the IPSE is to take in response to the event is embedded in the code that follows the accept_cmd statement. Loops, if-then-else, and nondeterministic select statements can be used to represent the desired sequencing of steps within an activity. A role player cannot send messages to entry points defined in ordinary accept statements. They are restricted to communication between activities.

The statement below illustrates an accept_cmd statement, which appears in the script of the design_system activity type.

\(^2\)Though the examples only show the definition of the activity “body”, the entry points of an activity also need to be declared in a specification part (as in Ada). Issues related to separate compilation, and the separation of specification and implementation have been ignored in this proposal.
accept_cmd first_phase_complete (sd : system_design)

annotation (''signal completion of the first phase'');

When control reaches an accept_cmd statement, the activity goes into a waiting state until the user sends a message. Messages can be sent in two ways – by picking an item out of a menu of activity specific commands, or by typing the command from the user interface. The name of the command is the same as that of the entry point, and its arguments have to match the formal parameters of the entry.

A menu corresponding to the current selected activity lists all the user commands that the script is waiting for at any instant. When a command is selected from the menu, the user is interactively asked to provide the values of the actual parameters. Alternatively, the user may type the command shown below

first_phase_complete (sys_design)

where sys_design is a synonym (see Section 3.5) for an instance of the system_design object type defined by the user assigned to the activity.

The (optional) annotation part contains a string that provides documentation for an entry point. The system user can query the annotation for an entry point to get a better understanding of the purpose of the message to be sent.

An activity can send a message to another activity, even if the receiver has not yet executed an accept statement for the corresponding entry point. However, a user cannot issue an activity specific command until the corresponding accept_user statement has been executed. This semantics allows the script to enforce a desired
sequencing of steps\(^3\).

Ordinary input statements are sufficient to lead a user through a sequence of steps and collect data. However, the semantics of the accept-cmd statement goes beyond simple data collection. Execution of an accept-cmd statement makes the activity go into a waiting state, returning control to the user who is then free to manipulate objects or invoke tools that (s)he has access to. In contrast, a read statement waits for the user to supply a value without relinquishing control. Secondly, accept-user statements can appear as input guards of nondeterministic select statements, giving the user a choice of commands to interact with the system.

Since interaction between a user and an activity can only occur through messages, an analogy can be drawn with object-oriented systems. A message is sent to an entry point, the code corresponding to which can be thought of as a method (e.g., as in Smalltalk [53]). However, in this model (which is similar to the Ada tasking model), an entry point can be defined more than once in different ways. Thus, the semantics of the method that gets executed in response to a message depends on the current state of execution of the script. Additionally, the points in time when a user can send messages to an activity can be controlled by the script, unlike conventional object-oriented interfaces where an user can send a valid message to an object at all times without any sequencing constraints being imposed.

\(^3\text{Planning approaches are less rigid. Sequencing constraints on steps are expressed using preconditions. Any attempt to perform a step out of sequence can be detected by plan recognition, and permitted if the preconditions of the step can be automatically satisfied by the system.}\)
However, because activities have their own thread of control, method inheritance cannot be performed.

5.3.3 Language Constructs for Defining Activity Scripts

The language constructs used for defining the logic of activity execution include those that have been used in defining triggers and tool envelopes. Ada constructs are used for sequential programming. Language constructs and operations described in Chapters III and IV are used to manipulate database and role objects. Tool envelopes are invoked as procedures. In addition, the nondeterministic select statement of Ada is also provided.

The *select* statement allows waiting for, and selecting a message to receive, from one or more alternatives. Each alternative is an accept or an accept.cmd statement (the input guard), preceded by an optional boolean condition (the boolean guard), and followed by sequential code (including nested select statements). Different alternatives are separated by the keyword *or*.

When a select statement is executed, all the boolean guards are evaluated first. Alternatives whose boolean guards evaluate to TRUE, or which do not have a boolean guard, are said to be open. If one or more messages corresponding to the open input guards have already been sent, one of the alternatives is nondeterministically selected, and the accept statement, as well as the code that follows (up to the next alternative, or the end of the select statement) is executed. If no message
select
  accept first_phase_complete
  (sd : system_design) do ... end;

or
  accept cmd terminate_sys_design ...

end select;

Figure 5.5: Example of a Select Statement

exists, the activity goes into a waiting state. In a select statement, if all boolean
guards evaluate to FALSE, a SELECT_ERROR exception is raised. An example
of a select statement is shown in Figure 5.5.

The following section presents a detailed example to illustrate the language con­
structs presented. Section 5.5 describes additional semantics of activity execution,
focusing on issues related to database transactions, exception handling, dynamic
modification of activity definitions during a project, etc. Section 5.6 presents high
level descriptions of some other typical project control policies and how the above
language framework can be used to model those policies.

5.4 An Example of Activity Coordination During System
Design

The system design example of Figure 5.1 is elaborated in this section, to illustrate
how activity coordination and communication policies are represented in a process
program. The policies for controlling the system_design_phase activity have been
5.4.1 Overview of the Subactivities of the Design Phase

The design phase is decomposed into three major activities, design related planning (budgeting, scheduling, etc.), design of the overall system architecture, and detailed design. System design and planning are started together. The subactivity of planning that deals with initial project planning related to budgets, etc., may begin independently. However, the development of schedules for detailed design can begin only after the major (top level) subsystems have been identified. This event is signalled by a set of messages. After system design is complete, a preliminary design review is held. If the review is completely successful, the system design activity is terminated, and detailed design begins. If the review team suggests that certain subsystems or interfaces be redefined, detailed design can still proceed on the approved subsystems, in parallel with iterations on the system design. As more and more subsystems are approved by further reviews, the list of newly approved subsystems are sent to the detailed design activity, which can then schedule their refinement.

Figure 5.6 shows the data objects that are instantiated to record the system design. It is assumed that designers use the generic object editor to instantiate and create links between these objects to form a network structure that represents the system organization. A tool (possible a graphical one) that is integrated with
type system_design is object
attributes
    system_name : nametype;
    top_level_subsystems : seq of subsystem_design;
.....
end object;

type design_subsystem is object
attributes
    subsystem.name : nametype;
    major_design_decisions : text;
    design_alternatives : text;
    subsystem_interface : interface_descr, subobject;
.....
    components : seq of uniontype {subsystem_design,
                              module_design}, subobject;
.....
end object;

Figure 5.6: Object Types for Documenting the System Design

the object base can also be used to create the database objects; the exact nature
of the editing tool is not relevant to this description. Figure 5.7 shows the flow of
messages between these activities (a subset of the activity hierarchy of Figure 5.1).
Figure 5.8 (in pages 230 through 233) shows the script for the activity, parts of
which are explained below.

5.4.2 The Prototypical System_design_phase Activity

The system_design_phase activity type is instantiated and initiated by its parent,
design_phase. It is assumed that at the time of its instantiation, requirements and
Figure 5.7: Message Flow Among Subactivities of System Design Phase

Specifications have already been defined as networks of object instances, each with a root object. These root objects, along with a reference to the parent activity, are passed as parameters. The activity is assigned to an instance of the role type system désign_phase_manager.

The system design phase activity needs to coordinate its execution with its siblings, instances of design planning and detailed design phase respectively. The parent activity sends references to these activity instances with a message which is received by the statement

```
accept init.data ( ... ) ...
```

The system.design.phase activity has two major steps that are initiated under
the control of the phase manager, a design followed by a review. This is followed by an iterative process where errors identified by the previous review are corrected and reviewed again.

The first step (see Figure 5.8 in page 231) that is initiated by the system design phase manager results in the creation and initiation of an instance of the design_system activity type. The system design phase manager sends a message to the system design phase activity, passing as a parameter the reference to the role instance that will be responsible for the overall system design. The role instance is created using the create_role operation provided in the user interface. Role creation is an example of resource assignment and could itself involve a nontrivial process that is carried out by the manager by consulting a resource database (e.g., see [42]). That process is not modeled here to simplify the example.

Following that event, the design phase activity goes into a wait state until it receives a message from the design system activity (at the first_phase_complete entry point in page 231), signalling the completion of a "rough" architecture definition. This message is sent by the system design activity in response to a command from the system designer, who has to identify the completion of the rough design, but does not have to worry about the policies that have to be enforced after such an event has occured. The code that follows the statement

```plaintext
accept first_phase_complete (...) ...
```

represents the clerical actions that are to be performed by the IPSE in response
to the event.

The design_planning activity represents the aggregation of two subactivities, initial_project_planning and schedule_detailed_design. The initial project planning activity involves the development of project budgets, overall project planning, etc., and is started in parallel with the system design phase activity. However, the schedule_detailed_design activity type is not initiated until the first cut of the system design has been completed. When the system design phase activity receives the above message from its child, it sends a message to the design planning activity, in order that the activity for scheduling detailed design can be started. This is achieved by the message

```
send des_plan.first_phase_complete (...) ...
```

Note that the entry point first_phase_complete is defined in more than one activity.

When the entire system design activity has been completed to the satisfaction of the system designer and the other team members (subsystem designers), the system design phase activity receives another message signalling the availability of the complete system design. This results in another message being sent to the design planning activity, which can start refining the detailed design schedule as a result.

At this point, the system design phase manager is allowed to initiate the preliminary design review (step 2 in page 232). Once again, the manager selects a role instance to conduct the preliminary design review, and the script takes care
of instantiating the appropriate activity, assigning the given role to the activity, and assigning read rights for the system design object to the review team manager. The review team manager is also given the control_acl right because it is his or her responsibility to delegate read rights to the team members who will conduct the review. The code that follows

```plaintext
accept_cmd start_pdr ( ... )
```

performs these functions.

The review itself can have two possible results, either everything was completely satisfactory to the review team, or certain aspects of the high level design such as the decomposition of systems into subsystems, the interfaces between them, etc. were not found to be acceptable. In the situation modeled in Figure 5.8, it is assumed that the only problem that the review team might find is that certain subsystems need to be redesigned, or their interfaces redefined.

The select command that follows waits for either of the two messages. In either case, a message is sent to the parent. In the situation where the review is completely satisfactory, the reference to the root of the system design object network is passed to the parent, which will then start the detailed design activity.

The `system_design_phase` activity itself may then terminate\(^4\).

\(^4\)The `system_design` activity may have to be restarted later in the project to handle requests for modifications to the system design as a result of requirements changes, or design flaws discovered in later phases. This may be modeled by either making the current activity go into a waiting state from which it can be activated later, or, by terminating the current activity and handling requests for modification by creating new instances of the `system_design_phase` activity type.
If the review is not completely satisfactory, the list of unapproved subsystems is sent to the parent. The parent activity (of type design_phase) initiates the detailed design activity, and informs the system design phase activity. When the activity receives the message corresponding to the statement

```c
accept det_des_started;
```

it allows the manager to initiate redesign.

The iteration between redesign and review is modeled as a loop (step 3 in page 233). After each review, the newly approved subsystems are sent to the detailed design activity, so that personnel may be assigned to work on detailing the design of those subsystems. Another instance of the redesign activity is then started for the remaining unapproved subsystems.

### 5.4.3 The Design_system and Design_subsystem Activity Types

The design_system activity (a subactivity of the system_design_phase activity), and its subactivity, subsystem_design, are partially defined in this section to illustrate further the interaction between activities and the users that are performing them. In particular, the design_subsystem activity illustrates how activities that occupy the leaves of the hierarchy (in the activity tree) typically let the user have control in manipulating a certain set of objects, and require direct interaction only to

---

5To simplify the program, it is assumed that approval is given only to top level subsystems and intermediate level decompositions (see Figure 5.6) are not considered separately.
type system_design_phase (parent : design_phase;
  reqs : requirements; specs : specifications) is activity
role system_design_phase_manager;
attributes

-- ****** child activities ******
sd_activity : design_system, subactivity;
pdr : prelim_design_review, subactivity;
sd_revision : revise_system_design, subactivity;
rd_review : redesign_review, subactivity;

-- ****** sibling activities ******
des_plan : design_planning;
dd_phase : detailed_design_phase;

-- **** public variable that shows current state ****
curr_state : (sys_design_inprogress, sys_design_complete,
pdr_inprogress, pdr_complete, redesign_inprogress,
rd_review_inprogress, revisions_complete, ...), public;

-- **** references to database objects ****
sys_design : system_design, public;
subs_for_redesign : set_of subsystem_design;
subs_approved : set_of subsystem_design;
pdr.rev_rep : review_report;
redesign.rev.reps : seq_of review_report;

-- ****** private attributes ******
sys.des.mgr : system_designer;
pdr.mgr : pdr_manager;
redesign_reqd : boolean := FALSE;

Figure 5.8: Definition of Attributes of System Design Phase Activity
begin

--- *** get activity references from parents ***

accept init_data (dp : design_planning;
    ddp : detailed_design_phase) do
    des_plan := dp; dd_phase := ddp;
end;

--- *** step 1. Start system design. ***

accept_cmd start_system_design
    (sd_mgr : system_designer) do
    sd_activity := create_activity (system_design, parent => self
        specs => specs, reqs => reqs, $role => sd_mgr);
end;

--- in specs => specs, lhs is formal parameter of system design
    and rhs is formal parameter of system design phase
curr_state := sys_design_inprogress;
initiate (sd_activity);

--- *** wait for messages from system design activity ***

accept first_phase_complete (sd : system_design) do
    send des_plan.first_phase_complete (sd); sys_des := sd;
    add_access* (sd, self.$role, {READ, CONTROL.ACL});
end;

--- Ref. to system design object assigned to public attribute.
    -- Read access given to all system design related objects.

accept refinement_complete (sd : system_design) do
    send des_plan.refinement_complete (sd); sys_des := sd;
    add_access* (sd, self.$role, {READ, CONTROL.ACL});
end;

curr_state := sys_design_complete;
delete_activity (sd_activity);

Figure 5.8: continued. Script for Controlling System Design
--- **** Step 2. Start preliminary design review ****
accept_cmd start_pdr (rev_mgr : pdr_manager) do
  add_access* (sys.des, rev_mgr, {READ, CONTROLACL});
  pdr := create_activity (prelim_design_review, reqs => reqs,
  specs => specs, parent => self, sd => sys.des, $role=>rev_mgr);
end;
curr_state := pdr.inprogress; initiate (pdr);

--- Wait for review activity to signal completion.
select
  accept design_fully_approved (rev_rep : review_report) do
    pdr_rev_rep := rev_rep;
  end;
  send parent.sdphase_complete (rev_rep, sys.des);
or
  accept design_partially_approved (rev_rep : review_report;
    unapproved_subsystems : set_of subsystem_design) do
    pdr_rev_rep := rev_rep;
    set_insert (subs_for_redesign, unapproved_subsystems);
  end;
  send parent.sdphase_partially_complete (pdr.rev_rep,
    sys.des, unapproved_subsystems);
  redesign_reqd := TRUE;
end_select;
add_access* (self.$role, pdr.rev_rep, READ);
delete_activity (pdr);

--- **** Perform iteration if PDR wasn't successful ****
if redesign_reqd then
  -- Wait for parent to start detailed design
  accept det.des_started;
  accept_cmd start_redesign;
  sd_revision := create_activity (revise_system_design,
    unapproved_subs => subs_for_redesign, specs => specs,
    reqs => reqs, $role => sys.desmgr, parent => self);

Figure 5.8: continued. Script for Controlling Review
curr_state := redesign_inprogress;
initiate (sd_revision);

-- *** Iterate until review is successful ***
loop select
  when curr_state = redesign_inprogress =>
    accept redesign_complete;
    -- It is assumed that no subsystems are created or deleted
    rd_review := create_activity (redesign_review, reqs => reqs,
      specs => specs, sd => sys.des, $role => rev_mgr);
    subs_redesigned => subs_for_redesign,
    curr_state := rd_review.inprogress; initiate (rd_review);
    delete_activity (sd_revision);
  or
  when curr_state = rd_review.inprogress =>
    accept design_fully_approved (rev_rep : review_report) do
      seq.append (redesign_rev_reps, <rev_rep>);
      add_access* (self.$role, rev_rep, READ);
    end;
    send dd_phase.revisions_complete (subs_for_redesign);
    -- Subsystems for which detailed design has to be started
    send parent.revisions_complete (redesign_rev_reps);
    curr_state := revisions.complete;
    delete_activity (rd_review);
  or
  when curr_state = rd_review.inprogress =>
    accept design.partially.approved ( ... )
    -- Send approved subsystems for detailed design and
    assign redesign task for others. Update attributes.
......
  end select;
end loop;
end if; -- if redesign_reqd
......
signal important checkpoints.

The design system activity is performed by the system designer. In the activity script defined in Figure 5.9, it has been assumed that the system designer identifies the top level subsystems of the main system, and distributes responsibilities for detailing out the preliminary design of the subsystems (including identification of next level subsystems, subsystem interfaces, etc.) to subsystem designers, who work in the context of a subsystem design activity.

At a certain point during the system design process, the system designer can decide that the "first phase" of design is complete, and informs the activity of that event by sending it a message. Any time before or after that event has been signalled, the system designer can create more top level subsystems and assign more subsystem designers to refine the design. When all subsystem designers have submitted their designs, the system designer can signal the completion of the "final version", which results in a message to the parent activity, causing a review to be scheduled (as described in the previous section). Figure 5.9 contains the script for this activity. The attribute declarations have been omitted.

Figure 5.10 defines the script for the subsystem design activity. Subsystem designers update the subsystem design object by creating new (lower level) subsystems, defining interfaces etc. All changes are made (by the object editor, or any other integrated tool) in the private database, and are checked into the public database once the designer is ready to share the design with other members of the
loop
  select
  when not fph.complete =>
    accept.cmd first.phase.complete (sd : system.design)
    do sys.des := end;
    begin update (all);
      undersend parent.first.phase.complete (sys.des);
      first.phase.complete := TRUE;
    exception ......
    end;
  or
  when first.phase.complete and num_active designers = 0 =>
    accept.cmd refinement_complete (sd : system.design)
    do sys.des := sd; end;
    begin checkin (all);
      change.owner* (sys.des, parent.$role);
      send parent.refinement_complete (sys.des);
      exit; -- get out of the loop and terminate script
    exception ......
    end;
  or
  accept.cmd assign subsystem designer (sd : subsystem.design;
    sd.designer : subsystem.designer;
    group.obj : system.design_group) do
    index := index + 1;
    seq.append (design_activities, create_activity (design.subsystem,
      reqs => reqs, specs => specs, sd => sd, ind => index,
      parent => self.$role, $role => sd.designer,
      grpobj => group.obj);
    add.access* (sd.designer, sd, WRITE);
  end;
  num_active designers := num_active designers + 1;
  initiate (design_activities [index]);
  or
  accept subsystem submitted (sd : design subsystem;
    ind : integer) do
    num_active designers := num_active designers - 1;
    seq.remove (design_activities, index, 1); -- deletes subactivity
  end;
end select;
end loop;

Figure 5.9: Activity Script for System Design
type design_subsystem (sd : subsystem_design;
    specs : specifications; reqs : requirements;
    ind : integer; grp_obj : system_design_group, public;
    parent : system_design) is activity
begin
    loop
        select
            accept_cmd submit_design;
            begin checkin (all);  
                change_owner* (sd, parent.$role);
                send parent.subsystem_submitted (sd, ind);
            exit; -- terminate
            exception ......
            end;
        or
            accept_cmd add_group_access;
            add_access* (grp_obj, sd, READ);
        end select;
    end loop;
end activity;

Figure 5.10: Activity Script for Subsystem Design
group. Before checking the objects in, access has to be given to the group. This may be performed by the subsystem designer using the user interface commands (the reference to the group object is a public attribute of the activity), or can be delegated to the activity by sending it an “add_group_access” message. According to the script defined in the figure, the message to add the group object to the access control list may be sent at the discretion of the user.

A more stringent policy could prevent checkin of a subsystem_design object (with a trigger) unless the above message is sent prior to checkin. When the activity is initiated, the script could set a particular attribute in the subsystem_design object (that the subsystem_designer’s view does not have access to). This attribute is reset in the code that follows the add_group_access entry. Since modifications to the subsystem_design object with the object editor can only be made within the scope of a long transaction (a restriction at the user interface level), this enforcement is possible. If the changes are made with an external (e.g., a graphical) tool integrated with the system, the restriction has to be enforced by an appropriate definition of the tool view, an issue that is being addressed in [140].

The example also assumes that additional objects that are created by the subsystem designer (such as other intermediate level subsystems) are properly linked to the parent objects so that the entire subsystem design is structured as a tree. Currently, that is not enforced as a constraint. Such enforcement can be implemented by a modify_checkin trigger (see Section 3.4) that checks that an appropriately
type system_designer is role
  can create object {system_design, subsystem_design};
  can create role {subsystem_designer};
  view of system_design is
    (can read all, can update all);
  view of subsystem_design is
    (can read all, can update all);

type subsystem_designer is role
  can create object {subsystem_design};
  view of subsystem_design is
    can read all,
    can update all);

Figure 5.11: Role Types of Designers

defined inverse attribute has a non null value at checkin time.

Definitions of the user role types system_designer and subsystem_designer are partially described in Figure 5.11. The system designer is allowed to create system_design and subsystem_design objects, as well as subsystem_designer roles. The system designer also has access to all attributes of system_design and subsystem_design objects. Subsystem designers can instantiate objects of type subsystem_design, and read and modify all attributes of such objects. However, they cannot instantiate system_design objects or subsystem_designer roles. These role definitions restrict the actions of the designers acting in the context of the above activities.
5.5 Issues Related to the Semantics of Activity Execution

This section discusses a number of issues related to the definition and execution of scripts. Some of the issues have been addressed in the design and the solutions are also described.

5.5.1 Short Transactions and Activity Execution

Every database access that is initiated by a script is treated as a short transaction. This default behavior can be overridden by enclosing a group of statements within an atomic . . . end atomic pair. However, since the execution of a script can continue across login sessions, there are restrictions on the statements that can be grouped together to be executed as one short transaction.

An executing activity can go into a wait state when a synchronous send, an accept or an accept.cmd statement, or a suspend_work or an abort statement is executed. At any such point, control returns to the user, who is free to terminate the session with the IPSE. Therefore, the system state has to be saved at such a point, and a group of statements to be executed as a short transaction cannot include these statements.

A “foreach” loop is always executed as a short transaction\(^6\) and therefore, cannot include the above statements in its scope.

\(^6\)The implementation has to use a cursor mechanism (see Chapter VI).
groups of statements declared to be atomic, it gives rise to nested transactions.
As mentioned in Section 3.4.5, without a nested transaction mechanism in the underlying database, such a feature cannot be supported, and the easiest option is to ignore directives that control the scopes of transactions if they are nested.

5.5.2 Long Transactions and Script Execution

In a tool envelope, statements to checkout and checkin operations are treated as nested long transaction operations. However, long transaction operations that appear inside triggers and scripts are executed on behalf of the user who initiated its execution.

All long transaction locks are owned by the activity instance, in the context of which the user is currently working. Therefore, even after the script of an activity executes its last statement, the activity cannot complete its execution until all remaining objects have been checked in by the user.

5.5.3 Scheduling Activities for Execution

When a waiting activity can continue execution as a result of the receipt of a message, it is put in a ready state. An activity in the ready state can be scheduled for execution. A number of different scheduling techniques can be used in the virtual machine. These are outlined below.

---

7In [57], the owner of a long transaction lock is the user. Since multiple roles are supported here, a single user cannot be the owner.
accept.cmd statement, the activity is immediately scheduled for execution, since generating an immediate response to the user command has the highest priority.

The major scheduling issues arise in the context of activities that are waiting for messages from other activities. When an activity goes into a ready state because a message that it sent in synchronous mode (was waiting for) is received (sent) by the script of another activity, the user responsible for the activity:

1. may not be logged on,

2. may be logged on, but working in the context of a different activity, or

3. may be working in the context of the same activity.

In case 1, if the activity is scheduled for execution and the script that is executed needs to interact with the user, invokes an interactive tool, or simply results in an error, user response would be required. An error situation can be handled by saving a record of the error in a persistent object (see Section 3.4.2), but there is no obvious solution to the other two problems. Therefore, a possible strategy for the process manager could be to not schedule an activity if the user is not logged on. However, if it is possible to detect (by static analysis) that a handler for a message from an activity does not involve tool invocation or user interaction, the activity could be scheduled. The advantage of scheduling activities as soon as possible is that the execution of a script may enable other activities to become ready. If execution is postponed till the user logs on, the progress of activities
becomes dependent on the user logging on regularly.

In case 2, an activity could be scheduled, irrespective of whether the corresponding script involves user interaction. Scheduling may be done whenever control comes back to the system (i.e. when the user issues an object editor command, or a tool invocation is completed). However, when the user is working in the context of one activity, suddenly receiving messages from another activity, or worse still, being forced to interact with a tool invoked by a different script, is less than ideal. This can be a problem in case 3 too, where the current activity is waiting on one or more accept statements while the user is working with tools and objects. In either case, an interaction is forced on the user if scheduling is automatic.

The best option is to allow the user to explicitly control the scheduling of ready activities (assigned to a role associated with the user's login id), since no surprises are involved. However, the user is then free to indefinitely delay execution of certain activities. A viable alternative is to schedule activity execution under user control, until the time the user wants to exit from the system, at which point, all remaining ready activities can be scheduled for execution.

5.5.4 Controlling Access to Activity Instances

The execution model assumes a default set of access rights to activity instances. The user assigned to an activity instance is the only person who can “execute” (i.e. interact with) the script of that activity. The user is also given default rights
to view public attributes of the activity.

Further control of read access to public attributes should be allowed to permit project members in "manager-type" roles to query the status of all subactivities of their activity, for example. If rights to read public attributes are controlled by an access control list as used for objects, then the critical design issue is one of deciding how rights to control the ACL (the control_acl right) are distributed.

If only the role player responsible for an activity is given the control_acl rights by default, then that project member (who might be playing a non-manager role such as that of a programmer) can control the visibility of some part of the state of the project by preventing others (including managers) from seeing activity instances. An alternative would be to allow class specific access to activity instances to be specified as privileges of role types, in a manner similar to that used in database management systems. A more complex (in terms of the efficiency of implementation) alternative would be to allow a role instance to only view all activity instances that are direct or indirect subactivities of the activity it is associated with.

An orthogonal problem, that of giving different role types different views of public attributes, can be addressed by extending the view definition mechanism of role types to also include attributes of activities. That extension is trivial and is not elaborated further.
5.5.5 Direct Manipulation of Activities

While the runtime system allows direct viewing of attributes of an activity, all other interactions with the user must occur through messages. This restriction is necessary because of the object-oriented nature of a script, but it significantly complicates the job of the script writer.

In scripts corresponding to "intermediate" level nodes in the tree of activity instances, it is imperative that the role player be allowed to terminate or suspend a subactivity, or to change the role of a subactivity to assign it to a different project member. Commands to do this would have to be provided as entry points associated with accept cmd statements.

In a conventional object-oriented programming language, this is not a significant problem, since all that is required is the definition of a message and the corresponding method that is to be executed. With the tasking paradigm however, the situation is more complex because the commands that can actually be received from the user at any point during the execution of the script is dependent on the accept cmd statements that have been most recently executed. If the option to suspend, terminate, or change roles of activities is to be made available at any synchronizing point, then any single accept or accept cmd statement has to be replaced by a select statement which presents the additional options. Therefore, the first accept statement in Figure 5.8 of page 231 following the instantiation of a subactivity has to be replaced by the code shown in Figure 5.12.
select
    accept first_phase_complete ......
or
    accept_cmd terminate_subactivity ......
 ......
or
    accept_cmd suspend_subactivity ......
 ......
or
    accept_cmd change_role ......
 ......
end select;

Figure 5.12: Allowing General User Commands At Any Synchronization Point

It is possible to extend the direct manipulation capabilities of activity instances to permit subactivities to be suspended and roles of subactivities to be changed by the role player performing the parent activity. However, direct manipulation capabilities for terminating subactivities can cause deadlocks and should not be permitted.

If a subactivity is suspended manually, and the parent activity’s script is waiting for a message solely from that activity, then, the script cannot continue execution until the child is restarted by the user using a resume_work command. However, in the same situation, a manually issued abort command would cause a deadlock because the parent script would never receive the message.

Changing the role associated with a subactivity to be directly altered can be allowed, because it typically does not affect the parent script. The major design
decision involves deciding when to make the effect of the role change visible to the original role player (who can no longer perform the activity as a result), if the role change occurs during activity execution. This problem also arose in the context of activity termination or suspension during execution (see discussion on page 213) and the same solution can be adopted.

5.5.6 Error Handling

Exception handling features similar to those described in Section 3.2.5 are also used in scripts. The system defined exceptions raised due to errors in accessing the database or manipulating activities have already been defined.

The semantics of exception propagation is similar to Ada. If data manipulation (which may cause trigger chains to be invoked) or tool invocation is initiated by a script, any exception raised, but not handled by the triggers or the tool envelope, should be handled by the script. If the data manipulation or tool invocation was initiated by the user from the user interface, unhandled exceptions are propagated back to the user.

When synchronous sends are used and the caller waits, the error propagation semantics are identical to that used in Ada. Exceptions raised inside the accept statement and not handled locally are propagated to the caller. Also, if the script containing the accept statement is aborted, a TASKING_ERROR exception is raised in the caller. When asynchronous sends are used, the caller is not affected
by exceptions raised in the activity that receives the message. If the receiver
terminates and messages for that activity still exist (which can occur because of
asynchronous sends), a TASKING_ERROR exception is raised.

5.5.7 Procedures and Parallel Threads of Control

Activity types can contain internal procedures. Such procedures can have access
to the attributes of an activity. Procedures cannot contain synchronous send,
accept, or accept.cmd statements, a restriction that is imposed to simplify the
implementation of saving the state of a script when a session is terminated.

Having parallel threads of control within an activity is an extension that has not
been considered in the current proposal. Multiple activity types (and instances)
can be used to model parallel subtasks of a single user's activity. However, a
user performing a subactivity will then be unable to access objects checked out by
the parent activity (since the locks are owned by the parent activity). A special
construct that represents a subtask which operates in the same scope as the activity,
and yet allows parallelism, is illustrated in Figure 5.13.

A subtask type can have multiple instantiations, and is related to the parent
activity by attributes. Being in the same scope allows scripts of subtasks to directly
modify attributes of the parent activity to represent execution states, without the
overhead of message transmission. Subtasks can also be used to model parallel
invocation of tools (one from each task instance) on behalf of the user.
5.5.8 Modifying Process Programs During Execution

The most difficult problem that needs to be addressed before a process-based IPSE architecture can be effectively used, is the issue of modifying process programs during execution. The need to modify process programs arises because of the following reasons:

1. An error is found in the script of an activity.
2. An exception raised in an activity is not handled by its script.
3. Two or more activity instances get into a deadlock.
4. Some parts of the overall process or the products produced are not known at the beginning of the project, and have to be added later.

In case 1, the error can be easily corrected if no user is actually in the middle of script execution. Cases 2 and 3 can be difficult to handle under the same
circumstances. For example in case 2, new code has to be written to handle the exception, and execution must be restarted from some previous point (an unhandled exception terminates the script). That in itself is not difficult to do since the program counter for script execution is maintained in the database by the virtual machine (see Chapter VI). However, prior to restarting execution, the state of the database must also be restored to its previous state.

If control has to be restored to the previous synchronization point (synchronous send, accept, or accept.cmd), and statements between such synchronization points are always grouped as one atomic unit, then the exception would abort the transaction (a default system action) and the database state is not corrupted. If restoration requires backing up further, manual modification of the database state would have to be performed by a user with superuser privileges. Programming environment support for that activity is required.

Handling enhancements to process programs, as necessitated by the requirements described in case 4, is relatively simple. Such enhancements typically involve adding new activity types, or adding scripts to activity types that are initially defined as stubs. The example of section 5.6.3 illustrates the need for such runtime enhancements. One policy that might need to be enforced during the maintenance phase involves running a set of test cases every time a module is changed due to a modification request. Since test cases are typed objects whose structure may not be standardized, and the output of the software being built is definitely not known
until detailed design is complete, the product types used by the script for such a maintenance activity cannot be defined in the beginning of the project.

Break points can be defined for such stubs so that if execution ever reaches such a script, the user cannot continue that activity, though other project members can continue executing different scripts. Extending or modifying scripts that are not in the middle of execution is easy since no state restoration operations are required. However, sophisticated incremental compilation and linking facilities are required in the programming environment for process programming. These issues are not addressed in this dissertation.

5.6 Further Examples of Process Programs

This section provides further examples of process programming by describing how the policies for controlling some other software development and maintenance activities may be represented using activity types and scripts. In most cases, only high level descriptions are provided and detailed scripts are omitted.

5.6.1 Problem Reporting and Change Control Procedures

A problem reporting and change control procedure is a set of policies and tools that are used to keep track of software errors reported by users of the system, and to process the errors by making modifications to the software system. It is an integral part of the software maintenance process, requires the involvement
McClure describes a high level change control procedure (see page 179 of [82]) that is illustrated in Figure 5.14. Bersoff [17] describes a more detailed change control procedure that involves a greater number of user role types. Variations of these basic procedures have been adopted into automated systems for software error reporting and tracking, that have been developed either as a component of a configuration control system (e.g., see [89], and the Bell Labs Modification Request system [117]), as a project management component of a software development environment (e.g., the SAGA project [46]), or as an isolated coordination support tool (e.g., XCP [123]).
Description of the Process

The basic process illustrated in Figure 5.14 is as follows. A report about a software error is submitted to the software maintenance group by a requester, by filling out a change request form. The change request form contains information about the type of change requested, the software system involved, the type of change requested, the urgency of the request, etc. Requests for repairs to software failures, as well as software enhancements may be included in this form.

Upon receipt of the change request, it is entered into a change request database, that contains status information about all active change requests under consideration or implementation. The change request is then given to the user liaison who is responsible for the support of the affected software system.

The user liaison conducts a preliminary study to determine the effort needed to make the change, the justification of the change, its feasibility, cost, etc. The purpose of the study is to gather enough information upon which to make an initial recommendation to accept, reject, or save the request.

The user liaison assigns a priority number for the request, and submits the change request accompanied by his recommendation to the maintenance leader for consideration. The maintenance leader may override the priority number assigned or the recommendation made by the user liaison, or may request that a more comprehensive study be performed before a decision is made regarding the request.

\[\textsuperscript{8}\text{Adopted from [82]}\]
Change requests accepted by the maintenance leader, and which require a minor effort to implement, may be scheduled immediately. Change requests that are not minor must be reviewed by the maintenance administrator.

Accepted requests are scheduled for implementation and assigned to the maintenance team by the maintenance leader. The user liaison informs the requester that the request has been scheduled. When the change is completed, the user liaison informs the requester and updates the change request database accordingly.

Automation of Change Control Procedures

One major issue that needs to be addressed when providing automated support for the above class of change control procedures is the effective integration of change control and development procedures. This involves keeping links between error reports and the source code and design objects that they impact.

The other aspect involves providing the user with an “in-tray out-tray” view of the change control system. With such a metaphor, a particular role player (say the maintenance leader) can log in to the system, get a list of all unprocessed change requests that have been sent by the user liaison, examine a change request, forward a change request to the maintenance leader or schedule it for implementation, etc.

In addressing the integration issue, a centralized database of project information is a major requirement. A data model such as the one proposed in Chapter III seems to be adequate.
The second issue can be very adequately addressed by modeling individual activities of role players with activity types and scripts. Activity specific commands can be provided to:

- examine and modify an unprocessed change request,
- forward it to the maintenance leader,
- return it to the user liaison for further evaluation,
- schedule it for implementation, etc.

Sequence valued attributes of activity types can be used to model "in-trays" and "out-trays" as queues, and message send and receive primitives can be used to forward change request forms from one role player's activity to another.

The use of activity types and scripts should be contrasted with the use of a specialized mailer system to achieve the same functionality in the Saga system [46]. In effect, specialized message queues implemented as message files, and mail system commands of the Saga system, are implemented using the data modeling and activity manipulation primitives of the process programming language, respectively. This approach should also be compared with the protocol mechanism of XCP [123], which is conceptually much closer to the process modeling paradigm.

The modeling of change control procedures as activity types in a process program also leads to better integration with design and development processes. The
latter processes are also modelled as activity types in the same process programs and the exact interactions (e.g., message flows) between them is explicitly represented. In the conventional approach, any relationships between change control and development procedures are buried in isolated shell scripts and tools distributed among the change management and development support systems. This further illustrates how the process programming approach leads to improved visibility and better understanding of the overall software process.

5.6.2 The Change Coordination Policy of Infuse

Infuse is a change management tool developed at AT&T Bell Labs [70], which provides automated support for enforced cooperation among a group of maintenance programmers, who are making changes to a large collection of modules in response to a maintenance request. Infuse is a generalization of CMU's Smile system [69]. Smile supports the concept of a main database where stable software is stored, and multiple experimental databases in which modules are reserved prior to modification. This is equivalent to the baseline and private databases of the long transaction model presented in Chapter III. Infuse introduces the concept of a hierarchy of experimental databases for recursively partitioning the total set of modules that are to be changed, and thereby forces the programmers to change the modules in a given order. A brief

*Such an implementation would follow the change control procedure described in the previous section.*
(and simplified) description of Infuse is presented next. The section that follows
describes how the change coordination process of Infuse is modeled with a pro­
cess program. The subsequent section contains a discussion of the limitations of
the process and data model proposals, that makes it impossible to duplicate the
functionality of Infuse with the process-based approach.

A Description of Infuse

An experimental database in Infuse consists of a collection of reserved modules.
The top level of the hierarchy of experimental databases contains the modules to
be changed in response to the request. This database is then partitioned into child
databases, where disjoint subsets of the modules are placed in each child. Modules
are placed in a particular experimental database according to some criteria that
partition the collection of modules. The partitioning is repeated recursively until
the experimental database at each leaf contains one or more modules assigned to
an individual programmer. Intermediate level databases in the tree are associated
with more than one programmer.

One possible criterion involves partitioning the modules based on the strengths
of the syntactic and semantic dependencies among the modules [103]. This ap­
proach attempts to put within one subtree of experimental databases those mod­
ules whose new changes will interact most strongly. The goal is to change and
check the intermodule consistency among subsets of the modules in an optimal
order that reduces the total number of iterations required.

The modules in a leaf database are changed by an individual programmer, and the deposit command is issued to signal that the change is complete. After all modules have been deposited in an experimental database, an analysis tool that checks that the modules are consistent with each other (with respect to symbol definitions) is invoked. Modules outside the experimental database are not considered. If the modules are consistent, they are deposited into the parent database. If they are not, all the programmers associated with the experimental database are informed. Deposit of the modules to the parent database is not permitted until they are consistent with each other.

The programmers negotiate changes to their modules that will resolve the conflict, and choose a subset of the modules that must be changed again. These are partitioned recursively as before and the process is repeated. The change process stops when all the modules are deposited back to the root database. This is illustrated in Figure 5.15 (taken from [70]).

When part of a hierarchy is deposited, inconsistencies may be detected, requiring repartitioning. This is referred to as the “yo-yo” effect. When there is a lack of cooperation among the programmers involved, this yo-yo effect can occur frequently. This can be monitored and appropriate managers can be notified.
Initial decomposition hierarchy

Modules C, D, E and F changed, analyzed and submitted to parent database

Inconsistency between C, D, E and F results in repartitioning

Figure 5.15: A Hierarchy of Experimental Databases in Infuse
A Process Program Representing the Change Policy

The hierarchy of experimental databases in Infuse controls the order in which modules are modified. This can be modeled by a hierarchy of activity instances, each one corresponding to a node in the tree of databases. The root and the intermediate databases are represented by instances of the "make_partition" activity type. The script of this activity invokes the automatic partitioning tool to perform an initial partitioning, and spawns subactivities that correspond to child databases. Leaf level databases are represented by the "change_module" activity type. A single programmer is associated with this activity, and a command is provided to deposit a module (for simplicity, it is assumed that the leaf level database contains only one module).

After partitioning the modules corresponding to an intermediate level database, the script of make_partition waits for all the child activities to complete. When all the child activities have completed, it invokes the "check_group_consistency" tool to check for intermodule consistency, and initiates repartitioning if an inconsistency is reported. The change_module activity invokes the "check_module_consistency" tool in response to the deposit command, and if the module is self consistent, it sends a submit message to the parent to signal completion of the individual task. The partial definitions of these activities is presented in Figures 5.17 and 5.18. Figure 5.16 contains the definition of data objects that are used to represent the partitions.
The root instance of make_partition is instantiated by another activity type that initiates the change activity. An object of type "problem_descrn" that describes the bug is passed to the root activity, and this is passed on to each of the subactivities. The "create_groupings" tool invoked in make_partition performs the partitioning of the modules based on the strength of syntactic and semantic interconnections of the modules. The interconnection of modules is represented by additional attributes of the module object that are not elaborated here.

It is assumed that one administrator is assigned as the role player for an intermediate level activity, and this person identifies the modules to be recoded in the case of an intermodule inconsistency. This is a departure from the implementation of Infuse, where each intermediate level database can be "owned" by many programmers. This simplification is made necessary because of the restriction that an activity can be associated with only one role player, and is further discussed in the next section. The remaining code should be self explanatory.

Limitations of the Process and Data Models

A number of limitations of the proposed architecture are brought out by this example, because a lot of the functionality of Infuse cannot be duplicated exactly by the above process program. These are described below.

As already discussed, only one role player can be associated with each intermediate level activity, and it is the responsibility of this role player to inform
type partition descr is object
    attributes
        module groups : set of union type
            {module set, unit module}, subobject;
    end object;

    type unit module is object
        attributes
            mod : module, subobject;
            mp : maintenance programmer, subobject; -- role
    end object;

    type module set is object
        attributes
            mod set : set of module, subobject;
            admin : administrator, subobject; -- role
    end object;

Figure 5.16: Data Objects for Representing Partitions
type make_partition (mset : set of module; pd : problem.descr, public; mp.parent : uniontype {make_partition, ...}) is activity;  
attributes partition : partition descr, public;  
    cons.chk.results : ..., public;  
    subtasks : set of uniontype  
        {make.partition, change.module}, subactivity;  
begin  
partition := create.groupings (mset); -- Role player is the owner  
loop  
    accept.cmd create.subtasks;  
    -- Programmers/administrators must be assigned for elements of  
    -- partition, using object editor. Iterate until complete  
end loop;  
foreach mg in partition.module.groups loop  
    if type.of (mg) = unit.module then  
        set.insert (subtasks, {create.activity (change.module, ...)});  
        add.access (mg.mod, mg.mp, WRITE);  
    else  
        set.insert (subtasks, {create.activity (make.partition, ...)});  
    end if;  
end loop;  
foreach st in subtasks loop initiate (st); end loop;  
loop -- Wait for all subtasks to complete  
    if not size (subtasks) = 0 then  
        accept.submit (st : uniontype ...);  
        set.remove(subtasks, st);  
    else exit;  
    end if;  
end loop;  
loop -- Repartition if necessary  
    cons.chk.results := check.group.consistency (mset);  
    if cons.chk.results.satisf then send mp.parent.submit (self);  
    else -- Identify modules to change. Repartition and repeat  
    end if;  
end loop; .......  
end activity;  

Figure 5.17: The Script of the "make_partition" Activity
type change.module (mod : module, public; pd : problem.descrn, public; mp.parent : make.partition) is activity;

attributes
mod.err.rep : module.error.report, public;

begin

loop

accept.cmd deposit;
mod.err.rep := check.module.consistency (mod);
if mod.err.rep.satisfactory then
begin
checkin (all); send mp.parent.submit (self); exit;
exception .......
end;
end if;
else prompt ("......");
end loop;
end activity;

Figure 5.18: The Script of the “change.module” Activity
the system as to which modules will be changed as a result of an intermodule inconsistency. In Infuse, more than one programmer can be associated with an intermediate level experimental database, though details about how multiple programmers interact with one experimental database are not provided in the paper.

A much more serious problem is the limited functionality of the long transaction model of Chapter III. When a programmer finishes the changes to a module, the changes have to be checked into the baseline before the parent activity (and tools invoked by that activity) can examine the changes. While the process embodied in the hierarchy of experimental databases is adequately modeled by activities, the checkin-checkout model is limited by only two levels of databases (as in Smile). This forces the changes to be made public to everybody at each stage in the iteration. What is required is a nested long transaction model, and the need for this has been already elaborated in the context of tool invocations in Chapter IV.

Infuse provides the notion of a workspace that cuts across the boundaries set by the hierarchy of experimental databases. This cannot be modeled using the mechanisms provided in the proposed architecture. Extensions to the proposal to effectively model these features are being investigated.

5.6.3 A Policy of an Individual Change Activity

A very common policy of change control dictates that after an individual module has been modified in response to a change request (e.g., as in the change_module
activity in the last section), and before it is checked in to the baseline, the module should be compiled, linked with the rest of the existing system, and a set of test cases should be run. If the results are not as expected, the change (and the submission to the baseline) should be aborted. Most program support libraries (e.g., see [7]) implement such policies using shell scripts that implement deposit commands.

In the proposed architecture, the above policy can be implemented as an activity specific command for depositing a module back into the baseline. The normal mode of deposit, the checkin command, can be disallowed with a modify_checkin trigger, that checks if the required set of tests have been run. A boolean attribute that is set to false on modification of the code, and which is set to true by the script on successful completion of the tests, has to be defined.

The execution of the test cases itself is performed by the code shown in Figure 5.19. It is assumed that the test cases to be executed for a module are related to it with a sequence valued attribute. The example itself is conceptually straightforward, but it illustrates some of the difficulties of the process-based approach. As discussed in Section 5.5.8, this example illustrates the need to be able to modify process programs during execution.

The object type test_cases can be defined only when the precise nature of the functionality of the system is known. Execution of the test cases also requires that specialized tool envelopes be defined to convert test cases to files which are input
......

accept.cmd deposit (mod : module);
-- compile module
-- link with other modules
for i in 1..size (mod.test_cases) loop
  tc := mod.test_cases[i];
  -- execute system with tc, compare with output
end loop;

Figure 5.19: Running Test Cases Prior to Checkin

to the executing program, and to interface to file comparison tools which compare test results. Therefore, the code for handling the deposit command can only be defined after a significant part of the development has been carried out.

5.6.4 Policies for Enforcing Interface Checking

In [145], Wolf et al. describe a set of notations and tools which allow precise checking of module interfaces, and the detection of various degrees of inconsistencies among these interfaces. One of the issues raised in the paper is the requirement of flexibility in an environment regarding the degree of interface checking that may be required in a project, and even within various phases of a project.

For example, when an interface is defined during the initial design phase, many details are omitted, and a very rigorous checking of interface consistency is meaningless because most of the checks would fail. However, during the coding phase of the project, interfaces have to be precise, and all possible checks should be carried
The flexible implementation of these policies can be very easily achieved in the process-based approach. In each phase, the code to handle the "deposit" command can call the appropriate tool fragments to perform the required degree of interface checking. Moreover, these tool fragments need not be provided for invocation by the user, and can be executed completely under the control of the scripts which execute with superuser privileges. The definition of these scripts is not outlined here. A possible combination of the tool fragments under program control is presented in [145].
CHAPTER VI

Implementation of the Virtual Machine on a Relational Database System

This chapter provides a high level description of the runtime system for the process programming language, focusing on the implementation of the object base and the activity model. The description is based on actual implementations of various subsets of the concepts presented in this dissertation, in research prototypes as well as commercial products.

A relational database serves as a backend object store that provides persistent storage, query facilities, and a (conventional) transaction mechanism. The virtual machine is implemented as a set of library routines that implement the data and activity manipulation operations of the process programming language. A compiler translates statements of the language into statements of a conventional (base) programming language, with calls to library routines. These library routines have embedded statements that manipulate the relational database. Some of the pro-

---

1Such a compiler does not exist yet. In the existing implementations, the translation has to be manually performed. In the actual implementations, the base languages have been PL/I, C and REXX.
cess programming language statements are also directly mapped to relational data
definition (DDL) and data manipulation language (DML) statements. The output
of the compiler is then processed by a preprocessor for the relational DDL/DML,
which replaces the commands to the relational database with calls to the runtime
routines provided by the relational DBMS.

Section 6.1 gives an overview of the runtime system, and presents alternative
scenarios for executing multiple copies of a process program. Section 6.2 describes
the mapping of the structures in the object model onto relational schemata, and
the runtime primitives that implement the data manipulation and long transaction
operations in terms of relational operations. Section 6.3 describes the runtime
system for the activity manipulation operations. The description focuses on the
implementation of the operations for message passing and execution control.

6.1 An Overview of the runtime system

Figure 6.1 shows the organization of the runtime system, and the relationship
between the process program for a project, an instance of the process program
executing on behalf of a project member, and the underlying relational DBMS.
The DBMS runs as a server process2, and each copy of the process program is
a relational database application that implements the operations of the process
program using operations on the relational DBMS. Different process programs for

2Actual implementations may use more than one process (e.g., Ingres [129]).
Figure 6.1: The Organization of the Runtime System
different projects use (logically) different relational databases, whereas different
executing versions of a particular process program act as multiple applications
that access the same relational database concurrently. The entire discussion in
this chapter centers around the latter case.

6.1.1 The Information Manipulated at Runtime

The relational DBMS is used to store two broad categories of information:

- Instances of typed objects. These are stored in “generated” relations (see
  Figure 6.1), and contain the following types of information:
  - Instances of data objects.
  - Instance of activities.
  - Instances of role objects.
  - Message queues.

- Generic runtime information. These are stored in “fixed” relations, and
  contain the following types of information:
  - Type information – such as the different types of objects, the names and
    types of their attributes, the triggers associated with database objects
    and attributes, etc.
  - Information about checked out objects, the log that records which at-
    tributes have been modified during a long transaction, etc.
— Information about executing activity instances – such as their execution states (active, suspended, etc.), the entry points at which an activity instance is waiting, etc.

The schema of a generated relation represents the relational mapping of the structure (i.e., the number and types of attributes) of an object in the process program. Tuples in these relations correspond to object instances. Since object structures vary from one process program to another, so do the generated relations.

In contrast, the fixed relations represent information used by the runtime system to control various facets of a process program's execution. This information has a predefined structure, that is mapped to fixed relations common to all process programs.

6.1.2 The Components of a Process Program

A process program has a fixed part and a variable part (see Figure 1.3 in page 24), which is further elaborated in Figure 6.2. The variable part contains translations of executable code such as triggers and activity scripts into a base language (e.g. PL/I, C, or any other language for which a relational query language preprocessor exists), with embedded calls to runtime routines that constitute the fixed part of a process program. Runtime routines manipulate the information in the different relations and perform the following broad classes of functions:
Figure 6.2: The Functional Components of a Process Program
• Translation of object manipulation operations on data, role and activity objects into operations on the underlying relations.

• Access control.

• Implementation of long transaction operations.

• Activity manipulation operations such as message sends, waits, etc.

• Object editing functions.

The first two items involves manipulation of the generated relations. The implementation of long transactions involves the manipulation of both the generated relations (for making private copies of baseline objects) as well as the fixed relations (e.g., to update the log). Activity manipulation operations that change the states of executing activity instances typically require certain fixed relations to be manipulated.

6.1.3 An Alternative Implementation

In the current design, the runtime system is included in each process program. While making the process program reentrant would reduce runtime space requirements, process programs for different projects would still link the runtime system code. Additionally, if tools are constructed to directly use the object base capabilities, they too would include the object management routines.
Figure 6.3: An Alternative Implementation Architecture
An alternative implementation of the runtime system is outlined in Figure 6.3. In that design, the runtime routines that perform object and activity management functions are encapsulated in a separate object and activity management system (OAMS), that is implemented as a standard database server. In that architecture, individual process programs are smaller, because while they still contain the variable part (triggers, scripts, etc.), the fixed part is centralized in the server. The object management component of the OAMS server uses the relational DBMS's features of persistence and concurrency control to implement the object management functions, and effectively acts as a data model translator. The OAMS also provides activity management functions, a feature typically not provided in DBMSs (see Taxis [90] for an example of an exception). Though this runtime architecture is cleaner, its implementation is far more complex. Current implementations of the system are based on the previous design.

6.2 Object Storage and Manipulation

This section describes the mapping of object structure into relations, and the translation of object manipulation operations. The implementation of triggers and long transaction operations is also described.

6.2.1 Mapping Objects into Relations

Each data, activity, role and group object is mapped into one or more relations. Since each object may have attributes that are sets or sequences of values, it is not
possible to have a 1:1 mapping between a non-first normal form object structure and a relational schema which must be in first normal form. The following mapping is used in this implementation:

- All the single valued attributes of an object are mapped into columns of a relation, whose name is derived from the object name. That relation is referred to as the base relation corresponding to the object.

- For each collection, a separate relation is defined. Each tuple of such a relation stores one element of a set or sequence and a surr_id field ties the element tuples to the tuple of the base relation that corresponds to the object. Tuples corresponding to a sequence contain an additional field (seq_no, as shown in Figure 6.4) that is used to maintain the order of the elements. The names of these relations are also derived from the object name.

In [93], this is referred to as horizontal splitting. Besides storing the single valued attributes, the root relation also contains system defined fields, the most important of which is the surrogate id of the underlying object. This value (a 4 byte integer in one implementation, a 22 byte string in another) is automatically assigned by the system at object\(^3\) creation time, and is used to uniquely identify an object instance.

\(^3\)The term object subsumes data, activity, role and group objects.
Figure 6.4 illustrates horizontal splitting with an example. Figure 6.5 shows how two object instances (each with multivalued attributes) are internally represented as tuples. In the example, surrogate ids are represented as 4 byte integers. The types of the fields of the relations are described using the keywords of SQL [64]. Normally, each attribute is mapped to one field. However, if an attribute's type is a union of two or more types, an extra field that stores the runtime type information is also defined.

An alternative mapping, which has been used in some of the older implementations of this system (and also in the implementation of Taxis [93]), uses the technique of *vertical splitting*. In that mapping, all the object types are mapped into a fixed number of relations, which store one attribute per tuple. In [93], it has been shown that vertical splitting is less space and time efficient than horizontal splitting. However, the advantage of vertical splitting is that the modification of existing object structure and the addition of new types can be achieved without defining new relations.

### 6.2.2 Mapping Object Model Operations to Relational Operations

The purpose of the runtime routines that perform object-oriented manipulation of data, role, group and activity objects, is to translate each such operation into one or more operations on the underlying relations. The nature of these translations depends on the mapping used.
type module is object
attributes
  name : string (20);
  implementor : string (10);
  design : module.design;
  interface : module_specification, subobject;
  implementation : module_body, subobject;
  test_cases : seq of test_case;
  lines_of_code : integer;
  .......
end object;

module0 (base relation)

<table>
<thead>
<tr>
<th>surr_id</th>
<th>name</th>
<th>implementor</th>
<th>design</th>
<th>interface</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>longinteger</td>
<td>char(10)</td>
<td>char(10)</td>
<td>longinteger</td>
<td>longinteger</td>
<td>...</td>
</tr>
</tbody>
</table>

module1 (relation for first sequence)

<table>
<thead>
<tr>
<th>surr_id</th>
<th>seq_no</th>
<th>test_cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>longinteger</td>
<td>integer</td>
<td>longinteger</td>
</tr>
</tbody>
</table>

Figure 6.4: Mapping an Object into One or More Relations
module0

<table>
<thead>
<tr>
<th>surr_id</th>
<th>name</th>
<th>implementor</th>
<th>design</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>&quot;stack&quot;</td>
<td>&quot;sarkar&quot;</td>
<td>34</td>
<td>56</td>
</tr>
</tbody>
</table>

module1

<table>
<thead>
<tr>
<th>surr_id</th>
<th>seq_no</th>
<th>test_cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>83</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>82</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>84</td>
</tr>
</tbody>
</table>

Figure 6.5: Object Instances and Tuples of Relations
With vertical splitting, only a few access routines are needed to access attributes of all object types. With horizontal splitting, potentially many access routines would be needed because each object type is mapped to one or more different relations. One way to avoid an explosion of access routines is to use a dynamic query construction facility\(^4\). This facility allows a relational query to be constructed as a string at runtime, which is then passed the query translator part of the DBMS. Using this feature, one access routine can be written to access any attribute of an object, irrespective of its type. The access routine has to consult type information at runtime to decide which relation, and which field within the relation, has to be accessed. That is the technique which has been used for the implementation based on horizontal splitting.

Simple Object Accesses

Figure 6.6 illustrates the translation of a simple assignment statement in the process programming language into a base language, with embedded calls to runtime routines. The base language that is used to illustrate the mapping is Ada\(^5\).

M1 and M2 are variables of type module (in a trigger, a tool envelope, or a script), and implementor is a string valued attribute of module (see Figure 6.4). The assignment statement is translated into an invocation of a generic “get” routine to fetch the value of M1.implementor into a local variable. The generic “put”

\(^4\)In SQL/DS, it is referred to as dynamic SQL [64].
\(^5\)The actual implementations have used PL/I, C, and REXX.
**** assignment statement in PPL ****
M1.implementor := M2.implementor;

****** Ada code generated *******
......
M1, M2, implementor.value, ...: value.type; -- Ada record type
......
g att r ib u te (M2, implementor.attrid, implementor.value, ......);
modify_attribute (M1, implementor.attrid, implementor.value, ...);

****** Query generated (dynamically) by get_attribute *******

select implementor from module0
where surr_id = /* integer pointer stored in field of M1 */

****** Query generated by modify_attribute *******

update module0
set implementor = /* string stored in field of implementor.value */
where surr_id = /* integer pointer stored in field of M2 */

Figure 6.6: Example of Code Generation
routine is then invoked to update the value of M2.implementor from the value in
this local variable. The Ada record type - value_type - is a generic record struc-
ture that is used to store values of the various types allowed for attributes. The
value_type record also contains the type information when the object's type is de-
fixed as a union. When union types are used, the compiler cannot always statically
check the validity of an attribute access, and the get_attribute and modify_attribute
routines have to perform the checks at runtime.

A large number of generic runtime routines for data object manipulation have
been defined and implemented. These include routines to create and delete objects,
insert and remove elements of sets and sequences, etc., which are not elaborated
any further in this chapter. The pseudocode for modify_attribute is illustrated in
Figure 6.12. Since each routine handles objects of all types, type information has
to be maintained by the system at runtime. This is discussed in Section 6.2.6.

More Complex Object Manipulation

While simple statements are translated into invocations of runtime routines, more
complex statements have to be directly translated into relational operations be-
cause generic routines are not sufficient. Examples of such complex statements are
queries, and "foreach" loops based on such queries.

General object-oriented queries are mapped into relational queries, according
to the scheme discussed in page 288. "Foreach" loops are translated into relational
database statements for defining cursors (e.g., see [64]), and the looping facilities of the host language are used to cycle through the resulting tuples. Since these are fairly standard techniques of relational database programming, they are not elaborated further.

While the basic translation is straightforward, a number of complications arise because of the presence of multiple copies of an object (due to the long transaction mechanism), the need to maintain a log (due to the delayed triggering mechanism), and the need to control short transaction boundaries (due to the arbitrary "length" of a trigger chain). These issues are addressed in Sections 6.2.3 and 6.2.4.

### 6.2.3 The Implementation of Long Transactions

There are two implementation issues that need to be addressed in the context of the long transaction mechanism: the maintenance of private and baseline copies of an object, and long transaction locks. These are described in this section. The implementation of general queries in the presence of baseline and private copies is also described.

**Multiple Copies and Checkout Locks**

Every instance of a data object can exist as either a baseline, or a private copy. In the mapping of objects to "generated" relations, a copy_no field in all the tuples corresponding to an object (see Figure 6.7) is used to reflect this distinction. The
Figure 6.7: Representing Multiple Copies of an Object
copy_no field has a value of 1 (for baseline copies) or 2 (for private copies). When a create_object operation is invoked (see Section 3.2.1), the newly created tuples are assigned a value of 1 in the copy_no field. In contrast, when the create_object_local operation is invoked, the tuples are created with a value of 2 in the copy_no field. If a baseline object is checked out for update, both copies exist simultaneously in the database. If a baseline object is checked out for read, only a lock is set, but no copy is made.

The checkout_w_id field of every base relation corresponding to an object type contains the surrogate id of the activity instance, in the context of which any instance of that type is checked out for update. This reflects the fact that the owner of a long transaction is an activity instance and not a user (see Section 3.3.5). This field also serves as the lock. When a baseline object is not checked out for update, this field has a null value. When it is checked out, the field of the tuple corresponding to both copy_no 1 and copy_no 2 are assigned the value of the activity instance. Figure 6.7 shows the internal representation of the baseline and private copies of the "stack" module, where the checkout_w operation was invoked in the context of activity id 6745.

Read mode checkout operations result in a separate count field being incremented by 1. The actual id of the owner is not important. A checkout_w operation cannot be completed unless the checkout_r_count field has a value of zero, or a value of 1 where the only reader is working in the context of the same activity. The
Figure 6.8: Relation for Storing Information About Checked Out Objects

<table>
<thead>
<tr>
<th>surr_id</th>
<th>type_id</th>
<th>checkout_mode</th>
<th>activity_surr_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>longinteger</td>
<td>integer</td>
<td>char (1)</td>
<td>longinteger</td>
</tr>
</tbody>
</table>

latter case results in a lock upgrade and the checkout_count field is decremented by 1. Similarly, a checkout_r operation raises an exception if the checkout_w.id field has a non-null value that is different from the surrogate id of the current activity instance. To perform these checks, the additional relation described in the following section is required.

**Recording Checkout Information**

Besides storing the copy_no and lock information in tuples corresponding to object instances, a "fixed" relation – $checked_out_objs – is used to store information about objects that have been checked out, in either read or update mode. Figure 6.8 shows the format of this relation. The relation contains the object type id, the surrogate id, the checkout mode, and the surrogate id of the owner activity. The checkout_mode field can have a value of "r" or "w". The part of this relation that contains information about objects checked out in the context of the "current" activity is cached by the copy of the process program that is managing that activity.
Controlling Access to Copies of Objects

When an attribute of an object is fetched, either through the object editor or by procedural code, the correct copy must be accessed (see Section 3.3.2). For example, in the assignment statement

\[ \text{M.implementor} := \text{'sarkar'}; \]

the reference to object M (of type module) should resolve to the private copy, if it is checked out for update by the activity instance in the context of which the above statement is executed. Otherwise, the base copy is updated, provided that the object is not checked out by any other activity. This is checked by the generic “put” routines by accessing the information in the relation $\text{checked}\_\text{out}\_\text{objs}$.

A more complex situation arises in the context of a general purpose query, formulated as a collection constructor. Figure 6.9 illustrates how the object model query

\[(m \text{ in module where } m.\text{loc} > 200) \ldots \]

is translated into a relational query. The complexity of the translation is caused by the presence of baseline and private copies in the object base. If the query is executed in the context of activity id 6745, the module name “stack” is included in the resulting set because the lines\_of\_code attribute of copy\_no 2 has a value of 300. However, if the query is evaluated in the context of some other activity instance, “stack” is not included in the set because the lines\_of\_code attribute of the baseline copy has a value of 190 (see Figure 6.7).
The first translation in Figure 6.9 uses the relational join and union operations. Those instances of type module which are checked out for update (mode field = "w") are joined with the tuples whose copy_no field has the value 2 in the corresponding generated relation (module0), with the additional condition in the collection constructor being used to restrict the resulting tuples. For all other objects, including those that are checked out in read mode by the current activity, tuples whose copy_no field is 1 are searched.

The second translation avoids a join, and is thus more efficient. It uses the fact that the checkout_w_id field of every base relation contains the surrogate id of the owner activity, to achieve the same result. When an object checked out for update is deleted using the delete_object_local operation (see Section 3.3.2), only the tuples corresponding to the private copy are deleted, whereas the checkout_w_id field of the baseline copy still contains the activity's surrogate id. Therefore, neither component of the union selects the tuple of the base relation.

6.2.4 The Implementation of Triggers

There are three issues to be addressed in the implementation of triggers, deciding which trigger(s) to execute immediately, controlling the scope of a transaction during the execution of trigger chains, and the implementation of delayed triggers. These are addressed in the following three sections.
-- ----- Object model query ------

(m in module where m.loc > 200) ......

-- ----- Translation 1 (using join) -----

select surr_id
from module0 m, $checked_out_objs c
where m.lines_of_code > 200 and m.copy_no = 2
    and c.owner_act = /* current activity id */
    and c.type_id = /* type id of module */
    and c.mode = 'w'
    and c.surr_id = m.surr_id
union
m.lines_of_code > 200 and m.copy_no = 1
    and c.owner_act = /* current activity id */
    and c.type_id = /* type id of module */
    and c.mode = 'w'
    and c.surr_id <> m.surr_id

-- ----- Translation 2 (without join) ----- 

select surr_id
from module0
where lines_of_code > 200 and copy_no = 2
    and checkout_w_id = /* current activity id */
union
lines_of_code > 200 and copy_no = 1
    and checkout_w_id <> /* current activity id */

Figure 6.9: The Translation of Object Model Queries
The Execution of Immediate Triggers

When a database update takes place, the runtime type information has to be examined by a trigger manager to see if any immediate triggers are activated by the update. Since the allowable trigger conditions in the proposal are trivial, no such separate trigger manager is necessary. Each runtime routine that is invoked to perform object base updates (e.g., modify_attribute in Figure 6.12) examines the type information to check if "pre" and "post" triggers should be executed as a result of the update. The implementation of the trigger manager is therefore distributed among the various database update routines.

Each trigger is translated into either a function or a procedure in the base language, which might call runtime routines, or directly invoke relational DML operations. The actual choice (i.e., function or procedure) depends on the base language. Triggers need to return error codes, the most important of which represents the occurrence of a deadlock. If the base language is Ada, the exception handling mechanism can be used, and therefore, the trigger can be translated into a procedure. In PL/I and C, where exception handling features are not supported, triggers are translated into functions which return predefined error codes. Each translated version of a trigger is assigned an index and in the type information (see Section 6.2.6), every attribute is associated with a list of procedure indices and the conditions (e.g., create, delete, modify_pre, etc.) under which they are to be invoked. This is the information that the runtime routines examines. If a
trigger is to be executed, the procedure with the appropriate index is invoked. The implementation of this table of procedures depends upon the base language. In C and PL/I, an array of references to procedures can be used. In Ada, which does not allow pointers to procedures and functions, a case statement which selects the procedure to be invoked based on its index has to be generated by the translator.

Different trigger procedures require a different number of parameters, as described in Section 3.4.1. In addition, each trigger is passed an integer parameter called \textit{self}, that contains the surrogate id of the object whose manipulation caused the trigger to execute.

**Trigger Chains**

When the (translated) code in a trigger invokes runtime routines to perform further data manipulation, more triggers may be invoked. Therefore, a trigger chain is implemented as a chain of procedure invocations. Since triggers are executed immediately, the procedure call mechanism of the base language can be exploited in the implementation of triggers. In contrast, the action equation mechanism \cite{67}, where equations triggered as a side effect of a rule are scheduled for execution only after the execution of the rule body completes, requires a queue to be maintained by the interpreter. The same is true of rule based systems.

The only other issue in the implementation of trigger chains is the control of the scope of a short transaction. This is either controlled with explicit \textit{commit}
or abort statements (see [64]), or is controlled by the runtime routines which are passed appropriate flags. The choice depends on the complexity of the object manipulation statement. The language semantics state that every atomic unit is retried automatically if it is rolled back due to a deadlock. The implementation of this involves enclosing each atomic unit in a loop, which iterates if any of the underlying relational operations that are involved in the implementation of the object model operation(s) results in a deadlock. A relational DBMS that includes deadlock detection and automatic rollback of transactions is required to implement the above semantics.

The above description is elaborated with an example. Consider the following statement which appears in the script of an activity:

\[
\text{M1.implementor} := \text{'sarkar'};
\]

Every statement in a script is treated as atomic by default (unless enclosed in a larger unit with the explicit use of the keywords atomic ...... end atomic). This statement is translated into an invocation of a modify.attribute routine. Each generic update routine takes an additional boolean valued parameter, that is used to determine if it should commit the (underlying relational database) transaction after performing the operation. In this case, the modify.attribute routine is invoked with the value TRUE, indicating that a commit is required after the modification is complete, as illustrated below:

\[
\text{modify.attribute (M1, ... , TRUE)};
\]
The logic of the modify.attribute routine itself is enclosed within an outer loop, which is repeatedly executed in the case of a deadlock and rollback, to automatically retry the transaction if the above parameter has the value TRUE. This is illustrated in Figure 6.12.

All invocations of runtime routines from trigger procedures should pass the value FALSE for the above parameter, because trigger chains are committed only after all trigger procedures have executed successfully. This is ensured by the compiler.

Figure 6.10 illustrates the code generated for the statement:

```
M1.implementor := M2.implementor;
```

when it appears in the script of an activity. It is a more detailed version of Figure 6.6. In this case, since the scope of an atomic unit is larger than the execution of one runtime routine, explicit code has to be generated to commit the transaction, or to retry it in the case of a deadlock.

Figure 6.10 also shows how the code generated for the same statement is simpler if it appears in a trigger. Triggers never commit in the middle of execution, and any DEADLOCK exception raised by a runtime routine is propagated back all the way to the caller that invoked the runtime routine that resulted in the first trigger invocation. It is the responsibility of the caller to handle the deadlock. The caller may be a script, a tool envelope, or an object editor function that is performing a data manipulation operation, and the code to retry the transaction is either
--- ***** assignment statement in script *****
M1.implementor := M2.implementor;

--- ***** Ada code generated *****
......
loop
begin
  get_attribute (M2, ...);
  modify_attribute (M1, ..., FALSE);
  -- runtime routines do not commit;
  commit_work;
  exit; -- exit loop only if no exception has been raised
exception
  when DEADLOCK => -- clean up and iterate again
  end;
end;
end;

--- *** Code generated when statement is in a trigger ***
get_attribute (M2, ...);
modify_attribute (M1, ..., FALSE);
-- no loop necessary since DEADLOCK exception is handled by the
code that caused the first trigger in the chain to be invoked.

Figure 6.10: Generating Code for Atomic Units
generated by the compiler (in the first two cases) or is already implemented (in the case of the editor). The DEADLOCK exception is predefined by the system, and is raised by runtime routines if the underlying relational database signals a deadlock error. The automatic handling of trigger chains and transactions can succeed only if trigger writers are not allowed to define handlers for DEADLOCK exceptions. This can be ensured by the compiler.

Invoking Triggers at Checkin Time

Triggers whose preconditions are modify_checkin, create_checkin or delete_checkin (see Section 3.4.1) are executed at checkin time. The relation $\text{checked\_out\_objs}$ (see Figure 6.8) contains information about objects which are created or deleted within a long transaction. When the creation or deletion of an object is checked in, any trigger associated with that object, which has a create_checkin or delete_checkin precondition, is executed by the runtime routine that implements the checkin operation.

$\text{log (base relation)}$

<table>
<thead>
<tr>
<th>obj_surr_id</th>
<th>attr_id</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>long integer</td>
<td>integer</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 6.11: The $\text{log}$ Relation
To invoke modify_checkin triggers, a log that records modifications made to individual attributes of checked out objects is maintained. Figure 6.11 describes the format of the $log relation. When an attribute of an object that is checked out for update is modified for the first time, and the attribute has an associated trigger whose precondition is modify_checkin, an entry is made in the log to record the modification. Subsequent modifications do not require any manipulation of the log because the delayed trigger is executed only after the last modification in the long transaction. However, if the object is deleted, all log entries for that object are removed, and a field of the entry in $checked_out_objs is set to record the deletion.

When an object is checked in, all the entries in the log that reflect modifications to attributes of that object are examined, and the corresponding triggers are invoked. The delayed trigger mechanism is thus implemented as part of the checkin operation.

Whenever an object valued attribute (representing a binary, 1:1 relationship) is modified in a long transaction, an entry is made in the log, irrespective of whether a modify_checkin trigger is attached to that attribute. This is required to update the reference count in the target object, information that is used to implement referential integrity (see Section 3.1.3). When a link is set between two objects, and the source object is in a private database, two cases arise. In the first case, the target object is checked out in read mode (it has to be, according to the restriction described in Section 3.3.4), and only a baseline copy exists. The reference count
of this field is incremented when the change to the private copy is checked in. In the second case, the target is checked out for update and is part of the same long transaction. In this case, a private copy exists, and a baseline copy may or may not exist, depending on whether the object was created locally within this transaction. In either case, the private copy’s reference count is immediately incremented for referential integrity for the private copies to be enforced correctly. The referential integrity for the baseline copy is incremented only at checkin time as before, and a log entry is made.

The work done by the system in propagating reference counts to maintain referential integrity may be thought of as the execution of built-in delayed triggers. In this case, the value of the object valued attribute is required at checkin time so that the appropriate reference count can be incremented. This value can be got from the private copy of the object since the object type and surrogate id information is already available in the log. The old target object is also required since its reference count has to be decremented, and this is got from the attribute value in the baseline object.

6.2.5 An Example of a Runtime Routine

Figure 6.12 outlines the pseudocode for the modify_attribute procedure. The various steps of the procedure implement the different semantics of object manipulation that have been described in Chapter III, by examining the type information,
procedure modify_attribute (object_value, attr_id, rhs_value, modifying_agent, ...) raises abort, deadlock;
begin
  while loop_more loop loop_more := FALSE;
  begin -- block
    Check if attr_id is valid for that object type;
    if modify_attribute invoked by object editor, and attribute
    is derived or object not checked out, then raise exception;
    if modify_attribute invoked by short transaction and object checked
    out by another long transaction, then raise exception;
    if union types are involved, check validity of assignment;
    execute modify_pre triggers if any;
    check declarative constraints;
    if attribute is object valued then
      save current attribute value;
      if target not checked out then raise exception;
      if attribute is a subobject then check for cycles;
    update database object(s) by generating relational query
      -- update copy_no 2 for checked out objects, and non-derived
      -- attributes; Update copy_no >= 1 for all other cases;
    if attribute is object valued then
      update ref. counts for old and new targets (copy_no 2)
      if they are part of same transaction;
      if attribute is a subobject then
        see if old target should be deleted;
      invoke modify_post triggers, if any;
    insert log entry if required;
    if commit_flag then commit work (SQL);
  exception-- abort exception propagated to caller
    when DEADLOCK => -- clean up in memory data structures;
      if commit_flag then loop_more := TRUE;
  end; -- block
end; -- loop
end modify_attribute;

Figure 6.12: The Pseudocode for the modify_attribute procedure
and by manipulating the various relations that contain type, instance, and generic runtime information.

6.2.6 Maintaining Type Information

Type information is used by runtime routines that create relational database queries dynamically, and has to be maintained at runtime. Type information is also required for implementing some of the built-in functions in the language (e.g., type.of), to implement runtime type checking for union types, and to provide online, object-oriented query facilities.

A number of fixed relations are used to store persistent type information. Conceptual and external schemas (views) are defined in separate compilation units of a process program and tools, and type information is collected from these units during compilation. A high level description of these relations is given below.

The $Objects relation contains one entry for each data, role, group, and activity object type. Each tuple contains the unique type id of the object, and a field that identifies the category of the object (i.e., data, activity, etc.).

The $Attributes relation contains one tuple for each attribute of an object. A type id field determines the parent object type, an attribute id uniquely identifies the attribute within the object, and other fields describe the type of the attribute, and whether it is single valued or a collection.

The $Constraints relation contains a tuple for each declarative constraint of an
attribute (e.g., range, nonull). The $Triggers relation contains one tuple for each trigger associated with an object or an attribute. The tuple contains information about the triggering condition, and the index of the trigger procedure that is generated by the compiler. Triggers attached to objects are associated with a dummy attribute for that type.

The index of the procedure that is the base language translation of the script of an activity (see following section) is stored in the $Objects relation. This field has a non null value only for activity objects. Since an activity only has one script, a single field is sufficient, and a separate relation such as $Triggers is not required.

### 6.3 Activity Manipulation Operations

There are three broad classes of activity manipulation operations — operations for the creation, deletion and manipulation of attributes of activity objects, operations for the initiation, termination, suspension and resumption of activity execution, and operations for sending and receiving messages. The sections that follow describe the various relations and runtime routines that are used to implement primitives for the above operations.

Two fixed relations and a number of generated relations are used in the implementation of the activity model. The fixed relations $all_activities and $activity_record are shown in Figure 6.13. These relations, as well as the generated relations, are discussed in the following sections.
6.3.1 Creation, Deletion, and the Manipulation of Attributes of Activities

Activity objects are similar to data, role and group objects in that they are also aggregates of attributes. Activities are different from other categories of objects because they can have attributes whose types are other activity types. However, this does not have an impact on the underlying implementation. The mapping of activity structure to relations is therefore similar to that for database objects, as described in Section 6.2.1. Activity creation (deletion) is mapped to the creation (deletion) of one or more tuples in these relations. As with other categories of objects, a unique surrogate id is automatically generated by the system. Attribute
access and modification is mapped into invocations of the "get" and "put" routines described earlier. An additional set of relations, which correspond to message queues for different entries, are generated for each activity type. This is described in Section 6.3.4.

Since activity objects have associated execution semantics, the creation of an activity has to be recorded in the fixed relation $all_activities. This relation contains one tuple for each instantiated activity. Each tuple is uniquely identified by the surrogate id, and contains information such as the surrogate ids of the owner (role instance) and the parent activity, the type id of the activity, its execution status, and the point in the script from which execution should begin if the activity is waiting, ready, or suspended. The status field (see Figure 6.13) contains integer codes that correspond to the various execution states – instantiated, executing, ready, waiting, suspended, and terminated (see Figure 5.3). This field also takes additional values, which are required to properly implement the suspend_work and abort statements.

When an activity is instantiated, a new tuple is inserted in $all_activities. All fields are assigned appropriate values except code_ptr, which remains null. The status field is assigned (the code for the) state – instantiated.
6.3.2 The Translation of Scripts into Base Language Procedures

The script corresponding to an activity type is translated into a procedure, in a manner similar to triggers. However, because a script can be suspended during execution, additional mechanisms are required to temporarily halt and resume the execution of a base language procedure across login sessions.

A script can only be suspended at points where synchronous send, accept, accept.cmd, suspend_work and abort statements are executed\(^6\). Since a suspended script can be resume execution from statements that follow the above operation invocations, base language labels are generated for these statements. A script is translated into a procedure with a number of labelled statements, a parameter called self that contains the surrogate id of the activity instance (similar to the parameter passed to triggers), and a parameter called $label which specifies the label of the statement from which execution should (re)start.

The first part of the translated procedure must be a set of statements that pass control to the appropriate statement in the script, depending on the value of $label parameter. With a base language like PL/I, a computed GOTO statement can be used, but with languages like Ada and C, a case statement that represents a jump table has to be generated. This is illustrated (for Ada) in Figure 6.14.

\(^6\)Even if a script tries to suspend or abort a child activity with an explicit statement, the actual suspension does not occur until the child activity's script reaches one of these statements. This semantics is described in Section 5.3.1.
type XXXX is activity

begin

pi.entry_point1 (...); ... statement group 1 ...;
accept entry_point2 (...); ... statement group 2 ...;
accept.cmd entry_point3 (...); ... statement group 3 ...
end activity;

**** Translated Procedure ****
procedure XXXX (self : integer; $label : integer);
case $label of
  10 : go to 10;
  20 : go to 20;
  30 : go to 30;
......
end case;
......
code for pi.entry_point1 (...);
if waiting required then
  save label 10 in $all_activities; return;
end if;
10 : code for statement group 1 ...... ;
code for accept entry_point2 (...);
if waiting required then
  save label 20 in $activity_record; return;
end if;
20 : code for statement group 2 ...... ;
code for accept_cmd entry_point3 (...);
waiting is always required for accept.cmd statement
save label 30 in $activity_record; return;
30 : code for statement group 3 ...... ;
set state to 'terminated';
end XXXX;

Figure 6.14: Sample Translation of a Script
A script is initiated or resumed in response to an explicit execution control operation in a parent script, a message send or receive operation, or a user command. The runtime routines that implement the above operations in a script may enable other scripts to become ready for execution. The activity manager schedules a ready process for execution by invoking the translated procedure, passing it a value for $label that is saved in one of the two fixed relations, which determines the point at which the script should resume execution.

If an executing script has to be suspended at any point, the address (label) from which execution should be restarted is saved in a relation (either $all_activities or $activity_record), and control returns from the procedure to the activity manager. Section 6.3.4 describes further the code generated for message send and receive operations, as well as for select statements, elaborating on how the execution state is saved.

6.3.3 Operations for Controlling Script Execution

Different runtime routines are provided to initiate, suspend, resume, and terminate an activity instance. The common action corresponding to each of these operations is the modification of the status field of $all_activities. Each operation may have additional side effects. Subsequent sections describe the actions taken for the above operations.
Initiate

When an activity instance is initiated, the status field is set to the code for the state - executing, and execution of the script is begun from its first statement. The first statement always has the label 1, and this is the value passed to $label by the activity manager. Execution continues until the script has to suspend or terminated, and the status field of $all_activities is changed accordingly before control returns to the activity manager.

Suspend_work

When a script executes a suspend_work statement to suspend the execution of a child script, a number of situations may occur. In the simplest case, the child activity is in a waiting state, and no role player is working in the context of that activity. The child's state is changed to suspended, and the parent script continues execution.

A more complex situation occurs when the child script is in an executing state. The execution may take an arbitrarily long time if the script has invoked an interactive tool. In this case, the status field of the child activity is changed from executing to to_be_suspended, and the status field of the parent activity is changed to waiting. When the child script's execution reaches a point where it can give up control, its state is changed to suspended, and the parent's state is changed to ready, so that it can be scheduled for execution. If a user command resulted
in the execution of the suspend_work statement in the parent script, the user can continue working in the context of the script, while the script itself waits for the suspension to become effective. The user can also log out of the system.

The third situation occurs where the child script itself is in a waiting state, but the user is working in the context of the activity. In this case, the status field of the child activity is changed to to_be_suspended and the parent is made to wait, as before. The child activity's state is changed to suspended only when the user hands back control to the system (as described in Page 213), at which point, the parent's script can be scheduled for execution.

When the state of an activity is changed to waiting because of the execution of a suspend_work statement, the label corresponding to the next statement is saved in the code_ptr field of the tuple corresponding to that activity in the $all_activities relation.

One further complication occurs because a script that is in a waiting state can be suspended by its parent. The script might have been waiting for a message to be received or sent, or for a suspend_work statement to be successfully executed, and when that event finally occurs after the script itself has been suspended, its state cannot be changed to ready. However, the event should be recorded so that once the script is resumed with a resume_work command, the state of the script can be appropriately adjusted. This is accounted for by saving the new status in the temp_status field when an activity is suspended. When the suspended activity
is finally resumed, its status field is updated with the contents of the temp.status field.

Resume_work

The implementation of the resume_work command is simpler. The child activity should be in a suspended state, and the status field is updated from the contents of the temp_status field. The parent script continues.

Abort

The implementation of the abort statement is very similar to that of the suspend_work statement in that, if the activity to be aborted is in an executing state, or a user is working in the context of the activity, then the abort operation takes place in two stages. First, the status field of the child activity is changed to to_be_terminated and the parent activity's state is changed to waiting. When the child activity's script reaches a point where the actual abort can take place, the activity is aborted, its state is changed to terminated, and the parent activity’s state is changed to ready. However, if the parent activity has itself been suspended in the meantime, the temp_status field is updated instead.

When an activity is aborted, the undo operation is invoked by the activity manager, to abort any long transaction that is executing in the context of that activity. The semantics of the abort statement is also more complex than the
suspend_work statement because when an activity is aborted, all its subactivities have to be aborted too (see Section 5.3.1). The activity manager implements this by recursively invoking the above procedure for all descendant subactivities.

6.3.4 Operations for Sending and Receiving Messages

Operations for sending and receiving messages are available in all message based parallel programming languages, and the implementation techniques are well documented. Techniques for implementing nondeterministic select statements with only input guards are also described in the literature (e.g., see [45]).

All implementation mechanisms require the definition and manipulation of queues corresponding to entry points at which messages can be sent, and the manipulation of task records that are used to save the execution state of a process before it is suspended. The major issue that is addressed in this section is the necessity of making messages and task records persistent. Since activities execute only when the corresponding role players are logged in, the interval between the sending and the receipt of a message may span login sessions. This is handled by implementing message queues and task execution records as relations. The sections that follow outline the format of these relations, and the implementation of the message passing operations.
Message Queues and Task Records

Each entry point defined in the script of an activity type has a fixed *signature* that defines the number and type of formal parameters for that entry point. The parameters define the format of the messages that may be sent to that entry point. One or more relations are defined for each entry point of an activity type. These relations represent the *common message queue* for that entry point, for all instances of that activity type. This design decision differs from typical runtime systems for parallel programming languages, where one message queue is allocated for each process, and was motivated by the need to reduce the total number of relations.

The mapping of an entry point definition into one or more relations is based on the same technique that is used to store attributes of objects. Single valued parameters are mapped into columns of one relation, while each set or sequence parameter is mapped into its own relation. Figure 6.15 illustrates the mapping for an entry point. The msg_seq_no field is used to order the arrival of messages, the surr_id field uniquely identifies the activity instance that the message was addressed to, and the entry_id generated by the compiler uniquely identifies an entry point of an activity type.

Figure 6.13 shows the format of the relation $activity_record, that is used to store information about the messages that a script is waiting for, when the corresponding activity goes into waiting mode. The use of this relation is outlined in the sections below that describe the implementation of the message passing
-- Entry point defined in object type XXXX

... entry YYY (m : module; dset : set of designers);

<table>
<thead>
<tr>
<th>surr_id</th>
<th>entry_id</th>
<th>msg_seq_no</th>
<th>sender_surr_id</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>longinteger</td>
<td>integer</td>
<td>integer</td>
<td>longinteger</td>
<td>longinteger</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>surr_id</th>
<th>entry_id</th>
<th>msg_seq_no</th>
<th>dset</th>
</tr>
</thead>
<tbody>
<tr>
<td>longinteger</td>
<td>integer</td>
<td>integer</td>
<td>longinteger</td>
</tr>
</tbody>
</table>

Figure 6.15: The Implementation of Message Queues
operations.

Sending Messages

When an asynchronous send statement is executed, the runtime routine inserts tuples in the message queue relations to store the message being sent. The seq_no field is assigned a value that is one more than the maximum value in the relation\(^7\). If the target activity is in a waiting state, and it is waiting for a message to this entry point, then its state is changed to ready. Execution of the script that performed the send operation is continued.

For a synchronous send operation, tuples are inserted in the message queue as before. However, the sender's script execution is suspended and the activity's state is changed to waiting. If the target of the message is waiting for a message on this entry point, its state is changed to ready. The sender will be made ready for execution once the receiver has executed the code to accept the message and the rendezvous is complete.

If the target activity is in a suspended state, and prior to that event it was waiting for a message to this entry point, then the temp_status field is changed to ready, and the status field is left unchanged. When a resume_work command is executed to restart that activity, its status field will be updated from the contents of the temp_status field, as described earlier.

\(^7\)Since a four-byte integer field is used for this purpose, a few billion messages to an entry point can be accommodating without the occurrence of overflow.
Accept Statements

When an accept statement is executed, the runtime routine that implements that operation examines the message queue for that entry point to test if any messages have already been sent. If so, then the tuples with the minimum value in the seq_no field (the first message in the queue) are extracted and assigned to base language variables that represent the parameters of the entry point. The tuples are also deleted from the message queue relations.

If the sender had executed a synchronous send when transmitting this message, the sender.surr_id field of the base message relation has a nonnull value. In that case, if the sender is still waiting, its status field is changed to ready. However, if the sender has been suspended in the meantime, only the temp_status field is changed to ready. If the sender has been terminated, no action is taken.

If no messages have been sent already, after executing an accept statement the activity goes into a waiting state. An appropriate entry is made in the $activity_record relation to indicate which entry point the script is waiting for. In this case, the label of statement from which execution should continue is stored in the code_ptr field of the $activity_record relation instead of the $all_activities relation. This makes the implementation uniform with the more general case where an activity can wait for more than one message at many entry points by executing a nondeterministic select statement. Tuples of $activity_record play the role of a task record in the implementation of processes in a conventional parallel programming
language such as Ada (e.g., see [45]).

Accept_cmd Statements

The implementation of this statement is very similar to that of an accept statement, except that annotations and the names of entry points are also stored in $activity_record. These fields are used by the activity manager to show the annotations and names of entry points to role players.

When the user issues an activity specific command by sending a message to an activity instance, the instance itself maybe in a waiting or ready state. The state cannot be suspended, because before the user can issue an activity specific command, it must return control to the activity manager, and any request to suspend the activity can be successfully carried out.

An activity is normally in a waiting state if it has only executed an accept_cmd statement. The activity manager responds to the user message by invoking the base language procedure mapping the script, with the label stored in the code.ptr field of $activity_record as a parameter. However, if the activity had gone into a waiting state by executing a select statement containing many accept and accept_cmd statements, then it is possible that a message corresponding to one of the accept statements was sent while the user was not directly interacting with the activity. In such a case, the activity is in a ready state. The user is informed about the pending message, and the part of the script following the accept statement is
select Statements

The implementation of the select statement is a slightly more complex version of
the implementation of the accept or accept_cmd statements. Code is generated to
evaluate all boolean guards, and for those guards that evaluate to true, the message
receive (accept and accept_cmd) statements are examined one by one. If messages
exist for more than one of the receive statements, one is chosen nondeterminis-
tically. If only one message exists, the selection is deterministic. If no messages
exist, all entries corresponding to the statements with true boolean guards are
recorded in $activity_record as before, and the state of the activity is changed to
waiting. In the last case, when a message finally arrives and is accepted, all the
entries in $activity_record are removed. The set of tuples that represent these
entries correspond to a conventional task record.
CHAPTER VII

Summary and Conclusions

This chapter summarizes the goals of the dissertation, justifies the programming language approach, evaluates the architecture, the proposed language features for data, activity, tool and user role modeling, discusses implementation issues, and presents additional research issues that were raised in each area as a result of this work.

7.1 Summary of Goals and Results

The goal of this research was to design an IPSE architecture that can provide customized support for enforcing the policies of various software development processes. The architecture is based on the following concept:

- At IPSE construction time, an explicit representation of project control policies is created.
- At IPSE use time, the above representation is executed on a virtual machine to provide project-specific support for enforcing those policies.
In progressing from the initial identification of the above research problem, to the language and virtual machine design that has been presented in this dissertation, the following major steps were taken:

- Existing software environments were analyzed to get a better understanding of the inability of isolated tools to enforce project life cycle control policies. Various project control policies, particularly those that are used to coordinate group activities, were studied. These examples were used to further motivate the need for the process-based architecture that has been proposed.

- A programming language paradigm was selected for representing software process knowledge. Alternative formalisms such as Petri nets, frame systems, rules, and planning languages were also studied before selecting the programming language approach.

- The major representational requirements for modeling software processes were identified. The decomposition of a process representation into data, tool, user role, and activity models has served to drive both the language and the virtual machine design.

- Two major language design issues were selected for investigation:
  - the problem of designing an appropriate data model for software engineering databases,
— the modeling of concurrency in the software process.

Simplifying assumptions were made about tools and user role models in order to present a complete proposal.

• The runtime system for the process programming language was designed and partly implemented to demonstrate the feasibility of the architecture.

### 7.1.1 Contributions of this Research

The following are the major contributions of this dissertation research:

• The identification of the four models that are required to describe the policies of a software development process, and the design of the IPSE architecture driven by those models.

• The design of language features for data modeling and manipulation. The major contributions in this area are – the design of data modeling primitives for modeling software project information, the definition of the syntax and semantics of database triggers and interactive design transactions, and the semantics of their interaction.

• The design of language features for activity modeling. The major contribution in this area is the use of parallel programming techniques for modeling the policies of coordination and communication among concurrent activities during software development.
• The design and partial implementation of the virtual machine for executing process programs, using a relational database backend.

The overall contribution of this research is the demonstration of the feasibility of a fundamentally new approach for building software environments. Philosophical issues regarding the construction of process models, and their use in constructing software environments, are being addressed by a number of other groups. While this dissertation has concentrated on only a subset of the process modeling problem (concurrency), issues ranging from the representation formalisms and the virtual machine design, down to the actual implementation of the runtime system, have all been addressed, representing a "complete" solution for that subset. This can be used as the starting point for further enhancements to the process programming language, as a testbed for building and studying the feasibility of executable process models, and for experimenting with more efficient implementations of the virtual machine.

7.2 Justification of the Programming Language Paradigm

While not many process modeling formalisms have actually been developed, those that have been presented range from transition/Petri nets, rule and plan based paradigms, to programming languages. There are two major aspects that have to be addressed in process modeling: representing the activities of a single user, and representing coordination policies of project teams. The efficacy of each of the
above formalisms is judged in light of those two contexts.

In modeling the steps that a single user should take in performing an activity, transition nets, the control flow of conventional languages, and rule based forward chaining systems are equivalent in the degree of rigidity that they impose on the steps. In contrast, plan-based systems can use backward chaining to recognize missing steps, and take automatic action if possible. In the context of software development, activities rarely contain a detailed specification of steps. In the typical step in leaf level activities, the user informs the activity model that some task is complete, at which point the script checks exit criteria, accepts or rejects the completion, and collects data if necessary. In that context, the need for backward chaining is limited.

Proponents of plan-based approaches for task support also claim that planning system interpreters have generic capabilities for dealing with plan failures, whereas, in procedural approaches, explicit exception handlers have to be written. If most exception handling is generic, the argument is valid. However, if different exceptions are to be dealt with in different ways, then the use of exception handlers as presented in the various examples is more appropriate, though tedious.

In dealing with parallelism, message based communication is typically used, though Petri nets, as well as the shared working memory paradigm of rule based systems (e.g., Genesis' activity model [107]) can also be used. In Taxis [28], scripts that model the ordering of steps in an event are represented as Petri nets, but
communication with other scripts is modeled with messages. It is possible to model the impact of one user's action on the possible actions of other users in a rule based system, where the first user's action fires a rule that sets working memory elements which enable other rules to fire. Since both programming languages and rule based systems are computation universal systems, the comparison of one to the other essentially boils down to a software engineering argument.

Flat rule based systems lack modularity. The same argument can be made against plan-based systems that contain a large number of operator definitions, each operator defining an allowable user activity (e.g., Grapple [62]). The typed object approach presented in this dissertation is much more modular than flat rule systems. The attributes of an activity clearly define the subactivity relationships, and from the script of an activity that is not too large, it is easy to understand the embedded policies of various steps, and the points of interactions between different activities.

In addition, it should be emphasized that the language presented here is not purely imperative, since not everything that the user does is under program control. Database objects can be directly manipulated using the object editor facility, with controls on user actions being defined by access control lists and the class specific rights defined in role objects. Embedded triggers represent code that executes in response to user actions, and is a fundamentally different control paradigm often referred to as access-oriented programming [126]. Even a script defined in
an activity object only comes into play when a user *directly* interacts with it by sending a message; a whole range of activities can be performed outside the control of the script. This gives a lot of flexibility to the IPSE designer, who can model life cycle control policies without having to explicitly represent all possible orders in which a user may perform an activity if such control is unnecessary.

In conclusion, the proposed language is modular, not purely imperative, uses a message based paradigm for modeling concurrency that is cleaner than a shared memory model, and allows specification of tasks with varying degrees of control over user actions with scripts and role type definitions. Database triggers represent integrity control policies, role type definitions represent access control policies, and scripts represent policies related to key events (typically at a higher level than the modification of an attribute). Such a multi-paradigm representation scheme is more general than a single (e.g., rule based, plan based) paradigm. Section 4.1.3 illustrates how some of the advantages of backward chaining, especially in the area of "intelligent" tool invocation, can be obtained by extending the tool representation features of the language.

7.3 Evaluation of Results, and Future Research Issues

This section evaluates the process-based approach, the language primitives for activity, data, tool, and user role modeling, and the prototype implementation and discusses research problems that need to be addressed in each area.
7.3.1 The Process Based Approach to IPSE Construction

While the process based approach allows experimentation with project control policies, and the analysis of the impact of different processes on software productivity, the benefits can only be obtained at the price of greater complexity in the IPSE architecture. The major impediment to the success of such an approach is the complexity of writing process programs. The great multitude of data objects, activity types, roles and tools in a typical large project would result in a process program of very large size. The need to express parallelism increases the complexity even further.

A more fundamental problem is one of extracting the knowledge of software processes in the first place. In an organization which undertakes a lot of large software projects, project control expertise only exists among a few key individuals, and is rarely written down in a document. Furthermore, these individuals are typically experienced managers, and not programmers who can write process programs. This gives rise to the classic knowledge acquisition problem where a "knowledge engineer" has to extract the knowledge from the expert.

The other unique requirement of a process programming environment is the requirement of being able to modify process programs during execution, and has been outlined in some detail in Section 5.5.8. An error in the process program (e.g., an unhandled exception, a database error, a deadlock) during its execution (i.e., while the IPSE is being used), cannot result in the entire program being aborted.
Therefore, the error must be corrected and execution must be continued with minimum loss of work. It may be reasonable to stop everybody from using the system while such recovery is being performed, since the situation is much more complex than the simpler scenario in databases, where only schema change operations such as the definition of new object types, or the modification of existing object types can be performed on the fly (e.g., as addressed in Orion [10]).

Research Issues in Process Based IPSE Architectures

While the process programming language presented in the dissertation is appropriate for coding, it is not an appropriate language for specification or design. Graphical specification languages (e.g., role activity diagrams [100], Rombach's graphical language [114]), as well as Williams' behavioral model [144] are appropriate starting points.

Very sophisticated programming environments need to be designed for process programming. The state of the executing program is always maintained in the database, and during program modification, these should be made visible to the process programmer so that recovery from errors, and the continuation of program execution from a previous state can be carried out easily. Typically, errors will occur in scripts that do not handle some exceptions, or which get deadlocked. The process programmer has to change the script to correct the error, continue execution from a previous point, and also undo changes to data objects that have
been erroneously updated due to the error in the script. The last task may be quite difficult if a script has an error (in the coding of an automated policy, say) which is not discovered for some time. However, the same problem can occur in a toolkit IPSE where a shell script encodes a wrong policy and updates a lot of files or databases erroneously.

Tools for debugging process programs will have more complex requirements than conventional debuggers. Since process programs manipulate databases extensively, appropriate interfaces to the database should be provided to allow examination of the database contents during program execution. The problem of debugging parallel programs is another research topic that is being actively addressed.

Since project control policies typically do not change very drastically from one project to another within one organization, there is great scope for reuse of process program components. Libraries of activity descriptions, tool envelopes, role and data type definitions can be maintained, and a process program for a project can be constructed by the selection of appropriate object types, and their subsequent tuning. However, since the data model does not support behavioral object orientation (i.e., a pure abstract data type approach to data manipulation), the usual benefits of an object oriented approach to reusability are not currently available. The pros and cons of a pure object-oriented data model are discussed in Section 7.3.2.
7.3.2 The Data Model

There are three major issues that have been addressed in the design of the data model, the primitives for modeling object structure and relationships, the trigger mechanism for constraint checking and propagation, and the long transaction mechanism. These are evaluated and future research issues in environment databases are outlined.

Object Structure and Relationships

The proposed data model can be used to represent both fine granularity objects (such as syntax trees) as well as coarse granularity objects (such as a module). However, the implementation technique of using an underlying relational database makes it unsuitable (from a performance point of view) for representing and manipulating fine granularity information.

The comparison between a data model based on objects and attributes, and the ER model has already been presented in Section 3.1.3. In the object model, objects are also used to represent higher order relationships, or binary relationships with attributes, in accordance with the principle of semantic relativism. When relationships can be represented as objects, schema integration can be facilitated because an object may be viewed as an entity in one view, and a relationship in another. This is not possible in the ER model. Additionally, part-of and is-a hierarchies are more easily introduced in the object model, and is less natural in
the ER model.

The proposed data model does not incorporate primitives for representing class hierarchies and versioning relationships. This decision was simply motivated by the need to keep the model and its implementation simple, and these are features that need to be incorporated in a realistic design.

The data model does not support behavioral object orientation, since attributes can be directly accessed. It is not at all clear as to whether a pure object oriented approach is appropriate for all domains. A database, by its very nature, is a shared information structure, accessed by more than one application (or tool), and mechanisms such as views already provide control of access to attributes in ways orthogonal to that provided by the object oriented paradigm. Many object oriented databases (e.g., Orion [10], VBase [97]) do not restrict access to individual attributes of objects (in effect, providing default methods whose names are the same as the attributes). However, they also allow the definition of useful methods that access or manipulate multiple attributes. Furthermore, an object oriented model is not appropriate for a hypertext oriented interface, which allows individual attributes to be manipulated and links to be traversed in generic ways.

The Trigger Facility

Triggers serve two major functions – they check for constraints, and propagate side effects of database operations. The trigger definition language is not as high level
as a declarative constraint language, but its imperative nature makes it more easy to specify side effects.

A declarative constraint language allows concise specifications of integrity constraints that have to be maintained on the information in the object base (e.g., [86]). The proposed trigger mechanism represents the other end of the spectrum because the precondition of a trigger can only be a simple update operation on an object or an attribute. For example, to represent even a simple constraint that imposes a relationship (say "\(\geq\") on the values of two attributes, two separate triggers would have to be defined, whereas one constraint specification would suffice in the declarative approach. However, the trigger mechanism provides flexibility in allowing the specification of when a trigger should be executed (e.g., immediately, after a short transaction\(^1\), or after a long transaction), representing different units of consistency.

The greatest advantage provided by the trigger facility is the capability of describing side effects of data manipulation operations. Declarative mechanisms can be extended to handle side effect propagation in response to constraint violation, but they become very unwieldy (e.g., [86]). As described in Section 3.5.2, the trigger definition mechanism, in conjunction with the generic object editor facility, can be used to implement the functionality of many special purpose interactive tools. The trigger facility allows the implementation of an IPSE interface based on the

\(^1\)The current proposal does not allow this case.
principle of incremental computation and immediate feedback, similar to syntax editor based programming environments such as Gandalf [94]. Of course, the usefulness of such a paradigm for tasks other than programming can be questioned, but examples such as history managers, presented in Section 3.5.2, do provide some justification for the feature.

Issues related to triggers and nested (conventional) transactions have been discussed in Section 3.4.5. Since a chain of trigger activations is treated as one short transaction, embedded tool invocations which define their own atomic units cannot be implemented without a nested transaction model.

The Long Transaction Mechanism

The extension of existing database transaction mechanisms to satisfy the requirements of interactive design transactions has not been adequately addressed by database researchers. Simple proposals such as the one by Haskin et al. [57], as well as the one proposed in this dissertation, are only a beginning. More complete proposals such as Bancilhon et al.'s [9] are too complex to be implemented or used. Alternative proposals such as split transactions [105] that do not have separate mechanisms (e.g., two kinds of locks) for long and short transactions, and still work within the bounds of serializability, have to be extensively evaluated before they can be incorporated into actual DBMSs.

The major advantages of the proposed long transaction mechanism are the fa-
cility to lock objects across login sessions, the permitted violation of serializability (though the checkin (all) operation can be used to enforce serializability if desired), and in conjunction with triggers, the ability to delay constraint checking and propagation until commit time. The equivalent of checkout and checkin operations are provided by any library or version control system in a software environment, and the long transaction mechanism integrates those operations into the DBMS.

The major source of complexity is introduced by the need for long transactions and short transactions to coexist and interact in the same database. This issue has been discussed in the context of a tool envelope in Section 4.1.1. The current solution that has been adopted is the following, if a short transaction accesses an object checked out in the context of a different activity, an exception is raised. Therefore, if the short transaction is part of a tool, it cannot proceed. The split transaction mechanism proposes an alternative solution where the long transaction can be committed temporarily (where the transaction is split), the short transaction’s request can be satisfied, and the long transaction can be restarted. The difficulty of identifying feasible split points would make the implementation quite complex.

The other issue raised in Section 4.1.2 is that of nested long transactions, caused by users invoking tools from the system interface, and by tools invoking more tools. The actual implementation of such a nested long transaction model is a conceptually simple extension of the prototype implementation. Since delayed triggers
would have to be executed for each level of checkin, the log becomes more complex. An alternative design in which tools do not invoke long transaction operations at all, and thus obviates the need for implementing nested long transactions, is also discussed in that section.

**Research Issues in Software Engineering Databases**

Bernstein [15] provides a detailed description of the special requirements of software environment databases. This section focuses on a few of the issues that have been raised by the proposed data model and language design.

Immediate extensions that need to be considered are the incorporation of class hierarchies and versioning. Issues related to inheritance (both single and multiple) have been adequately addressed in the literature, and its incorporation into the model is much less of a problem than its efficient implementation. One issue that has to be addressed is the refinement of triggers in the same way that methods of superclasses are refined in subclasses (e.g., see [126]).

Version modeling has been addressed in some recent proposals (e.g., Orion [27], Iris [14], XSQL [39]), but more work remains to be done in incorporating and extending those proposals for software development, and integrating them with the long transaction mechanism. The current long transaction mechanism does not incorporate versioning, and a checkin operation after modification of a private object results in the baseline object being overwritten. New modes of checkout, with
the explicit purpose of creating new revisions, will give rise to a greater degree of concurrency, since a baseline object checked out for the purpose of creating a new revision is equivalent to a checkout operation. Furthermore, representational capabilities of generic objects and contexts (see above proposals) needs to be compared with sophisticated version and configuration description facilities (e.g., DSEE [80]) to assess their adequacy.

The trigger mechanism needs to be extended with more powerful preconditions, and the capability to execute different triggers as separate transactions. The HIPAC [47] has addressed some of these representational, as well as implementation issues. Alternatives to the trigger paradigm, for example the extension to attribute grammars as implemented in the Cactis DBMS [61], need to be compared.

The design of a query language has been ignored in this proposal. The most complete proposal for an object oriented query language known to the author is available in Orion [11]. Object oriented databases do not have a well developed underlying theory as the relational model. The notion of relational completeness has no analog in the object oriented world. In contrast, the theory of normalization has been incorporated in non-first normal form data models (e.g., [50]), and since the structural component of the object oriented data model is a subset of that model, the revised normalization theory can be used to produce good database designs.
7.3.3 The Activity Model

The activity model is central to the process program, since all actions take place in the context of an activity instance. The major issue in activity modeling is the nature of the representation language. The proposed model is based on an imperative script language, but as argued in Section 7.2, this does not make the process programming language imperative. Various issues related to activity modeling have been outlined in Section 5.5, where the proposal has been evaluated with respect to the complexity of real life models, and the limitations of the script based formalism. This section concentrates on future research issues.

Future Research on Activity Modeling

With arbitrarily complex scripts, the disadvantages of a purely procedural language become more conspicuous. The exact sequences of steps have to be defined by the IPSE designer, and all possible exceptions have to be handled. The flexibility offered by a planning approach to task support (e.g., in office systems such as [31]) is undeniable. The modularity offered by the representation of activities as typed objects and their organization into subactivity hierarchies, needs to be combined with the planning approach. An ideal combination of paradigms would involve replacing the procedural script language with operators with preconditions and postconditions, where plan recognition can be used to recognize missing steps and perform the actions automatically if possible. Appropriate preconditions can
be used to represent nondeterminism (as is currently represented by the “select” statement (see Section 5.3.3)). However, the combination of message send and receive operations with a planning model requires further investigation.

The complexity of the proposed activity model is caused by the close interaction that typically occurs between activities, subactivities, and sibling activities, with messages being exchanged. This also severely restricts the degree of manual manipulation of the activity model that can be permitted outside of the control of the script. For example, if a script is waiting for a single subactivity to complete, and the subactivity is manually terminated, the wait statement raises an exception in accordance with semantics that is typical of CSP like languages [59]. This is illustrated in the example of Section 5.5.5. Whether plan based approaches can offer greater flexibility in specifying control flows needs to be investigated further.

In the proposed language, messages to activities can only be sent by other activities, or by the user. This is not general enough. For example, the completion of a module modification step could be synonymous with its first successful compilation, which can be detected by the compiler tool, or even by a trigger that reacts to an attribute modification by the compiler. Therefore, providing message sending capabilities to database triggers and tools seems to be a useful extension. However, there is a major software engineering issue that needs to be considered here. In general, a single database object can usually be manipulated in the context of more than one activity. Since the notion of a context is not related to scope
rules (database objects are typically global), whether a message from a trigger to an activity is valid cannot be statically checked. The same is true of messages from tools.

In an earlier version of the language, execution of subactivities was controlled by preconditions that were limited to logical expressions on state variables of parents, even though the internal scripts were procedural (see [110]). In that model, activities were explicitly created (as in the current model), but activity initiation was controlled by preconditions. This was abandoned in the current version because:

- Making the parent script procedurally control the execution of its children did not require the introduction of additional concepts such as preconditions, and kept the activity manager simple.

- The combination of a precondition based mechanism to control the initiation of activity execution did not mesh well with the use of a message based approach for synchronizing script executions.

There is an advantage to the precondition based approach, if the scope of preconditions is extended to permit conditions on any database object. For example, a script could be designed to “interpret” a PERT chart, and schedule activities in accordance with the constraints expressed in the network. With an imperative language, this would be simple enough though tedious. The script would iter-
ate through the network, creating and initiating activities whose predecessors are complete (or which do not have predecessors), waiting for each such activity to complete, at which time, preconditions of successor activities are checked. One could imagine an alternative representation of this logic with preconditions of activity types, expressed in terms of attributes of the nodes of the PERT chart object that represent instances of such activity types. When an actual instance of such an activity type is created, the PERT chart node is passed as a parameter, which is used to evaluate the precondition. The issues that need to be addressed to allow efficient implementation of such preconditions, expressed in terms of database objects, are similar to those raised by more complex preconditions of triggers. The application of multiple query optimization techniques to solve this problem has been addressed in the HIPAC project [47].

7.3.4 The Tool Model

Language features for representing operations of external tools have been presented in Chapter IV. The proposal concentrates on existing file based tools, does not consider composite tools (i.e., a tool that consists of other subtools), and only briefly mentions the extensions necessary to represent relationships between tools to incorporate "intelligent" tool invocation facilities. Most of these issues are being addressed in a separate dissertation [140].
Research Issues in Tool Integration

The most important requirement of a database approach to tool integration is a view description facility, using which, a new tool can be provided with a restricted view of the underlying object base. Such views become part of the schema (or meta information) and can be shared by more than one tool. View descriptions are not part of the process program, since views are not used by the activity descriptions. However, the tool manager component of the virtual machine (which is nonexistent in the current proposal where only file based tools are considered) does need to have access to the meta information that relates tools to the views they use. In the proposal in Chapter IV, the envelope extracts the required view for the tool and therefore, the “view description” does becomes a part of the process program.

Besides addressing the design of a view definition language, [140] also addresses the representational for describing complex tools (called cotools). This represents a fundamentally different approach to integration in contrast to the architecture of Odin [29] and Marvel [68], where the complex tool explicitly coordinates the invocation of its component tools instead of depending on a generic relationship interpreter. The reader is referred to [140] for further details.

The issue of “intelligent” tool invocation needs to be addressed further. The primary requirement is the development of language primitives for representing relationships between tools, either directly, as in Toolpack [98] and Odin, or via pre and post conditions, as in Marvel and Agora [18]. The essence of the approach
is a backward chaining inferencing mechanism (though Odin doesn't describe the paradigm in those terms), that incrementally invokes tools in response to user requests. Agora uses a full fledged planning approach to tool invocation, and is able to handle situations that Make [48] cannot because its understanding of inconsistency is limited to "time of last change".

7.3.5 The User Model

Most of the work on role definitions presented in Chapter IV is very preliminary. Lack of expressiveness is not the major issue here; efficiency of implementation is. Minsky's send-receive transport model [85] allows complex role specific access control schemes to be specified in terms of preconditions of actions. Sandhu's variation of this model, the schematic send receive (SSR) model [119], is simpler since it is type (and not predicate) based, and is also amenable to more efficient implementations.

The major limitation of the model proposed in Chapter IV is its lack of control over transport of privileges. Thus, whoever has the CONTROL_ACL right for an object can give away all rights to that object to any other role player. The SSR model allows more fine granularity control over which role type can give what kind of access rights to which other role type, by defining the equivalent of typed channels categorized by sender and receiver types. In addition, at use time, the sender and receiver must have capabilities for sending and receiving such
access rights. The SSR model is capability based. Whether an access control list-based mechanism can be extended to incorporate such features needs to be further investigated.

The modeling of group hierarchies, and the traversal of such a hierarchy at runtime to determine the correctness of an access, is a capability that has been proposed in systems other than this one (e.g., PCTE+ [21], CAIS+ [88]). While it is a powerful mechanism, runtime efficiency of its implementation is a major issue that has not been tested.

The proposed model does not handle executable objects (e.g., objects of type "load module"). Since the objective was to design a data model, and not a virtual operating system (which both PCTE+ and CAIS+ are), this omission is obvious.

The access control mechanism does not allow the specification of restrictions that limit access to certain object types only through the invocation of certain tools. The ability to define different views for users (via the object editor) and tools (via envelopes) does provide certain limits on access. However, what is required is a more general capability where tool objects (in addition to role objects) can appear in access control lists of data objects, forcing access to go through the appropriate tools. This is a different approach to incorporating an abstract data type facility on a "conventional" database, and has been adopted in both PCTE+ and CAIS+.
7.4 Conclusions

This research was based on the hypothesis that a software environment architecture should be designed as an interpreter or execution mechanism for an explicit representation of the software processes that are to be followed in a project in which the environment is to be used. The goal of the thesis was to substantiate this hypothesis by designing a language for modeling software processes, by representing examples of realistic software processes to demonstrate the adequacy of the language, and by designing and implementing a prototype runtime system for the language.

This dissertation has addressed each of the goals successfully by focussing on the representation of policies for multiuser coordination and control, while ignoring the more creative aspects of software development such as the processes by which people actually do design. While much more work needs to be done in the development of more high level languages for process programming, and the development of sophisticated programming environments for such languages, this initial effort has served to highlight both the benefits and the difficulties of the approach.
Bibliography


