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  800/521-0600
Integrated tool support for object-based environments

Venugopal, Vasudevan, Ph.D.
The Ohio State University, 1990
INTEGRATED TOOL SUPPORT FOR OBJECT BASED ENVIRONMENTS

DISSERTATION

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

By

Vasudevan Venugopal, B.S., M.S.

The Ohio State University

1990

Dissertation Committee

Prof. Neelam Soundararajan
Prof. Jayashree Ramanathan
Prof. Ponnuswamy Sadayappan

Approved by

N. Soundararajan
Adviser

Department of Computer and Information Science
ACKNOWLEDGMENTS

I would like to thank my advisor, Jay Ramanathan for all the technical and personal support she provided. Her insights and feedback were largely responsible for most of the ideas in this thesis, while her patience, indulgence and persuasion ensured that I finished what I had started. Neelam Soundarajan helped me out as my official advisor during the last year of my thesis. I am eternally grateful to him for that and for tolerating the imprecision of a systems thesis. Thanks to Ponnuswamy Sadayappan for being on my committee and also for teaching me what I know about Computer Architecture.

Thanks to Ronnie Sarkar for being the benevolent senior student. Thanks to Venkat Ashok, Bill Hochstettler and the other folks at UES for providing a wacky "real" world outside the academic confines. Special thanks to Priya, Rajiv and Sunil for the technobabble, the idle gossip, junk food and good times.

I would like to thank Prof. B. Chandarasekharan for teaching me all about Artificial Intelligence. His perspicuity and insight has always been and will always be an inspiration to me.

Special thanks to my mother for being there when I needed her.
VITA

June, 1962 .............. Born, Madras, India.

July, 1984 ............ B. Tech, Electrical Engineering,
Indian Institute of Technology, New Delhi, India.


PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science, Software Engineering.

Minor Fields: Artificial Intelligence, Computer Architecture.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS ................................................................. ii

VITA ........................................................................................... iii

LIST OF FIGURES ................................................................. vii

CHAPTER 

<table>
<thead>
<tr>
<th>I</th>
<th>Introduction .................................................................................. 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.1 The Integration Problem in Environments .................. 1</td>
</tr>
<tr>
<td></td>
<td>1.2 The Tool integration Problem .................................... 9</td>
</tr>
<tr>
<td></td>
<td>1.3 Tool Integration:Existing Approach and Changing Trends .. 10</td>
</tr>
<tr>
<td></td>
<td>1.3.1 The Toolkit Approach ........................................... 10</td>
</tr>
<tr>
<td></td>
<td>1.3.2 Implicit Assumptions of the Toolkit Approach .......... 16</td>
</tr>
<tr>
<td></td>
<td>1.3.3 Rationale for a move towards a Project Repository .... 19</td>
</tr>
<tr>
<td></td>
<td>1.4 Integrated Tool support in Object-Based Environments .. 22</td>
</tr>
<tr>
<td>II</td>
<td>Specifying a single Tool .................................................. 30</td>
</tr>
<tr>
<td></td>
<td>2.1 Introduction .................................................................. 30</td>
</tr>
<tr>
<td></td>
<td>2.2 Data Model .................................................................. 32</td>
</tr>
<tr>
<td></td>
<td>2.3 Aspects of the Tool Modelling Language ................... 34</td>
</tr>
<tr>
<td></td>
<td>2.3.1 Specifying the tool Interface .................................. 36</td>
</tr>
<tr>
<td></td>
<td>2.3.2 Specifying Tool Implementations ......................... 38</td>
</tr>
<tr>
<td></td>
<td>2.4 Views ......................................................................... 39</td>
</tr>
<tr>
<td></td>
<td>2.4.1 Theory .................................................................... 42</td>
</tr>
<tr>
<td></td>
<td>2.4.2 Language Primitives ............................................ 48</td>
</tr>
<tr>
<td></td>
<td>2.5 Facets ......................................................................... 59</td>
</tr>
<tr>
<td></td>
<td>2.5.1 Active/Passive tools ............................................ 59</td>
</tr>
<tr>
<td></td>
<td>2.5.2 Tools and Transactions ........................................ 61</td>
</tr>
<tr>
<td>Chapter</td>
<td>Section</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>2.6</td>
<td>Examples</td>
</tr>
<tr>
<td>2.7</td>
<td>Model for Display and Interaction</td>
</tr>
<tr>
<td>III</td>
<td>Tool Composition</td>
</tr>
<tr>
<td>3.1</td>
<td>Cotools</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Motivation and Abstract Model</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Comparison with other Ideas</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Extensions to the idea</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Motley subjects</td>
</tr>
<tr>
<td>3.2</td>
<td>Nested Tools</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Notation and Definitions</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Call and Split-Join disciplines</td>
</tr>
<tr>
<td>3.2.3</td>
<td>A Two Phase Checkout Protocol for Split-Joins</td>
</tr>
<tr>
<td>3.3</td>
<td>Language Primitives</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Cotools</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Nested Tools</td>
</tr>
<tr>
<td>3.4</td>
<td>An Example</td>
</tr>
<tr>
<td>IV</td>
<td>Tool Management</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
</tr>
<tr>
<td>4.2</td>
<td>A Birds Eye View of Planning</td>
</tr>
<tr>
<td>4.3</td>
<td>Tools and Plans</td>
</tr>
<tr>
<td>4.4</td>
<td>The simple planning problem</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Basic Steps</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Operator Selection</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Constraint Satisfaction</td>
</tr>
<tr>
<td>4.5</td>
<td>Language Primitive</td>
</tr>
<tr>
<td>4.6</td>
<td>Example</td>
</tr>
<tr>
<td>4.7</td>
<td>Extensions</td>
</tr>
<tr>
<td>4.8</td>
<td>Related Ideas</td>
</tr>
<tr>
<td>V</td>
<td>Implementation</td>
</tr>
<tr>
<td>5.1</td>
<td>Compiling tool specifications</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Tools</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Issues in the implementation of views</td>
</tr>
<tr>
<td>5.2</td>
<td>Implementation issues in Tool Composition</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1  Pipes, Super-Pipes and Ports .......................................................... 12
2  Tool modelling in file-oriented and object-based environments .... 24
3  Tool composition in file-based and object-based environments .... 27
4  Tool Management in file-based and object-based environments .... 28
5  Tools in file-oriented and database-oriented environments ........ 37
6  A complex object DAG and its rearrangements ....................... 41
7  Alternative ways of implementing a tool ...................................... 60
8  An environment for module programming .................................. 66
9  Alternatives for modelling interaction ....................................... 78
10  An abstract model of Cotools ...................................................... 86
11  Cotools in terms of more familiar ideas .................................... 88
12  Pipes and tools in the testing process ....................................... 92
13  Constraints on a unit tester view .............................................. 121
14  Constraint Satisfaction Process for "Used" Modules ................. 122
15  Categories of class constraints ................................................. 125
16  User strategies and their use ................................................... 127
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>The Tool Coordinator</td>
<td>133</td>
</tr>
<tr>
<td>18</td>
<td>A schematics of the Tool Compiler</td>
<td>136</td>
</tr>
<tr>
<td>19</td>
<td>Data structures in the tool compilation process</td>
<td>137</td>
</tr>
<tr>
<td>20</td>
<td>PseudoCode for a cotool, the basic model</td>
<td>145</td>
</tr>
<tr>
<td>21</td>
<td>System calls for backward reasoning</td>
<td>149</td>
</tr>
<tr>
<td>22</td>
<td>Modified view generation process</td>
<td>150</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

1.1 The Integration Problem in Environments

Integration [15, 21, 62, 49] is a fundamental and thorny problem in software environments. If environments are thought of as collections of cooperating tools, then the goal of integration is to provide a seamless look and feel to the environment. Seamlessness could manifest itself in terms of data, control, interface, or overall behaviour of the environment. In fact, the integration problem can be broken into several categories based on the particular kind of seamlessness promoted by that category of integration amongst environment components. The complexity of achieving integration in Software Environments stems from the complex, structured nature of the software development activity, the multiplicity of users and tools involved, and the information-rich nature of the software development process [63, 31, 57, 54, 8].

Integrated support in environments can be thought of as the overall goal of every environment, and the fundamental distinction between environments and collections of isolated tools. Tool Integration will be introduced as a sub-problem of the integration problem in the next section. First, the integration problem is
elaborated upon by an enumeration of its components:

- **Interface Integration**: One aspect of an integrated toolset is to present a uniform interface to the user, regardless of the service provided by the tool. There is a tradeoff between the level of interface integration and environment extensibility. Structure-editor environments such as GNOME [27], AGAVE [51] and MENTOR [19] provide a uniform interface by a single “unparse scheme” [26]. But they do this by means of a single internal object representation (trees), thus compromising extensibility. AGAVE also limits the scope of the environment by imposing a single conceptual model of activities (editing). On the other hand, window management toolkits such as SUNVIEW [79] and X [69] provide useful, general purpose graphical primitives without attaching any semantic interpretation to the resulting depictions. Environmental display integration schemes such as CHIRON [87] have to not only propose a depiction vocabulary suited to the domain, but also have to tie depiction with the manipulation of the object(s) being depicted. Typically, most such schemes separate what is being presented (model) from the actual depiction (view). In the pioneering SmallTalk system, a “controller” maintains consistency between the depiction and the underlying object. The model-view-controller (or MVC) paradigm [28] is a very general principle, and has spawned numerous specific approaches such as DESCARTES [70], and BALSA [9]. It is also important to note that the efficiency require-
ments for display schemes in SDEs are almost as stringent as in simulation and algorithm animation [9], as that is exactly the sort of assistance that SDEs would provide in the coding phase of the software development process.

In general, an interface-integrated environment provides an expressive set of rendering abstractions to the tool builder, and yet a uniform look to the user interface of tool users.

- **Data Integration**: Data integration promotes seamlessness in the environment with respect to data. This means that different tools should be able to operate on each others data and should have a common language for expressing the structure of data. The simplest form of data integration is *data interoperability*, or the ability of one tool to access the data of another. Even this simple form of integration is often missing in collection of isolated tools because the tools may have *private databases* for storing data that inaccessible to other tools. Design tools such as PSL/PSA [80] are a case to point. Beyond this, one needs to allow sets of tools to cooperate in the creation of project data objects and collectively maintain constraints on the aggregate data. For example, a data flow diagram produced by a data flow editor and a requirements document created in a document preparation system should be maintainable by an environment so that *traceability* is maintained between
the two. In other words, a data flow diagram is invalidated if its associated requirements document is changed or becomes obsolete due to a change in customer requirements. Traceability is a big issue in environments and maintenance of traceability is one of the primary goals of the next generation of environment repositories. An added aspect of data integration is the ability of the environment to support a common model of versioning, configuration management and other properties of data that are orthogonal to the structure of data itself. Since it is often impractical to impose a single canonical form of data in the environment, it is customary to provide high-level specification mechanisms for data translation across data structures [76, 50, 61].

- Control Integration: Control integration deals with the capacity amongst environmental tools to exchange data and services. The data exchange being referred to here is of a fine granularity and typically amongst concurrently operating tools. UNIX files can be thought of as providing data integration by providing a common model of all data as character streams. In contrast, UNIX pipes provide control integration because they imply a control discipline (concurrency, incrementality) amongst the tools connected by a pipe in addition to the data transport semantics. Another kind of control integration involves the ability of tools to invoke services of other tools. This seems like a trivial "procedure call" facility, but is often complicated by interactions with other features of tools (such as atomicity) and by features of
environments (such as distribution and heterogeneity). While the semantics of nested procedures may be trivial, the semantics of nested atomic actions are far from trivial. The overall effect of control integration is to provide the ability in the environment to build new tools by composing existing tools.

Software environments have taken a variety of approaches to the problem of control integration. "Do-what-you-wish" environments such as UNIX [37] provide weak models of tools with a high degree of composability. That is to say, a tool is just a program to UNIX. However, any form of reuse and composition that a program can go through, also holds for a tool in UNIX. Attitudes in programming environments, on the other hand, range from the "closed-door-approach" (i.e., these are all the tools you get and don't you dare ask for more), to the "leave-it-in-the-shop" approach (i.e., it can be done, but leave it to the experts, i.e., the environment designers). Of course, closed programming environments [81, 82, 25] do provide an extremely well thought out, well knit set of tools. However, one has to rely on the foresight of the environment designer for a rich tool set.

The tool fragment approach bridges the gap between closed (no new tools) and open environments (extensible toolset) in an interesting way. It does not allow the addition of "primitive" tools (tool fragments), but allows new composites to be specified in terms of existing tool fragments. The basis of this is that there exists a set of primitive services that is used by all
tools, such as a parser, lexical analyser and so on. While this assumption is true in very specific classes of software (such as mathematical software), software development tools are typically large and the toolset varies considerably depending on the methodology being used for software development, the nature of the application domain and so on. Nevertheless, the assumption that the environment is composed of a set of small, highly reusable tool fragments is very attractive [56]. It permits one to think of an “open” environment as a “closed” set of building blocks (tool fragments) with an open set of tools (fragment composites). This in turn allows the tool support model to completely avoid the issue of tool modelling and concentrate entirely on tool composition.

*Process Integration:* Large-Scale software development is typically associated with a discipline or process [63, 31, 57, 54, 8]. Environmental support for the software process guarantees a certain harmony amongst users of the environment in that it prevents users and tools from creating and manipulating work products of the software development cycle in haphazard ways. Process integration includes ensuring that tools conform to the data model (such as DoD-2167), so that only legal data is produced. It also implies control over tools based on process policies. This ensures that a coder cannot change a module while it is being reviewed or is part of some downstream activity such as testing or maintenance. This in turn allows different role players in the
environment to work cooperatively in a disciplined manner. For example, a tester can be assured in a process integrated environment that the modules he is testing or their test plan will not change on him while he is in the process of testing the module. The software process as a whole defines work products, software development operations, roles and role implications for various users in the software enterprise, and constraints and manipulation criteria for work products.

- **Plan Integration**: The *model* of tools in an environment is as a manipulator of environment objects. The *object model* in an environment defines the structure and consistency criteria for data objects in the environment. Plan integration combines the tool and object models of an environment to provide intelligent assistance to environment users. Plan integration [24, 56, 35, 6] automates the creation of objects and the maintenance of their consistency by automatically invoking tools on objects that are inconsistent, or invoking tools to produce missing objects. Typically, the invocation of a single tool may not be sufficient to satisfy the constraint, and the environment might have to formulate a *tool plan* to do so. The creation of complex sequences of tool invocation (tool plans) is done by applying some *generic reasoning paradigm* to deduce the need to apply various tools in a certain coordinated manner. A plan-integrated tool environment understands all the implicit combinations of tool plans in addition to the explicit plans provided to it by
tool and environment builders.

From the tool users point of view, plan integration allows the user to fire tools at incorrect times on incomplete data. By finding ways and means of producing the right environment for the tool firing (in terms of producing the required data and meeting the required constraints), a plan integrated environment completes the user's tool plan.

From the tool builders point of view, plan integration obviates the need to specify connections between the tool that he is writing and other tools that are already present in the environment. Since various forms of plan dependencies are already encoded in the environment, the tool writer can assume that the runtime monitor of the environment will use these plans to complete the tool program in suitable ways.

Despite the impressiveness of the generic goals of plan integration, its benefits are limited by the vocabulary in which plan fragments are expressed and the inferencing capabilities of the environment's planner. Typical benefits of plan integration are limited to resembling that of the "make" facility on UNIX. Another major flaw with plan integration is that its uncontrolled, "opportunistic" processing makes it somewhat attractive in isolation but awfully hard to mesh with other forms of integration in a multi-model environment. In many ways, it is more suited to environments where plan
integration is the major component of the support provided by the environment.

The most compelling argument for integration is an efficiency argument. Without interface integration, users would take longer to learn the use of tools because they would have a tool-specific look and feel. Lack of data integration would lead to redundant data entry. Lack of traceability would lead to inconsistent environment states where information about entities may be missing because traceability was not maintained. Lack of process integration would lead to destructive interference between user tasks and so on.

1.2 The Tool integration Problem

The tool integration problem can be thought of as the integration problem as seen by a specific class of environment user, the tool builder. A tool builders task can be thought of as consisting of:

- **Tool Modelling**: This refers to the task of representing and implementing a single tool in the environment. The facilities provided for tool modelling by the environment enable the tool builder to: use abstraction mechanisms to separate tool interfaces from tool implementations, specify tool data requirements and data transformation mechanisms for inter-tool data exchange, and specify overall properties of a tool such as concurrency and recovery.
• **Tool Composition:** Tool composition facilities allow the tool builder to reuse existing or already specified tools in specifying new *composite* tools. Composites may be cooperative composites which allow a set of existing tools to operate as a cooperative environment, or control composites, where a tool has invocations to services of other tools embedded within it.

• **Tool Management:** This feature is almost identical to what has been described in the previous section as plan integration. Except that while plan integration can be a database facility which is activated on object access, tool management is triggered on tool invocation by a user. Tool management deals with the use of tool and object models to come up with tool plans that correct an erroneous tool invocation by the user. A user may erroneously invoke a tool whose arguments are either missing or inconsistent in some way. Tool management generates tool plans to restore consistency and to generate missing objects.

1.3 Tool Integration: Existing Approach and Changing Trends

1.3.1 The Toolkit Approach

The two main approaches to tool integration in environments so far, have been the *Toolkit* approach and the *Integrated* approach. The integrated approach [19, 25, 52] treats the entire programming environment as one large monolithic program, with hardwired ways of glueing environment components together. The integrated
approach is very specific to the particular environment to which it is being applied. It relies on a great deal of knowledge about the nature of the components being glued together. It is thus unsuitable for SDEs which by their very nature are highly open and extensible. The basic philosophy of the toolkit approach is to think of the environment as being made up of a large number of small tools that can be coupled together in various ways. Toolkit approaches treat tools as building blocks and provide generic "glues" independent of the size and shape of the building blocks. The toolkit approach is consistent with the openness and extensibility goals of SDEs. Since the vocabulary of the toolkit philosophy is so general, it has lent itself to numerous interpretations. Some of these are described below and illustrated in Figure 1. However, the toolkit approach makes certain unstated assumptions about the organization, architecture and goals of environments which do not hold for SDEs. These are described in section 1.3.2 and the reasons for their inappropriateness is discussed in the section 1.4.

- **UNIX Pipes**: UNIX treats programs and tools as equivalent. Pipes [37] are a mechanism to allow concurrent, incremental operation of UNIX tools. Each tool in UNIX is modelled as having two specialised ports, one for input and the other for output. Pipes connect the input and output ports of tools to other tools, rather than the terminal, thus allowing linear configurations of tools to be created. The greatest merit of pipes is the minimality and generality of the integration model, and the orthogonality of integration and
UNIX pipes

MUPE-2 extended pipe

Port Based Specifications

Figure 1: Pipes, Super-Pipes and Ports
specification. Minimality is attained by the extremely general definition of tools. Also, the composition mechanism reduces data to its most rudimentary form, a *character stream*. Thus, the resulting data integration is very general since data of any form can be converted into a character stream (and back). Since the UNIX features for specifying and composing tools are orthogonal, tool code need not be written with composition plans in mind. This also makes the tool highly reusable.

These very features are also responsible for the weaknesses of the pipe mechanism, namely that:

- Assuming a two-ported model of tools allows only those forms of tool composition that result in linear tool configurations. This is an extremely restrictive form of control integration.

- The low-level and weak typing of inter-tool communication is inefficient and error-prone. Its inefficient because tools have to convert data from language structures to streams and vice-versa. Weak typing in pipes allows for error-prone composition schemes with type mismatches.

- Pipes handle data integration to the extent that they provide a common standard for data transfer. However, they do not help the tool writer in tackling the data transformation problem.
— Pipes can only model clear producer-consumer relationships between tools in a configuration.

— Only one instance of a tool type can participate in a pipe configuration. In other words, the same tool cannot appear more than once in a tool configuration.

— Even within UNIX, pipes work only for stream-oriented tools.

• Enhanced Pipes Enhanced pipe schemes are a class of schemes meaning to eliminate some of the deficiencies of pipes while retaining their advantages. A principal contributor to this category is MUPE-2 [43]. Pipes in MUPE-2 carry input groups of input fragments which are processed by tools into output groups composed of output fragments. Both groups and fragments are treated as types in the MUPE-2 environment. Thus, the tool specification is strongly typed in terms of the groups it can operate on and the fragment transformations that it affects. MUPE-2 allows the incremental operation of tools on group streams in the same manner as UNIX tools work incrementally on character streams. In addition, operations in the MUPE-2 pipes may be repeated.

The limitation of the MUPE-2 pipe is that tools are restricted to be structural transformers. Thus the communication mechanism is strictly specified at the expense of a restricted tool definition.
Plug-Oriented Approaches} Plug-oriented approaches [76] obtain the benefits of both the above approaches by separating the tool model into a process invariant and a number of port specifications. The process invariant represents the computation performed by the tool and can be any program. The port specification is a declarative specification of the type of information flowing through that port. The direction of information flow depends on whether the port is an input or an output port. Port-oriented tool models obtain control integration by allowing tools to "plug" into each other by port-to-port connections. The environment integrity is enforced by utilities that ensure that connections connect ports of the same type. Port-based tool configurations can be multi-dimensional, since a tool can plug into a number of tools (the connectivity of a tool is bounded only by the number of ports it has). Moreover, the communication is in terms of structures, not characters. By making a distinction between invariant and port in the tool model, the model is made extensible to handle data transformation. One could include data transformation abstractions between any input port and the tool invariant data structure, or between the tool invariant data structure and an output port. Thus, the modified tool language would have a data transformation sub-language, a process invariant specification sub-language, and a "wiring" (inter-port connection) sub-language.
The plug-oriented approach is the most complete form of the toolkit approach. Its deficiencies lie in the untruth of the assumption that it makes (in common with all other toolkit approaches).

1.3.2 Implicit Assumptions of the Toolkit Approach

Several architectural characteristics have been so common to environments that they have been assumed to be true and self-evident. Worse yet, many existing solutions to the tool integration problem have assumed these characteristics to hold without ever stating them, thus lending a false universality to their solutions. Some of these assumptions are stated below. Section 1.4 elaborates on how some of these assumptions have been violated in SDEs, thus requiring a re-examination of the tool integration problem.

- **Process-Based nature of the Approach** Most recent approaches to environments can be categorised as process-based. The essence of the process-based approach is to concentrate on agents in the environment (tools/users) as processes that manipulate data. The process-based approach emphasises the tool/user process and de-emphasises data. Thus, the facilities in process-based environments are towards facilitating the writing of processes. Data access policies are considered to be private to each process that manipulates

---

1There is an unfortunate clash of terminology between the software-methodology connotation of process-based, and the meaning given to the term in this section. The methodology connotation will be assumed to hold throughout the thesis, save this section.
the environment. The data repository is therefore considered a passive entity that does not enforce any data integrity policies beyond what is required by the operating system. Thus, while there is an emphasis on being able to create and manipulate the data that environment processes need, the data has no semantics of its own, and is interpreted in any way that the environment process considers appropriate.

The justification for adopting such a policy is as follows:

- Data of a certain type belongs predominantly to a single process. It is therefore not inefficient to let the process interpret it.

- Data manipulation policies, such as garbage collection, are often much more difficult to implement generically. On the other hand, a tool that manipulates a certain kind of data has much deeper knowledge about whether a certain data item is needed in the future, or can be garbage-collected.

- **Power Tools for Power Programmers** The basic goal of most programming environments, and SDEs of the past has been to provide the programmer an enhanced, more powerful toolset with no accompanying restrictions [71]. The underlying assumption being that what programmers need is smarter and smarter tools, and that any control exercised over the use of these tools is detrimental to programming productivity since it hinders the programmer's
creativity.

- **Environment = Tool + data** This relates to the previous point about power tools. Most previous environments limit their notion of support to tools and data. This meshes with their philosophy of non-intervention as stated above.

- **Quasi-Independent Users** Most existing environments underplay the need for coordinating the concurrent use of the environment by several users. They do this in varying degrees, from denying the existence of multiple users, to the optimistic assumption that multiple users typically have separate domains of manipulation in the database. Their model of multi-user activity is one that is *serializable*. They assume a quasi-independence amongst concurrent users which makes it unnecessary to explicitly support multi-user policies.

Again, this stand has some justifications in domains such as programming and CAD. These environments are primarily meant for supporting single-user activity. While multiple users might use the environment simultaneously, their activities are not sufficiently tightly coupled to warrant multi-user models for cooperative work. Their activities get serialised in an adhoc manner without environmental control.

- **The size assumption** Since tasks such as programming are extremely creative and knowledge-intensive, it is reasonable to assume that large amounts of factual information are not being created. As a consequence, while program-
ming environments need to support complex data types, they do not need to support large amounts of information. By database standards, the size of the data space supported by PEs is rather small. Consequently, main-memory management techniques can be used for data management in PEs. This is also a justification for adopting a process perspective towards environment design.

1.3.3 Rationale for a move towards a Project Repository

The "information-management" perspective in SDEs has led to a revamping of environmental cliches. Some of the more significant changes are listed below. Many of the assumptions of the toolkit approach have been based on the idea that SDEs largely support the coding activity and largely the activity of a single user. Many of these assumptions are violated in environments that need to support comprehensive, multi-user software development. These changes are explained below and lead to motivation for a single project repository with the facilities conventionally associated with commercial databases. The reason why this database needs to be an objectbase is subsequently dealt with.

- Constrained Environment for Tool Invocation: Programming environments and SDEs of the past emphasized on the philosophy of power tools for power programmers. A need to support process integration conflicts with this philosophy. Supporting the process implies that programmers need to be con-
strained in whatever manner is specified by the software process. Tracking and maintaining these constraints has been shown to be a task requiring non-trivial reasoning on the part of the environment [33, 34]. This means that a single repository is needed where constraints on data can be specified independent of the user or tool which created or manipulates the data.

• **Nature of Information Support** The variety and amount of data generated in SDEs introduces a modelling problem and a management problem. The size and complexity of the data space necessitates the use of a specialised module with capabilities for managing large amounts of data. It also implies that the data repository needs to support querying and indexing facilities to efficiently and associatively access environment data.

The size assumption of the toolkit approach is violated by the fact a tool may need to query a potentially large information base to find the data it needs. All this data cannot fit into main memory, so secondary storage management and indexing facilities to store and query information in secondary memory need to be provided. The argument for a process-based approach for environments is weakened because data management policies in SDEs are too complex to be replicated in every environmental agent. The centralization of data also means that inter-agent communication has to change from being "channel-based" to "view-based". Channel-based communication depends
on the ability for agents to create private channels of communication in a pairwise manner. This could violate the policy enforcement goals of SDEs.

- Multi-User Coordination The quasi-independent user assumption is violated in SDEs. The support model in SDEs is inherently multi-user. Support policies regarding process steps (ex: design, coding ..) explicitly relate different users [63] who may be concurrently involved in activities. Serializing multi-user activities based on data dependencies would reduce the support model in SDEs to the "waterfall model", whose deficiencies are well chronicled. Fine-grained concurrency control policies of the kind provided by databases are required.

Thus, the project repository that an SDE needs possesses the secondary data storage and management, clustering, indexing and querying features traditionally associated with databases. Beyond this requirement, a software engineering database also needs to be based on a semantic data model [36] and needs to be active. Limitations of the relational data model and its inability to handle the data modelling requirements of SDE data motivate the need for a semantic data model. An active database is required to store procedures for deriving data within the database, and to maintain data related constraints automatically. The exact nature of a software knowledge-base is elaborated on in [18, 58, 50, 20, 30, 32, 48, 8].

\[2\] Since the object-oriented data model is widely accepted, and an object-oriented

\[3\] A knowledge-base is a database with embedded procedural information
database with triggers satisfies all the abovementioned requirements, it is not unreasonable to assume that the environment architecture is centered around an **active, object-oriented database**.

### 1.4 Integrated Tool support in Object-Based Environments

This thesis proposes a tool integration strategy that is more suited to object-based environments. What follows is a brief description of some features that a tool integration scheme needs to provide and their motivations from the SDE viewpoint.

- **Tool Abstractions**: The multi-user nature of the environment and the fact that it is database-centered affect the language primitives for representing tools. Several tools in SDEs such as resource schedulers and project management tools, need to be operations that are concurrently invokable by different users. Thus the conventional abstract data type primitive needs to be augmented by language primitives for specifying concurrency and scheduling. There is nothing fundamentally new about this, and any number of abstractions that are found in concurrent programming literature [2] would suffice. The choice dictated by our view of SDEs is described in chapter II.

- **View-Based Tool Mode**: Tools interface to the database rather than to each other in a database-centered environment. This implies that tools have to
use database primitives to access and store information. Typical database mechanisms include query languages for data retrieval and manipulation, and views as a way of deriving the tool schema from the project schema. However, there has been little work on the use of views in Object-Oriented Databases. This is partly due to the fact that a major motivation for views in typical database applications was due to the “non-semantic” data models (such as the relational model) used in databases. In effect, views were supposed to reorganise information that was stored in a distorted manner by the database due to integrity maintenance processes such as database normalization which were essential to machine-oriented database models.

However, the existence of semantic data models does not change the fact that tool schemas are not identical to the database schema, which is the essential motivating factor for views. A notion of Object-Oriented Views is introduced in chapter II. The facilities provided by object-oriented views, language primitives for defining them, and their role in tool modelling is also discussed in chapter II. A comparison between stream-based and view-based tool models is presented in figure 2.

- **Database-Oriented Tool Implementation Schemes:** The fact that SDEs are database-centered implies that tool specifications will now compile into database primitives (as against operating system primitives). Since the DBMS does
Theme:

#Tool model = Tool Abstraction + Stream Discipline  
#Stream Discipline = Stream Type + Stream Abstr. + Stream Mode(s)

Variations:
Tool Abstraction - Program, Editor, Type transformer
Stream Type - Character, object, trees, graphs
Stream Abstraction - Queue, Set, Multiset...
Stream Mode - Broadcast, Varicast...

---

File based (stream based)

Tool Model = (External) Tool Object + Database Views

---

View based tool model

Figure 2: Tool modelling in file-oriented and object-based environments
not completely shield the operating system, some operating system primitives will still filter into the tool code. However, the DBMS is a system whose architecture is significantly different from current day operating systems. Consequently, tool integration strategies for SDEs have to face a different set of problems.

It is notable that a DBMS does not necessarily make the tool integration problem any easier. The fact that the environment is database-centered solves some problems and introduces others. For most traditional tool integration languages, a major component of the tool language is the modelling component. The modelling sub-language is typically further split up into two components. The "internal" component allows the creation and manipulation of software development structures. The "external" component specifies the manner in which these structures are to be stored in files. The external model of data is used both to store data in and retrieve data from the file system. In object-oriented databases, the presence of strong modelling capabilities, an expressive query language, and a structured canonical form makes the inclusion of specialised modelling constructs in the tool language redundant. Unlike tools, however, database applications are not geared towards reuse and composition. Mapping tools to database applications is therefore far from easy. Databases provide no mechanism for naming and invoking database applications from other database applications. In addition,
the mapping of database applications to transactions, and the difficulty in specifying the semantics of nested transactions [47] makes the mapping of tools to applications a far more difficult issue.

- Event-Based Control Integration: The case has already been made for view-based integration due to the fact that policy enforcement is hard to implement if tools have private communication channels. In an active, database-oriented environment one can implement notification events that notify the tool of data changes of interest. Since all the tools interface to a single database, these events can play the role of channels. This form of control integration, and its coexistence with the more conventional schemes for nested tools are described in chapter III. A comparison between cooperative composition in file-based and object-based environments is presented in figure 3.

- Tool Management in an object-based environment: The change in the vocabulary for specifying a tool, and the finer granularity of data modelling lead to a greater scope of assisting the user through tool management. The changes in vocabulary are dealt with in Chapter IV. The most interesting aspect is that the finer granularity of data modelling leads to the need for radically more powerful reasoning facilities than backward chaining, which is the form of reasoning used by Make [24], a tool for tool management in a
Figure 3: Tool composition in file-based and object-based environments
Figure 4: Tool Management in file-based and object-based environments

Object Model - Files
Tool Model - Files->file
Consistency Criteria- Timestamps
Generic Reasoning - B.Chaining

Object Model- Obj/prop/attr.
Tool Model- Ops/Views /View constraints
Consistency Criteria-View Constraints
Generic Reasoning - B.Chaining+..
file-based environment (UNIX). The need and ability to express constraints on groups of object called *class constraints*, the ability to specify and use *computational preferences* of the user as *strategies*, and the consequent requirement not only for conflict resolution in backward chaining but for more powerful reasoning techniques such as *constraint posting* and *blackboard-based reasoning* is dealt with in chapter 4. A superficial comparison between the vocabulary of tool management in file-based and object-based environments is presented in Figure 4.
CHAPTER II
Specifying a single Tool

2.1 Introduction

Tools can be roughly described as collections of related operations. The operations are typically related by commonality of purpose, similar environment setup requirements, or the fact that these operations are typically performed in a tightly interleaved manner. Language issues in tools are related to issues faced both by conventional programming languages as well as database programming languages (DBPLs). Tools perform compute-intensive manipulation of complex main-memory structures. These concerns of modelling complex main-memory data structures and their manipulation are similar to the concerns of conventional programming languages. However, tools in a database-centered environment also perform data-intensive computations on persistent information in the database. In this sense, tools are like database applications. Persistent information in conventional environments is mapped to weakly typed, unstructured entities such as files. This typically creates a "semantic gap" between the semantics that the tool needs to associate with the information and the reality of persistent stores. DBPLs deal with
the issue of providing persistent storage of data with the data modelling capabilities of conventional programming languages. Of course, there are many storage, query processing and concurrency concerns in DBPLs that have no parallels in conventional programming languages. A comparison of the modelling concerns of programming languages and DBPLs is made in [7], and the unique characteristics of DBPLs are described in [4]. The concern in tool modelling is to marry a suitable language abstraction for tools with an adequate way of characterising tool operations on the database.

Rather than propose yet another data model and DBPL, we adopt the data model proposed in ORION [5] and the ORION-1sx data manipulation language as the substrate for the tool modelling scheme. This is not limiting in any way since the features of the data model used in this thesis are those found in most object-oriented data models, both in object-oriented languages such as SMALLTALK and in a number of proposed data modelling schemes. However, the constructs of the tool modelling language are meant to be extensions of the ORION-1sx data manipulation language, and are therefore heavily influenced by its syntax.

The data abstraction that we choose to represent tools roughly resembles "modules" of MODULA [85]. Like modules (and unlike conventional Abstract Data Types(ADTs)), tools do not define a "type". Also, tools are not purely operational abstractions, bundling together not only operations but also types and variables as part of the tool declaration. We do not make any firm commitments about the
(non-)instantiability of modules, thus differing slightly from typical module languages. Even the use of modules in DBPLs is not without precedent, RIGEL [65] being a case to point.

The structure of a tool module, however, is much more hardwired than modules. A tool module declaration consists of declarations for operations, facets, views, and properties. Section 2.2 is a brief digression to the vocabulary of the ORION data model, which is rather heavily used in this thesis. Section 2.3 describes the language primitives in some detail, and provides the context for discussions on the specifics of tool modelling. Section 5.1.2 describes the concepts underlying views and view-based programming. Section 2.5 describes some useful facets, and the motivation for representing them in the tool specification. Specifically, some important facets describe the concurrency and recovery alternatives for tools, and are described in some detail in section 2.5. Finally, section 2.7 motivates the need for a model of interaction for environments which is at a higher level than the conventional primitives provided by databases and operating systems. It outlines one possible approach that could be taken in this direction.

2.2 Data Model

This section is a brief review of pertinent aspects of the ORION data model, which has been adopted as the data model of the database for the purposes of this thesis. ORION is an object-oriented data model. In other words, all conceptual entities
are modelled as *objects*, which comprise of a private memory made up of the values of a collection of *attributes* or *instance variables*. The value of an attribute is itself an object. A primitive object, such as an integer, has no attributes. The behaviour of an object is encapsulated in *methods*. Methods consist of pieces of code that manipulate or return the state of an object. They are part of the object definition. Both methods and attributes are manipulated by sending *messages* to the object. In that sense, ORION is *message based*. Complex objects can contain other objects as attributes, ad infinitum. Objects that are attributes of a parent are said to be related to the parent by a PART-OF relationship.

All ORION objects are *instances* of one or more *classes*. Classes contain the method and attribute definitions that are applicable to their instances. In keeping with the uniform treatment of classes and instances, classes can have attributes and methods of their own, which are shared by all instances of the class. The terms *class attributes* and *class methods* are used to refer to the attributes and methods that belong solely to the class. For example, ORION provides a *make* method for every class which is used to create instances of that class. Classes are themselves grouped in to *class hierarchies* to reduce the specification and storage needs of classes. A class hierarchy is a hierarchy of classes in which an edge between a node and a child represents an IS-A relationship; that is, the child node is a *specialization* of the parent node (and conversely, the parent is a *generalization* of the child [74]). The parent is called a *superclass* of the child and the child a *subclass* of the
parent. The attributes and methods of a class (collectively called properties) are shared (inherited) by all its (direct and indirect) subclasses. Additional properties may be specified for each of the subclasses. In reality, the class hierarchy of ORION is a lattice. A class can have multiple superclasses, and the ORION data model provides rules for conflict resolution in case the properties inherited from multiple superclasses conflict with each other. Further details about the ORION data model can be obtained from [5].

2.3 Aspects of the Tool Modelling Language

A tool is similar to a MODULA module in the sense that it presents a set of classes, operations, constants and variables to its clients or invokers. Like modules, it provides the facility of information hiding in that the clients of the tool are privy to the interface without being able to influence or view the implementation. This also leads to greater flexibility in choosing and modifying implementations, if required. The module-like vocabulary of tools comprises of the terms tool, operations, classes and properties. A tool defines a set of operations that its clients can invoke and class definitions that have been made available to clients by being included in the interface specification. Properties are data that belong to a tool and are global to all operations of the tool. Tool properties represent the

1 The language primitives may offend the sensibilities of many purists. However, since we are proposing extensions to the ORION model, it is only appropriate that our constructs should be embeddable in, and an extension of the orion-lsx language which is the data manipulation language for ORION. Orion in turn is built on top of COMMONLISP and is meant to be seamlessly integrated to it. Thus the LISP like syntax.
state of the tool, in some sense.

The features that set tools apart from conventional modules are views and facets. These language features are a direct consequence of the database-oriented nature of tools. Views are a mechanism for specifying the interface between programs (tools) and the database, or in database parlance, specifying the external schema in terms of the conceptual schema. This isolates the tool writer from irrelevant complexities of the conceptual schema, and provides access control by filtering out inaccessible parts of the conceptual schema from the tool view. This concept of views as a virtual database for database programs is not completely suitable for tools, or for object-oriented environments, for that matter. It seems more suitable to think of the tool’s concept of the database as a set of view-objects. A view-object is an object like any other, except that it is derived from some object(s) in the underlying database. Each view-object encapsulates the tool’s conception of a complex network of objects in the database. The set of view-objects in the tool specification collectively represent the tool’s virtual database. A view-object specification is a type specification. The actual view-object instances are derived at runtime, based on the parameters passed to the tool. Thus views implement a form of information hiding in that the invokers of a tool operation need not be aware of all the components of a view. They just have to provide

---

2 A concept of the same name and somewhat similar goals was proposed in [84].

3 Since view-objects connote the same concept as views in other worlds, we will henceforth use the term View to mean view-objects.
enough information to the invoked tool for it to be able to derive the view object using the view specification, which turns out to be the identity of the root of the view. Figure refTV illustrates the analogies between a filename-oriented interface that tools have to the file system, and a view-oriented interface that tools see in database-centered environments. Section 5.1.2 deals with the conceptual and language issues in defining and using views. Facets describe some properties of the tool that are orthogonal to the client interface provided by the tool, such as concurrency and recovery disciplines. Facet specifications are used by the tool compiler to map the tool to a suitable database implementation. While the list of tool facets is extensible, its interpretation and effects are hardwired into the tool compiler.

2.3.1 Specifying the tool Interface

Def-tool-Interface declares the components of the tool interface. The interface to a tool has four optional parts for declaring views, operations, facets and properties respectively. The interface description of a view includes the view name and the base object type of the root of a view. The meaning of these terms is explained in section 5.1.2. The interface specification for an operation specifies the operation name, arguments and their types, and the return type for the operation. It is important that the types of the arguments be views, as this is a basic tenet of view-oriented programming. Facets are specified as name value pairs, and tool
properties are declared in a manner similar to property declarations for ORION objects. The *initform* in the property declaration is a piece of lisp code that initializes the property.

```lisp
(def-tool-interface ToolName

    [:views ((ViewName RootBase)*)]

    [:Opns ((OpName ((Var ArgType)*) [Ret ArgType]*)*)]

    [:facets ((FacetName FacetVal)*)]

    [:properties ((PropName PropType [initform]*)*)]
)
```
2.3.2 Specifying Tool Implementations

Tool implementations are specified using the `def-tool-implementation` construct shown below. A tool implementation consists of view definitions followed by operation implementations (similar to procedure definitions in which declarations precede code). Constructs for defining views are explained in the section on views. Operations are specified using the `def-Opns` clause. The code for operations is like any old LISP code except that it manipulates views and properties in addition to the normal doings of LISP programs.

```
(def-tool-implementation ToolName

    [:Views ViewCode]

    [:Opns OpCode]

)
```

```
(def-Opn ToolName OpName

    [:Interface ((Var ArgType)* RetArgType)]

    [LambdaList])
```
2.4 Views

Views have traditionally been a mechanism for controlling access to a database in a multi-user environment [16, 78, 65, 72]. The issues in defining a view specification scheme for object-oriented environments hinge on specifying hitherto unspecified formalisms for object-oriented views, and on specifying the stances that this view specification language takes on certain traditional issues pertaining to views that transcend data models and domains of environment use.

The first point is non-trivial because most view definition schemes of the past have been geared towards machine-oriented data models such as the relational model. The nature of the theory of views is closely related to the nature of the theoretical underpinnings of the data model of the database. For example, view definition schemes for relational databases model views as relations. This allows the relational model to apply to views themselves. Thus a significant change in the data model, such as a move to the object-oriented model, suggests that views themselves need to be now couched in terms of objects. Since the object-oriented model is network-based in contrast to the set-based nature of the relational model, the theory of views in object-oriented environments needs some revision.

Fundamentally, view definition languages are transformational in nature. In other words, they specify not only the form of the view as seen by an environmental agent (user or tool), but also the algorithm to transform operations on the
view into operations on the underlying database. The traditional language issues in view definition languages pertain to the representational adequacy of the view definition component of the language, transformational ambiguity in the transformational component of the language, and an implementational efficiency issue of view materialisation versus view virtualisation. On the one hand, we want the view language to be arbitrarily powerful. However, while the language may be able to express all kinds of views, the view operations might not be transformable to operations on the objects from which the view was derived. This might either be because the transformational component of the view definition language is too weak to express complex transformations between views and the database, or because the transformational rules are "implicit" and there exist several ways of translating between a given view and the objects it was derived from. In many earlier databases, redundancy has been considered an evil to be avoided at all costs. Thus, replicating database information in "concrete" view objects is an implementation technique that has been repugnant to the "virtualists" who believe in treating views as virtual objects that are created on demand [78]. This is inefficient in situations where views are updated frequently, and this wisdom is of dubious value in active, object-oriented databases, where consistency maintenance between different forms of information is easily achieved by triggers.

These and other issues are dealt with in this section which describes the theory behind tool views, necessary language primitives and limitations.
**Vehicle part Hierarchy**

![Diagram of vehicle part hierarchy]

**Some Complex Object Views**

Figure 6: A complex object DAG and its rearrangements
2.4.1 Theory

This section proposes a theory of views that is suitable for the object-oriented model. One can model any complex network of objects (classes or instances) as directed-acyclic graphs (DAGs). Nodes of the DAG correspond to objects, while links correspond to relationships between objects. Two commonly occurring types of networks in OO environments are class lattices and complex objects. The former can be modelled as DAGS in which the nodes are classes and the links represent IS-A relationships. The latter can be modelled by complex object DAGs in which nodes are instances and links represent PART-OF relationships. Nodes are also associated with a color signifying the structure associated with a node. In other words, if a non-colored node represents the type “Person” colored versions of the node can represent a “set” or a “queue” of persons.

Views can be thought of the result of DAG transformations applied to DAGs representing some part of the underlying database. DAG transformation operations, hereafter referred to as DAG operations can be of two classes: node translations and DAG rearrangements. Depending on the kind of DAG being transformed, we have complex object views and class lattice views. Figure 6 presents examples of some views. Because of the rich semantics of IS-A and PART-OF relationships, and the tremendous disparity between their implications, the exact set of allowed DAG rearrangement operations and node translation operations is dependent on the DAG type. In fact, while the theory covers both classes of Dags, the seman-
tics of class hierarchy rearrangement is complex enough to warrant attention in and of itself. We concentrate here on complex object views, although a subsequent section speculates on the issues in, and utility of class lattice views.

Complex Object Views

As mentioned earlier, a complex object type can be described by a colored DAG, where the nodes represent types, links represent constituent relationships between types, and the color represents the "structure" (such as set-of, stack-of, ...) if any, associated with the type. Figure 6 shows a DAG representation of a complex object and some rearrangement views (uncolored nodes represent objects without an associated structure). While arbitrary rearrangements on complex object DAGs are possible, many such rearrangements do not retain the virtualizability property, i.e., queries on views are not translatable algorithmically into queries on base objects. We choose to limit ourselves to rearrangement operations which preserve this property. Queries on complex objects are assumed to be of the form shown below.

```
GET attributes of target record type WHERE predicate.
```

---

4It must be emphasised that a "virtualisable" theory of views does not mean that views will be virtual in our implementation. In fact, we choose to materialise views for the purposes of manipulation while translating queries on view objects by query modification into queries on base objects. This scheme has the limitation that the result of view-based queries cannot themselves be manipulated.
As it turns out, queries can be classified into categories based on the fact that they can be processed on the basis of a *traversal scheme*. Given a traversal scheme, there are certain rearrangement operations which one can apply to complex object DAGs without losing the property of virtualisability under that traversal scheme. We can then say that a complex object view is *processable* in a traversal scheme under the allowed operations. Presented below are two traversal schemes. Also listed are the rearrangement operations under which processable complex object views can be created from the original complex object DAG. What follows is summarised from Kim's thesis [38]. Proofs for the informal claims made above can be found there. Since the allowed node translation operations are independent of traversal schemes and such, they are presented following the discussion on traversal schemes.

- **Childpart-Only Traversal**: Under this scheme, if target attributes belong to a part P of a complex object, query predicates may only refer to attributes of *parents* of P, i.e. objects which are directly or indirectly have P as the domain of a property. Under this scheme, a DAG rearrangement is virtualisable iff the following rule is obeyed.

  Let $S$ and $S_d$ denote the schema and the DAG rearrangement view on $S$ respectively. $S_d$ is a *processable* view, iff for each part $P$ in $S_d$, the set of direct or indirect parents of $P$ in $S_d$ is a subset of corresponding
Childpart-only traversal allows the following rearrangement operations on DAGs.

- **Hide P**: Hiding a part P in a DAG attaches the children of P to the direct parents of P, thus eliminating P from the modified DAG.

- **Promote P to Q**: Promotion leads to P becoming a direct descendant of Q. However, the axiom of childpart-only traversal implies that P must have been an indirect descendant of Q in the first place.

- **Childpart-Parentpart Traversal**: In childpart-parentpart traversal, if the target attributes of a query belong to a part P, query predicates may contain only attributes of parentparts or childparts of P, or P itself. Under this scheme, a DAG rearrangement is virtualisable iff the following rule is obeyed.

Let $S$ and $S_d$ denote the original schema of a composite object and the DAG rearrangement view respectively. $S_d$ is a *processable* view, iff for each part P in $S_d$, $AD(P \text{ in } S)$ is a superset of $AD(P \text{ in } S_d)$. Where $AD$ is the union of the set of all ancestors of P and all its descendants and P itself.

This traversal scheme permits one more rearrangement operation in addition to the ones permitted by the Childpart-only traversal scheme.
- *Exchange P,P'*: The exchange operation allows two parts on the same branch to exchange positions. Two parts \( P \) and \( P' \) lie on the same branch iff \( \text{AD}(P) \) is either a subset or a superset of \( \text{AD}(P') \).

**Node Translation**

Node translation operations allow the nature of a node to be manipulated without any rearrangement to the DAG itself. The translation occurs between a "base node" (or a node in the underlying DAG) and a "view node" (a node in the complex object view). Node translation deals with deriving the structure of a view node from its "identity map" in the underlying DAG. Our scheme allows two translation operations, *attribute projection* and *attribute morphosis*. Attribute projection operations lead to the "projecting in" of the attributes mentioned in the projection specification, and the consequent "projecting out" of all other attributes of the underlying node. Sometimes, however, we may not only want to derive an object by selecting relevant parts of a base object, we may actually want to change the form of the derived object. This is carried out by "attribute morphosis". In terms of the DAG model, an attribute "morphosis" changes the color of a node in the derived graph. The issue of *morphosis* is much more complex than the attribute
projection operation. This is because, specifying, creating and maintaining correspondences between diverse structural abstractions (such as between a stack and a set) is a non-trivial problem. We introduce a language feature called *morphisms* in the next section that encapsulates such information, which just begins to attack the structural transformation problem.

**Class Hierarchy Views**

The DAG model does not force any restrictions on the nature of the objects being represented in the DAG. Thus, it applies equally to class lattices as well as complex objects. So, one can postulate "class views" that are rearrangements of the class lattice DAG. However, the added semantics that classes have beyond vanilla objects suggests that the uniformity is somewhat false. For example, classes have a "template" and an "extent" implication. If class A IS-A class B, then the template of class A includes the template of B, and the "extent" (i.e. the collection of all instances that belong to this class) of class B includes the extent of class A. That being the case, the exchange operator discussed for complex object views would make no sense for class lattices. This is because, while flipping PART-OF relationships can be justified, IS-A relationships have much deeper meanings, and implications of flipping IS-A’s are hard to understand.

However, the idea of class lattice views is very powerful. Classes of objects represent classes of information. Class lattice views can be used in role-based
environments to determine “who” can see certain classes. This is orthogonal to complex object views which then determine “what” someone can see of a class.

2.4.2 Language Primitives

Views

A Complex Object View (labelled by a view name) represents a “derived” network of objects. This derivation is limited to “rooted” networks, and therefore both the view and the underlying object network are rooted. The view specification construct shown below includes a specification of the class of the root base object (the root object of the underlying complex object network) as well as the class name of the root of the derived object network. The actual correspondence between derived objects in the view, and the base objects they map to, as well as the topology of the rearrangement of the underlying complex object network is specified by a derived class specifications.

(def-view ViewName
  :root RootDerivedType
  :base RootBaseType
  :notify TorNil
)

\footnote{In keeping with the COMMONLISP style, we liberally use “keyword” arguments in our constructs to avoid the need to supply arguments in a particular order. Keyword arguments are passed in the actual call by preceding the arguments by the matching keyword.}
Derived Classes

*Derived Classes* represent the specifications of components of a complex object view. An instance of a derived class is always derived from an instance of a specific *base class* (i.e., a class defined in the public schema, and thus global to all views). Thus, there is a *derivation map* between a derived class and some base class, as well as an *identity map* between instances of the derived class and exactly one instance of the base class which is specified by the derivation map. Thus, objects in a view cannot be "joins" (in the relational sense) of underlying objects. This is a small price to pay for avoiding the ambiguity caused by join-like operators in view definition schemes [16].

Presented below is the syntax for defining the derived classes associated with a view. The :view keyword is followed by the view for which this derived class specification holds. Views also serve the purpose of "scoping" derived class declarations. The name of a derived class need be unique only within a view. The DAG transformations carried out by the view are a little less clear from the language specification than they are from a visual presentation of the DAGS involved. This is because the each derived class specification can only refer to "local" rearrangements, i.e., derivations pertaining to the base type to which it corresponds and to
the direct descendants of that base class. The overall DAG rearrangement has to be specified in terms of a whole set of such local rearrangements. A derived class specification includes not only the local DAG rearrangement specifications but also the node translation procedure, i.e., a specification of how to obtain a node in the derived DAG from another node in the underlying DAG.

One can think of a derived class as consisting of derived properties and derived methods. Some of these properties may be themselves be objects, thus representing links in the DAG model while scalar properties represent aspects of nodes in the DAG model. Thus, property derivation in the language could map to either node translation or DAG pruning in the graph model. "Derivation" is used in a generic sense here, and refers to copying refinement or augmentation of base object specifications. The clauses :virtual, :synthesized and :own represent three different categories of derived properties. The :methods clause deals with derivations pertaining to methods. Both properties and methods are assumed to be absent by default. In other words, any property or method not mentioned in some derivation clause in the derived class specification is projected out of the view specification. The clauses :catch,:throw,and :allow specify the global DAG rearrangements that the view carries out. They collectively implement the aforementioned hide and promote operations. The complexity of these operations makes us hesitant to extend the language with capabilities for the exchange operation. The :constraints clause specifies inviolable constraints that must be obeyed in the view derivation
process. Violation of these constraints leads to the abortion of the view derivation process and thus to a "null" view.

Derived properties, derived methods, and details about the rearrangement operations are described later in this section, and in that order.

(def-derived-class DerivedClassName BaseClassName
            [:view ViewName]
            [:virtual VirtProplist]
            [:synthesised SynthPropList]
            [:own OwnPropList]
            [:methods ListofMethodSpecs]
            [:notify TorNil]
            [:catch TagLiteral PropName CatchMethod]
            [:throw TagLiteral]
            [:allow DerClassName PropName]
            [:constraints ConstraintList]
        )
Derived Properties

Virtual Properties

*Virtual properties* are properties in the derived class whose specifications are identical to a property of the same name in the base class. In fact, such properties may not even need to occupy storage in the derived object, thus the label "virtual". Since there is no specification needed for virtual properties, their names are collectively specified as a list in the :virtual clause.

Own Properties

Own properties are private to the derived class and are not "anchored" by any property in the derived class. A possible use for such properties is as placeholders for view-specific computations. All the own property specifications pertaining to a derived class are specified as a list in the :own clause. Each element of the list is of the form shown below. An own property specification is associated with a domain, specified by the :domain clause, and optionally an initial value and/or an initialisation procedure. These are specified by :init and :initform clauses respectively.

\[(PropName
 :domain DomainSpec
 [:init InitialValue] \]
Synthesized Properties and Morphisms

Synthesised properties are anchored in some property in the base object. However, unlike virtual properties, the specifications of synthesised properties are different in the derived class from the specifications of their anchors. Thus, a synthesised property specification includes a derivation specification in addition to domain and init specifications. Depending on whether the synthesised property is scalar or structured, one associates triggers or morphisms with the property so as to maintain consistency between the synthesised property and its anchor. If the property is itself object-oriented but not structured, then the derived class specification of the property itself takes care of anchoring the property. The workings of get and set triggers are quite similar to trigger behaviour in many data models, and the details are omitted here. However, morphisms are a relatively less familiar concept.

Morphisms are used when there is not only a derivation but also a structural transformation involved. Suppose we want to derive a "queue of SalaryView" from a property that denotes a "set of person". Here SalaryView is a view of a person object in which only the person name and salary are selected. There are two
transformations involved here. Firstly, for each person in the set, we need to extract a SalaryView object and anchor it in the underlying person object. Secondly, we need to create a "queue" from a "set". Even more important, we need to map queue operations such as enqueue and dequeue to set operations such as insert or delete. Note that the manipulations on any element of the queue are automatically mapped to its anchor by the instance maps between the derived instances and their base anchors. Morphisms are special objects that encapsulate the semantics of structure-to-structure transformations. In particular, when derived and base objects are bound to a morphism object (associated with a morphism type, of course), they can be thought of as clients of the morphism. Clients of a morphism send "snapshots" of any operation in the client to the morphism. The morphism object translates the snapshot into corresponding operations on the other client and invokes the operation on the other client to restore parity between the clients.

This scenario can be generalised to morphisms that can handle more than two clients. While the above description fixes the interface of the message handlers of all morphism objects, the actual translation has to be currently hardcoded. In other words, there is no high-level primitives to specify the operation-correspondence map between clients of a morphism. Needless to say, there can be many morphism types that translate between the same two structures. For example, many different morphism types might translate between "sets" and "queues" with different initialization semantics for the queue and a different operation-correspondence map.
between the set and the queue.

The language construct shown below illustrates the keyword arguments for each of the facilities discussed above. The :domain :structure and :init clauses specify the domain, structure, and initialisation behaviour to be associated with the synthesised property. The :get and :set clauses associate get and set triggers with scalar properties. In case the derived property is structured, one can associate a morphism with the property by using a :morphism clause. :Base and :basestruct clauses the domain and structure associated with the anchor property in the base class. This construct describes an element of the synthesised property list.

(PropName
  :domain DomainSpec
  :structure StructSpec
  :init InitialValue
  :get getTrigger
  :set setTrigger
  :base BaseDomainSpec
  :basestruct StructSpec
  :morphism MorphismSpec
)
Derived Methods Derived methods are of two kinds, virtual and own. Synthesised methods do not exist in the model because there is no way to confirm that a method is "synthesised" from an underlying method and yet related to it. Since methods are procedural code, synthesis relationships between base methods and derived methods are hard to detect and enforce. It should be noted, that one can conceive of synthesised methods in other data models which model message handlers more richly, such as the model provided by VBASE [3], which allows "pre" and "post" methods to be attached to method code. As before, virtual methods are identical to some method in the base type, while own methods are private to the derived class specification. The construct shown below serves to declare both own and virtual methods. The :own clause is used to introduce the interface specifications and code associated with the method.  

(def-derived-method MethodName

(Self DerClassName [Classp])

[:own (LambdaList Code)]

)

DAG Rearrangement

While the graph representation shows no sequentiality in the creation of the com-

---

6 The reader should consult [77] for many of the LISP-oriented aspects of the syntax, such as LambdaLists in the derived-method construct.
plex object view, the view is essentially created top-down. There are two options on how rearrangements can be affected in such a scenario. One can first determine the topology of the graph without rearrangements, and then perform all the rearrangements in one step. An alternative is to specify the rearrangement with the component which is the target of the rearrangement. The latter strategy is implemented by :catch and :throw\(^7\). Catch sets up tagged catchers. Catchers represent locations at which rearranged objects will subsequently attach themselves. The tag is a globally-unique (global to the view, that is) identifier for the particular attachment point. The :throw clause, if embedded in a derived class specification, implies that the object, after derivation, needs to be "thrown" at the tag specified in the throw clause. This causes attachment of the newly created object to the object which has a catcher for the throw tag. It is important that the throw succeeds the object derivation. The catch specification also includes a property name and a catch method. The object thrown with a particular tag is attached to the property mentioned in the catch clause for that tag. The catch method performs the actual attachment of the thrown object to the catcher property. The catch methods are automatically passed the thrown objects as arguments. Thus catch and throw implement the promote DAG operation and can be used to simulate the exchange if need be.

Often, objects performing a throw are parts of subtrees that are pruned at

\(^7\)The idea of catch and throw clauses is borrowed from the catch and throw constructs in COMMONLISP
a higher level. For example, in Figure 1a), the "engine", which is attached to "vehicle" after rearrangement, is actually part of a subtree in the original DAG, which is pruned in the rearrangement. To enable objects in pruned parts of the DAG to perform throws and such, we have the :allow clause. The :allow message can percolate down one node of a subDAG without actually causing derivations to occur in the nodes it flows through. Thus, the "drivetrain" can allow the derivation message to flow through it to the engine. The engine then gets translated into a view of the engine node and subsequently "thrown" to the vehicle. However, the allow message passed through the drivetrain object without causing any derivation.

Finally, the hide operation described in the theory is also not directly implemented. It can be simulated by having the object to be hidden to allow the derivation message to flow through it. The immediate subobjects of the hidden object should be programmed to throw their self-derivation to the parent of the hidden object (using the now familiar tag scheme). The reason for not creating a specialised "hide" construct is because, while it is easy to describe the object that is to be hidden, one has to describe the new attachment points for the children of the hidden object. This is exactly what is provided by the catch clause. In effect, a hide construct will end up looking very much like a catch clause, and the throws made by the hidden object's children will be implicit, and hard to pinpoint.
2.5 Facets

Since the separation between data and program is dimmed in object-oriented environments, a tool specification can be mapped in several ways to database primitives. The exact mapping is decided by hints in the tool specification which help determine a unique implementation for the tool. These hints are called facets. While the set of facets is extensible, the interpretation of facets is hardwired into the tool-compilation environment. We describe two facets here that pertain to the active/passive nature of tools and the mapping of tools to transactions. Figure 7 illustrates three ways of compiling a tool module into database entities.

2.5.1 Active/Passive tools

The active/passive nature of a tool is determined by the value of the InvocationDiscipline attribute of the tool. Active tools are uninstantiable and possess scheduling disciplines for handling requests from multiple agents of the environment. One could choose any one of the many available abstractions available in concurrent programming such as monitors, serializers, path expressions [2] and so on to represent the scheduling discipline. We choose a convention that each operation of the tool is independently scheduled and access to shared data by different operations of the same tool is handled by database mechanisms for concurrency control such as locks and transactions. This fixed policy would suffice for SDEs, accurately represents the implementation restrictions on active tools in a database-centered environ-
Figure 7: Alternative ways of implementing a tool
ment, and requires no specialised language primitives for representing scheduling disciplines.

2.5.2 Tools and Transactions

Transactions [29] have traditionally been the principal concept for describing the disciplined manipulation of data in a multi-user environment. The classic transaction mechanism is a highly overloaded construct [42]. It has three distinct purposes: as a logical unit for database programming, a unit of atomicity, and as a unit of recovery. Transactions are logical units in database programming in the sense that local state information can persist during the duration of a transaction and can be seen by all steps in the transaction. A transaction is atomic in that the steps of a transaction appear to other users/transactions to execute as a unit or not at all. Since this property ensures that concurrently executing transactions can be equivalently expressed as a sequential execution of the same transactions, this property is called the serializability property in database literature. The minimal set of commands offered in a transaction-based environment are commit and abort. Commit results in the successful completion of a transaction, causing its modifications to be made permanent and therefore visible to other transactions. Aborting of a transaction leads to the complete cancellation of its effects.

The short transaction concept described above is unduly restrictive for many domains, including tool modelling in CASE environments. An underlying premise
in the transaction serializability discipline of short transactions is that transactions will be short and conflicts resolved quickly. This requirement is unduly restrictive in situations where transactions need to hold locks for long durations of time. Such situations can occur either in a "conversational" application where user interaction is a long computation by database standards, or in cases where due to a combination of interactiveness and a compute-intensive nature of the application, transactions run for long periods of time. These are often called long-lived transactions (LLTs) or long transactions (LTs).

**LLTs: Themes and Schemes**

Almost all LLT schemes have sacrificed serialisability to obtain greater concurrency. The justification for this is that in many domains, the real-world semantics demands disciplined interleaving of data manipulation tasks, not complete serialisability. Two such approaches have been *negotiation based cooperation* and *programmed cooperation*. The former is rife in Computer-Supported Cooperative Work [67], where transactions are long due to their interactive, user-controlled nature. In that scenario, potential resource conflicts such as requests for locked objects, are handled by interactive negotiation between the lock holder and the requestor. *Reservations* and *floor-passing* in RTCAL [67], and *soft locks* in GORDION [22] can serve such a functionality.

This kind of flexible negotiation is not possible amongst database applications.
In such a situation, one has to adopt a more programmed strategy for access transfer of objects amongst cooperating applications. *Programmed cooperation* has been the dominant paradigm for LLTs in CASE and CAD/CAM domains where LLTs are created by large, compute-intensive, interactive applications. Three major LLT schemes for programmed cooperation are the Kim and Lorie scheme of hierarchical *semi-public* databases [40], the Dittrich scheme of “group databases” [17], and the scheme of Klahold et al [41]. They are based on the idea that the application possess “semi-public” or private databases. These are logical databases that are owned by a single application, as opposed to the *public* database which is shared amongst all applications. Desired objects are atomically *checked-out* from the public database into the semi-public database of the application by atomically copying the object from the public database. The objects in the semi-public database can thus be manipulated indefinitely if necessary, since the check-out mechanism is session-independent. Objects that are no longer needed are *checked-in* by applications to the public database. This leads to the copying back of the modified object to the public database. All three schemes are based on the *check-in/check-out* model outlined above. The distinction between these schemes is in the *sharability* of semi-public databases, and the nature of ordering enforced between cooperating transactions. The Kim and Lorie scheme is the strictest and most complex of the three. It insists on unique ownership of semi-public databases. Thus, two cooperating applications cannot share a semi-public database. One of them (say A)
must check-out the desired object(s) from the other transaction (say B). A is then "dependent on" and a "child" of B. In this manner, parent-child hierarchies (not lattices) of transactions are created, and must follow certain rules relating to the nesting of transactions. Check-in of objects is always to one's parent database, irrespective of where the check-in was from. It is the parent's duty to check-in not only its objects but also those of its children. Both Klahold and Dittrich allow the notion of groups and group transactions. Group databases hold information and data common to a cooperating team. Groups can be members of groups, leading to user-group lattices, rooted at the public database. There is the notion of a group administrator who decides the appropriate moment to check-in a group database to its parent group(s). The group notion handles multiple parents in that a group's objects are deemed to automatically check-in to the parent group from which they were checked out.

Our LLT model

Our LLT model is based on the notion of semi-public databases for tools, similar to the above mentioned LLT models. An entire tool is mapped to an LLT. The semi-public database of a tool contains all its view objects along with the base objects they are derived from. The state of the semi-public database is savepointed from time to time by commits of the ongoing short transaction. The semantics of savepointing is to commit the resources handled by the short transaction with-
out releasing any resource locks. *Long transaction commits* cause checking-in of the semi-public database into the public database. We do not actually provide a command for savepointing in our tool language. Savepointing is done at the end of every tool operation. This is intuitively reasonable since a tool operation is a semantically significant unit of tool execution. The end of a tool dialogue leads to a long-transaction commit. At this point, the semi-public database is checked-in to the public database and the objects in the semi-public database are garbage collected. A standard command "End-Session" is provided for every tool to indicate the end of tool dialogue. The nesting aspects of our transaction model are postponed till the next chapter since nested transactions appear only in situations where tools call other tools.

### 2.6 Examples

#### 1. A Module Programming Environment

This example deals with an mini-programming environment for a module based language such as MODULA-2 [85]. The informal structure of a module is as shown below:

```
Class Module is

Properties

    Name : ModName;
```
Procedure Name | Interface         | Documentation
----------------|------------------|-----------------
p11              | p11(arg1:t1,....) | This procedure
p12              | p12(....)        | .................

Figure 8: An environment for module programming
Exports : set-of Item;

Imports : IdSet;

Impl : ModuleBody;

Procedures: set-of Procedure;

Objcode : Code;

Methods

Compile();

Get-ObjCode() returns code;

Add-Procedure(ProcName);

...........

...........

end Module;

Class Procedure is

Properties

Name : ProcName;

Interface : Signature;

Code : SourceCode;

...........

...........

Procedures : set-of Procedure;

Methods
add-procedure(...)  
delete-procedure(...)  
end Procedure;

Now, a module coding environment may provide two useful views to the module programmer. It could provide him with a "module graph" view, by which he could observe the graphical structure of the module-procedure hierarchy. A "library" module could be used by the programmer to get a view of the interfaces of all the procedures he has coded, irrespective of their position in the module hierarchy. This could be used by the programmer to check in potentially useful procedures to a "procedure library" for future reuse. Various other useful views include a views for viewing source code, coding schedules for the module components, coding productivity views showing when various procedures were completely coded and tested, and so on. Figure 8 shows a partial user interface for a module programming environment.

Module Graph View

The module graph view specification follows. The goal of this view is to expose the structural design of a module without allowing the user of the view access to any of the details of the view, or any methods to compile the module or change its

---

8Actually, the graph view is makes much more sense in representing the structure of a system in terms of its constituent modules.
interface specification. On the other hand, methods to add and delete procedures are “virtually” introduced into the view. This allows the user of the view to change the procedural structure of the module without changing the code.

(def-view ModGraph
   :root ModIcon
   :base Module )

(def-derived-class 'ModICon 'Module
   :view ModGraph
   :virtual '(Name)
   :synthesised '((Procedure :domain ProcIcon
                     :structure set-of
                     :base Procedure
                     :basestruct set-of
                     :morphism settoset1))
   :own '((Icon :domain Icon :init ModPixrect)))

(def-derived-class 'ProcIcon 'Procedure
   :view ModGraph
   :virtual '(Name)
   :synthesised '((Procedure :domain ProcIcon
                     :structure set-of
                     :base Procedure
                     :basestruct set-of
                     :morphism settoset1))
Library View

This view illustrates the use of rearrangement operations. The whole procedure hierarchy is completely flattened to present a "set" of procedure interfaces, irrespective of the original position of the procedure in the hierarchy.

(defun-derived-class 'ModLib 'Module
  :view LibraryView
  :virtual '(ModName)
  :synthesized '((LibProcs :domain ModProcs
                   :structure set-of))
  :catch LoadTag LibProcs AddToLibProc
  :allow LibProc Procedures)

(defun-derived-class LibProc 'Procedure
  :view LibraryView
  :virtual '(Name Interface)
  :throw LoadTag
  :allow LibProc Procedures)
2. Unit Tester

Unit Testing [59] is a phase where a programmer tests his own module after it has compiled successfully and passed the code review. For a unit test to proceed, other module which it uses have to be available and ready. In bottom-up testing this is assured since the testing discipline ensures that modules are tested only when the modules they depend upon are ready. Let us assume however, that we allow random testing in our environment. Thus, the programmer is free to test his module in an approximate or incomplete environment. To help the programmer simulate reality, the environment provides stubs and stub generators. Stubs are simulations of modules that have not been completely coded. Stub generation algorithms could be as trivial as just printing a message such as “You are in module X...”. However, one can have more intelligent stubs which either simulate the module exactly for a fixed set of inputs and use some sort of interpolation to figure out the behaviour for all other inputs. Alternatively, one could directly interpret the design specification to generate a stub. Random testing also requires a driver to set up the environment for the module being tested before invoking it. A driver can be as trivial as a call to the test module. However, modules with complex arguments require non-trivial (and therefore non-automatable) drivers. We assume, for the time being, that the driver is automatically generated and available for the RunTest command.

The UnitTester tool has two main operations to create the load module and
execute the test plan respectively. Obviously the real-world tool would contain operations to browse test results, manually modify stubs and drivers, obtain scheduling information about dependent modules to see if postponing the test by a day or two would help, and so on.

CreateTestLoadModule links the object code of all the modules involved, and with the driver code to create the executable. RunTest runs the test plan from the Unit development Folder on the executable.⁹

Shown below is a language specification for the tool interface and associated views. A pointer to the LoadModule object is stored in a tool property for future use by operations such as RunTest. The tool runs as a long transaction and is passive. The tool implementation is not shown as there is nothing notable about it. Conceivably, it dumps the code in a file and calls a compiler to generate the object code. Obviously, the strict intermodule type checking has to be implemented within the environment. We do not dwell on the details as such tasks have already been accomplished in module interconnection languages of the past such as INTERCOL [83].

The UnitTestView used by CreateTestLoadModule represents the module being tested and the modules it depends on as two different derived classes. The view also expresses the constraint that the module undergoing the test should have compiled successfully and the object code should be up to date. The modules being

⁹The UDF is a standard object used in software specification to hold all information about a module.
used by the module being tested must either have already been tested or stubs
should be available to simulate the behaviour of these modules.

The constraints on this view are fairly complex and beg the question “What
if the constraints are not met...”. This is where plan integration can play a role.
The environment can use its knowledge about tool functionality to help automate
the constraint satisfaction process. More is said about this in Chapter 4.

(def-tool-interface UnitTester

  :views (UnitTestView,Module)

  :Ops ((CreateTestLoadModule ((T UnitTestView)))
         (RunTest ((T UnitTestView) (U UnitDevFolderView))) . . )

  :facets ((InvocationDiscipline Passive) (UCC Transaction))

  :properties ((LoadModule Module))

)

(def-view UnitTesterView

  :root TestUnit

  :base Module)

(def-derived-class TestUnit Module

  :view UnitTesterView

  :virtual (Name Impl Object)

  :synthesised ((Uses :domain UseStubs
                  :structure set-of

)
:base Module

:base struct set-of

:morphism setTo set1)

:constraints (Impl Object

(RELATION ≤ Impl.timestamp

Object.Timestamp))

)

(def-derived-class UseStubs Module

:view UnitTesterView

:virtua l (Name Impl Object Stub

Source-Tested)

:constraints ((RELATION (or Source-Tested Stub))

Object (RELATION or

(RELATION ≤ Impl.Timestamp

Object.Timestamp)

(RELATION ≤ Stub.Timestamp

Object.Timestamp)))

)
2.7 Model for Display and Interaction

The conventional “stream-oriented” model of interaction supported by most languages is low-level, synchronous, and sequential. It is synchronous in that a display is treated as a service call of some sort to a logical presentation device. Since the data passed in the tool-user dialogue is of an unstructured type supported by the language, the interaction model is characterized as low-level. Since there is no explicit support for dialogues between a user and several concurrently running applications, one can characterise the display model as being sequential.

This model is obviously unsatisfactory for workstation-based, database-centered, software development environments. Workstations support interactions between a user and multiple applications. SDEs additionally require some way to concisely specify the presentation structures and presentation disciplines involved in a tool-user dialogue. A summary of the modelling issues raised in workstation-based environments follows:

- The display device is treated as a passive device that is exclusively controlled by a single application for the duration of the interaction. This model does not hold for workstations which model concurrent dialogues between multiple application and the user.

- Tools in an SDE interact with the user through complex presentation structures. These are tedious to model using the low-level primitives provided
by conventional languages. One needs not only to have higher-models of presentation, but also some technique for dialogue specification.

- The notion that the lifetime of an interaction unit be limited to the duration of the presentation request is unsatisfactory for SDEs. Some presentation requests, such as setup requests for a tool may be made at the beginning of the user-tool dialogue but might need to last for the entire duration of the tool interaction. Others, such as data requests, may need to follow the conventional paradigm of interaction.

- Presentation structures in SDEs are typically anchored. In other words, they are not purely presentation structures, but depictions of database objects. This is true of SDEs that are not database-centered as well, such as GANDALF, PECAN, GNOME and so on. This is in contrast to application such as image-processing where the display could be the end product of the application. Thus, the presentation primitives must make sense in the context of the modelling primitives (structures and operations) provided by the data model. For example, in an object-oriented data model, presentation schemes should facilitate the presentation and manipulation of complex object networks. One can think of the depiction as providing a visual programming language for object manipulation, which parallels the normal programming language provided to applications for database manipulation.
- Extending the thought that a depiction serves as a visual programming language for users, a depiction should not only allow the user to manipulate database objects, but also to invoke tools on them. The latter is significant because, as we have seen, the mechanics of invoking operations of tools differs considerably from that for invoking operations on an object. The recursive relationship between display and tools (i.e., that tools can create depictions which can allow invocation of the same tool on objects in the depiction) complicates the relationship between tools and the presentation agent.

Presented in Figure 9 are some possible "models" for modelling the presentation agent and presentation policies. Part a) shows the much maligned "stream model" of display. The "fill-in-the-form" model of part b) models the presentation agent as an active application that coordinates multiple depictions, and is high-level in the sense that it models display at the object level. However, passing large objects "by copy" is inefficient, and the changes are not reflected in the database copies. Part c) acknowledges that the object(s) being displayed reside in the database, and are modified in step with the depiction. However, it models the presentation agent and the tool as cooperating in an undisciplined manner through the shared view object. Since this would require a great deal of additional synchronization mechanisms in the data model, we reject this model. Part d) depicts the presentation request in the same way as c) but augments the depiction semantics with a "check-out" of the object being depicted from the tool's database to the local database of the
Figure 9: Alternatives for modelling interaction
presentation agent. This way, during the period that a view is being depicted, it is not accessible to the tool that requested it's display. Part d) still does not accurately represent the semantics of tool invocation requests that originate from the scene being displayed. These tool invocations would require the tool to regain control of the view object so that it can service the request. Part e) of the figure explicitly models callbacks, which are operations modelled in a depiction that map to explicit calls to some tool. This is in contrast to graphical operations which are considered part of the presentation agent and have no consequences on the "state" of the environment.

None of these figures describes the nature of the display scheme and the display abstractions used by the presentation agent. We believe that the editor based model outlined below provides an adequate depiction model for the granularity and level of abstraction required by tools in SDEs. This models borrows heavily from substrates for interface design such as IMPULSE-86 [75] and object-oriented frameworks such as IDA [86].

An Editor-Based Display Scheme Each genre of display can be thought of as a hierarchy of editors. Editors mediate between editees(objects that are edited by the editor) and the user. Higher-level editors display complex objects by delegating the depiction of the object's components to lower level editors. The lowest-level editors display primitive objects.

Editor instances are instantiated from editor classes by template instantiation.
An instance of an editee type is bound to an instance of an editor type to create an instance of a depiction type, or a tableau. Type information pertaining to an editor consists of instantiable and uninstantiable parts. Uninstantiable features include EditorName, EditSchemeType, EditorMenu, and EditorCallBacks. Instantiable parts include Editee, EditSchemeInstance, ToolBinding and so on.

The EditScheme is different from the type of the editee (which is a particular kind of view). It has more to do with a general display paradigm such as textual, graph-oriented, or dataflow-oriented paradigms. They determine the global properties of a display such as where the subeditors are placed, when they are instantiated, and so on. The EditMenu represents the messages that the user can send to a particular editor. Operations tied to the menu might be purely graphical or may have a side-effect of causing changes to the database (such as object creation). EditorCallBacks are messages that the user can send to the tool that initiated the dialogue. The actual tool instance to which the message is sent is decided by the tool instance bound to the editor via the ToolBinding property.

The Display Scheme used by a tool in a dialogue is a combination of an Editor type and an object pointer. From these pieces of information, the Presentation Manager (display tool) creates the actual editor instance. Editors in this scheme map both to views and controllers of the Smalltalk MVC scheme.
Since tools are fairly large pieces of code, it is useful to think of ways to use existing tools in the process of building new tools. The process of building tools using other tools as building blocks is known as tool composition. To accomplish this, the environment has to provide composition mechanisms within the environment. Language primitives for such composition mechanisms need to be incorporated into the tool modelling language so as to allow the tool builder to reuse existing tools in the tool writing process. In terms of the four sub-problems of tool integration discussed in Chapter 1, tool composition deals with control and data integration. Plan integration is dealt with separately in Chapter 4.

In this chapter, we discuss two tool composition mechanisms, Cotools and Nested tools. The former is predominantly a data integration mechanism, and the latter a control integration mechanism.
3.1 Cotools

3.1.1 Motivation and Abstract Model

Tools can be effectively reused if they are used as building blocks in larger composites. From a data integration viewpoint, a composites can be thought of as being formed from the plugging together of component tools. The goal is also to achieve composition of existing tools with no modification to the specifications of the tools being reused in the new tool composite. Ideally, the tool implementations of tools being reused should not be changed, at any rate the tool specification should be reused in the composite. This orthogonality of tool modelling and composition leads to a very clean and comprehensible composition mechanism, much in the spirit of UNIX pipes.

In a database-centered environment, all tools interface to the database and there are no explicit communication channels between tools. The language primitive for plug-oriented tool composition has to invent equivalents of channels in a database. The possibility explored in cotools is to think of "view intersection" as creating channels of communication between tools. One can think of many views being derived by many tools from the same set of base object(s). Thus, several view objects are dependents of the same base object. The base object can then be thought of as a bidirectional channel through which the components of several views communicate. Any change made to a dependent of a base object is immediately propagated
to all the dependents of the base object.

A cotool is a set of tools whose views might intersect. Two independent tools which have to coordinate access to a common object by database mechanisms such as locking and transactions. In contrast, member tools of a cotool group manipulate common objects in an interleaved manner. The cooperation discipline of the cotool specifies the manner in which cotools coordinate with each other. Modelling members of a cotool group as being truly concurrent makes things complicated as one has to introduce synchronization primitives at the base object level. The alternative of modelling a cotool as being composed of quasi-concurrent tools is simple and intuitively more appropriate. This means that only one tool is manipulating a base object at a time. Any changes made by that active tool are propagated atomically to all the other dependents of the base object being manipulated. The appearance of concurrency arises from the fact that different tools in the cotool group are active at different times, and although their activities are multiplexed, all members of a cotool are simultaneously interacting with the user. Intuitively, the quasi-concurrent model is adequate because at any specific point in time, a user can be carrying on a dialogue with only one member of the cotool. As the primary purpose of introducing cotools is to be able to construct single-user environments out of extant tools, the quasi-concurrent model makes sense.

Figure 10 illustrates the abstract model of cotools. The views being manipulated by component tools are modelled as private databases of the tool, and all base
objects belong to a group database which is global to the cotool group. The group database is distinct from the public database which is accessed by all tools, whether or not they belong to the cotool group in question. Base objects in the group database may be unshared or concurrently accessed by several members of the cotool group. In the latter case, the above discipline of manipulation is enforced.

In addition to the concurrency discipline outlined above, the cotool model possesses certain other behavioural properties:

- **Passive Members** Since the scheduling of tool operations is not in the hands of any member tool, it follows that only "passive" tools are allowed by this cotool model.

- **Non-Preemptive, Unbiased Scheduling**: The scheduling policy for cotools is fixed. It is non-preemptive in that once a tool operation is begun, it cannot be interrupted until it terminates. On termination, the member tool gives up control to the cotool scheduler. There are no scheduling priorities associated with members of a cotool. In that sense the scheduling is unbiased. This is compatible with the intuition behind cotools because the demand for tool operations are really generated by the user. Therefore, assigning programmed scheduling priorities to cotool members is pointless.

- **Single Recovery Unit**: While the activities of a cotool are concurrent, they represent a single activity being carried out by the user. It is thus appropriate
that the entire activity of the cotool commit or abort as a single unit. This in turn means that either the member tools are procedural (and therefore have no specified recovery behaviour), or their recovery behaviour is consistent with that desired of the cotool. Namely, if the overall transaction behaviour desired from a cotool is a short transaction, then the member tools should themselves have either no recovery behaviour specified or be specified to be short transactions themselves. If the overall recovery behaviour of the cotool is to be a long transaction, then the member tools must have a recovery behaviour which is a long transaction.

3.1.2 Comparison with other Ideas

Discussed here are three ideas that contribute to the aggregate notion of cotools. While each of these ideas expresses one aspect of the cotool model well, it says nothing about the other aspects of the cotool model. Together, they can be looked upon as expressing the essential elements of the cotool model in terms of existing approaches to concurrency, recovery and coordination. Figure 11 illustrates the ideas described below.

Coroutines

In some sense, member tools in a cotool suspend and resume like coroutines [13, 53]. Coroutines are like subroutines, but allow transfer of control in a symmetric way rather than in a strictly hierarchical way. Control is transferred between coroutines
Figure 10: An abstract model of Cotools
by means of the resume statement. Execution of resume is like a procedure call; it transfers control to the named routine, saving enough state information for control to return later to the instruction following the resume. (When a routine is first resumed, control is transferred to the beginning of that routine). However, control is returned to the original routine by executing another resume rather than a return. Any coroutine can transfer control to any other coroutine in the coroutine set by naming the coroutine to activate as an argument to resume. In a sense, coroutines are concurrent processes in which process switching has been completely specified.

In the sense of being quasi-concurrent the concurrency behaviour of coroutines is similar to cotools. However, the fact that the process switching in coroutines is embedded makes it different from cotools. Embedded process switching is unsuitable for tool composition because the composition discipline has to be foreseen while constructing the tool. This not only makes the tool less reusable (by adding a "context" to its construction), it makes the context switching deterministic, and thus biased. Thus some tool in a cotool set can be programmed to "hog" time. This goes counter to the behaviour we want from cotools, which is to schedule member tool invocations based on user demand.

SMALLTALK dependents

A kind of dependency supported by the SMALLTALK Object class performs a coordination task similar to cotools. Specifically, via the dependents [28] mechanism, an object B can be linked to an object A in a manner such that B can be informed
Figure 11: Cotools in terms of more familiar ideas
of changes occurring to A. Upon being informed when A changes and the nature of the change, B can decide to take some action such as updating its own status. The concrete mechanism supported by SMALLTALK requires all clients of an informer object to register themselves with the informer. The informer broadcasts any changes to all the objects registered as its dependents. It is the responsibility of the clients to provide a change method as part of their specification, which gets triggered in response to the change broadcast by the informer.

The view intersection discipline of cotools involve a change notification similar to the dependents mechanism described above. In fact, at an object to object basis, the protocol followed by base object and the derived objects that depend on it is identical to the dependents mechanism. However, view intersection is implicit and applies to views rather than the explicit registration and a single object basis of the dependents mechanism.

**Sessions and Transaction Groups**

As was noted in chapter 2, databases generally use the notion of transaction to represent both as a unit of concurrency and as a unit of recovery. The notion of sessions in ORION [5] is an exception to the rule and allows a concurrent application to be a single recovery unit (which is similar to cotools). ORION allows several concurrently running UNIX applications to belong to the same session. All applications of a session share locks, and only one member of a session is active at any point in time. Also, any member of a session can commit the results of the
computation so far, thus leading to the committing of the results of all sessions to the database.

Sessions are similar to cotools in the fact that they allow concurrency in single unit of recovery. However, the notion of sessions does not provide any mechanism for members to have multiple views of the database while operating concurrently. It also does not provide any coordination discipline a la coroutines, which is specified by the cotool model. Transaction groups [73] are more general than sessions. The constraints on a group of transaction in a transaction group are specified as constraints on a concurrent history. However, transaction groups do not address the issue of multiple views either.

3.1.3 Extensions to the idea

Quasi-concurrency in cotools is a design decision that trades expressive power for simplicity. While it properly expresses the semantics of single-user environments, it is desirable to allow true concurrency amongst multiple members of a cotool in many situations. Two such examples are cooperative multi-user environments and object-oriented pipes. In the former situation, it is unnatural to constrain multiple users of a cotool to a protocol which does not allow two users to concurrently interact with members of the same cotool. In the case of object-oriented pipes, lack of true concurrency is more of an efficiency issue. Object-oriented pipes are

\[1\] If each tool/transaction is thought of as having an associated history of operation invocations or data manipulations, then a cotool is associated with a concurrent history due to the concurrency in the activities of member tools.
basically like UNIX pipes in that concurrently operating processes communicate with each other via directional channels. The modification in object-oriented pipes is to enforce strong-typing on channels. While the idea of cotools is in contrast to channel-oriented paradigms, it is useful to allow tool views to intersect on a particular object such that one tool produces components of that object and others consume the object. The production and consumption behaviour is virtual and is simulated by morphisms in views.

The example in figure 12 illustrates an example with both the above characteristics. One can think of the process of module testing as being performed by a set of tools that communicate with each other through object-oriented pipes. Each tool performs one stage of the testing process, and the composite tool interacts with several users (role players in the testing process). Concurrency in tool operation is therefore also desirable due to the multi-user nature of the tool composite.

An informal description of the testing process for a module is as follows. Stubs need to be generated for modules which are used by the module to be tested but are not yet coded, and a driver needs to be generated for running the entire test. The complexity arises in the generation of test cases by analysing the code to be tested. The is done in the following stages: control flow graph generation, path enumeration, generation of test cases that will exercise the enumerated paths by path domain analysis, generation of a suite of expected behaviours (set of expected outputs) by symbolic execution of the test program, running of the test cases (could
Generate control flow graph and enumerate paths.

Perform path-dependent symbolic evaluation to generate path domains.

Use path domain to generate a set of test cases (test plan).
Generate expected results.

Run test.

Compare expected and actual results.

Figure 12: Pipes and tools in the testing process
be automated by a tool), and comparison of expected and actual results. Shown in figure 12 are the various tools that would be involved in the process and the objects in the database through which they communicate. The communication obeys a production-consumption protocol of pipes. However, these pipes are created by morphisms and are streams of objects rather than streams of characters. Stub writing, test plan generation and test execution are not performed by the same person, thus the tool is a multi-user cotool. The example is largely synthesized out of a description of the applications of symbolic evaluation in the testing domain by Lori Clarke et al [10]. The reader is directed to that reference for details of the testing process and a more elaborate description of the definitions and modalities of the testing process.

3.1.4 Motley subjects

Cotools and transaction: long and short

While describing the cotool abstractions in the previous subsections, the exact recovery behaviour of the tool composite was glossed over because it did not radically affect the fundamental notion of cotools. However, from a pragmatic point of view, it is significant whether the overall cotool is a short transaction or a long transaction. This has to do with the fact that it is possible to implement the extended cotool model if the member tools involved are long transactions, while it makes it hard to visualise the implementation of the extended cotool model for
cotools that are short transactions without fundamentally changing the nature of the short transaction concept.

There is a race condition associated with concurrent activities accessing shared information. In the context of cotools this has to do with two member tools M1 and M2, simultaneously modifying objects D1-B and D2-B respectively, which are dependent on the same base object B. Assuming that D1-B and D2-B are the only objects dependent on B, the cotool coordination mechanism does not handle this situation. If only D1-B were changed, the coordination mechanism states that the base object B will be changed through the view mechanism, and that all other dependents of the base object in question (namely D2-B) will be sent a change message. A similar interpretation is given if only D2-B were changed. However, concurrent changes to D1-B and D2-B lead to concurrent, intersecting chains of activities which would either lead to some inconsistency or are handled by some conflict resolution mechanism that serialises the activities by rolling back (and possibly retrying) one of the activities involved.

The notion of long transactions is itself based on a lower level atomic action primitive (maybe even short transactions). In the checkin/checkout model of long transactions each checkin, checkout property access or property modification in an object is a lower level atomic action. Thus it is left to this atomic action primitive to serialise the intersecting activities referred to in the previous paragraph. Conversely, short transactions are thought of as the low-level atomic action primitive
and are based on locking which is not an atomic action in itself. While one could conceive of a modified *multi-threaded short transaction* concept, the conventional short transaction concept would run into some trouble in the above situation.

Better change notification for views

The abstract model of a cotool explains the manner in which views belonging to member tools are changed by modifications to views belonging to completely different tools. It does not explain however, the manner in which a tool T manipulating a view object V would be *notified* of a change in its view that the tool T was itself no responsible for. This can be implemented by having a single *ViewState* object for each view which maintains a list of *dirty* objects (i.e objects that have changed since the tool last saw it). The tool can then periodically check the ViewState object for a view V to find out which of its component objects changed, rather than traverse all the objects in a view and compare their values to old values that have been cached someplace. The distinction between the tool T changing an object in view V and that happening through the actions of another tool can be made in the system because change messages get sent to objects of a view V only if the object in view V is modified as a result of the actions of a tool T' that does not own V. Thus, allowing only change messages can append to the list of dirty objects in the ViewState object solves the problem.

This solution is rather clumsy in that the *notification* abstraction is replaced by the idea that a tool busy-wait on all its ViewState objects if it has nothing else
to do, waiting for a change to a view which may lead to display update chores for
the tool. This is remedied by introducing a *wait-on-views* primitive into the tool
language which actually waits in a non-busy-wait manner and is awakened on view
changes. This again is easily doable using operating systems primitives, and is one
of the *yawn* issues in operating systems.

**Do we need selective waits?**

In the case where a member tool of a cotool is interactive, and the user is idle,
one would want the tool to sleep and be awakened by either user input or view
changes. This requires that a *selective-wait* primitive (a la ADA) be introduced into
the tool language which allows the tool to wait on either kind of event (view change
or user input). While the language issue here is fairly trivial, the implementation
issues could be non-trivial because event-based user interface toolkits such as X
implement their own event queues and their own primitives for waiting on the
user-interface event queue alone. Some low-level modifications involving merging
event-based primitives of the DBMS and the user interface toolkit would need to
be performed to implement the selective wait primitive in the tool language.

### 3.2 Nested Tools

Issues in tool nesting pertain to the concurrency and recovery implications of *nest-
ing*. From the point of view of invocation, it seems adequate to provide *call* and
*split-join* forms of invocation. The latter involves parallel operation of invoker and
invoked tools until a “join” is performed by the invoker. If both tools have an associated recovery behaviour (specifically long transaction), then the semantics of transaction nesting need to be specified. The next two subsections introduce a notation for representing transactions and use the notation to describe the semantics of call and split-join. A two phase checkout protocol is presented under which split-join relationships can be enforced. The notation is borrowed from [39].

3.2.1 Notation and Definitions

Definition: A global database consists of a public database and several private databases of active transactions. The set of values taken by elements of the database is known as the state of the database. The set of all possible states is denoted by S.

Definition: An integrity constraint C is a predicate on S, C: S → TRUE, FALSE.

Definition: A database operation (operation for short) O is a mapping from S to S: O: S → S.

Definition: A transaction is a 3-tuple (T, O, C) where

- T is a set of transactions or operations.
- O is a partial order on T
- C is the integrity constraint preserved by the transaction.

At the deepest level of nesting, transactions consist solely of operations.
Definition: M denotes the set of all allowable operation names.

Definition: The closure \((T, O, C)^*\) of a transaction \((T, O, C)\) is the transaction \((T^*, O^*, C)\) where:

- \(T^* = (M \cap T) \cup \left( X(t', o', c) \right)^*\) and \(X = \{ t', o', c \mid t', o', c \in T - M \}\).

- \(O^*\) is the partial order induced on \(T^*\) defined as follows: Let \(a\) and \(b\) be operations in \(T^*\). In \(O^*\), \(a \prec b\) if one of the following holds:

1. \(a, b \in T\) and \(a \prec b \in O\).
2. There exist a pair of transactions \((t_1, o_1, c_1)\) and \((t_2, o_2, c_2)\) such that \(a \in t_1^*, b \in t_2^*\) and \(t_1 \prec t_2\) in \(O\).
3. There exists a transaction \((t_1, o_1, c_1)\) such that \(a, b \in t_1^*\) and \(a \prec b \in (t_1, o_1, c_1)^*\).

Definition: An execution \(e\) of a transaction \(t = (T, O, C)\) is a transaction \((T^*, O', C)\) where \(O'\) is a total order such that \(O'\) implies \(O^*\). That is, if \(a, b \in T^*\), and \(a \prec b \in O^*\), then \(a \prec b \in O\). We denote the set of all executions of \((T, O, C)\) by \(E_t\).

Definition: An execution \(e\) of \((T, O, C)\) is correct if for all states \(s \in S_C\), \(e(s) \in S_C\).

Definition: A protocol \(P\) is a mapping from the set \(E_t\) of executions of \(t\) to TRUE or FALSE. If \(e\) is an execution, the \(P(e)\) if \(e\) is an execution legal under
A protocol P is **correct** if for all executions e in $E_t, P(e)$ implies e is correct.

Determining protocol correctness is an NP-complete problem.

### 3.2.2 Call and Split-Join disciplines

Let $(T_{Ci}, O, C)$ be a transaction where:

- $T_{Ci}$ contains $T_{Cd}$, where $T_{Ci}$ calls $T_{Cd}$. ($T_{Cd} = t', o', c$).
- $O$ is some partial order.
- $C$ is a set of integrity constraints.

The *call* relationship between $T_{Ci}$ and $T_{Cd}$ obeys the call semantics under protocol P if $P(e)$ is true only if e is equivalent to a correct execution of $((T_C - T_S), O'', C)$ where $O''$ contains:

- All elements $a < b$ of $O$ such that $a, b \in T_{Ci} - T_{Cd}$.
- All elements $a < b$ of $o'$.
- All orderings of the form $a < b$ where $a \in T_{Ci} - T_{Cd}$ and $a < T_{Cd}$ appears in $O$, and $b \in t'$.
- All orderings of the form $a < b$ where $b \in T_C - T_S$ and $T_S < b$ appears in $O$, and $a \in t'$. 

The split-join relationship between $T_{C1}$ and $T_{C4}$ obeys the call semantics under protocol $P$ if $P(e)$ is true only if $e$ is equivalent to a correct execution of $((T_C - T_s), O'', C)$ where $O''$ contains:

- All elements $a < b$ of $O$ such that $a, b \in T_{C1} - T_{C4}$.

- All elements $a < b$ of $\alpha$.

- All orderings of the form $a < b$ where $a \in T_{C1} - T_{C4}$ and $a < T_{C4}$ appears in $O$, and $b \in t'$.

- All orderings of the form $a < b$ where $b \in T_C - T_s$ and $J(T_s) < b$ appears in $O$, and $a \in t'$. It is also necessary that $T_s < J(T_s)$ (in other words, the join takes place after the split).

Obviously, a call is a special case of a split-join in which no operation occur between the split and the join in the caller. Therefore, only the protocol for split-join is presented below, in the next subsection.

### 3.2.3 A Two Phase Checkout Protocol for Split-Joins

The following is a protocol that can be used to implement the split-join relationship amongst tools (transactions). For simplicity, only update-mode checkouts are considered. The discussion can easily be generalised to include read-mode checkouts.

We use the term *client* for the invoker of the split, and the term *subcontractor* for the transaction created by the split.
**Definition:** We think of each client and subcontractor transaction as being associated with a **client database**, a **private database**, and a **subcontractor database**. The private database is not shared with any other transaction. The subcontractor database is used for communication between the client and the subcontractor. A transaction places private data for check-out by subcontractors in the subcontractor database. Each transaction has a unique subcontractor database. The client database of a transaction is the subcontractor database of its client or the public database. More than one transaction may share a client space in a cooperative environment.

**Definition:** As before, we use the terms **checkin** and **checkout** to represent inter-database data transfers. We add the notion of **checkout enable** to represent the act of moving data from a private space to a subcontractor space. **Checkout disable** is the reverse.

**Definition:** A **two-phase checkout protocol** is said to be satisfied if there is one phase of checkout enabling of data followed by a phase of checkout disabling. Once, any data is checkout disabled by the subcontractor transaction, no data is checked in by the subcontractor transaction from its client space.

### 3.3 Language Primitives

#### 3.3.1 Cotools

The language specification for a cotool describes the following things:
• The cotool identity, its members, and the arguments with which the cotool is invoked.

• The startup command with which each member tool is invoked to interact with the user.

• The manner in which startup arguments are derived from the arguments in the cotool invocation. Since the operation specifications for the member tools already specify the definitions of the view arguments for all operation, the actual derivation of the tool startup arguments from the cotool arguments need not be specified. All that needs to be specified is the particular cotool argument from which a tool startup argument is derived.

The actual syntax of a cotool definition is shown below.

```
(def-cotool CotoolName
  :arguments ArgList
  :members MemberSpecList)

MemberSpecs
(ToolName :startup (StartupCmd :arguments (ArgName
    :derived-from BArgName)))
```

In the def-cotool statement, :arguments denote the arguments that accompany a cotool invocation, and a description of the member tools appears in the :member
clause. For each member tool, there is an associates startup clause which describes the manner in which the member tool should be started up when the cotool starts up. This is done as part of the:startup clause for each member tool description. Since the arguments to member tools are views, it is necessary to describe the base(cotool) arguments from which the tool arguments are to be derived. This is done in the :derived-from clause for each argument description in the member tool description.

3.3.2 Nested Tools

Nested tools are calls to tools embedded within other tools. Therefore the language primitive required for nested tools is a facility by which a call to a tool can be embedded in another tool's code. The invoke-tool primitive shown below serves this purpose. In addition to invoking a called tool, the invoke-tool construct has to allow for modes of tool invocation (specifically call or split-join).

There is some non-uniformity in the nesting primitives in that while an ordinary call requires no explicit join between the invoker and the invoked, the split-join facility needs the addition of an explicit language construct for synchronising the caller and called tool activities. To handle this, the invoke-tool primitive is made to return a tag which uniquely identifies the particular tool invocation in question. This tag is later used by the join construct to identify the particular tool invocation with which the caller tool wishes to synchronise (the caller tool may have called
several tools in split-join mode, thus several tagsd may be active). For a normal
call, a null tag (or nil) is returned by the `invoke-tool` construct.

Shown below are the constructs for tool invocation and caller-callee synchro-
nisation(join) respectively:

```
(invoke-tool ToolName OpName Mode Arg)
```

```
(join TagName)
```

### 3.4 An Example

The module programming environment example of chapter 2 also serves as an
eexample of cotools. The environment as a whole can be thought of as a cotool made
up of a set of member tools. Each window on the screen is actually a tool interacting
with the user using a tool-specific view and a tool-specific dialogue. In this case the
views are iconic, editing and library-oriented respectively. The tools operate quasi-
concurrently (in that only one window in the workstation is active at any time),
have different views of the same set of objects (a module-procedure hierarchy), and
represent a single user activity (and thus constitute a single transaction).

The cotool semantics automatically correlates changes in the active view (say
iconic) to corresponding changes in other views. The user can however interact
with the member tools in an interleaved manner. A partial language specification of the module programming environment cotool is shown below.

(def-cotool MiniProgEnv
    :arguments System
    :members
    (IconTool
        :start-up (Init
            :arguments (IconView :derived-from System)))
    (EditorTool
        :start-up (EditorInit
            :arguments (EditorView :derived-from System)))

      . . . . .
      . . . . .
    )
CHAPTER IV

Tool Management

4.1 Introduction

The complexity of the information manipulated and the size of the tool name space in SDEs introduces a management problem. Due to the size of the tool space, it is hard for the environment user to discern the correct time to invoke a tool and the right data to pass it. While the high-level tool specifications of Chapter 2 lend a uniformity to tool specifications, the size of the tool space makes it difficult to remember the functionality of individual tools. Similarly, while views and automatic view translations reduce the complexity of tool invocation in a view-oriented environment, the inherent complexity of views makes it hard to foresee if they will be successfully derivable and whether the tool invocation will in fact be successful. The task of assisting the user(s) in tool invocation is referred to as Tool Management.

While passive devices such as on-line manuals can be used to deal with size of the tool space, ensuring the success of tool invocations is a more difficult task. Invocations of one tool need to be coordinated with others to be successful. To
assist the user in successfully invoking a tool, the environment has to ensure that the tool coordination discipline has been followed. This in turns breaks into a representation problem and a reasoning problem. For the environment to enforce a coordination discipline, it has to be represented in the environment in the first place. The environment then needs to reason with or interpret this information to accomplish the stated tool management goals.

Due to the opportunistic nature of tool manipulation, representation of tool coordination disciplines have been descriptive rather than prescriptive. In other words, rather than represent the tool invocation pattern over the entire software development lifecycle, it is preferred to simply represent high-level information about the tasks accomplished by individual tools along with some associational information about closely related tools. Thus, one needs problem-solving techniques that can reason with this information, as opposed to being able to execute a tool coordination program.

A popular approach to tool coordination has been to view it as a planning problem [6, 33]. Planning is a form of problem-solving where a goal is attained using a set of operators by coordinating the application of the available operators on the world state. Planning thus uses descriptions of domain operators to coordinate their application and attain a stated goal. Rudiments of planning are outlined in 4.2 and the correspondence between planning and tool coordination is discussed in 4.3. Details of the tool planning problem are discussed in 4.4, language aspects
in 2.4.2 and comparisons to other approaches in 4.8.

4.2 A Birds Eye View of Planning

A plan can be thought of as a program of actions that can be carried out to achieve goals. Planning is the process of creating (and optionally executing) plans. The typical vocabulary for expressing domain knowledge in planning consists of operators and a state schema. Operators are parametrised templates defining the possible actions in the domain. The state schema consists of a set of predicates that describe the state of the world for that domain. Operators represent the unit for describing actions. Operators are further described in terms of preconditions that must hold in order for the action to be legal, a set of effects(goal(s) and side-effects) that defines the result of performing the operation.

The plan to achieve a goal(desired state), given an initial state, is constructed using the following steps:

- Choosing the best operator to apply next based on available heuristic information.

- Applying the chosen operator to compute the new world state.

- Detecting a solution when it is found.

- Detecting deadends and abandoning them with as little computational effort as possible.
The following questions represent design decisions that need to be made regarding the details of planning. They help determine the particular subclass of planning (and therefore the kind of planner) that applies to the situation being dealt with.

- What is a problem? What does it mean for one to be reduced?

- Is the choice of reduction deterministic?

- How are simultaneous subproblems handled?

- Is execution necessary?

- Is the universe predictable, i.e. can the effects of operators be symbolically represented, or can they be determined only at execution time?

- How are planning and execution interleaved?

- How are errors at planning and execution time handled?

In addition, the nature of the actions performed is profitably classifiable along three dimensions: Problematicity, Monasticism, and Parasitism [45]. The problematicity of a task can be primitive, where the task is carried completely by a single operator in the environment, or problematic where the task requires a plan of execution. In terms of Monasticism, a task is inferential if it has no effects on the real world, and worldly otherwise. “Think of a humble Indian”, for example is
an inferential task. Parasiticism distinguishes goals of an action from the policies that are secondary and based upon achieving the goal in the first place. “Win the war without alienating the middle class” contains winning as a goal and not alienating the middle class as a policy.

The above questions and task classifications help to distinguish between various kinds of planning and planner architectures [64]. For example, interacting subproblems imply a need for nonlinear planning [66], where a problem cannot be solved by solving its subproblems in any arbitrary order due to goal interference amongst the subproblems. In unpredictable universes, or “non-simulatable” situations, the effects of an operator cannot be fully chronicled in a symbolic manner. For example, the only way to find out if some source code will compile correctly is to do it. In addition, a domain can be benign or malignant. One might irreversibly compile a piece of code before its time has come, but the effects of such a breach of protocol are tolerable as compared to irreversibly dropping a bomb or inadvertently launching a missile. In benign domains, operation scheduling is only a matter of efficiency. In malignant situations, it is a matter of correctness (which translates to survival, sometimes). Thus planners in malignant domains have to be cautious and have no freedom to guess a possible course of action. In many real-world situations, the completely carefree method of totally interleaving scheduling and execution of actions is recommended, where reversing an action is either unnecessary, a harbinger of bad things anyway, or a facile task.
This section is meant to serve not so much as a compendium of planning information as a background for justifying some of the decisions made subsequently about planning in the CASE domain.

4.3 Tools and Plans

The planning problem in tool support boils down to finding an invocation schedule for tool operations that would attain the stated goal. The difference between this and the general statement about planning is the replacement of the term "operator" by the word "tool". In tool-based environments, tools are considered to be the operators provided by the environment. In addition, a goal is often stated as "achieve a successful invocation of operation X of tool Y on arguments ..." as opposed to "achieve world state s". While the former is less expressive, it is easier to plan with, and will be the primary concern in this chapter. The former will be referred to as the simple tool plan and the latter as the complex tool plan for a certain problem.

If one were to augment the tool representation discussed in chapter 2 with a statement of the effects (goals and side-effects) achieved by tool operation, the analogy between plan operators and tools would be complete. In this section, we deal with the details of how a plan operator and a tool are similar, and the planning problem induced by tool invocation.

A tool consists of as many tool-operation pairs as there are operations defined
for the tool. Each tool-operation pair corresponds to an operator in the plan space. The view arguments needed by the operation can be thought of as preconditions for the invocation of the operation. What is currently missing from the tool representation is a statement of the changes in the state schema effected by a successful tool operation. We represent the postcondition of a successful tool invocation (goal) as class expressions associated with each input view argument that is passed to the tool operation. The class expressions associated with a view collectively specify the domain of influence of the tool on that view. Class expressions specify the effect of the tool operation on instances of a particular (derived) class in the associated view. The effect is specified in a constraint language, and quantified based on whether the class expression applies to all members of the class in the associated view, or only to some of them. Thus class expressions are quantified predicate expressions.

The constraint Language

The rudiments of the constraint language and some associated definitions follow.

- A constraint can be thought of as consisting of a Domain and a Predicate.

The domain of a constraint is a set of attributes of the object with which the constraint is associated. The constraint applies to the values of these attributes and/or their descriptors. Descriptors are entities that describe secondary aspects of an attribute, such as timestamps and sizes. The predicate represents the expression that must be true in the domain for the constraint
to evaluate to true.

- The constraint language allows two kinds of predicates, existential and relational. The EXISTential predicate is a unary predicate whose domain is a single attribute. It requires that a value exist for that property. Relational predicates express relational constraints that must hold over an arbitrary domain. We assume for simplicity that relational constraints span only two attributes.

Definition: A Pod $P=(R,U)$ associated with an attribute $X$ consists of an existential constraint on $X$, a set $R$ of all the relational constraints containing $X$ (or one of its descriptors) in their domain, and a set $U$ containing the union of the domains of all the constraints in a pod. Intuitively, a pod represents all the constraints that must be satisfied during the generation of attribute $X$.

Definition: Every attribute $A$ in a class $C$ is associated with a generator set $G$. $G$ represents all the attributes of $C$ that are used by some tool to generate $A$. A tool uses some attribute $A'$ to generate $A$ iff the existence of $A'$ is a precondition for tool operation and $A$ exists as a postcondition. Intuitively, constraints on an attribute $A$ should be dealt with only after any constraints related to its generators are satisfied.
4.4 The simple planning problem

4.4.1 Basic Steps

The simple planning problem attempts to achieve the goal of a successful tool invocation. The steps are as follows:

1. The initial operator and its arguments are chosen by explicit operation invocation.

2. The goal of successfully applying the operator has the successful derivation of input views as preconditions.

3. The derivation of any single view object can itself be thought of as a hierarchical goal, since successfully deriving any object in the view consists of successfully deriving its subobjects. One can think of a generic DERIVE operator as achieving the derivation for any derived-class base-class pair.

4. The DERIVE goal for any object fails if a constraint associated with that derived class is violated in the derivation process. Constraint checking is itself done based on the heuristics outlined in the last section. A pod is checked simultaneously, and constraints associated with the generator set of an attribute are checked before that of the attribute itself.

5. Violation of constraint(s) sets up further goals of attempting to satisfy the violated constraint(s) by invoking other tools on the object that violated
the constraint. Specific policies for planning further tool invocations are as follows. When a constraint is violated, an appropriate operator(tool) has to be chosen to rectify matters. This is done by choosing the least abstract operator that is compatible with the constraint. Operator application leads to another iteration through steps 1 to 5 ad infinitum(if one is unlucky).

The basic steps illustrated above do not discuss operator selection and constraint-based reasoning in any detail. These are dealt with in the next two subsections.

4.4.2 Operator Selection

Operator Selection can be thought of as a form of means-ends analysis [64], a technique typically used for operator choice in most forms of planning. The central idea in means-ends analysis is to choose an operator that decreases the distance between the current world state and the goal the most. Distance is naturally a figurative term, and the success of means-ends analysis lies in picking a suitable distance function.

In the case of tools, operators are invoked to satisfy violated constraints on some object(s). Thus satisfying the constraint can be viewed as the goal and the selected operator(tool) is one that satisfies the goal(or is most likely to do so). Let constraint C on object O have been violated. The operator selection process should pick a tool that can be invoked on O(i.e on a view derived from its base object, say B) and whose invocation leads to the satisfaction of constraint C.
One way of detecting this is to try and match the stated postconditions of the operator and the constraint that was violated. If it is desired that the environmental assistance be directed, then it is helpful to augment the operator selection process with the least abstract operator policy. This policy states that in case several operators are available that could conceivably satisfy the constraint, the operator that is selected should do as little as possible beyond satisfying the constraint.

4.4.3 Constraint Satisfaction

There are two issues in the constraint-based problem solving outlined above. They pertain to tracking the constraint satisfaction process and handling dependencies, circularity and interactions between constraints. For the simple planning problem we can ignore circularity and interactions between constraints, especially because they rarely seem to occur in tool-based problem solving in the first place. However, it is quite common for the constraints associated with an object to have a partial ordering amongst them based on the generator set relationships mentioned earlier. For example, a module object may have a conjunctive constraint that the source and object code should exist. Obviously, the former is dependent on the latter. One might well ask as to why constraint scheduling is being considered when operator scheduling was deemed unnecessary in the outline of the planning process. The answer relates to the fact that the constraint space pertaining to an
object is small enough and simple enough to be able to compile the schedule.

The constraint scheduling process involves associating a pod dependency table (PDT) and a pod tree (PT) with each object. The pod dependency table represents the dependencies between constraints based on generator set relationships. The pod tree represents the structural hierarchy amongst pods in the pod set pertaining to an object. The pod tree is the expression tree corresponding to the object type it is associated with. The constraint satisfaction terminates successfully if the pod tree has evaluated to true. A pod is said to be free if all its predecessors in the PDT are either satisfied or defunct. A pod is said to be defunct if it is unnecessary to satisfy the pod so as to satisfy the pod tree. For example, in the expression (b OR c OR d), b and c are defunct as soon as d is satisfied.

Any constraint that is free and non-defunct can be scheduled for satisfaction. Constraints are satisfied by means of operator application. The method for choosing a suitable operator were outlined in the previous section.

4.5 Language Primitive

To be able to perform the sort of problem-solving outlined above, it is necessary for the system to understand tool computations in terms of the conditions they require and the effects they have. While the former is expressed by tool views and constraints specifications within views, the latter has not been represented in the language yet. The def-tool-effects primitive introduced below, is used to represent
the effects of (tool, operation) pairs to the environment. Using this information, the tool management component can perform the tool selection step of the constraint satisfaction process and also ensure that a tool has executed successfully.

The syntax of the \textit{def-tool-effects} command is shown below, where "" represents an arbitrary number of repetitions of the appropriate construct.

\begin{verbatim}
(def-tool-effects ToolName

   (OpName

      (OpArg

         (DerClass BaseClass PodSet)

      )

   )

)
\end{verbatim}

4.6 Example

Let us return to the example of a Unit Tester dealt with in chapter 2. There are constraints on the view that a unit tester tool has. These constraints have to be fulfilled before the unit tester can perform the CreateTestLoadModule command, for example. Figure 13 illustrates some of the constraints on the module being tested as well as the modules it "Uses". As outlined in the previous section, these constraints are implicitly partially ordered, and the order needs to be followed for efficient constraint satisfaction. For example, the constraint formula for the "Used"
modules includes constraints on both source code and object code. It consists of both primary constraints (such as EXISTS(object)), and secondary constraints relating to timestamps. Constraints may be satisfiable by the invocation of other tools (Stub-Exists can be fulfilled by a stub generator) or satisfiable only by user activities (as in Source-Tested).

On invocation of the Tester(CreateLoadModule), the derivation of the Tester-View commences. Constraint violation such as the non-existence of object code, non-existence of stubs etc. leads to the invocation of tools to fulfill these constraints. The tool selection and invocation process is termed the Constraint Satisfaction Process. The constraint satisfaction process for this example is illustrated in Figure 14. Figure 14 illustrates the following aspects of the process:

- organization of constraints into pods
- pod ordering by using dependency relationships
- tool selection for constraint satisfaction

All the aspects of the unit tester specification other than a statement of tool effects were discussed in chapter 2. The use of def-tool-effects to state the expected effects of the CreateTestLoadModule command is shown below.

(def-tool-effects UnitTester

   (CreateTestLoadModule
(UnitTestView
(UnitTest Module (Object (Source.ts ≤ Obj.ts)))))
(UseStubs Module (Object (OR (Object.ts ≤ Stub.ts)
(Object.ts ≤ Source.ts)))))
)

4.7 Extensions

Some enhancements can be made to the representation described earlier that are incremental without being revolutionary. These are described here under the headings Strategies and Class Constraints. Strategies are generic rules about tools that can be used to guide the overall reasoning process. Class Constraints describe a category of constraints that apply to whole classes rather than to instances of the class individually.

Class Constraints

In contrast to the kinds of constraints described earlier in this chapter, there exists a class of constraints that has to be evaluated across some or all the elements of a derived class in a view. At present, two varieties of constraints can be identified
FORALL UseModules: (Source-tested OR Stub-Exists) AND
(Object.ts > Source.ts) OR
(Object.ts > Stub.ts)

Figure 13: Constraints on a unit tester view
Figure 14: Constraint Satisfaction Process for "Used" Modules
within the scope of class constraints: *derivational* and *generational*. Both these apply to a particular property in a class.

A *derivational* constraint states that a property A must be derived from the same property B in all instances of derived class D in a view V. Obviously, A and B have to be valid properties of D. This constraint is used in the situation where the *generator set* of B has a cardinality greater than 1. In other words, if B can be generated from one of several other properties in class D, a derivational constraint restrict the derivations that can be used to generate B. Referring back to the example above, one could have a constraint that the Object property for UsedModules be generated either from the Source property or from Stubs, *but not from both* for all UsedModules in the UnitTesterView. In other words, let the module being tested uses two modules A and B which correspond to Ad and Bd in the view. Then we cannot allow the situation in the view where the object code for Ad in the TesterView is generated from source code that is generated by the programmer of module A, while that of Bd is generated from a stub. If all used modules have not been coded and reviewed, the derivational constraint prevents source code from being used for compilation purposes and forces all object code to be generated from stubs.

A *generational constraint* restricts the tools that can be used to generate property B. It could state, for example, that the same tool be used in the generation process for all instances of the class D. For example, one may have a constraint
that the same tool be used for the Unit Testing Process to generate the Object
property, where the tool may be:

- A simple stub generator, one that generates a stub that prints a you are in
  procedure ... message.

- An interpolation-based stub generator, one which picks up a discrete set of
  recommended behaviours (input/output pairs) from the Unit Development
  Folder (UDF) of the module, and generates the stub for the module using
  statistical interpolation techniques to generate a procedure which handles
  inputs other than those prescribed in the UDF.

A subcategory of the generational constraint is the transformational constraint
which states that only tool T may be used in generating property B from property
A. Note that the transformational constraint does not imply a derivational con­
straint. It does not force B to be derived for A only. It merely states that in the
event that B is generated from A, tool T should be used. Some of this is illustrated
in figure 15.

Strategies
During the reasoning process described in this chapter, there might be some con­
cerns that might be constant irrespective of the tool being used, or the exact point
in the problem-solving at which one presently is. These concerns, called strategies
refer to desirable or undesirable tool properties that should be kept in mind during
Class Constraints
(on all instances of
derived class C of
view V)

Derivational Constraint
A property T must be derived from the
same property G for all inst. of C.

Object for all used modules
must be derived from Stub.

Generational Constraint:
The same tool \( \psi \) must be used to
generate the property P for all inst.
of C.

The stub should be
generated using the same
tool (ex: a simple stub
generator or a complex
stub generator).

Transformational Constraint
The same tool \( \psi \) must be used to generate
property P from the same generator property(ies).

Figure 15: Categories of class constraints
the problem-solving process. Strategies are of three kinds: inhibition, prohibition and cost metric.

Inhibition strategies outline properties of tools that deem them undesirable. For example, in a particular computation, tools that are long transactions may be undesirable since they would take too long. If an inhibition strategy is active with respect to a certain tool property, a tool with that property is chosen only as a last resort. A prohibition strategy prevents certain kinds of tools from being used in the chaining process. For example, rather than disallowing non-idempotent actions such as user interaction from being present in tool code, one may just set up a strategy that prohibits interactive tools from being invoked in the reasoning process. Cost metrics specify the relative cost to be associated with a certain tool property during the tool selection step in the process of backward reasoning. One may want to try the most complex tool first in a tool selection step (to attain the best solution to a constraint), or the least complex tool (to attain the quickest solution), or the least time-consuming one (therefore giving preference to batch tools), or interactive tools first (therefore involving the user a lot in the chaining process), or the best-tool only (if we want to be uncompromising about the quality of constraint satisfaction) and so on. The use of strategies is illustrated in figure 16.
Figure 16: User strategies and their use
4.8 Related Ideas

Various environments in the past have had components that provide the sort of intelligent assistance alluded to in this chapter. The vocabulary for expressing coordination plans is very closely tied to the vocabulary of the environment on the one hand, and the ultimate goal of the coordination plan on the other. Some examples of coordination tools are described below.

Make

Make [24], for historical reasons, has been the most popular tool for expressing coordination plans relating to other tools. Make's knowledge about environment objects/files is classified into targets and generators. Tools produce target files form a generator set. Make possesses a rudimentary form of constraint knowledge based on timestamps which it uses as a criterion for tool invocation decisions. A tool is invoked if its target is outdated, i.e., its generator set has been updated since the target was generated.

The above forms of system information are specified in a makefile. The dependencies form a tool graph in which nodes are objects and edges the tool commands that produce them. While this graph is not explicitly represented, the make tool uses this implicit graph to ensure that the timestamp constraint between targets and generator sets is obeyed for all intermediate products of the system generation process as well as the final system module.
The limitations of *make* lie in its rudimentary and coarse-grained knowledge about environment objects, simplistic inferencing process, lack of knowledge about operator effects, and some rather arbitrary representational weaknesses in the way a *makefile* is specified. In addition, the fact that the tool graph is not explicitly represented prevents *make* from checking the integrity of the tool graph in some ways. These are better handled in environments which explicitly represent tool graphs.

**Toolgraph Approaches**

Environments such as TOOLPACK [55], ODIN [11] and KEYSTONE [12] explicitly represent the tool graph that is implicit in *make*. This allows them to discover redundancies, conflicts, and cycles in the tool graph and adopt a suitable approach in dealing with these issues. ODIN for example avoids using a path with cycles, and picks the *cheapest* path to execute if multiple paths are available. It also uses space-time tradeoffs in generating intermediate objects in the system generation process. The user can specify a maximum on the space that can be taken up by the intermediate products, and ODIN deletes derived intermediate objects if the space limit is exceeded using a *least recently used* strategy.

KEYSTONE is an extension of ODIN that deals with the problems of distribution of the coordination plan across several machines. The main addition to the tool graph due to distribution is the addition of *flavors* to the notion of object...
instances. The same tool operating on the same object produces derived objects of different \textit{flavors} based on the machine on which the tool was run. \textit{Flavors} might be incompatible across machines due to low-level architectural differences. On the other hand, if flavors are compatible, then it may be more efficient to create a derived object on a faster machine and move it to a compatible slower machine than to move the source to the slower machine and generate the derived object.

Representing such heuristics about object mobility and the optimal ordering of object motion and derivation requires additional constructs beyond the centralised tool graph language. Some of these issues have been addressed in [12].

\textbf{Other Planning Approaches}

While the planning approach proposed in this chapter is essentially for centralized database-oriented systems, other projects [6] have focussed on the problem of distribution in conventional file-oriented environments. In distributed heterogenous systems, you not only have the "flavors" problem alluded to above, but also a major efficiency problem. This can be remedied by searching for parallelism in the tool plan, i.e to find disjoint goals that can be satisfied in parallel. In addition, the tools differ from types in that they are \textit{physical resources} that may have only a limited number of instances. Tools are software, and thus have \textit{flavors} themselves. Thus, there is a significant knowledge representation problem in representing and using this sort of information about tools.
The combination of derivational, efficiency, parallelism and resource management knowledge leads to inferencing demands well beyond the kind built in to make. In fact, the planner used in [6] uses the most complex and general form of planning, constraint satisfaction.

Configuring distributed applications is one of the more exciting research problems to be tackled in the area of intelligent assistance, and the complexity of the problem would justify environments which are dedicated to providing this form of support. The varying demands of distributed systems make it imperative that intelligent assistants for them be highly tailorable and that their inferencing not be hardwired as in Make and other tools.
CHAPTER V

Implementation

Elements of a language for tool support have been described in the past three chapters. An environment supporting such a language has to provide static support for translating these specifications into lower level code, and provide run-time support for managing the tool space. In a database-centered environment, the tool compiler has to translate the tool specifications discussed in chapter 2 into a database vocabulary of database objects, applications and transactions. Run-time support needs to be provided for the implementation and scheduling of tool invocations by users and other tools.

In a manner similar to conventional languages, we can split the support environment into a tool compiler that compiles the tool language specification into lower-level code, and a tool coordinator which provides run-time support for tool execution. The tool coordinator can be looked upon as a component performing some of the scheduling and communication management functions performed by the operating system in an OS-based environment. In a database-centered environment, the coordinator can be implemented as a component replicated in all
tools but manipulating shared system data. This is in contrast to an operating system which has to be implemented as a single system that coordinates multiple users since there is no underlying database to provide concurrency control mechanisms over a shared data repository. Figure 17 presents a high-level picture of the tool-coordinator idea.

The object code generated by the tool compiler and used by the tool coordinator
consists of *meta-objects* or *tool-schema objects* which represent the environment's knowledge about the tool, *tool objects* that implement the functionality of the tool, and *view objects* which represent the *data* manipulated by the tool. In addition, the desired concurrency and recovery behaviour of the tool can be adjusted by compiling it into *internal* database objects (i.e. that belong in the database) or as *external* objects (applications) linked to the database that have an independent concurrency and recovery behaviour. The exact set of *system* objects generated in the compilation process, and its use by the tool coordinator is described subsequently.

The organisation of the implementation description parallels the organisation of the thesis in that the implementation issues are also described in terms of the *modelling, composition* and *it management* of tools. The run-time support required for an integrated tool environment is not fully described in this thesis. Most of what is discussed are the structures that would be used by a runtime system to regulate and coordinate tool invocation.

### 5.1 Compiling tool specifications

#### 5.1.1 Tools

The specification of a single tool is compiled by the tool compiler into a *tool descriptor* object. This object includes attributes that serve simply as information about the nature of a tool, as well as providing information about system objects that constitute and collectively implement the tool. Included under the latter
category are the identity of object classes and external applications into which
the tool has been compiled, descriptions of tool operations in terms of their argu-
ment views and the objects in the database that implement these views, attributes
providing handles to the objects implementing the tool's properties, and several
objects that implement the mechanics of tool invocation in situations where the
tool is a separate address space.

The tool manager(s) maintain tool tables to maintain run-time information
about tool instances (in case of unshared, instantiable tools) and tool/operation
state in the case of shared tools in which operations need to be scheduled. A local
tool table is maintained by the tool manager associated with each user which keeps
track of instantiations of tools created by this user, and a global tool maintains
information about shared tools and their operations, such as whether the operation
is being currently invoked by any activity and the identity of such an activity.
Figure 19 shows some of the data structures used to implement tools and their
invocation and scheduling. Figure 18 illustrates the tool compilation process.

The discussion above and the data structures shown in figure 19 deal with views
at the level of view objects. In other words, views are assumed to have compiled
into objects and appropriate methods in these objects provide access to necessary
components of the view in question. Aspects of the implementation of views (view
objects) is discussed in section 5.1.2.
Figure 18: A schematics of the Tool Compiler
### Compile-Time Information

#### ToolSpec
- **String ToolName**
- **ToolDesc TDesc**

#### ExecObject
- **String FileName**
- **Oid MailBox**

#### OpDesc
- **String OpName**
- **ArgDesc Args[MAX_ARG]**

#### Tool Description
- **String ToolType**
- **OpDesc Op[MAX_OPS]**
- **OID PropObject**
  - **OneOf**
    - **Oid TypeObj**
    - **ExecObject FName**
    - **ExecObject**
    - **ExArr[Op_Max]**
  - **Impl_Description**

#### ArgDesc
- **String ViewName**
- **Oid ViewObj**

---

### Run-Time Information

#### GlobTableEntry
- **String ToolName**
- **String OpName**
- **STATE ToolOpState**
- **ActId Owner**
- **Oid MailBox**
- **Oid PropObject**

#### LocalTableEntry
- **String ToolName**
- **Oid ToolObject**

---

Figure 19: Data structures in the tool compilation process
5.1.2 Issues in the implementation of views

This section deals with some of the issues that are faced in implementing views on an ORION-like data model.

The basic classes

The basic classes that might be used to implement the view behaviour outlined earlier are Derived, View, and Morphism. Every derived class in a view is implemented with the class derived as a superclass. From derived, it inherits all the properties and methods needed to implement the identity map to a base object, actually derive the derived objects, binding of the necessary morphism(s) and any features needed to solve the problems outlined below. The class Morphism implements the generic behaviour of morphisms. Specifically, it has object pointers to all its clients (currently two only), an “init” method for initialising the derived object structure and a translate method that accepts a snapshot of the operation on any client and translates it to a corresponding operation on the other client(s). It is also beneficial to implement a view class, whose instances represent concrete view objects. An instance of a view object primarily holds a pointer to a derived complex object. The view object is a global object that is known to all the derived objects in the view. It is used to implement catches and throws. The view object maintains the tag table during view derivation and helps with the actual handling of throws. The view object also provides a handle to the root instance of the view, and a changed method which informs one about “dirty” view objects. This might
be used for incremental display updation of view objects.

Forwarding

There is an issue of how one can implement the forwarding behaviour associated with virtual properties (methods and attributes). In such situations, the “forwarder” can be thought of as a proxy or deputy [60, 44] of the object that services the forwarded message. SMALLTALK provides a cute, albeit primitive exception handling mechanism which can be used to elegantly implement forwarding. If a message is not understood by an object, the doesNotUnderstand message is sent by SMALLTALK to the receiver. One can reimplement the doesNotUnderstand message to accept the unhandled message script as an argument and forward it to the appropriate base object. Unfortunately, the error handling model of ORION does not duplicate the doesNotUnderstand feature of Smalltalk. We thus have to adopt the approach of generating forwarding code for each virtual property declaration. The forwarding code is invoked on attempting to access virtual properties. This is clumsy in “pure” OO models that have no encapsulation mechanisms because the implementation is viewable by accessors of the object.

Loopholes for methods

We can view the derived classes as presenting an “abstract” interface of the underlying object and performing “information hiding” in a manner similar to Abstract Data Types (ADTs) in programming languages such as CLU, ADA etc. Thus, like these languages, views have to provide a mechanism for methods (analogous
to operations in ADTs) to go below the interface and be able to view the implementation, which in this case is the base object. We provide a BASE message to method implementors to access the base object corresponding to a derived object, similar to the way SELF is used by methods in an object to access the object itself.

**Copymaps and Derivation Conflicts**

The *copymap problem*, stated in [46] is a problem in deriving graph structures from other graph structures. The problem occurs because an object in the lattice can be the target of a link from many other objects. This leads to two complementary problems. The *identity map ambiguity* problem deals with the fact that naive derivation strategies may lead to multiple derived objects corresponding to one base object. This can happen because a derivation of an object could take place for each path leading to the object. This violates the identity map assumption on which our view theory is based. The *derivation conflict* problem may occur because two parent objects may wish to map the same "sub"object into different derived classes. Notice that these two problems are opposites of each other! Derivation conflict requires that the same base object map to many derived objects to satisfy the needs of all its referrors. A proposed solution to the copymap problem [46] would tag already derived objects and prevent multiple derivations. However, this would disallow the above solution to the derivation conflict problem which requires many derivations of the same underlying object.

We choose to address the copymap ambiguity problem in the manner suggested
by Mittal [46], and look the other way regarding the derivation conflict problem, which is rather like the “join-view” problem in relational databases. Derivation conflicts illustrate the current limitations of our view definition scheme. It should be noted that the limitation is not in the language. The language can express views in which derivation conflicts exist. We have not completely thought out the details of solving this problem on an ORION-like framework.

5.2 Implementation issues in Tool Composition

5.2.1 Cotools

This question has to be dealt with along two dimensions: the nature of concurrency, and the transaction behaviour desired. In the former, one can think of the restricted model of cotools which is quasi-concurrent (coroutines) and suited for local environments or the truly concurrent model suited for multi-user, multimachine environments. The transaction behaviour desired of a cotool may be short transaction for efficiency reasons, or long transactions for longevity reasons.

Architectural restrictions in databases make it difficult, or at any rate unnatural to implement multi-threaded short transactions. We will therefore only discuss the remaining three combinations. Typically, database applications map into a single OS process and a single database transaction. The issue in implementing the basic cotool model, meaning quasi-concurrency and short transaction behaviour, is that of simulating the concurrency model of a cotool (coroutine) in a single
OS process. The issue in implementing the extended model of a cotool (i.e. with truly concurrent member tools) boils down to implementing multi-threaded transactions in a database and using them in straightforward ways to achieve extended cotool behaviour. Again, since quasi-concurrency is a restriction on true concurrency, we will not discuss the details of implementing the quasi-concurrency, long transaction combination. This is easily realised as a restricted form of true concurrency and long transaction combination which is described herein. The short transaction, quasi-concurrency combination has been described under the basic model section, and the long transaction, true concurrency combination is described as an extended model in the next two subsections.

5.2.2 On cotools (simple model)

Cotools are represented in the environment as cotool objects. All Cotool object classes have the class Cotool as a superclass. The class Cotool provides a class attribute Members which holds the identity of the member tools of a cotool, and an instance attribute Assoc which associates member tool names with concrete instances of these tools (represented by Oids of tool objects). The Cotool class also provides a StartUp method for cotools which creates concrete instances of a cotool, and initialises member tools by creating their startup arguments and invoking their startup operations.

As discussed above, the limitations of current databases require that the cotool
be a single (*heavyweight*) OS process due to the limitations of current transaction mechanisms. Thus individual threads of control for member tools need to be *lightweight* processes simulated within the heavyweight process that constitutes the cotool process. This is ironic because simulating lightweight processes (*lwps* or *thread*) in a heavy weight process is considered a "kludgy" way of implementing lightweight processes and adopted only by industrial-strength operating systems such as SunOS4.0. It is frowned upon by systems unfettered by "real-world" concerns like MACH [14] which tend to provide kernel support for lwps.

In *non-preemptive* implementations of lightweight processes, a thread gives up control without terminating by *yielding*. However, the thread relinquishing control has no say in scheduling decisions, which are handled by the *lwpscheduler*. The scheduler uses *thread priorities* in a hardwired scheduling algorithm which is *preemptive across priorities* and *non-preemptive within a priority* [1]. In other words, a thread with a priority lower than the highest cannot be scheduled, and the scheduler chooses a thread non-deterministically amongst the group with the same(highest) priority. The cotool concurrency model is deterministic with control passing from the cotool(a separate thread) to that of a selected member tool and vice-versa. Thus priorities of the cotool thread and member tool threads have to be rearranged at runtime to attain this deterministic behaviour. This means, for example, that all member tools cannot have the same scheduling priority since this introduces non-determinism into the scheduling and would not allow the co-
tool thread to schedule the member tool that it chooses. Some of this is more precisely stated in the cotool pseudocode in Figure 20. The yield points in member tools are not user specified but are decided by the tool compiler policy. One can provide a Yield command in each tool by default, that does nothing if the tool is not a member of a cotool. Alternatively, one can transfer control to the cotool switchboard after every operation in the member tool or have some such hardwired policy for context switching.

The second implementation problem lies in how the cotool decides to context switch between (quasi)concurrent member tools. One way is to have an explicit cotool window in the interface by which the user schedules and the cotool obeys. Another is for the omniscient cotool object to figure out the member tools whose views were changed by the tool that yielded most recently. This can be incorporated into the startup logic because the cotool has access to the properties of all member tools. Thus, it can poke the view-objects of member tools to see if any of the views are dirty (modified since the last view refresh). The two methods can be combined and the cotool can schedule only if the user has no specific opinions on which tool to schedule next. Ideally, scheduling should be done in real estate mode, i.e. the member tool owning the window under the mouse should be activated. However this is not possible because display aspects of views are private to the member tools.
CoTool Object

Properties:

Assoc: array of (TName, Thread_Id, ToolObjId);
View_Assocs: array of (ViewObj, ToolName);

Methods

StartUp(Cotool Args);

CoTool_StartUp_method_pseudoCode

#Set own thread to highest priority; /*Stay running*/

#foreach member in the cotool
    derive tool arguments form cotool arguments
    create a thread and attach the tool function(<ToolName>_FUNC)
    in suspended state
    Note the (thread, toolname, tool instance) association in the Assoc
    property of the cotool.
endfor;

#foreach member in the cotool
    run the start-up method (in the command loop) and yield to the
    cotool startup
endfor;

#display cotool select window
#do
    resume command loop of the tool selected by user
    /* the tool code automatically yields after every command*/
forever;

Figure 20: PseudoCode for a cotool, the basic model
5.2.3 On Cottools (Extended model)

In the basic model in which the problem was to fit the concurrency model into the recovery primitives. The roles are reversed here. The issues in implementing long transactions on top of databases that do not support them has been discussed in great detail in [68]. Concurrency amongst cotool members can be attained by simply making them separate database applications. The issue here lies in being able to fool the long transaction manager into believing that the member tools belong to the same long transaction. The implementation details of long transactions are discussed in barely enough detail to make the rest of the discussion comprehensible. More details can be obtained in [68].

As mentioned before, the checkout behaviour of long transactions results in duplication of the checked-out object if it is to be modified. This allows the modifying transaction to modify the object over long periods of time while allowing other transactions to read the old, outdated copy without having to block, as would have happened in case of short transactions. However, since triggers chains, which are short transactions, should be allowed to access any kind of object, checked-out or otherwise, the conventional locking mechanism has to be overlaid by the long transaction mechanism. In other words, before any object is locked in short transaction mode, the long transaction manner should be consulted to see that the object is not checked out by a user different from the one on whose behalf the
trigger chain is running. This allows trigger chains started off by a user to freely access his objects but obey conventional locking rules if they are accessing another user’s objects.

Since users and their activities are identified by their activity identifiers, all that needs to be done is for the cotool coordinator to start off the various member tools with the same activity identifiers, thus making them seem identical to the lock manager. Any short transaction deadlocks that would occur due to simultaneous trigger chains emanating from different member tools, is assumed to be handled by the deadlock detection and resolution mechanism of the database. Of course, it is assumed that the database has such a mechanism.

5.3 Modelling tool management

In parallel to the description in chapter 4, the implementation issues here can be divided into those concerning basic reasoning (i.e., local reasoning) and those pertaining to extended reasoning (or semi-local and global reasoning). The next two subsections discuss these two forms of reasoning individually.

5.3.1 Basic Reasoning

Fundamentally, the backward chaining paradigm for tool invocation has been studied and implemented in several systems such as MAKE, MARVEL, ODIN and such. The issue here is to dovetail the backward reasoning with view creation so that it is triggered on every constraint violation by invoking system calls to:
• Obtain pods associated with properties for each derived class.

• Obtain alternative generators which can satisfy unsatisfied pods, and apply them in accordance to some prioritising scheme.

• To rederive relevant parts of a view after a tool has been applied to satisfy a constraint. This is needed because a tool T2 may be applied to satisfy constraint C in view V of tool T1. However, since view V is private to tool T1 the changes made by tool T2 on base objects such that constraint C is satisfied for view V are not reflected in the current view V. Parts of V have to be rederived to update it with respect to the underlying database.

• Determine when pods of a derived class are satisfied or defunct, and when the constraints on an object are completely satisfied so as avoid unnecessary computation.

The required system calls are shown in figure 21 and the modified view generation process pseudocode in figure 22.

5.3.2 Extended Reasoning

Extended reasoning encompasses the ideas of class-based reasoning and of global strategies discussed in Chapter 4. The implementation issue in class-based constraints is one of efficiency. Handling a derivational constraint is easy. Let there be a derivational constraint on property B of a class C, namely that it must be
POD(PropName, DerClassName) returns
NULL or PodType; /* returns the pod associated
with a property*/

FREE(PodName) returns True/False;

CURRENT(Oid) returns PodType; /* call returns the
pod that is the current focus of att.*/

PROPS(DerClassName) returns SET_OF(PropName)
/* set of properties assoc. with derived classes*/

GENERATORS(PropName, DerClassName)
returns SET_OF(PropName) /* alternative generators
for a pod/property*/

DEFUNCT(PodType) returns True/False

SATISFIED(Pod, Oid) returns True/False

COMPLETE(DerClassName, Oid) returns True/False
/* computation complete*/

Figure 21: System calls for backward reasoning
Modified Instantiate View method for views:

Instantiate View(ViewName, BaseRootId)
{ Return(Derive(RootClass(ViewName), BaseRootId))); }

Derive Method:

Derive(DClassName, BaseOid) returns DClassType or NULL
{ 
  DObj = Make_Instance(DClasssName, BaseOid);
  Make_Instance(ConstDClassName, DObj);
  while not COMPLETE(Obj)
    for ANY $propname IN DClassName
      SUCH THAT FREE(POD(propname, DClassName), Oid)
        $pod = POD($propname, DClassName);
        if not(DEFUNCT($pod, Oid)
          HERE: Derive_Property(Dobj, $propname);
      
      if not(CONSTRAINT_CHECK($pod, Oid))
        { APPLY(TOOL_SELECT($pod, BaseOid, TYPE(BaseOid)), Dobj))
          UNTIL CONSTRAINT_CHECK($pod, Oid)
            or TOOL_SELECT(...)=NULL;
            longjmp(HERE);
        } else SATISFIED($pod, Dobj);
    end_for;
  end_while;

Figure 22: Modified view generation process
derived from property A only. Then if any constraint in the pod associated with property A is violated, the tool management utility must only fire a tool that derived B from A, even if other ways of deriving B are available. Similarly, a generational constraint is handled by not allowing the invocation of tools that do not fulfill the generational constraint, even if they can generate B to satisfy its pod.

Strategies are even more simply handled. One can merely store the strategy as a globally accessible *Strategy Object* for any iteration of the tool management utility. If a tool's facets matches any of the *prohibit* class facets in the *Strategy Object*, its invocation is prohibited. A similar logic can be applied for inhibit strategies. The tool prioritisation process (represented by the *TOOL-SELECT* procedure in figure 22) can be parametrized to work on a cost criteria specified by the strategy object rather than a hardwired cost criterion.
CHAPTER VI

Thesis Contributions, Future Research

This chapter is organised into two sections dealing with thesis contributions and future research issues respectively. The discussion within these sections is divided identically into subsections on Tool Modelling, Tool Composition and Tool Management respectively. Subsections in section 6.1 partition the contributions of this thesis into the broad categories into which the tool integration problem has been divided in this thesis. Future research issues have also been thought of with the same perspective, and therefore the same classification is reasonable.

6.1 Thesis Contributions

As was mentioned in chapter 1, the thesis assumes as reasonable, the axioms behind the move to Software Development Environments and the resulting changes in environment architecture and associated technologies. The most important architectural change is the move from file systems to databases. The thesis reexamines the tool integration problems in the light of this change.
6.1.1 Tool Modelling

The most significant consequence of the move to databases is the move to a view-oriented model of tools. In file-oriented environments, the internal structure of data objects handled by tools is not modelled. It is either implicit in the object identity (file name) or unnecessary due to the fact that it is private to a single tool. Scoping and access control is handled by operating systems structures (directories) and the tool is considered an extension of its invoker and therefore allowed access to any objects accessible by the invoker.

Databases model the structure of persistent data to a much finer granularity than filesystems by means of a data model. Unlike filesystems which have additional features such as directories to scope and aggregate data, databases are essentially a uniform sea of data. Views in databases serve to model the structure of tool data, and translate between changes to tool data and the underlying data from which they are derived. Thus views serve both to represent what data a tool has access to and what form it sees the data in.

Views may be seen as two-way data translators. The problem of representing views and translating between views and the underlying database is the well known and well worn problem in databases known as the view update problem. Essentially a view definition language can be measured in terms of representational power, conciseness of specification, view-base uniformity, ambiguity and efficiency of translation and commitments made by the view definition scheme on view ma-
terialization. While many of these issues have been addressed for traditional data models such as the relational model, the motivation, form and content for views has not been adequately studied in databases with non-traditional data models such as the object-oriented data model.

The thesis models tools as agents that access the database through views. It proposes an abstraction for tool representation (a tool model) which tailors standard data abstractions mechanisms such as abstract data types or modules to represent the fact that these abstractions interface to the database. This is done by the fact that tool operations manipulate (query and modify) view objects rather than directly manipulating any and all objects in the underlying database. Facets are used to specify orthogonal aspects of the tool that are used by the tool compiler to compile the tool into database primitives, as well as by the tool management utility to measure the cost in terms of time and computation of invoking a tool (operation).

A critical issue that is discussed but not completely solved is the issue of representing tool-user interaction. Issues here include a discussion of formalisms for representing presentation schemes, modelling the tool-user dialogue and efficiency issues of object exchange between the display server and the tool client.

6.1.2 Tool Composition

The discussion in tool composition is divided into cooperation mechanisms and mechanisms for tool nesting. Specifically, the thesis proposes a mechanism known
as a cotool which combines a set of previously specified tools into a cooperative unit. The inspiration for cotools comes from UNIX pipes which allow the composition mechanism to be truly orthogonal to the modelling mechanism, and incremental operation of member tools. Cotools can be thought of as an adaptation of the pipe idea to databases that additionally work at a higher level of abstraction (objects) than pipes. They also allow an infinite fan-in and fan-out of tool configurations. Pipes and other channel-based composition mechanisms limit the number of tools with which a certain tool communicates to the total number of output streams that are associated with the tool. The cotool mechanism allows a tool to communicate with a potentially infinite set of tools.

The issue of nested tools is not completely discussed, but used instead as an excuse to address a well-known database issue, that of nested transactions. Nested tools require nested transactions, but the exact definition of nested transactions needs to be tweaked to be tailored to the tool world. In addition, other forms of nesting which cannot be reasonably included in the general notion of nested transactions (a la Elliot Moss) are discussed. This includes multi-threaded transactions, and mixed-mode transactions.

6.1.3 Tool Management

The idea of tool management, or executing tool plans incrementally and efficiently was first introduced by the MAKE tool. The same philosophy is adopted in the
thesis except to modify the vocabulary to an object-oriented database perspective. This includes moving from files to objects (to views), and from a uniform view of data to a separation between derived data (views) and base data.

The basic backward reasoning paradigm of MAKE has been extended to include the possibility of several ways (tools) of achieving the same goal. In addition, the effects of a tool are explicitly represented and verified. The need for new forms of reasoning has been identified through examples in the domain of Software Engineering. The need for Class Constraints which are semi-local forms of reasoning (as opposed to the local nature of backward reasoning), and strategies which represent global parameters in the reasoning process have been identified. It should be emphasised that the major thrust and contribution of this research is not to find new forms of reasoning but to find a basis and a context to use it in a domain. To that end, the tool management model is pushed only as far as is necessary to handle "believable" examples.

6.2 Future Research

6.2.1 Tool Modelling

The solution taken to the view update problem in this thesis is only a first cut. More powerful and useful approaches need to be discovered to fulfill the demands of software environments. Also, an integrated approach needs to be taken with respect to both class view and instance views. It may be interesting to investigate
the effects of distribution and heterogeneity on view representations as well.

The other big issue in tool modelling that has not been adequately addressed is a theory of user interaction. Some problems faced here are unique to databases, and need to be addressed. Allowing a display server to interface to the database is efficient in that copies need not be made of displayed objects. However, database concurrency control mechanisms do not allow the fine-grained interleaving of object manipulation that is required between the display server and the client application. Enumerating useful facets, and their effect on tool implementations constitutes a useful and unexplored issue.

6.2.2 Tool Composition

It is clear that current techniques for concurrency control are too rigid and brittle to handle the needs of design environments. Some work is being done in the area of Computer Supported Cooperative Work, an area which concentrates on building tools to support loose cooperation amongst groups of human problem-solvers. Concurrency control mechanisms developed in this area [23] need to be adapted to the needs of software development environments. In particular, software development activities are not as serialised as current transaction mechanisms force them to be, nor are they as finely interleaved as CSCW work.

Tool composition provides a declarative way of specifying an aggregate activity. It may be worth investigating whether cotools or any other composition
mechanisms proposed for database-centered environments could extend easily to multi-user, multi-workstation environments. For example, the cotool idea as stated in this thesis would be impractical in distributed situations since it would lead to a lot of network traffic. Whether this is an inherent problem with the idea or with its current variation needs to be looked at. The same goes for nested tools as well.

6.2.3 Tool Management

Future research in tool management needs to deal with enhancements in the kinds of reasoning involved due to multi-user issues, distribution and heterogeneity. Co-operative multi-user environments require knowledge representation formalisms that represent multi-agent problem-solving processes. Specifically, trying to represent this discipline in terms of multi-agent planning seems promising.

Distribution introduces efficiency and resource management issues, while heterogeneity introduces extra constraints such as object-machine incompatibility. In addition, heterogenous environments tend to be multi-language as well, which in turn implies that the tools could be written in multiple languages. Thus tools need to be characterised in terms of the language they are written in, and the language of the objects they deal with. In terms of knowledge representation, the component representing aspects of distribution and heterogeneity is orthogonal to the reasoning component and can therefore be expressed independently. The reasoning however, becomes very complex because the MAKE-like reasoning alluded
to earlier cannot be performed independently of the problem solving needed to handle distribution and heterogeneity. This is evidenced by the fact that the "distributed MAKE" in the AGORA project has to use the most complex constraint satisfaction techniques to fulfill conditions that span several different machines.
BIBLIOGRAPHY


160


[28] A. Goldberg and D. Robson. *Smalltalk-80, the language and its implementation*. Addison-Wesley publishing co., 1983.


