Automatic Generation of Software Design Tools Supporting Semantics of Modelling Techniques

José Artur Ferreira da Silva e Vale Serrano

Submitted for the degree of
Doctor of Philosophy (Ph.D.)

Department of Computing Science
University of Glasgow

June 1997

Abstract

In this dissertation we propose a method to obtain formal high-level specifications of modelling techniques based on graphical notations, such as Dataflow Diagrams (DFD), State Transition Diagrams (STD) or Entity Relationship (ER) diagrams, and from them automatically generate supporting design tools. The generated tools support the visual representation, the semantics and usage of the specified modelling technique.

A formalism, designated as *Visual Concepts*, defines a model comprising a physical component (which includes a visual representation); a semantic component, expressed as constraints; and a usage component, which is related to the checking of the constraints.

The Visual Concepts formalism was used to derive a formal specification language - VC-t. The language has been developed on top of a predicate logic and is aimed at expressing the semantics of modelling techniques. Because it is a formal language, the semantic specifications produced are unambiguous. Complete VC-t specifications have been produced for the modelling techniques DFD, STD and ER.

The usage of the modelling technique may also be expressed in a specification. For example, in a DFD every process must have, at least, one input dataflow and one output dataflow. During the editing of a diagram, it may be useful to allow this constraint to be violated. To express such usage aspects, a novel theory of semantic constraints has been formulated. This theory is used to produce refinements of initial VC-t specifications. In this process, the constraints are labelled following a classification which includes *hard*, *soft*, *hardened* and *deferred*.

From a specification, a design tool is automatically generated by a compiler. The result is an interactive design tool supporting the semantics of the underlying modelling technique. The generated tool can be customised in accordance with the user's level of expertise.

A generic visual language for diagram editing has been created which may be instantiated for a particular modelling technique. The instantiated visual language is used within the generated design tool. It provides non-obtrusive guidance observing the semantics of the underlying modelling technique. The tool user is allowed to produce intermediate inconsistent states to facilitate diagram editing.

A prototype, comprising a compiler for the specification language, a multi-purpose graph tool and a configurable design tool was implemented to prove the feasibility of the complete approach.

The prototype compiler generates executable code. The target of the compiler is the persistent programming language Napier88. The architecture of the compiler has been designed as two separate components: a front-end (the parser), and a back-end (the code generator). This approach gives independence from the target language.

The code generator comprises a number of translation rules to produce Napier88 from a VC-t specification. The prototype compiler only incorporates some of the translation rules to demonstrate that executable Napier88 can be generated automatically. The remainder of the translation rules have been applied manually to produce Napier88, to demonstrate that the rules are correct.

The multi-purpose graph tool, designated as *GraphTool,* can be used with the class of applications using graphs as their main data structures. It provides support for visual and interactive representations of an application. The application acts as a client of the GraphTool. A protocol has been defined to connect the client application to the GraphTool. Any number of visual representations can be associated with the application. Maps are used for this purpose: to change the representation it is only necessary to change the map. The GraphTool includes a constraint manager used to associate constraints with the representations objects and check these at run-time.

The configurable design tool has been developed as a client for the GraphTool. It includes generic components for diagram editing and a drawing canvas. The functionality of the canvas is fully provided by the GraphTool.

Three possible directions for further research are suggested. The *first* relates to the visualization of semantic constraints. Constraints could be specified interactively by means of a visual representation. The visual representation could then be used by the system to show the constraints to a user (possibly using animation). The *second* direction proposes a new approach for code generation that bridges the semantic mismatch existent between the specification and the implementation. This is noticed, for example, in the problem of change propagation from the implementation back to the specification (retro-propagation). This new approach proposes solution to this and other problems. The *third* direction sees visual concepts as components. They become small, reusable units of specification. Following similar principles to those used for reusable code components, specification components would be created. These units of specification could then be stored and reused across a range of applications.

We believe that this dissertation contributes to, and opens new directions in the field (still in need of research effort) of providing frameworks and systems for the specification of modelling techniques and the automatic generation of design tools supporting them.

Dedicated to my parents for their love and support
and to my grandfather, Prof. José Vale Serrano, for initiating me in Science.

Acknowledgments

I would like to thank my supervisor, Ray Welland. It is mainly thanks to him that I was able to write the present dissertation. Every meeting we had resulted in a powerful recharge of my motivation.

My thanks to Richard Cooper, for supporting my enrolment as a Ph.D. student in this department and for his supervision on the first stages of the work. His interests on databases and user interfaces proved very useful throughout the research work.

Malcolm Atkinson has the ability to listen to any problem you might have in your research, and in a glance point out a solution (normally an excellent one). Thank you Malcolm. But beware of Malcolm's ideas: they can be overwhelming.

Many thanks to Alex Bunkenburg for his invaluable suggestions for the formal components of the work.

I must thank Phil Gray for introducing me to the area of Human Computer Interaction. His ideas were always a fruitful source of inspiration.

Darryn Lavery provided me with a bridge to cross from the high-level concepts and theories of programming to the details of systems implementation. It is not easy to find someone with such a holistic view. Thank you Darryn.

Thanks to Paul Philbrow for being able to mediate my frequent conflicts with the computational platform.

I am also thankful to the M.Sc. students Choy Lai Fun and Alvin Lam for testing part of the system and improving its user interface.

I acknowledge the financial support given by the Portuguese governmental institution JNICT in the scope of PRAXIS XXI and the Portuguese research institute INESC.

Table of Contents

1. Introduction and Motivation

The main objective of the research reported in this dissertation is to automatically generate tools to support software design using conceptual Modelling Techniques (MTs). We focus on those techniques which are based on graphical notations, such as: Dataflow Diagrams (DFD), State Transition Diagrams (STD) or Entity Relationship (ER) diagrams. We explore the formal description of these MTs, the automatic generation of diagram editing tools and the ways in which constraints can be applied to guide the user of such tools during the design process.

It is possible to define a *meta-method* that is able to capture the semantics associated with diagram based MTs. The meta-method is supported by a formalism - the *Visual Concepts*, from which a number of *conceptual frameworks* have been conceived in order to capture the following components of a MT: its notation (the representation component); its concepts structure (the semantics component); its dynamics during user interaction (the usage component). Using this meta-method, a MT designer can produce specifications of a given MT. From those specifications it is possible to automatically generate a dedicated design tool to support the MT. The tool's user interface conforms to the specified notation described in the representation component of the specification. A *visual language* is also generated as part of the design tool which, by the use of *constraints*, follows the *semantics* of the MT. A new theory of constraint definition and management makes it possible to determine the process of *usage* of the MT.

Examples are provided in the meta-modelling literature (as discussed in Chapter 3 'Literature Survey', Section 3.4 on page 25) proving that it is possible to define the static aspects (representation and concepts) of a MT. However, few reports are available on the definition of the dynamic aspects. In our context, the dynamic aspects are the implications of the concepts' semantics in the usage of the MT, e.g. which user actions are allowed for a given diagram configuration or how should the design tool guide the user through the editing task.

The dissertation proposes a new approach for the specification of MTs based on diagrammatic representations, such as: State Transition Diagrams (STD), Data Flow Diagrams (DFD), Petri Nets or Entity Relationship (ER). These specifications are to be used in the automatic generation of supportive interactive design tools. A specific-purpose formal language - the VC-t ('VC' stands for Visual Concepts, the formalism in which the language is based and 't' stands for textual) is used to produce the MT specifications. The VC-t language is able to express the semantics of the MTs by the use of constraints (semantic rules based on a form of predicate logic). These specifications are parsed and executable code is automatically generated. The generated code implements dedicated design tools supporting the given MTs. A new visual language in addition to a novel constraint specification and management theory defines the usage of the MTs at diagram design time. The generated design tools may be tailored in the way they enforce the MTs semantics to take into account the level of expertise of the user.

Another way of using this approach is the definition of MTs for specific application domains; we call these Application Specific Modelling Techniques (ASMTs). Several studies [Marttiin95; Wynekoop93] show that some companies find it more profitable to tailor standard MTs to suit their own application domain than to use general purpose commercial products.

This results from the fact that current commercial CASE-Tools are not completely satisfactory. A CASE tool is normally designed to support various MTs integrated in a complete design methodology [Brinkkemper93]. However, diagram editing facilities provided by CASE tools do not provide all the answers to our problems. The design tools are hardwired, hence if a MT has to be tailored to accommodate the needs of a particular domain of application, the CASE tool will not support the changes. Similarly CASE tools cannot be extended to support new MTs, the diagram editors provided by the CASE tools do not have an abstract model of interaction that can be applied across different MTs, they are normally designed to be oriented and exclusively dedicated to a specific modelling technique.

In most cases, developing from scratch a design tool to support an ASMT would not prove cost-effective for the company. This is when an approach such as the one we propose becomes valuable. The approach supports the production of simple, safe, formal, specialized conceptual MTs provided with a graphical representation to obtain diagrammatic models of applications.

To satisfy the requirements expressed above, the new MT should be:

- easy to use and understand (simplicity);
- solid and idiot proof; possible misuses, mistakes and inconsistencies must be readily detected and explained in the form of feedback to the user (safety);
- sound and complete to its purpose; semantics of constructs must be well-defined (formally defined);
- restricted to the specific domain; superfluous constructs must be eliminated (specialized).

There can be, potentially, a new MT for each new application domain. We need, therefore, to investigate the fast, simple and effortless generation of design tools to support MTs.

The goal of the work described in this dissertation is to provide a solution to the problems identified above. The definition of a MT is based on a specifically designed formal language (VC-t) which can be parsed to automatically generate supportive interactive design tools from specifications. The automatic generation process is grounded on a new theory of semantic constraints which constitutes an approach to the definition of MT usage. This dissertation is the result of the research work dedicated to pursue that goal.

1.1 Structure of the Dissertation

The dissertation is structured in four parts together with some appendixes. The first part, the 'Introduction', includes Chapter 1 (the present one) and Chapter 2 'Overview and Architecture' which gives the overall conceptual architecture of the system and a summary of each of its components. Chapter 3 is a literature survey of the areas related to this dissertation which focuses on other research work that provided the bases and the inspiration for our approach.

Chapter 4 'The Visual Concepts Formalism' starts the second part of the dissertation - the 'Theoretical Foundations'. Chapter 4 describes the formalism from which the three conceptual frameworks that support the whole dissertation were derived; the three following chapters each describe a different conceptual framework. Chapter 5 'Specifying the Semantics of Modelling Techniques' explains the VC-t specification language; it includes a tutorial for the language and a description of its formal aspects. Chapter 6 'The Generic Visual Language for MT Editors' concerns the way diagrams are interactively built and how the modelling task is supported by a design tool. Chapter 7 'Specifying Usage by Semantic Constraints' describes a new theory for the classification and specification of constraints and how they are used to guide the user through the editing process in accordance with the semantics of the underlying MT.

The third part of the dissertation is dedicated to 'System Design and Implementations', it begins with Chapter 8 'Compilation of Specifications and Automatic Code Generation'. A compiler was implemented for the VC-t language, it features two individual parts: a front-end, that includes a lexical analyser, a parser (syntactical analyser) and a semantics analyser for type checking and scope rules; and a back-end, which consists of a code generator. The compiler and all the automatic code generation process is presented in Chapter 8. A generic tool to support interactive graph based applications is presented in Chapter 9 'The GraphTool'. The generated design tools use the facilities provided by the GraphTool. The third part of the dissertation ends with Chapter 10 'A Prototype' which includes a section on the technology used, with: the Napier88 persistent language and its programming environment; the UIMS (User Interface Management System) supporting the prototype's interactive features - TkWin; the Lex and YACC Unix tools. Next in the chapter is the current status of implementation and finally an example of a design session, from the VC-t specification of a MT to the generation of a supporting design tool.

The 'Conclusion', the fourth part of the dissertation, includes Chapter 11 where the contributions given by this dissertation are outlined, the limitations of the approach are stated, some improvements are suggested and exciting future research directions are identified.

The following appendixes are included. Appendix A 'Guide to the VC-t Syntax' gives a simple description of the VC-t syntax. Appendix B 'Complete VC-t Specifications of Modelling Techniques' where specifications for DFD, STD, ER and OMT's Objects Diagram are included. Appendix C 'Types For The Representation Level - A Generic Lexicon For Interactive Graph Based Systems' presents a generic collection of type declarations, which are used by the system's representation level but may also be used by any interactive graph based system. Appendix D 'The Compiler's Front-End: a Parser for the VC-t Specification Language' consists of the lexical and syntax specifications (for Lex and YACC, respectively). Appendix E shows the code generated for a VC-t specification of STN.

1.2 A Note on Styles

The following styles are used in the dissertation:

- *italic* - used to introduce or refer to a concept;
- *italic* is also used to emphasize a part of the text;
- **bold** - definitions;
- 'single quotes' - denote a term, as in: we will use the term 'designer' to refer to the person who writes VC specifications;
- 'single quotes' is also used to include part of a text (citation) or specification, as in: the constraint 'entities must be named' must now be enforced;
- 'single quotes in the font used for code' - executable code;
- "double quotes" - a string, as in: the label on the icon is "Person".
- [brackets] - a bibliographic reference, as in: [Welland90]

2. Overview and Architecture

The overview includes three sections: *writing of a formal specification* for a Modelling Technique (MT), which includes describing its semantics and tailoring its usage, i.e. how a user can produce diagrams with the MT; the *generation of a design tool* to support the specified MT; and finally the *run-time operation* of the generated system and the way the final user interacts with it.

The three sections correspond to the stages that a method designer must go through to obtain a working design tool from a formal specification of a given MT (see Figure 1).

The *first stage* consists, firstly, in using the VC-t formal specification language to describe the semantics of the particular MT. Next, the designer may choose to tailor the usage of the specified MT by classifying the semantic constraints in terms of their checking and enforcement. This subject is covered by a novel theory of semantic constraints which allows the designer to refine the VC-t specifications. This means that the designer may determine how a diagram is drawn by a MT user. For example, the designer can determine that the objects of a particular VC type must be created before the objects of another VC type; or that a property (e.g. 'name') of some VC type must be given a value when an object of that type is created. The *second stage* consists of the compilation of the VC-t specification produced in the first stage. The output of the compiler is a Napier88 program that implements a design tool supporting the underlying MT. In the *third stage* the designer will test the generated design tool. The tool is analysed to certify that it correctly supports the semantics of the MT.

The production of a design tool is an iterative process. The designer may change the specification by returning to the first stage. A new compilation takes place and a design tool is generated which is then tested.

Figure 1 shows the architecture from a generation-time perspective (aspects regarding run-time operation are omitted). Each component of the architecture is described in a section of this chapter. A brief explanation of how the system works at run-time is given in Section 2.3 on page 15.

2.1 Producing Specifications for Modelling Techniques' Semantics and Tailoring Their Usage

A new formalism to express diagram based MTs - the Visual Concepts - is presented. A generic visual language to be used in the tools supporting MTs and a specification language to express the semantics of MTs, have been defined based on that formalism. A first

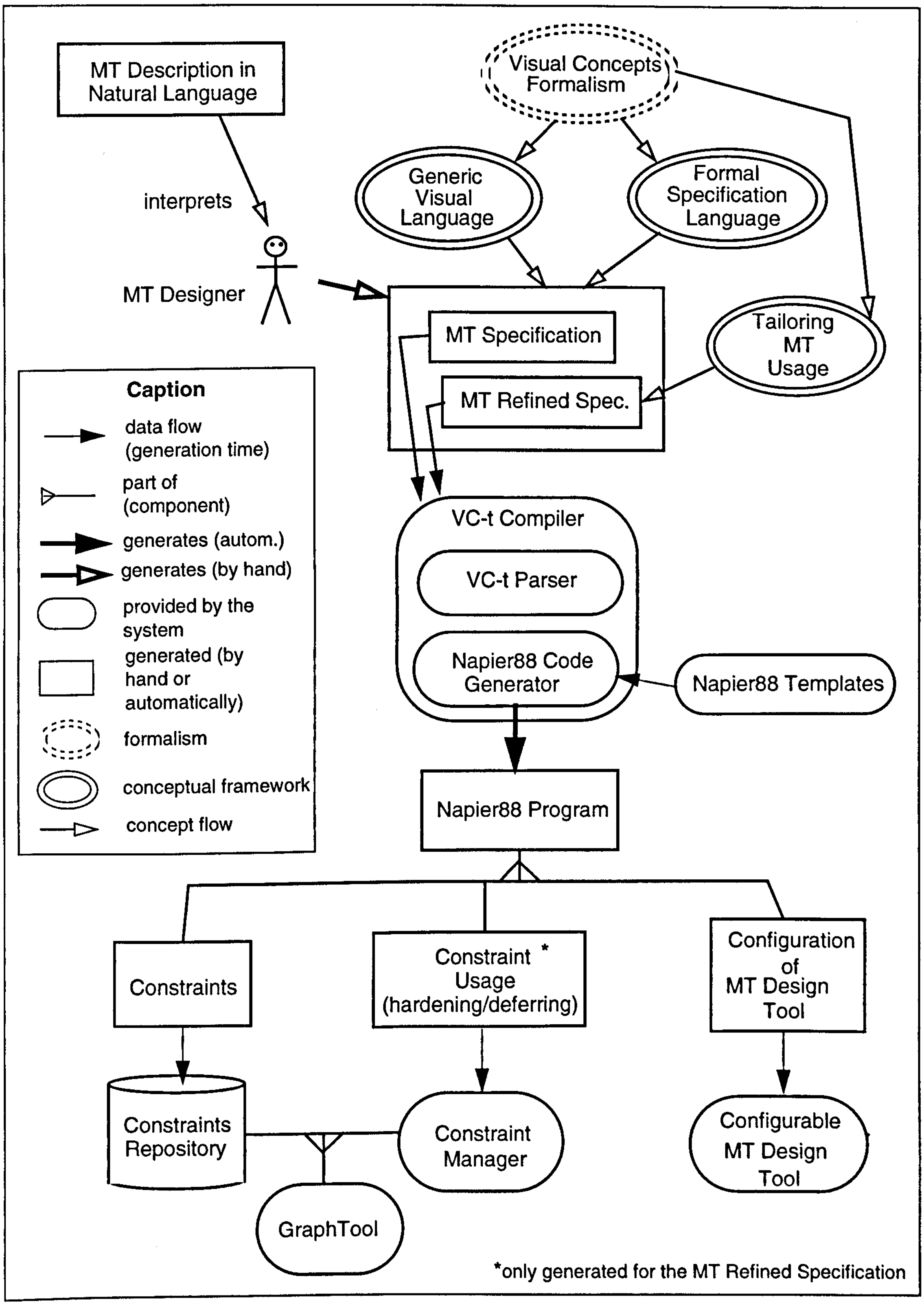


FIGURE 1. The overall conceptual architecture at generation-time.

specification for a MT can obtained with the specification language, however it does not include information regarding the way of using the MT - only the default usage is implicit in the specification. To allow the MT designer to tailor the usage, a theory of semantic constraints has been developed for the refinement of a MT specification. The production of a MT specification is an iterative process in which each new specification obtained can be parsed and, from it, a prototype can be generated.

2.1.1 The Visual Concepts Formalism

The new formalism, *Visual Concepts*, is able to describe MTs such as the Entity-Relationship (ER) technique, State Transition Diagrams (STD), Dataflow Diagrams (DFD) as well as propriety techniques developed for a specific purpose. It is based on Visual Concepts (VCs), which are typed units of specification encapsulating information on the physical, semantic and usage components.

The VC model is based on the use of constraints to express its semantics. They are called *semantic constraints* and are rules, on the VCs' properties or the whole diagram, that can be checked for validity and, optionally, enforced. An implementation of the VC formalism, called Visual Objects, to be used in the system's prototype, has been produced. Detailed presentation in Section 4.4 on page 42.

2.1.2 A Generic Visual Language to Draw MT Diagrams

We must define how the final user can produce diagrams with the design tool. For this purpose, a visual language is needed. The drawing process will be directed by the semantics of the underlying MT. As MTs have different semantics, there will be as many visual languages as there are MTs. However, there are commonalities amongst the various visual languages for they are all based on the same diagrammatic layout and all of them are implemented by semantic constraints. It is then possible to define a *generic visual language* which can be instantiated for each particular MT.

It is then possible to produce a visual language for a given MT by instantiating the generic visual language with the set of semantic constraints defined for that MT. However, current diagram editing tools based on constraints always display a trade-off: on one side the constraints are enforced to maintain the consistency of the diagram; on the other side the constraints force the user to follow a given path on the design process, limiting his/her freedom. If all the constraints are enforced at design time the user is unable to introduce any inconsistency in the diagram. In a normal editing process, the user deliberately introduces some temporary inconsistencies, e.g. delaying the specification of the attributes in an ER diagram until several entities have been drawn. If this is not allowed by the system, the user will have the sensation of wearing a straitjacket. However, if no constraints are enforced it may provoke the user to feel lost in the design task, without any semantic guidance. A decision on the number of constraints to be enforced at design time is then made to achieve a balance between the two extreme situations; as a result, a number of constraints are relaxed (deferred until explicit user request for a total diagram check). We believe this approach is inadequate and we present a revolutionary solution to that problem.

The proposed generic visual language is based on the VC formalism and, more specifically, on the Visual Objects implementation of it. It is able to embed the full semantics of a MT. Yet, it does not limit the user freedom during the editing task. Detailed presentation in Chapter 6 'The Generic Visual Language for MT Editors'.

2.1.3 A Formal Specification Language for Visual Concepts Semantics

In order to build specifications of MTs, a language for the Visual Concepts formalism has been designed and its syntax formally specified - the *VC-t Specification Language*. Specifications obtained with this language can be parsed, and MTs' dedicated design tools automatically generated from these specifications. A number of requirements for this language have been identified and are summarized in what follows.

The language must have a scope wide enough to cover a meaningful range of MTs. By a 'meaningful range of MTs' we mean a selection of widely used MTs which cover both static/architectural and dynamic aspects of application domains, using different paradigms (e.g. Semantic Data Modelling and Object Orientation) and applied in a broad variety of contexts (e.g. database applications, process modelling, hardware design). It must be expressive enough to capture the semantics of the MT; this is done by the use of constraints. The specifications obtained with it should be readable. It has to be more than a theoretical exercise - it must be practicable and simple to use.

A specification of a MT, obtained with the VC-t language, includes two main sections: the 'Preamble', where all the declarations are made (this includes the VCs and their properties); and the 'Semantic Constraints', where each constraint is specified by a statement in a form of predicate logic with equality.

In the specification, a natural language description is associated with each constraint; its main purpose is to give semantic feedback to the user of the design tools which are generated at the end. They also add legibility to the specification.

The Lex and Yacc Unix tools were used in the construction of a compiler for the VC-t language. To produce the compiler we obtained a formal description of the language in BNF. The complete Lex specification and Yacc specification (the BNF description) are included in Appendix D.

Specifications obtained with the VC-t language are, at the same time, formal descriptions of MTs and high-level descriptions of the computational system that is generated to support the MT. These descriptions have advantages when compared to natural language ones: they are formal, therefore they have a well defined meaning and consequently they are unambiguous; the VC-t language has explicitly been developed to express MTs, thus, being specific, it is expected that the specifications are more concise and have a more consistent structure and terminology than a specification obtained with a highly general-purpose language like the natural one. These advantages are important when producing specifications to be used as input to a computational system; still, from the user point of view, they can be totally negated by a single disadvantage - lack of readability. It is important that the specifications may be understood and used as a means of sharing information within a team. This will also allow for them to become an insight to the generated tools. In the design of the language this has also been taken into account.

Formality must be employed only as a mathematical tool to guide the design of the language, never as an end in itself. The main goal is to obtain a language that may be used by someone who is already able to specify a MT using natural language; the use of a special purpose language should not make the task of specifying a MT more complex.

The readability, the easy understanding and easy production of specifications were tested by using the language to specify a number of established and well known MTs. VC-t specifications were produced for the following MTs: DFD and STD (both techniques are used to express dynamic aspects of a universe of discourse); and ER (a Semantic Data Modelling technique). The complete specifications are included in Appendix B on page 159. Detailed presentation in Chapter 5 'Specifying the Semantics of Modelling Techniques'.

2.1.4 Tailoring the Usage of a MT

The simple enumeration of the semantic constraints on a MT specification obtained with the VC-t language, as presented in the previous section, is not enough to express the usage of that MT. For example, if the MT designer decides to specify that, for a STD, the 'initial state' must be the first state to be created in a diagram, or that writing the names of states can be delayed until several states have been drawn, then a more expressive constraint based specification must be used. The initial VC-t specification determines a default usage; but for a more elaborated definition of the way a user may draw a diagram, a configurable constraint checking mechanism becomes necessary.

A novel theory of semantic constraints has been elaborated to allow the designer to express the usage of a MT. The designer specifies how the constraints are enforced to define the possible drawing paths a user may follow when producing a diagram. Drawing paths are sequences of user actions and they are determined by the constraints that are enforced for each diagram configuration.

Refined MT Specifications

As mentioned above, the initial VC-t specification determines a default MT usage. This means that a design tool can be produced from it which provides a default usage, yet it will only be suitable for debugging purposes. A description of the MT usage must be given by the designer, so that a truly useful design tool can be produced.

The new theory provides a mechanism for the refinement of VC-t specifications. The constraints may be labelled as 'hard', 'soft', 'hardened' and 'deferred'. The VC-t language has been extended to support these constraint classes. The refined specifications can also be parsed but now design tools which incorporate the tailored usage of the MT are generated.

This theory gives the possibility of tailoring the usage of a MT. Moreover, it allows constraint checking to be configured at editing-time in accordance to the expertise of the user by providing a mechanism to enforce, or defer the enforcement, of constraints. Detailed presentation in Chapter 7 'Specifying Usage by Semantic Constraints'.

2.1.5 The Iterative MT Design Process

Once the description in natural language of the MT has been interpreted and translated

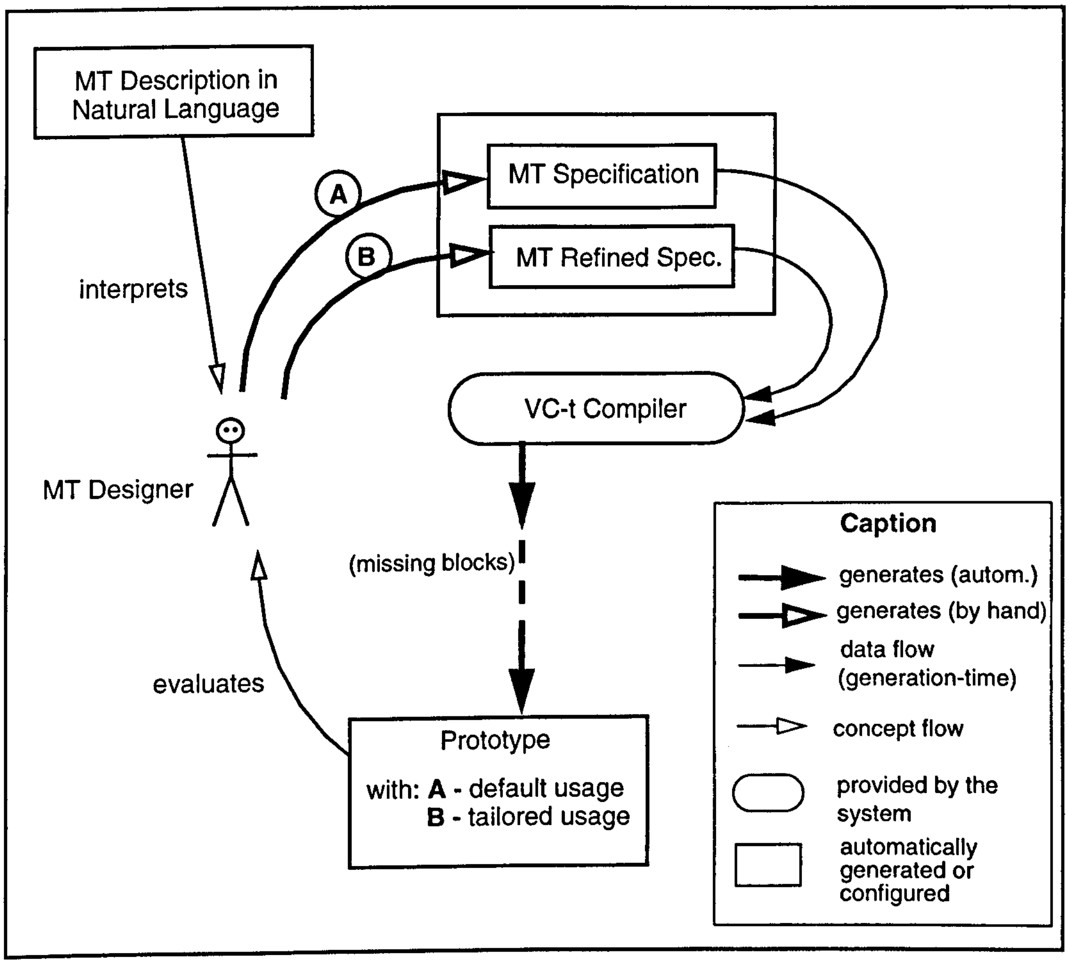


FIGURE 2. The iterative design process.

into a formal specification by the designer, a first prototype can be generated from it. The designer may then evaluate the prototype and, if necessary, change the specification and generate a new prototype. This process can be repeated any number of times until a suitable prototype is generated.This iterative process is represented in Figure 2 by the cycle marked 'A'.

The interaction with prototypes generated in the cycle 'A' conforms to a default usage of the underlying MT. After a satisfactory prototype with a default usage has been obtained within cycle 'A', the designer might want to tailor the usage of the MT. This is done by refining the specification produced in cycle 'A'. Information concerning constraint checking and enforcement is included in the specification - the designer has entered cycle 'B'. Within cycle 'B' the same iterative process explained above for cycle 'A', can be performed by the designer. The design process stops when a satisfactory system is obtained.

It is important to notice that all the changes within the iterative cycles for prototype improvement are carried out at the design level, not at the generated code level. The system obtained by this iterative process may satisfy all the requirements, but it might happen that it still needs further improvements or some extra features. In this situation it is necessary to change by hand the code generated automatically. Our approach does not provide mechanisms to automatically propagate changes made to the code (at the implementation level), up to the specification (at the design level). A possible direction to the solution of this nasty problem is pointed out in Chapter 11.

2.2 Generating Design Tools for Modelling Techniques

Once the MT designer obtains an initial or refined version of a specification, it can be used as input to the automatic generation process. The MT specification is compiled and Napier88 code is generated. Not all the code necessary to implement the design tool is generated; part of it is provided as templates and part is provided as configurable tools. The aspects related to the compilation process, constraint management and configurable tools are outlined below.

2.2.1 The VC-t Compiler

One of the basic guidelines of this thesis is to provide widely applicable theoretical and practical frameworks, rather than closed architectures and systems. The design of the compiler follows that guideline in that it can be applied to different technologies and programming languages.

The compiler is composed of a front-end and a back-end. The front-end is a parser for MT specifications; the back-end is a Napier88 code generator. The front-end is completely independent from the back-end and, therefore, from the particular programming language generated. If instead of Napier88, another output language is to be used, only the back-end has to be replaced. For output programming languages which offer similar constructs, the back-end will be based on similar sets of translation rules (discussed below), thus facilitating its replacement. Detailed presentation in Chapter 8 'Compilation of Specifications and Automatic Code Generation'. Chapter 8 discusses the compilation process and the principles of code generation.

Parsing MT Specifications

A parser for the VC-t language has been implemented based on the BNF description of its grammar. The Unix tools Lex and YACC have been used. Correctness checks are done at the lexical, syntactical and semantics levels. The semantics level refers here to those aspects related with type checking and variables' scope.

Generating Napier88 Code

A goal we have tried to achieve is that the generated code must be human-readable. By *readable code* we mean that it not only has a nice layout, but also that it may be easily understood by a Napier88 programmer.

The generated Napier88 coded is guaranteed to be correct, but it is far from being optimal. At this stage we want primarily to provide a prototype that works correctly and demonstrates the ideas stated in the thesis without trying to achieve any performance goals.

Not all the necessary Napier88 code is automatically generated, part of it is *hard-wired* in the system to be used in the automatic generation process. This includes *templates*, structures with chunks of pre-written code and place-holders to be filled with the automatically generated code; and *configurable modelling tools* tailored by the code generator with MT specific code.

Because the generated tools use persistent technology, both the data implementing the MT constructs (metadata), namely Icons and Connections and associated constraints; and the data produced with the generated design tools, can be made persistent. Therefore, the metadata-dictionary and data-dictionary found in traditional modelling systems, are replaced by repositories constituted by the data structures from which metadata and data are instantiated.

The code generation is guided by a set of translation rules that map syntactical constructs of the VC-t specification language to Napier88. A set of rules have been defined, for instance there is a rule which applies to the VC-t production rule *naturalExpression naturalComparison naturalExpression* defined for the non-terminal *simpleBooleanExpression*. A description of the complete set of rules and their application in the generation of the code for a design tool is given in Section 8.3.3 on page 99.

We have identified some general principles in the translation rules and interesting problems in the code generation process. The correctness and simplicity of the translation were our main concerns; the efficiency of the generated code was left for future stages of development.

2.2.2 Checking Semantic Constraints

A diagram operation is a transformation of the presentation (the visual component of the design tool's User Interface (UI)) resulting from a user command. A diagram has an underlying graph structure, thus the following classification of operations available to the user: *graph preserving* operations and *graph altering operations*. In graph preserving operations only the layout of the graph is changed; for instance moving an icon or changing the font of a label. In graph altering operations both the layout and the structure of the graph are changed; for instance, deleting or creating a connection.

Constraints are checked after any graph altering operation on a diagram object; so their satisfaction can be seen as a post-condition on the operation. Constraints can be classified according to the kind of diagram objects they apply to: icons, connections or labels belonging to either icons or connections.

Two problems must be solved in constraint management: determining which constraints must be checked for each diagram operation, and determining which constraints must be referred to by each VC type.

The complexity of these problems can be largely reduced when a number of diagram drawing assumptions are considered. E.g. 'the operation icon deletion provokes the deletion of all connections glued to the deleted icon'. This assumption allows us to reduce the number of constraints referred to by the icon that would otherwise be checked both for the deletion of the icon and the deletion of the connection. A detailed and formal study of these problems is presented in Section 8.4.

2.2.3 The GraphTool

The GraphTool aims at supporting the development of applications based on diagrams which are visual representations of graph structures. The tool provides generic facilities to manipulate graphs and their visual representations. It includes a library with standard graph operations, a repository for Visual Objects (the classes of diagram constructs, which can be Icons or Connections), a configurable representation manager, a repository for constraints and a constraint manager to express Visual Objects semantics.

The architecture of a GraphTool based application comprises three levels: the application level, consisting of the application specific code; the representation level, where visual representations for graph structures with user interaction mechanisms are defined; the abstract graph level, holding the graph data structures.

Several representations can be associated with the same graph. The connection between graph and representations is supported by map structures, one map for each representation. Changing the current map will change the representation.

The procedures to manipulate the graph structures and their representations are parameterized with application defined types, for this reason they cannot be provided as a library. To solve this problem, a number of generators (procedures that generate other procedures) have been included in the GraphTool. These generators provide a default functionality. They define for instance the creation of an icon or a connection, or the way a Visual Object can be moved by the user. Any generator may, however, be replaced by a user defined one; this way the GraphTool is fully configurable by the user.

In order to connect the GraphTool to the application in hand, a three element protocol has been defined; it includes: an interactive drawing canvas; a number of procedures defined in the structure*GraphToolFunctionality* which act as functionality connectors; and a callback mechanism allowing the GraphTool to execute procedures provided by the application.

The Visual Objects, i.e. the components of the diagram displayed on the canvas, besides their appearance, also have semantics; this is implemented as a set of constraints which are stored in a constraint repository. A constraint manager is included in the GraphTool to check, validate and enforce the constraints over the Visual Objects at editing-time.

As the GraphTool was implemented on top of a persistent system [Morrison94], it offers support for diagram persistence. Diagrams are inserted in a store, incrementally, during the drawing process. Note that when a repository is mentioned, such as the constraint repository, it means a sub-section in the global hierarchical structure of the persistent stable store (in Napier88 these sub-sections are called *environments*).

Although the GraphTool had been developed as part of the prototype implemented to demonstrate the ideas on this thesis, it can be used in a large variety of contexts. We believe that reusability and genericity should always be a concern in software design. Detailed presentation in Chapter 9 ‘The GraphTool’.

2.2.4 The Configurable MT Design Tool

A Configurable MT Design Tool is provided by the system. It can be configured to support any MT whose specification is used as input to the generation process.

The tool includes an interactive drawing canvas for diagram editing, a palette of icons and connections with an import mechanism to load new Visual Objects representations, a text display window for feedback on constraint violations, and a menu-bar with standard editing commands.

The tool does not provide the functionality for diagram editing; instead it acts as a front-end to the GraphTool by establishing links to its functionality connectors. The configuration is done at two levels: appearance, by importing the appropriate set of Visual Objects representations to the palette; functionality, by establishing the links to the appropriate GraphTool functionality connectors. This tool was obtained from the project described in [Fun96].

2.3 How the System Operates at Run-Time

When a VC-t specification of a MT, provided by the MT designer, is successfully parsed, a working system is generated. In Figure 3 is represented the architecture of a working system.

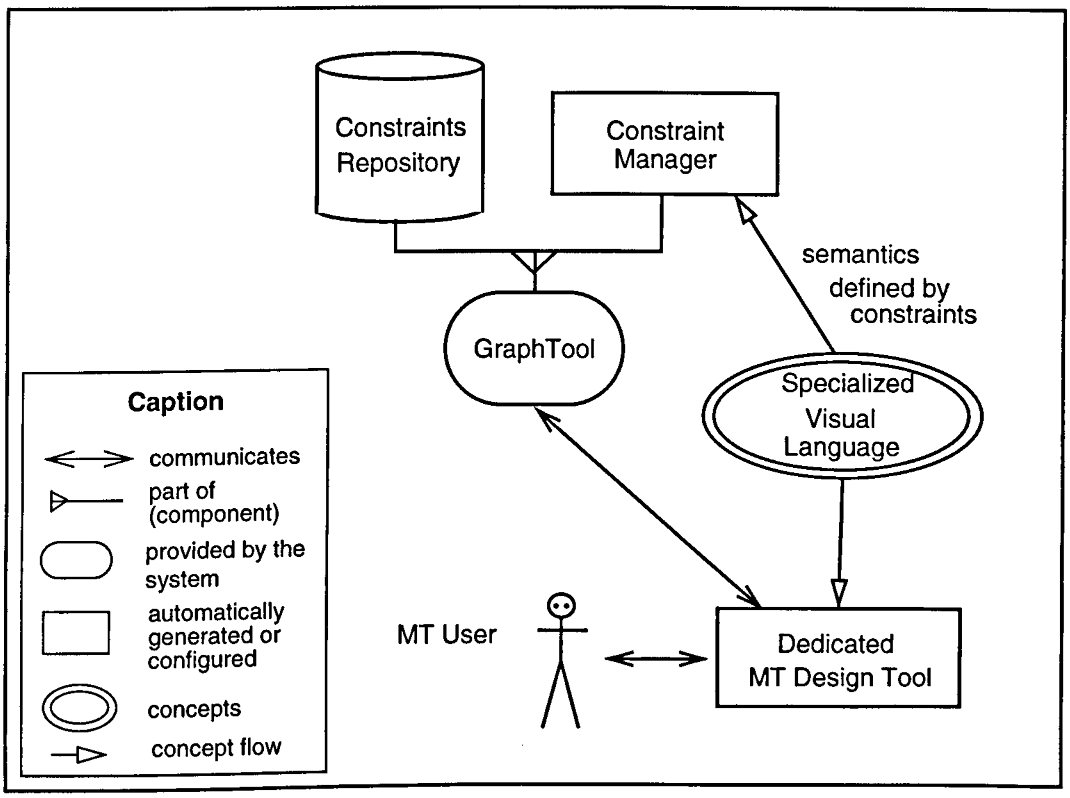


FIGURE 3. The system at run-time (editing a MT diagram).

The working system comprises:

- a *Constraints Repository* where the generated constraints are stored, a *Constraint Manager* to determine which constraints are to be checked for each diagram editing situation (both are part of the GraphTool);
- an instantiation of the Generic Visual Language, called the *Specialized Visual Language;*
- a Dedicated MT Design Tool where diagrams can be interactively edited.

2.3.1 Obtaining and Testing the Working System

The MT designer must test the obtained system to ensure that the constraints reflect the semantics of the MT being specified. Testing is done by experimenting with the design tool. Because the VC-t is based on predicate logic it would be possible to use a system to prove the correctness of a specification. However, this is out of the scope of our work.

After each iteration in the design cycle a working system is obtained. It is important to notice that the generated system is not a prototype in the sense that it only implements part of the functionality of the system to be obtained as a final product. In fact, the final product is just another refinement (the last one) of the system produced iteratively during the design process. Once a system is generated that upon testing appears to meet the requirements, i.e. it completely and correctly reflects the semantics of the specified MT, then it is called 'the final product'.

2.3.2 Interfacing With the User

The GraphTool implements the graphical appearance and the lower level of the UI (the dialogue) of the Dedicated MT Design Tool, which consists of commands for visual graph manipulation, such as *create icon* or *delete connection.*

The higher level of the UI (the functionality according to the MT semantics), is implemented by the constraint manager. The functionality defines, for instance, when and under which circumstances the creation of a 'transition' in a STD may occur.

2.3.3 A Dedicated MT Design Tool

The Dedicated MT Design Tool is obtained from the Configurable MT Design Tool presented in Section 2.2.4. The design tool conforms to the input specification, written by the designer, for the underlying MT. This design tool includes the icons and connections depicting the MT concepts and also incorporates a *specialized visual language* which is specific to the MT semantics.

The generated design tool can be used to edit a diagram corresponding to the specified MT. When a constraint is violated, the user will always be given some kind of feedback. If a soft constraint is violated a visual feedback is given by the design tool by highlighting the Visual Object(s) involved; if the violated constraint is *hard* then a message is issued by the system showing the constraint description in natural language (which is given by the designer in the VC-t specification).

A detailed presentation of the implemented prototype is given in Chapter 10.

2.4 Evaluation

As with its overview, the evaluation of the approach must also be carried out in three levels:

- obtaining specifications: expressiveness and scope of VC-t language; human-readability of the specifications - this was proved by producing concrete examples, such as specifications for DFD, STD and ER.
- automatic code generation: correctness of code and consistency with specification - proved by compiling and executing generated code for the specifications above and by, afterwards, testing the generated design tools for correctness and consistency with the specification.
- design tools: correctness of appearance and fulfilment of usability requirements - proved by observation of related work and analysis of the reported problems. By proving that the generated design tools solve those problems we are able to state that they meet the desired requirements.

It is important to stress that we are not interested in producing a system with an excellent UI including eye catching and elaborated interaction features, such as rubber-banding, snap-into-grid or 3D feel. These features can improve the usability of the system but they are not relevant to the research topics covered in this thesis. Moreover, they can always be added later to the system, with a small impact to it. We are, in fact, interested in producing a system that conforms to the semantics of the underlying MT, and which functionality mirrors that semantics in a non intrusive way, by providing a new visual language based on semantic constraints. In addition, the system includes a user-configurable constraint setting and gives the user adequate semantic feedback.

3. Literature Survey

3.1 Introduction

One of the hard problems about doing a literature survey in the area of software engineering, specifically when it covers several disciplines such as conceptual modelling, graph-grammars and constraint based systems, is coping with the diversity of terms and concepts used by the different research communities and even inside the same community. As an effort to be as consistent as we can when writing about related research, we will try to preserve the terminology used in the surveyed literature, but also mention the term adopted in this dissertation whenever we find this necessary. In what follows we include some text extracted from the literature; this text is delimited by single quotes. All the claims and statements are based on the publications cited; whenever this is not the case it will be explicitly mentioned.

3.2 Evolution From the ECLIPSE Design Editing System

The design editing system we will examine in this section was developed as part of the ECLIPSE system [Bott89]. ECLIPSE is a result of the Alvey Software Engineering Programme; it consists of an Integrated Project Support Environment (IPSE). One of the main characteristics of an IPSE is the presence of a central database (repository) which holds all information concerning any given project and through which all the tools communicate data.

Being an IPSE, the ECLIPSE Design Editing System [Welland88; Beer88], which will be designated ECLIPSE-DE from now on, was developed subject to a main requirement: the produced design diagrams should be stored and manipulated in the ECLIPSE database.

Further requirements were the ability to support multiple methods and the provision of diagram checking during the design task. It is here that the ECLIPSE-DE can be compared with our approach. We too intend to achieve a generic approach which is applicable across a range of methods (or modelling techniques) and that supports the designer during the modelling task.

In what follows, a brief analysis of those aspects that can be related between the two approaches is given. The objective of this section is not to provide a description of the ECLIPSE-DE but rather to identify some research directions that might not have been fully developed in that system and how we think the VC approach can provide a possible solution.

3.2.1 The Description Language

The language used in the ECLIPSE-DE to describe a design methods is called GDL (Graph Description Language). In the same way as our VC-t language, GDL is used to describe those methods which exhibit a node&link graph structure. Both nodes and links are typed; they are distinguished by their visual representation. GDL includes the feature of allowing new types to be derived from pre-defined ones; thus giving rise to a type hierarchy.

One of the requirements that guided the design of the VC-t language was ‘simplicity’. An immediate advantage is the ease of use of the language. It is also easier to reason about a simple, clear and non-ambiguous specification. Non ambiguity is a direct result of applying formality to the language design.

GDL is a very expressive language, but we believe ‘expressiveness’ should not compromise ‘simplicity’ in a context where the goal is to facilitate the development of design tools - not make it more difficult. The specifications obtained with GDL seem far more complex than the those obtained with VC-t. Two possible reasons for this to happen can be pointed-out. The first one is that VC-t is supported by a very simple formal mathematical basis, namely elementary set theory is used to describe VC types and a form of predicate logic with equality is used for the semantic constraints. An explicit formalization of GDL is not given in the literature. The second reason is that the main purpose of VC-t is to be able to express the semantics of a modelling technique, whereas GDL indiscriminately expresses semantic aspects, layout information, e.g. aspects related to how labels are placed in a diagram (inside a box, below it, etc.), rules to ensure the production of good quality designs (this is related to metrics provided by some design methods), and also project specific rules such as the maximum number of characters allowed in a label.

The existence of a formalism (the VC Formalism) in our approach from where all the frameworks are derived, constitutes a conceptual backbone for all the development process, from writing a specification to the generation of a modelling tool. Such a formalism, if applied to the ECLIPSE-DE, would probably simplify its concepts (which seem somewhat diffuse) by placing them in an overall structure.

3.2.2 The Assertions/Constraints

In GDL, constraints are called assertions. A check of an assertion produces one of three results: true, false or non-applicable. The following classification of checks on assertions is proposed. There are four types of checks: connectivity, layout, semantic and completeness. However it is not clear how the type of check is determined and for certain cases it seems that a check can belong to more than one type.

In our approach a comprehensive and detailed theory of semantic constraints has been developed. Based on that theory we elaborated a process of hardening and deferring constraints as a means of expressing the usage of a modelling technique.

GDL also describes checks according to timing; they may be: implicit, immediate and user-instantiated. Only the two first classes have been implemented. The approach does not provide a policy to determine which assertions are suitable to be user-instantiated. It should be noted that the use of user-instantiated checks requires the inclusion of an explicit validation phase of the diagram, as opposed to a validation occurring simultaneously with the editing task. In our approach all constraints are checked at editing time - there is not a distinct validation phase. In the dissertation we give the rationale for having taken that option and present a visual language based on that principle.

In [Beer88], as a conclusion, the following statement is made: 'to provide a meta-system capable of generating syntax-directed interfaces for many different methods would require such an enormous amount of specification concerning the behaviour and mode of operation of each editor as to defeat the purpose of having a tailorable system at all'. In this dissertation we show that this not only is possible but also extremely desirable.

In the ECLIPSE-DE it is assumed that 'the designer knows best' and therefore s/he is granted maximum freedom during the design task. Excessive freedom may result in diagram configurations with high levels of inconsistency obtained during the editing session. Such situations should be avoided and we address that problem by proposing a way of configuring the modelling tools to control the amount of freedom given to the designer.

3.2.3 Compilation of GDL Descriptions and Code Generation

The output of the compilation phase of a GDL description consists of a number of tables stored in files. Four tables are generated: a nodes table, a links table, a labels table and an assertions table. These tables are used to drive the generic design editor at execution time. The GDL compiler does not generate an intermediate representation (such as a syntax tree).

The assertions table is obtained by translating the method assertions in the GDL description into strings in Reverse Polish, a notation that describes expressions in postfix form. Reverse Polish was chosen due to its aptitude to be used in the evaluation of expressions. An interpreter executes the translated assertions at editing time. This approach provides no independence from the underlying implementation platform; the output tables are directly used to drive the generic design editor.

In the VC approach, the syntax tree generated by the compiler constitutes the intermediate representation from which executable code is generated. If the underlying platform changes, it is only necessary to update or replace the code generator (the back-end of the compiler) which will be using the same intermediate representation to generate the code in the new target language, for instance C++ or Java.

3.2.4 Storage

Although the initial idea had been to integrate the ECLIPSE-DE prototype in the ECLIPSE system through its central database, this was not achieved because the database was not available at the time. The prototype interfaces to the UNIX file system rather than to a project database [Beer88].

A canonical representation (ADI) [Sommerville86] was used to achieve independence from the tool to the underlying storage mechanism. This approach minimizes the effort required to port the ECLIPSE-DE to a new storage system - only the ADI interface would have to be changed.

3.2.5 Conclusion

In conclusion, it seems that the main directions for evolution for the ECLIPSE-DE project are the provision of a well-defined and consistent conceptual framework for constraint classification and checking, and also the inclusion of a code generator providing independence from the design tools implementation language (target system). We believe that the VC system achieved considerable progress in both those directions.

3.3 Hekmatpour, Ince and Woodman's Work

3.3.1 Formal Specifications for Software Systems Prototyping

In his PhD dissertation [Hekmatpour87a] Hekmatpour describes a system for rapid software prototyping named EPROS as opposed to the conventional 'life cycle' model of software development.

The following aspects of software prototyping are covered by EPROS: functionality, which expresses what the software system must do and is based on execution of specifications written in META-IV, the formal specification notation of VDM; human-computer interface, based on a textual representation of State Transition Diagrams.

The system uses an executable formal specification language, EPROL, which combines the two notations mentioned above, i.e. *functionality and dialogue notations*; and also the *design notation*, to be used in the refinement and modularisation of the software system under construction, which includes features such as abstract data types, functions and a formalism called 'cluster', addressed below; and the *implementation notation* which is based on a hybrid of C and Pascal and is strongly typed. The integration of these notations aims to compensate for each other's shortcomings.

The language also includes a formalism called 'cluster' which is a form of modularisation based upon the generalization of procedural abstraction. It is used when functions and procedures are inadequate. Using the 'cluster' mechanism, the programmer can extend the already existent facilities provided by the EPROS system. A goal of the use of 'clusters' is software reuse.

The EPROS architecture features both an EPROL interpreter and a compiler which generates LISP code. Using the interpreter it is possible to browse both dialogue and functional specifications. The final products, implemented in LISP, are then executed by a module called 'executor'. Only the executor communicates with the window manager and the I/O subsystem which collectively support the dialogue mechanism of EPROL.

The systems development is performed in a top-down fashion (from abstract to concrete) and iterative or cyclic. The result of each cycle is an executable EPROL specification of the system which can be converted into a working prototype.

When the user is fully satisfied with the exhibited behaviour of the system the dialogue specification and the functional specification can be integrated and the final prototype is obtained.

Relation to Our Work

It is not clear to what extent intermediate prototypes can be tested by the user. According to the paper: 'the executor has the role of executing finished products'. Also, only finished products in LISP may access the window manager and the I/O subsystem through the 'executer' module. So, apparently, with intermediate prototypes it is not possible to generate output or interact with the system. In our approach there is no concept of refinement of specifications. However, they can be written incrementally and for each obtained specification a working prototype can be readily and automatically generated using the full set of graphical and interactive features provided by the development environment.

The domain of application of EPROL is not restricted. For the specification formalism to be able to describe any possible application, it has to be very generic and include a vast and rich set of concepts. The result is a necessarily complex specification language. Moreover, in order to be generic it must be able to support any software mechanism. The 'cluster' formalism can be used to program these mechanisms, but this is a time consuming task.

When the domain of a specification formalism is restricted to a given kind of application it is possible to provide a shorter set of constructs and also high-level software mechanisms. Our approach is geared towards this context. For instance, it seems to us a very difficult enterprise to build a design editor based on diagrammatic notations, such as the ones generated by our approach, with the prototyping system described.

There is only one case study presented which includes both functional and dialogue specifications, and it does not include a detailed discussion on the User Interface (UI) of the generated prototypes or of the final system. Only a screen dump shows an aspect of the UI, which is form based and hierarchical.

The work described is more concerned with the formal aspects of the approach then the usability of the produced systems. For us the important focus of research is not as much the formalisation in itself but more the way it can be used to achieve concise, consistent and unambiguous specifications which can also be read by a parser. The automatic generation of useful design editors from these specifications is also a vital component of our approach.

3.3.2 Formal Specifications of Modelling Techniques and Diagram Editing

The paper presented at the European Software Engineering Conference (ESEC) [Hekmatpour87b] describes a language for the specification of Modelling Techniques (MTs) called PSN (Picture Specification Notation). PSN is to be used within a prototype of a software tool building system. A graphic editor which is driven by PSN specifications of MTs in a way that guarantees syntactic correctness is included in that system.

Mathematical Based Formalisms Versus Grammars

We claim in Section 4.3.2.1 on page 41 that grammars are not the best approach to specify modelling techniques; their work backs our claim. After some experimentation with grammars, they encountered a number of difficulties. It is difficult to elaborate a suitable grammar for that purpose and even more difficult to build a parser for it. A grammar notation is also hard to understand, use and verify. The notion of production rules is not enough to express all concepts of MTs. Also, a grammar cannot ensure total syntactic correctness, as an example the authors claim that it cannot prevent circularity of processes in a DFD. A grammar based formalism is not appropriate for handling incomplete diagrams.

Changing into a mathematically-oriented formalism, PSN, made it possible to overcome those difficulties. Another advantage is that being supported by a sound mathematical basis it lends itself well to verification. The same reasoning can be followed in regards to the VC formalism and the specification language derived from it, the VC-t.

Separation Between Components of a Graphical Notation

In the PSN based approach the graphical notation 'G' of a MT is seen as having three components: lexical, which denotes the symbols used in G; syntactic, the rules governing the combination of symbols in the production of a diagram; and semantic, 'which denotes the meaning attributed to each syntactically valid picture in G' (extracted from the paper). The paper does not explain how the semantic information is used in the system.

This definition differs from the one used in our approach in which the syntactic level dictates the valid geometrical relationships between any graphical objects, e.g. 'a shape can only be connected to a line style from its perimeter, not from its centre', or 'shapes cannot overlap'. These syntactic rules are independent of the type of the shapes, i.e. 'processes' in

DFD or 'entities' in ER, or the type of the line styles, i.e. 'dataflows' in DFD or 'relationships' in ER. The semantic level corresponds to the MT rules on its concepts, e.g. 'a DFD must have at least one external entity which provides input to the system'.

We believe this is a neater structure in what it provides a clear separation from the geometrical relationships, which are valid for all notations, and the semantic aspects, which are characteristic of each notation.

PSN specifies the rules of a notation, i.e. the syntax according to their definition. The alphabet (symbols) is defined separately using an interactive editor. VC uses a similar approach: the symbols (shapes and line styles) are defined using a graphical objects editor while the rules (constraints) are specified in VC-t language.

MT Usage, Visual Language and Diagram Validation

A major advantage of our approach is that it supports the generation of design editors which include a well defined visual language. The formal specification in VC-t is used to derive the usage aspects of a MT, i.e. the way a diagram of that MT is drawn. These usage aspects are provided to the user by the visual language.

In the work by Hekmatpour and Woodman no reference to the usage aspects of an MT is made. The paper does not include any discussion on this topic or on the interaction of the user with the graphic editors. We do not recognize the use of any form of visual language or structured dialogue.

Although the syntactic correctness of the MT diagrams obtained with the PSN driven editors is guaranteed, there is no concept of inconsistent states: the diagram is either correct or incorrect. The specifications obtained with PSN do not have distinct and identifiable assertions or constraints; i.e. there is a single parameterized predicate which includes all the rules of the MT. It is therefore impossible to perform continuous or incremental validation of the diagram being edited.

In the editors obtained with our system, in order to give more freedom to the user, inconsistent diagram states are allowed during the editing process. The visual language supports the editing task by giving semantic feedback to the user on the current state of the diagram. This is based on semantic constraints which are continuously being checked during the diagram editing. The diagram is therefore incrementally checked for validity.

Expressing Object Refinement

An important aspect of PSN is its ability to express refinement. An object can be refined into a diagram and a logical connection is always maintained between the two. This is possible even when the diagram is expressed in a different notation of that used for the object. Our formalism does not include such a feature at the moment. We do intend to extend the formalism to make it able to capture this kind of object refinement.

Code Generation

In our approach, executable code is automatically generated from VC-t specifications of MTs.

PSN specifications are translated into an intermediate format which can be interpreted. It cannot be translated or compiled into a widely used programming language, therefore it is not portable across implementation platforms. PSN is a very expressive formal language, but due to its expressiveness it is also too complex for automatic code generation, i.e. the designer is able to write specifications from which the generation of code would be virtually impossible. For example, the 'functions' section in a PSN specification allows the designer to specify in an algorithmic form the computations that must be used in the rules. This seems very difficult to be supported by a code generator.

The implementation of the system using PSN was still under way at the time the paper was written.

Conclusion

In spite of being a very expressive language, PSN was not designed for the purpose of code generation. In our approach we require an expressive formal specification language but one which can also be used in the automatic generation of executable code. Moreover, the generated diagram editors must include a visual language satisfying usability requirements such as the intentional introduction of diagram inconsistencies between valid configurations. This motivated a profound research into semantic constraints, their specification and management. The generation of usable interactive design editors that truly support the editing task using the semantics of the underlying technique is a main goal in our approach. This was not pursued in Hekmatpour and Woodman's work.

3.4 The Metamodelling Community

The work done by Cooper on Configurable Datamodelling Systems [Cooper90], although following an approach that is more system oriented - the system uses a toolkit of high level modelling primitives - rather than metamodel based, provided the motivation for the development of our formalism to describe MTs. The toolkit includes a set of data modelling primitives and a set of user interface primitives. Specific data models can be composed, by a user, out of these primitives. For any data model, a user interface can also be built. The system also allows for the specification of constraints which are part of the data model definition.

A good overview of concepts and systems for metamodelling is presented in [Alderson91]. The paper stresses the necessity of providing computer support to the process of tailoring methods to specific users' needs. It then explains how meta-CASE technologies tackle this challenge. Blaha also addresses metamodelling in an introductory way

[Blaha92]. The OMT object model notation is used to obtain a number of restricted meta-models of some widely known MTs.

Alderson defines a method as a number of inter-related techniques and notations for constructing a complex self-consistent information product. Our work is mainly focused on the process of specifying MTs (which encompasses both concepts of 'techniques' and 'notations' as used by Alderson) in a simple, quick but also formal way and how to automatically obtain supporting tools from the specifications. This approach allows the user of the system (the MT designer) to tailor existing MTs by changing their presentation or semantics.

A large amount of research is currently being done on method specification. However, its scope is related more to method integration rather than to tool generation. The relevance to our work is on the way methods are specified. A study on CASE tool integration performed over a number of large organisations in the United States is presented in [Rader93]. The study revealed that the majority of organizations either use particular CASE tools as and when necessary with no tool integration, or use clusters of CASE tools integrated to support a part of the process. More developed integration technology, such as framework-based integration or multiple integrated CASE tool clusters are not common. The use of complete integrated CASE environments is still just a target for the current research work. A language for the definition of a variety of MTs called MDL (Model Definition Language) is proposed in [Atzeni93]. The goal of this metamodelling approach is the translation of schemes from different models which were previously defined in MDL. One of the interesting points is the possibility of the automatic generation of a Schema Definition Language (SDL) from the corresponding MDL definition (unfortunately not presented in detail in the paper). In [Brinkkemper93] the integration of the diagrams resulting from the various editors provided by a CASE tool, e.g. entity relationship, dataflow diagrams or structure charts, is discussed. A framework, using a new construct named ViewPoints, for the development of systems requiring the use of multiple methods (which include notations and development strategies) in given in [Nuseibeh92].

Two research groups have been producing work which includes aspects that are particularly related to this PhD dissertation. Some of this work is briefly described below.

3.4.1 The Dutch Work

A formal language able to express constraints is presented in [Hofstede93]. The language is called LISA-D (Language for Information Structure and Access Descriptions) which is a formal extension of RIDL (Reference and IDea Language). LISA-D is based on the conceptual modelling technique PSM (Predicate Set Model), an extension of PM (Predicate Model) which in turn is a formalization of NIAM. The paper claims that 'a conceptual data modelling technique should not only be capable of representing complex structures but also rules (constraints) that must hold for these structures'. It is said that LISA-D is (in principle) also applicable to other object-role modelling techniques such as ER or FDM.

The feasibility of a completely flexible CASE shell is discussed in [Hofstede96]. A CASE shell is defined in the paper as a method independent CASE tool, which may be instantiated with a specific method to become a CASE tool supporting that method. The term 'flexibility' is used to refer to the extent to which users are able to adapt a tool to their working style. This definition differs from the one we use in this dissertation, in that we make a distinction between the user of the tool, and the user of the metamodelling system (MT designer). Flexibility is then defined at two different levels: the metamodelling level, where it designates the extent to which a method or individual MTs can be specified, and the modelling level, where it means how well the modelling system derived from the metamodel can support different ways of producing model diagrams. The paper states that although a number of CASE shells have already been produced and even commercialised they do not support the modelling process.

Three orthogonal dimensions are identified in a CASE shell repository: method level versus application level; process versus product (also referred to as 'way of working versus way of modelling' [Wijers90]); conceptual versus graphical knowledge.

The first dimension, also referred to as 'types versus instances', is not considered to create a hard problem. The 'process' part of the second dimension relates to the tasks to be performed during the modelling work. Tasks are classified according to their size: *large tasks*, for instance 'perform the Business Area Analysis' within the Information Engineering method; and *minor tasks*, e.g 'add an external entity to a diagram' in the DFD modelling technique. It is claimed by the authors that most state-of-the-art meta-modelling techniques do not address the way of working. In our approach we tackle the problem of supporting minor tasks. The MT usage can be specified by classifying semantic constraints as soft, hard, hardened or deferred. A flexible visual language can then be generated to support the modelling process.

The 'product' part shows the structure and relationships between the information modelling products. Obtained models must generally satisfy complex rules imposed by the MTs. To capture such rules a powerful constraint modelling technique is required. For that purpose the approach uses both graphical representation of constraints in PSM, e.g. total role or uniqueness constraints, and the constraint modelling language LISA-D for constraints that cannot be expressed graphically (which are in fact the majority, as they declare).

'A way of working and a way of modelling are closely related'. The metamodelling must be able to express the existing relationships. The constraints used in our approach to specify the 'way of modelling', i.e. the structural part, are then applied in the specification of the 'way of working' or 'MT usage' as explained above.

The third dimension relates conceptual to graphical knowledge. As stated by the authors, this is 'particularly important for CASE shells'. This specifies 'how models appear on the screen and how actions can be performed on these represented models' [Hofstede96]. However, also in the same paper it is said that 'an area that has never been addressed, to our knowledge, is the dynamic side of the representation of graphical knowledge'. This PhD dissertation presents a detailed study on the graphical representation's dynamics including constraint checking procedures as a result of diagram operations which are supported by an underlying graph library.

3.4.2 The Finnish Work

The work of the team led by Lyytinen is done in the context of the development of a CASE shell called MetaEdit. A CASE shell is defined as a tool that can be customised by users to support their own preferred methodologies [Smolander91].

In [Marttiin95], as a motivation to this work, the weak support given by CASE tools to the users' native methods and methodologies is mentioned.

Although taking a different approach, MetaEdit has a common goal with our work in that they both support high-level specification of methods, or modelling techniques, using an easy to use specification language.

MetaEdit is a metamodelling editor based on the OPRR (Object Property Role Relationship) data model [Smolander91] (it is in fact a meta-metamodel for it is used to obtain metamodels of methods or modelling techniques). OPRR offers a graphical notation with which methodology models can be constructed. MetaEdit can be used either as a CASE shell [Tolvanen93] or alternatively as an interface to other CASE shells by generating their input configuration files in a (semi)automatic way [Smolander91]. In the latter situation the output generator of MetaEdit translates methodology specifications to formats needed in CASE shells.

The process of method adaptation is described in [Tolvanen93]. During this process a formal model of the method is derived. Note that this is also the approach we take in this dissertation, it can then be considered as a metamodelling approach.

A tri-dimensional metamodelling framework is proposed (equivalent to the one used later in [Hofstede96]). The three dimensions are: type/instance, conceptual/representational and statics/dynamics. However the third dimension is not explored in the paper. This dimension states how a method should be followed, i.e. how and in what order of tasks, the method produces its products (representations). This is one of the topics of our work; the user (method designer) may express usage information through the classification of semantic constraints.

A new version of the MetaEdit tool, called MetaEdit+, has now been released [Kelly96]. The tool is now multi-user and multi-platform. It uses the conceptual meta-metamodel GOPRR, described in [Tolvanen93] and also [Marttiin95], which is an evolutionary extension of OPRR. It allows multiple representations of the same conceptual object (for instance, graphical, matrix or text) and even different graphical representations of the same object in any given representation paradigm. The letter 'G' added to the name stands for Graph, a concept to express an aggregation of a certain set of objects and their relationships. An extension of OPRR that is particularly related to our work, is the possibility of attaching integrity checking rules to properties, in addition to normal type rules. Although it is possible to express some integrity rules (or constraints as we call them) when modelling a given MT, they must be very simple ones. The following rule for DFD is given as an example in [Kelly96] 'a string property must be a dotted sequence of numbers'. This rule forbids combinations such as 'Fred' or '2.'. More complex integrity rules cannot be expressed, such as 'there cannot exist two dataflows with the same name and in the same direction which share the same origin and destination'.

The literature indicates that a desired direction for the work in MetaEdit+ is 'to increase the capabilities to describe integrity constraints within and between method specifications'. We believe our work has provided a positive contribution to this topic.

3.5 Graph-Grammar Based Approaches

3.5.1 Introduction

There are not many common points between the Graph-Grammars (GGs) approach and the one followed in our work. There are however common objectives and therefore an analysis of the advantages and drawbacks presented by each approach should be carried out. We should explain why GGs were not included in our research work.

The generic term 'graph-grammars' refers to a variety of methods for specifying (possibly infinite) sets of graphs or sets of maps [Ehrig86]. This involves defining their structure and a number of graph transformation rules [Rekers94]. The term 'graph rewriting' is preferred by the author of [Sleep90]; he considers 'graph grammars' to be a narrower term.

GGs are being applied to software engineering in a number of topics which are relevant to our work; these include software specification and development, incremental compilers, etc. [Ehrig86].

The following paragraph, taken from [Nagl90], summarises the role of GGs in software engineering applications: 'GGs are used for internal, high-level programming. This means that the effects of concrete tools (such as those of a software development environment) or abstract tools (such as formal specifications of semantics) on internal data structures are operationally specified by a language for manipulating attributed node and edge labelled graphs, one essential of which is rewriting by rules. From these specifications the software architecture and the implementation of tools can be derived'.

Initially, it seemed that GGs could constitute the ideal supporting conceptual tool for our work. For this reason, in the beginning of this research work, the possibility of pursuing a GG approach was taken into account. However, after a survey of the field, we felt that the implementation of a software system following a GG approach is still a complex task. In what follows we give support to this statement.

Why Could GGs Have Been Useful to Us?

'Graphs combine the advantage of intuitive understanding with mathematical feasibility. Rule-basedness is considered to be a powerful and manageable method for specifying the dynamic and behaviour of systems. Putting graphs and rules together yields graph grammars' [Kreowski90].

The importance of combining graph-like structures and rule-based systems is also asserted in [Schurr95]. It is said 'nevertheless, their symbiosis in the form of graph rewriting systems or graph-grammars are not yet popular among software engineers. This is a consequence of the fact that graph-grammar tools were not available until recently and the lack of knowledge about how to use graph-grammars for software engineering purposes'.

We couldn't agree more with the importance of rule-basedness and the use of graphs stated by Kreowski and Schurr - we use both in our system. However, for the reasons that will be given below, instead of a GG formalism we have created our own logic-based formalism - the Visual Concepts - which, for the way it was designed, we knew it could be implemented by us.

Nagl defends that 'GGs are better for modelling; they offer mechanisms to specify complex operations that cannot be found in the data modelling discussion' [Nagl90].

As stated in [Kaplan90], 'designing a visual language based around graph rewriting and building an associated environment seems an excellent potential application of graph grammar technology'. According to Kaplan, the area of visual languages is an application domain where GGs could be directly accessible and manipulated by the user. It seemed that our generic visual language for diagram editing could be based on this technology.

Why Have We Not Used GGs?

Our goal when selecting a formalism was to be able to use it as a tool in our research, not focusing our research on the formalism itself. What has been said in favour of GG approaches is very much supported by theoretical reasoning rather than concrete systems or implementations.

Nagl says: 'a basic software layer suitable for GG implementations has to be developed in order to be used at different sites. Furthermore, integrated GG specification environments and, finally, transformation tools for getting an implementation from a GG specification (compilers, generator tools) have to be developed' [Nagl90]. According to Nagl, 'there must be some help for the complex translation from a graph grammar specification to an efficient implementation in one of the conventional programming languages'. At the moment there was no efficient automatic translation for a GG specification; it would have to be carried out by hand. A GG was seen only as a 'conceptual tool to be used in paper and pencil mode'. He also declares the necessity of some basic tools, for instance a general purpose graph storage.

The RWTH group in Aachen, Germany, has been working with GGs for more than 15 years. Their experience in the area of software engineering stems from two projects:

IPSEN [Nagl86] and PROGRES [Schurr95]. The latest developments of the PROGRES based system [Schurr95] seem to indicate that GGs are moving in the right direction. A specification obtained with the PROGRES language can be executed by an interpreter which shows the effects of the graph rewriting operations. The specification is checked for consistency and a stand-alone prototype can be automatically generated from it. This prototype is built on top of the nonstandard database system GRAS and has a Tcl/Tk based interface. The PROGRES compiler is able to produce 'easy-to-read' C or Modula-2 code from which a final implementation can be derived. 'The language PROGRES, its tools and the GG engineering methodology are a first step [...] to establish graph rewriting as a new specification and programming paradigm'. At the end of the paper it is declared that 'we have to admit that currently available tools and techniques are far from being as mature as (for instance) logic-oriented or functional centred tools and techniques'.

'Will the future bring the breakthrough? In my opinion, there is a fair chance.' [Kreowski90]. We believe more evidence is still needed.

3.6 Visual Interaction and Diagram Drawing Support Systems

The principles of visual programming and a proposed taxonomy are given in [Shu88]. According to the book, '*visual programming languages* allow users to program with visual expressions. [...] used to accomplish what would otherwise have to be written in a traditional one-dimensional programming language'. We subscribe to this view in this dissertation. A diagram is obtained instead of a textual representation. The visual expressions are the representations of the MT constructs; they are called Visual Objects. They include a semantic component which is expressed by constraints. This is based on Chang's theory of generalized icons, presented in [Chang90], in which icons are composed of a logical part (the meaning) and a physical part (the image).

Tinkertoy [Edel88] is one of the early attempts to produce a visual programming environment. Lisp programs are built interactively with icons and flexible interconnections. Tinkertoy converts textual Lisp expressions into iconic expressions and vice versa, i.e. textual and iconic editing are interchangeable. The way the Tinkertoy prototype evaluates iconic expressions is by first converting them to Lisp code which is then executed. The environment does not offer any possibility of customisation and there is no flexibility in the editing process.

EcrinsDesign [Adreit91] is a graphical tool for the drawing of diagrams of a semantic data model based on ER. This is a single MT system, in contrast to our case. Its most interesting feature is the support of the dynamic aspects of the design. The objects' type may evolve with the user's actions in conformity to a set of constraints defined at the tool level. The objects' evolution is specified in a fixed way by a state transition automaton. In our opinion this manner of defining the dynamics of the design task is not scalable, the number of states in the state transition automaton would grow very rapidly if more complex MTs were used instead of the ER based one. Moreover, it is not suitable for our approach; we require a generic mechanism that can support any description of dynamics.

Once a diagram has been obtained it is then analysed and translated into a DDL (Data Definition Language) representation. Again in this system, full diagram validation is carried out in a separate stage and is performed only when the user explicitly requests it. During the design process only a subset of constraints are enforced to give some assistance to the user; the remaining are *relaxed* to increase the flexibility during the editing of a diagram. The visual language we propose in this dissertation represents a considerable progress when compared to the constraint relaxing technique.

A functional visual language, denominated VisaVis, is proposed in [Poswig92]. When editing a visual program, the user is informed by the system about the possible connections between graphical components in order to avoid syntactical errors. Visual feedback is also given by the editors generated in our approach: the user is always kept informed of the status of each visual object so that all diagram inconsistencies are visually noticeable.

A formalism for the specification of graphical languages is presented in [Rekers94]. The proposed formalism is a graph-grammar specialized towards the definition of graphical syntax. The specification formalism for the graphical languages is graphical itself. The syntax rules are formulated in terms of typed graphical objects and spatial relations.

In Rekers' approach the editing phase is separated from the analysis phase. This does not happen in our approach: the diagram is checked at design time. The analysis is further split into two phases: *graphical scanning*, when the graphical objects are identified and their spatial relationships are determined; and *graphical parsing*, when the spatial relationships graph resulting from the first phase is used to derive constructs in terms of the graphical language 'L' under consideration. The result of the second phase is an abstract syntax graph according to L.

The paper mainly deals with the graphical scanning phase for which a high level syntax specification formalism was developed which is able to express the majority of existing graphical languages. A tool (not implemented at the time) can then generate graphical parsers according to such a specification.

The main advantage of this work is the readability of the specifications of graphical languages, because they are graphical themselves, i.e. in a specification, the syntax rules of a graphical language are expressed graphically.

The approach does not support partial validation, i.e. the validation of a diagram in an inconsistent state would fail. In our approach the validation is continuously performed and the inconsistencies are shown to the user by means of visual feedback. This can be done because not all constraints are enforced during validation. In Rekers approach all syntax rules are enforced during graphical parsing.

A generator for diagram editors, DiaGen, is presented in [Viehstaedt95] and [Minas95]. The system is meant to overcome some limitations presented by state-of-the-art systems.

For instance, with Garnet [Myers90; Myers92] the structure of valid diagrams has to be maintained by the programmer using the system. From a specification, DiaGen will generate a diagram editor for some class of diagrams. The examples given in the paper are Nassi-Shneiderman diagrams (NSD) and State Transition diagrams (STD); it is not clear if more complex diagrams could be expressed.

It is stated that the generator should be based on a formal model: hypergraph grammars are used, which '(unlike context-free grammars) are able to describe multidimensional relationships between diagram elements'. In a hypergraph the edges are hyperedges, i.e. they can be connected to any (fixed) number of nodes. It is said the edge *visits* these nodes. A familiar directed graph can be seen as a hypergraph in which all (hyper)edges visit exactly two nodes. The layout of the diagrams is defined on a high level by constraints which are defined over attributes assigned to the nodes and edges of the hypergraph. However, with this approach, a diagram edited by the user is always in a consistent state (a hypergraph in the formal model); as already mentioned, we regard this as a disadvantage.

A useful paper for anyone researching the area of supporting tools for MTs is the one by Thomas Green on 'Cognitive Dimensions of Notations' [Green89]. It proposes a number of dimensions which characterise notations and can be used to guide the design of tools to support them. Some examples of these dimensions are: *premature commitment*, one example being a constraint based notation which requires the user to indicate firstly the total number of constraints to appear in the specification; *viscosity/fluidity*, a viscous notation is one that resists to local changes which occurs when the information structure constituents contain many interdependencies; *role-expressiveness*, a notation that displays its plan structure clearly is called 'role-expressive'.

The cognitive dimensions can be used at the specification level for the VC-t language and its compiler; they may also be applied to the generic visual language and the generated design tools.

3.7 Other Software Design and Development Frameworks and Systems

[Budgen92] describes the transformation of models obtained with diagram based techniques (MASCOT and STD) into a more formal specification (CSP/*me too*) which can be executed. An experimental CASE tool, termed Experimental Design Workbench (EDW), has been built to support the whole design and transformation process.

The paper does not generalise the approach to several MTs, it is restricted to the use of MASCOT and STDs. As a future development it is suggested that this work could be generalised and employed within the GOOSE project. It is this project that will be briefly discussed below.

In [Reeves95] a system (GOOSE) to support software design in a complete way is presented. It extends what a CASE tool normally provides, in the following ways:

- It captures all the views of a system. It uses a viewpoints framework, which includes functional, behavioural, structural and data modelling viewpoints. Because GOOSE is more dedicated to real-time systems than to data intensive applications, the initial implementation did not include a data model viewpoint.
- It provides eight tools, including the following which are particularly relevant to our work: a consistency tool, which gives the user information on the violation of the consistency rules amongst viewpoints and within a viewpoint; and an execution tool to animate 'behavioural' and 'structural' viewpoints.
- It describes the evolution of the design using a D-matrix (Design matrix), which keeps information regarding all the elements of a design stage in the design process.

D-matrices are used to support any forms of design behaviour including those which are opportunistic. According to what is mentioned in the 'Conclusions' section, the support for opportunistic design does not seem to have been fully accomplished. Constraints governing the design process have only been slightly addressed. Formality is regarded as important and is appointed as a direction for future development. We consider that our system contributes towards the previous points since it proposes a solution to the problem of providing help during the design task without limiting the user's freedom, this is done using formal specifications of semantic constraints.

3.8 Constraint Based Systems

A constraint describes relations that must be maintained [Borning87]. For our approach we need a formalism able to express complex constraints and a supporting system to store and evaluate the constraints at execution time.

We explored the possibility of using a constraint based system such as ThingLab [Borning81; Borning87] or Garnet [Myers90; Myers92]. However, we concluded that these systems were not suitable to support our approach. The reasons for this claim are presented below.

Most constraint based systems are geared towards the specification and management of geometrical constraints in graphical applications. Amongst the applications of constraint-based languages and systems, the following are named in [Borning87]: geometric layout, physical simulations, user interface design, document formatting, algorithm animation, design and analysis of electrical circuits. The number of constraints can amount to hundreds or even thousands.

Constraints are defined as rules on values; these rules are expressed as equations. The following example is given in [Zanden91]: 'A designer might write the following equation to position a circle 10 pixels to the right of a rectangle: left = my_rect.right + 10'.

Constraints express relations between the graphical objects in a design in terms of their intrinsic properties, such as position, length or colour. The constraints are defined at the presentation level of the graphical interface (the lexical level) and determine the behaviour of the graphical objects (the syntactic level) as a result of the maintenance of spatial relations between those objects.

Constraints in our approach are quite different in nature. They also express relations between objects, but now the properties are extrinsic, in the sense that they are independent of the particular presentation of the objects. Only the meaning of the objects is now relevant. In a design there will be ER 'entities' instead of 'rectangles' and 'attributes' instead of 'circles'. An arrow in the STD modelling technique means a 'transition' and its relevant properties are, for instance, the type of the 'states' connected at its origin and destination. The rules are on types instead of being on values. For example, the following constraint is defined for the STD modelling technique: 'The same pair of states cannot be connected by transitions with the same direction and the same transition condition'. Which is specified as:

```
'FORALL t1, t2 : Transition • t1, t2 BELONGING Transitions(std) IMPLIES
    (( origin(t1) = origin(t2) AND
    destination(t1) = destination(t2) AND
    transitionCondition(t1) = transitionCondition(t2)) IMPLIES
        t1 = t2 )'
```

Constraints are defined at the semantic level, which relates to the meaning of the objects in the application domain, rather than at the lexical level. There will be some consequences at the syntactic level resulting from the enforcement of the constraints, namely in the form of visual feedback regarding state changes in the objects (e.g., an object might be highlighted in some way). These consequences result from constraints on the semantic of the objects (the level of abstraction above) and not from constraints on their presentation (the level of abstraction below).

This fundamental difference between the two types of constraints is reflected in the interaction with the user at design time. While in a constraint based system the spatial relations specified by the constraints are automatically maintained [Myers90], in our system the evaluation of constraints will only decide on the semantic validity of the user actions. That is, the constraints assert that a user action conforms, or not, to the semantic of the objects. Based on the result of the evaluation, the action may be accepted or rejected; in both cases, the system may give feedback to the user. However, the evaluation of constraints do not imply the automatic repositioning of objects as in the case of constraint based systems.

We decided to develop a specification language and build its supporting system from scratch. The language is based on a form of predicate logic and the constraints are expressed as propositions about the properties of objects.

One advantage of predicate logic is its expressiveness. We have lost in efficiency: a powerful constraint solver is always included in constraint based systems. For instance, Garnet is able to 're-evaluate 3500 constraints per second on the IBM RT PC implementation' [Myers90]. However, in our domain of application the number of constraints to be evaluated after a user action is normally around ten! As a result, even with a very inefficient constraint solver (when compared to Garnet's one), as the one featured on our current implementation, the system's response times are adequate.

In conclusion, the constraints in our approach are fundamentally different from those used in the cited constraint based systems. This difference is also reflected in the way constraints are evaluated at design time. For these reasons we opted for developing our own constraints specification language and supporting system. Although we have not used a constraint based system, we believe that, since this study had to be done, it should be included in the literature survey.

3.9 Graph Editors

We require a specialised tool to support the user interaction with visual graphs. Visual representations of graphs have been extensively used in computing science applications. The elements represented by the nodes range from the most concrete, such as gates in a digital circuit diagram, to the most abstract, such as classes in a conceptual schema; while 'the edges have been used to represent almost any conceivable kind of relation, including ones of temporal, causal, functional, or epistemological nature' [Harel88].

As noted by David Harel, the relevant information - and this is true also for our work - is 'nonquantitative, but rather of a structural, set-theoretical and relational nature'. This means that locations, distances and sizes are not relevant, only spatial relationships are significant, for example object connectedness, as in 'node N_a is connected to node N_b by edge E_a'. Therefore, the diagrammatic paradigms we are interested in are topological, rather than geometrical; they are termed by Harel 'topovisual formalisms'.

Because the tool we have mentioned above is to be used in our research work as an underlying layer to support editors for MTs, it must satisfy a number of specific requirements, including:

- providing support for multiple and interchangeable visual representations of the same graph structure;
- including a repository of constraints which are associated with diagram objects;
- offering a fully configurable functionality;
- allowing the User Interface (UI) to be provided by the application using the tool (client application), the tool must only specify the communication protocols,

meaning that it must be able to support any graph-based application with any UI as long as the latter conforms to the established communication protocols.

Most of the work reported in the literature is concerned more with graph visualization methods rather than to support for interactive visual graph editing. This means that current research is dedicated mostly to graph drawing algorithms or efficient visualization techniques (e.g. 'fisheye views' [Sarkar92]) and not as much to generic architectures to support interactive graph based applications. Amongst the more relevant work in this field is the one produced in the scope of 'Diagram Server' (DS), a tool with several facilities for managing diagrams. The architecture of DS is discussed in [Battista90]. An extensive description of the tool within the context of parametric graph drawing can be found in [Bertolazzi92].

Although DS is mainly dedicated to the automatic layout of diagrams, it also includes advanced diagram editing facilities that are relevant to our work. Some of these facilities are:

- independence from the system (the client application) that uses DS, the client does not need to change its internal structure in order to use DS;
- ability to associate callbacks with diagram objects;
- allowing for multiple representations of the same object within a diagram and also multiple diagrams representing the same underlying graph.

There are however some limiting aspects that make DS inadequate to support our approach. For example, in DS the UI is pre-defined. We need a tool that does not impose a particular UI. For enhanced flexibility, the UI must be provided by the client application. Also, DS has a non-customizable functionality; we would like a tool in which the way operations are performed by the user over the diagram may be defined by the client application.

A tool that satisfies the requirements discussed above has been designed and made. We called it GraphTool and present it in this dissertation in a dedicated Chapter.

4. The Visual Concepts Formalism

4.1 Visual Concepts - a Formalism for the Specification of Modelling Techniques

concept *n. Philos.* an idea or mental picture of a class of objects *formed by combining all their aspects.*

In The Concise Oxford Dictionary of Current English [Allen91]

We propose a formalism, called **Visual Concepts**, to be used in the specification of Modelling Techniques (MTs) based on diagramming notations, such as the Entity-Relationship (ER) technique, State Transition Diagrams (STD), Dataflow Diagrams (DFD) and propriety techniques developed for a specific purpose. Visual Concepts (VCs) are typed units of specification. Each VC encapsulates information on the physical, semantics and usage components of a MT construct, e.g. an ER Entity or a STD Transition. In the context of this PhD, the Visual Concepts formalism supports three frameworks: a *Formal MT Specification Language*, the *Generic Visual Language* and ways of *Tailoring MT Usage*. Using those frameworks, specifications may be produced for MTs and, from them, dedicated design tools can be automatically generated.

VCs cover more aspects of a construct or object than what is proposed by the Object-Oriented (OO) approach [King89; Stroustrup88]. Besides the OO's attributes and methods, VCs are able to capture the visual representation, meaning and behaviour of a MT construct. They still provide encapsulation but no inheritance mechanisms are supported, both for simplicity of implementation and because we did not find it necessary to have inheritance in our approach.

4.2 Key points of Visual Concepts

- The Visual Concepts formalism captures both the semantics and the graphical notation of a MT. This conforms to the ideas presented in [Shu88].
- Visual Concepts are aimed at the automatic generation of design tools from formal specifications of MTs.
- A single Visual Concept is obtained for each construct in the MT. This maintains the semantic information encapsulated in the Visual Concepts, preventing its dispersion throughout the specification. As a consequence, code generation and con-

straint management are facilitated, in that it is easier to establish the references to the constraints at generation time and to determine which constraints to check at editing time. Moreover, it makes it easier for the generated design tools supporting the MT to provide semantic feedback to the user.

4.3 Components of a Visual Concept

A **Visual Concept** has the following components:

- Identification - unique name;
- Physical - including properties and visual representation;
- Semantics - a set of constraints;
- Usage - dynamics at design-time (when drawing diagrams or models).

The physical, semantics and usage components are discussed in more detail in the following sections.

4.3.1 The Physical component

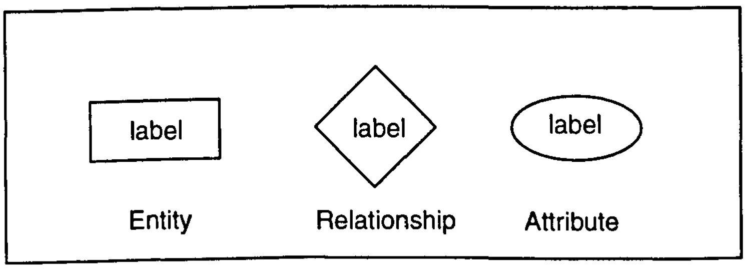


FIGURE 4. ER Modelling Technique constructs.

The **physical component** includes visual representation and properties.

The **visual representation** is a reference to one or more graphical objects, for example an ER Entity is depicted by a rectangle (see Figure 4) and a STD Transition by an arrow. A VC visual representation may refer to several visual objects; this happens, for instance, when different views are defined, e.g. a full view and an simplified view. Consider the notation used in the Object Model view of the OMT methodology [Rumbaugh91], shown in Figure 5. Two visual objects may be used: a complete view, including the class name, attributes and operations; and a simplified one featuring only the class name.

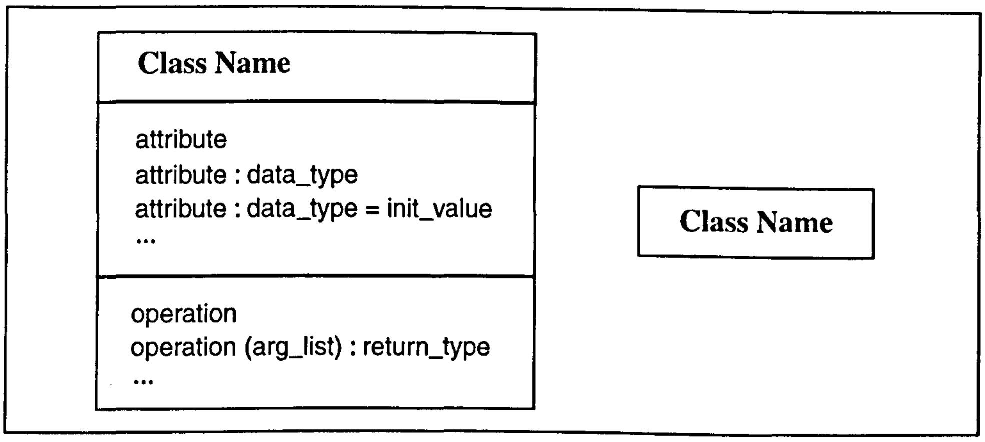

Class Name
attribute
attribute : data_type
attribute : data_type = init_value
...
operation
operation (arg_list) : return_type
...
Class Name

FIGURE 5. Summary of OMT's Object Model notations for classes.

The **properties** are the static characteristics of VCs. Some properties are reflected in the visual representation as a way to display information from the application being modelled. This can be done by labels, for instance the name displayed in a label of an ER Entity, displays information about that Entity on the application's Universe of Discourse (UoD), e.g. "Person"; or by any other modifiable characteristic of the visual representation, for instance its colour. For example, the property 'name' of the ER Entity would be specified as: 'name: Entity -> String'. Multiple labels can also be defined for a VC, for instance, as we have seen, in Rumbaugh's OMT, the Object Model allows multiple labels in the attribute section. Other properties of a VC may give information on which other VCs are attached to it; or may allow the comparison of the VC with other VC instances of the same type, for instance the 'equality' property on ER Attributes may be expressed as: 'equal : Attribute x Attribute -> Boolean'.

4.3.2 The Semantics Component

The **semantics component** consists of a set of semantic constraints, expressed in a form of predicate logic with equality.

A **semantic constraint** is an assertion either on the whole diagram or on the properties of a VC type; the latter may affect the VC itself and/or its relationship with other VCs. For example, the following simple constraint is included in the semantics component of the VC 'Entity' defined for the ER technique:

'Each Entity must have a unique name'

which can be formally expressed as:

'$\forall e1, e2{:}Entity \bullet e1, e2 \in Entities(er) \Rightarrow$
$(name(e1) = name(e2) \Rightarrow e1 = e2)$'

Where 'er' is an ER diagram.

The constraint can be read as: 'for all e1 and e2 which are entities of an ER diagram, if the name of e1 is the same as the name of e2, then e1 and e2 must be the same entity'.

The semantics component of a Visual Concept expresses the semantics of the corresponding MT construct. The semantics of MT constructs are normally described using natural language (e.g. English), whereas the semantics component of Visual Concepts is a formal specification and therefore unambiguous which makes it easier to reason about.

4.3.2.1 Specifying Semantics by Grammars or Semantic Constraints?

Instead of constraints we could have used a grammar to specify the semantics component of VCs. Why have we chosen constraints? There are good reasons for this which we will now discuss.

Preserving semantic information

The problem of using grammars (both constraint-grammars or graph-grammars[1]) to specify the semantics resides in the fact that the information expressed by the grammar, is no longer tractable once the code is generated. As a result of this, the generated editor is unable to give back to the user any meaningful information - semantic feedback - at editing time.

Our approach of semantic constraints encapsulates all the relevant constraints of a given Visual Concept inside it. Moreover, the objects' identification is kept throughout the software development cycle. This way the semantic information is preserved from the specification to the final code across the generation process.

Encapsulation can also be found in the paradigm of object-orientation. However, the most common object models (for instance, Eifel, Smalltalk or C++) do not provide support for the specification of semantics [Paredes93] or the visual aspects of objects [Levialdi93]. Semantics are normally embedded in the code of the object's operations; the visual aspect is usually provided by graphical libraries such as Motif, Interviews, GoPath, Athena or Tk. The advantage of our formalism is that both semantics and visual representation are built-into the Visual Concept model. This idea is related to the field of Component Programming. In this paradigm, interactive components have a user interface and a specific behavior or functionality. As Visual Concepts they provide encapsulation but they do not provide inheritance. However, Visual Concepts offer a powerful support to semantic definition through the use of constraints which cannot be found on Component Programming based approaches [Jazayeri95; Udell94].

1. Graph-grammars are implemented by a graph-parser. Constraint-grammars are implemented by a parser plus low-level constraints.

Inspecting specifications

The main advantage of grammar specifications, over constraint based ones, is that they are easier to validate and to check for their completeness. However, producing a grammar specification can become a very tedious task and its complexity makes it accessible only to experts. Conversely, the concept of semantic constraint is easy to grasp and it provides a more intuitive way of producing specifications.

A specification solely based on constraints is typically difficult to inspect. A possible solution to this problem is to provide a demonstration mechanism for the VCs. This could be done using an animation based technique. Such a technique would allow for the visual inspection of VCs semantics (see Section Section 11.3, "Future Research Directions," on page 148).

4.3.3 The Usage Component

The Physical and the Semantics components described above are used as the foundations of the VC-t Specification Language (Chapter 5). A Visual Language framework is then described in Chapter 6 using an implementation of VCs, the Visual Objects (VOs - which are the topic of the next section). A notion of state is also introduced in Chapter 6, and associated with VOs and diagrams to express their dynamics. The way a VO evolves through changes on its state is fully developed in Chapter 7, where the usage component is presented.

In the **usage component**, VC constraints specified in the Semantics component, are divided into the classes: hard, soft, hardened and deferred. This process determines the dynamic aspects of the VOs in a diagram during the design task. The purpose of specifying the usage component is improving the usability of the generated design tools and tailoring their use to suit a particular user or class of users.

We will not detail the presentation of the usage component here as it is covered in Chapter 7. To make it possible to define the usage component of VCs it was necessary to develop a new theory of semantic constraints, which is described in detail in that Chapter.

4.4 Visual Objects - An Implementation of the Visual Concepts Formalism

This section describes Visual Objects (VOs), which will be used as the components of the Generic Visual Language (described in Chapter 6).

Each different interactive graphical system has its own terminology for the classes of objects it uses. We tried to classify our Visual Objects in a generic way, so that this same classification may be used in the development of other systems. The concepts and structure presented may constitute a generic lexicon for interactive graphical systems.

A **VO** is a VC that can be displayed on the computer screen and that reacts to user generated events. A VO is, therefore, an implementation of a VC.

A **diagram** is a set of VOs connected to each other in accordance with their semantics. It must be noticed that some semantic constraints cannot be associated with a particular VO, for example 'a STD must have an Initial State'; this constraint cannot be associated with the VO 'Initial State' because it must be checked before its creation. Another example is a connectivity constraint, such as 'all the states in a STD must be connected'. These constraints are classified as Diagram Constraints and are associated with the diagram itself.

VOs are described by a hierarchical structure of objects. From the bottom to the top of the hierarchy we have the following kinds of objects: *Graphical Objects, Visual Object Definitions* and *Visual Objects.* The section below presents an example of this hierarchical structure showing how it can be instantiated for the ER modelling technique.

4.4.1 An Example: The ER Modelling Technique

The hierarchical structure of VOs presented above when applied to the ER modelling technique is shown in Figure 6. For simplicity, only the VOicons are presented, a similar instantiation could be done for the VOconnections.

At the top we have the instances of the Visual Objects (VOs) appearing on the screen. For example, 'Person'. Each VO has a reference to an instance of a Visual Object Definition (VOD), in this case the VOD of 'Person' is the ERentity. The VOD includes a reference to a Graphical Object, which for the 'ERentity' is a LabelledRectangle. The Graphical Object defines the appearance, its shape and label (font and position), and the operations available on this kind of object.

4.4.2 The VOs Hierarchical Structure in Detail

At the bottom of the hierarchy are the *Graphical Objects*, which can be *Shapes*, complex geometrical figures which may include labels, for instance the OMT class notations shown in Figure 5; or *Line Styles*, complex lines which may include labels, e.g. single labelled arrowed line, or double labelled line as used in ER cardinality connections. The Shape type includes an image and a structure to describe label slots which are to be filled for each particular shape instance. For this purpose, procedures are defined to deal with the labels. It makes sense to define label operations at the Graphical Object level because it is here that the labels are known (their number and position). However these are just operation definitions and no implementations are provided at this stage.

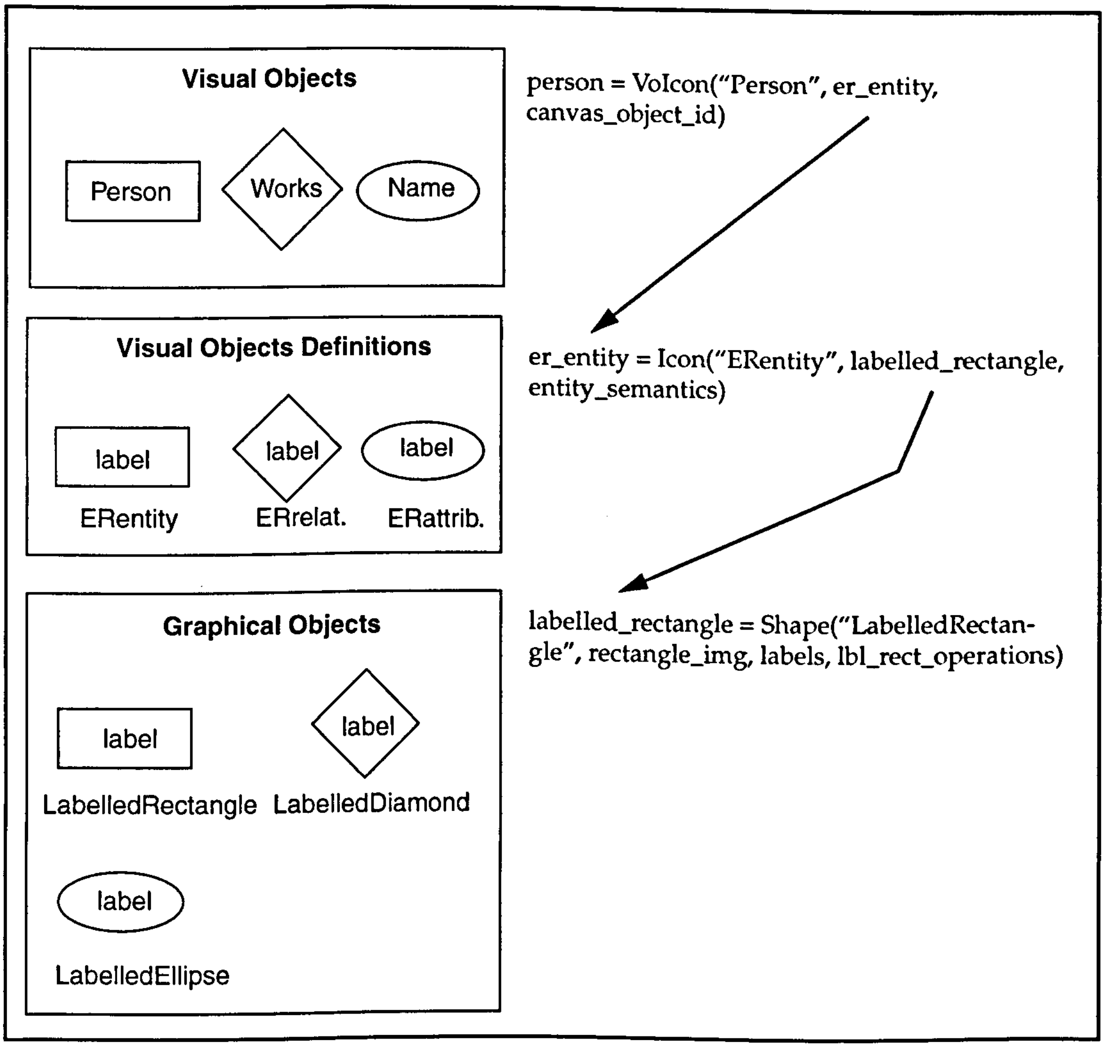

Visual Objects
Person
Works
Name
Visual Objects Definitions
label
label
label
ERentity
ERrelat.
ERattrib.
Graphical Objects
label
label
LabelledRectangle
LabelledDiamond
label
LabelledEllipse
person = VoIcon("Person", er_entity, canvas_object_id)
er_entity = Icon("ERentity", labelled_rectangle, entity_semantics)
labelled_rectangle = Shape("LabelledRectangle", rectangle_img, labels, lbl_rect_operations)

FIGURE 6. VOs Hierarchical Structure for the ER Modelling Technique (extract)

Note that we could consider other layers further down in the hierarchy. Below the Graphical objects there would be the Generic Geometric Objects. These include: unlabelled polygons, such as rectangles, squares, circles, ellipses; unlabelled lines, such as dashed line, dotted line, single and double arrowed line; and text labels. This layer could be supported by another layer, even lower: the Polylines. These consist of arrays or lists of line segments. Polylines are themselves composed by sequences of points (pixels in the screen). Our approach defines the layers above the Graphical Objects (inclusive), as they were described; the GraphTool (see Chapter 9) supports those layers, while a UIMS (User Interface Management System) supports the layers below.

No implementation is given for any operation and graphical appearance associated with the Graphical Objects and Visual Objects Definitions. Only at the top of the hierarchy - the Visual Objects - the appearance, operations and interaction capabilities (events and callbacks) are implemented, typically by a UIMS. This design option promotes the GraphTool to be independent from the system's platform or architecture being used.

Further up in the hierarchy we have the *Visual Object Definitions*, which are composed of a Graphical Object (the physical component) and semantics (the logical component). Visual Object Definitions can be *Icons* or *Connections*. Both Icons and Connections have their semantics expressed by constraints. The Graphical Objects included in Icons (their physical component) are Shapes, whereas the ones included in Connections are Line Styles. At this level are defined the constructs (concepts) of MTs; for instance, 'ER entity', 'ER relationship' or 'DFD dataflow'. In Figure 6 only the Icons are represented.

Finally, at the top of the hierarchy are the *Visual Objects* (VOs), which are either *VOicons* or *VOconnections* (also referred to simply as Icons or Connections, if it is clear from the context that we are not mentioning Visual Object Definitions). Each VO corresponds to a Visual Object Definition with two added characteristics: it can be displayed on the screen and it has user interaction capabilities.

The VO structure includes a name, a VO Definition, a structure to hold the values given to the labels, and elements for screen representation and user interaction. The last component is implemented by a UIMS; we have used a persistent version of the Tk toolkit (see Section 10.1.2 on page 138). In our implementation the screen representation and interaction capabilities are given by TkWin, a Graphical User Interface Management System (GUIMS) provided in the Napier88 system. Diagrams are drawn in a canvas (a widget included in TkWin's toolkit). Canvas Objects are used to implement the interactive component of VOicons and VOconnections. For VOconnections, drawing procedures (which are now part of the Napier88 graphical library) are also needed to produce the line style associated with the VO - these implement the procedure definitions included in the LineStyle type (data types are shown in Appendix C).

The Visual Objects can be combined, by joining a pair of VOicons with one or more VOconnections, to form diagrams. A diagram is supported by a graph data structure which provides the necessary information on how the Visual Objects are linked.

5. Specifying the Semantics of Modelling Techniques

5.1 VC-t - A Formal Language for the Specification of Modelling Techniques' Semantics

5.1.1 Introduction

VC-t is a formal specification language to express the semantics of Modelling Techniques (MTs). It is an implementation of the Visual Concepts formalism in which the MTs' constructs are specified as sets. The main purpose of the language is to express the semantic component of the MT and for that it uses a form of predicate logic with equality.

The language is not intended to be a general purpose one. We believe that a simpler language, which produces clearer and more readable specifications can be obtained if its scope is well delimited. We have established the following language requirements which were used as a guide to its design:

- its scope must be large enough to cover widely used MTs such as STD, DFD or ER;
- it must be expressive enough to capture the semantics of the MTs;
- the specifications produced with the language must be parseable (computer-readable);
- the specifications should also be human-readable - this promotes and facilitates the creation of clear and easy to understand specifications;
- code generation (for the target language) must be possible from any legal specification;
- the language has to be more than a theoretical exercise - it must be practicable and easy to use.

The specification language should also be able to capture the variations of any of the more established MTs in general use or application specific MTs (ASMTs) (usually being company defined).

Both the well known MTs and the ASMTs, cover a variety of application domains, such as office databases or industrial process modelling, each focusing on a given view of the domain: static, dynamic or architectural. The VC-t language must be able to capture the semantics of any of those MTs. In the case of the ASMTs, the specification task may have an additional benefit: uncovering eventual faults or inconsistencies in the semantics of the MT.

The semantics of the MT being modelled are expressed through constraints, constituting the semantic component of the VCs which model the MT constructs.

Because it has been intentionally designed to express the semantics of MTs as described above, the VC-t language provides the necessary expressive power and, simultaneously, it has the potential to lead to more readable and clear specifications than a general purpose language.

A number of specifications of some well known MTs have been produced which prove the expressive power of the language, e.g. Data Flow Diagrams (DFD), State Transition Diagrams (STD) and Entity Relationship (ER) diagrams. The readability of those specifications can be confirmed by the reader looking in Appendix B.

A compiler has been designed and implemented for the VC-t language so specifications can be parsed and their correctness checked at the lexical, syntactical and semantic levels. This is carried out by the front-end of the compiler and is independent of the software system used to support the design tools (target language). The tools are automatically generated by the back-end of the compiler which produces code for the chosen target language. If the software system is changed, only the back-end of the compiler has to be replaced. The lexical and syntactical specifications for the VC-t language are presented in Appendix D. The syntax is expressed by a BNF description.

The language is aimed at the automatic generation of tools. We concluded that the use of constraints is the right approach to build specifications intended to be used for the automatic generation of design tools (which themselves are also constraint-based). In Section 4.3.2.1 on page 41 we gave support to this claim.

The last language requirement on the bulleted list refers to its ease of use. We want the user to be able to easily and quickly obtain a specification. That is only achievable if the language has a steep and short learning curve. We must have in mind that the language users will be modelling technique designers (or at least have a fair knowledge of software modelling) but will not necessarily have any expertise in logics or constraints/semantic rules definition.

5.1.2 The Structure of a Formal Specification

Having a description in natural language of a MT, the VC-t specification language is then used to formally express the MT. In this way, a specification is obtained which includes the following aspects of the MT:

- its concepts and their properties;
- the semantics, expressed as constraints.

These two aspects are described in two main sections: the 'Preamble' and the 'Semantic Constraints'. The 'Preamble' contains declarations of all the VCs, which include their graphical representation and properties. Any necessary auxiliary sets must also be declared. In the section 'Semantic Constraints' each constraint is specified by both a description in natural language and a corresponding sentence in a form of predicate logic with equality. The natural language description adds legibility to the specification and is used to give meaningful messages to the final user during the editing process (*semantic feedback*). A formal specification is also needed, as a non-ambiguous description of the constraint, for the automatic generation of executable code.

Consider for example the STD modelling technique. Its specification includes the concepts: 'StartState', 'IntermediateState', 'FinalState' and 'Transition'; a property of any State is the 'name' and properties of Transition are the 'transition condition', 'origin' and 'destination'. Semantic constraints determine which concept configurations are allowed when drawing a diagram. As mentioned above, a constraint has both a description in natural language, for instance, "on a STD, a FinalState cannot have out-transitions or loop-transitions", and a formal specification. The formal aspects of the VC-t language will be presented later, but just to give a flavour of a constraint specification, for the above constraint we would have:

```
FORALL f : FinalState • f BELONGING Finals(std) IMPLIES
    NOT EXISTS t : Transition • t BELONGING Transitions(std) AND
        (origin(t) = f)
```

The constraint can be read as: for each FinalState 'f' in a STD diagram there cannot exist a Transition 't' which has 'f' as its origin. This includes both the case of a loop-transition, in which the destination of the Transition 't' is also the FinalState 'f', and the case of an out-transition, in which the destination of the Transition 't' is not the FinalState 'f'.

We have obtained several complete specifications, included in Appendix B, which constitute a meaningful representation of the MTs' state-of-the-art, and therefore assert the expressiveness of the VC-t specification language. Below we introduce a simple example of how to use the VC-t language to express the semantics of a non-standard MT. This example constitutes a tutorial for the VC-t language: the way the example specification is built is presented step-by-step with explanatory annotations. This is meant to be just a 'getting started' approach to the language, formal aspects being discussed later.

5.1.3 A Simple Specification

A very simple Modelling Technique (MT) was invented for which a detailed non-formal description in natural language (English) will be presented. Afterwards, an explanation of how to obtain a formal VC-t specification from this natural language description will be given.

5.1.3.1 The SimpleMT

This is a very simple MT, created just for exemplification purposes, which has only three constructs, the 'State', the 'StartState' and the 'Event'. The graphical representations of the constructs are shown in Figure 7.

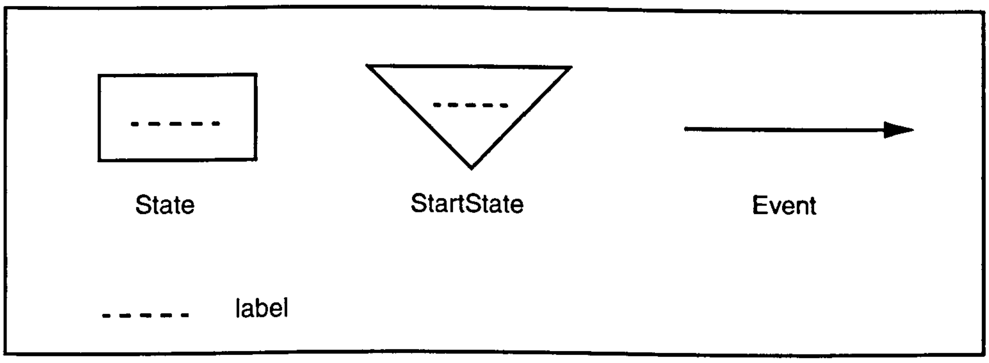


FIGURE 7. The SimpleMT graphical representation.

The State is depicted by a rectangle and the StartState by an inverted triangle, both have a label with a name, which is unique amongst their instances; the Event is depicted by an arrow.

A SimpleMT diagram is, structurally, a connected graph in which the nodes and edges are graphically represented, respectively, by icons and connections. In the SimpleMT the icon types are the StartState and the State; the only connection type is the Event.

In a diagram there must be one and only one StartState (instance of the StartState Icon type); the other nodes (one or more) are States (instances of the State Icon type). The StartState can only be connected to States by outgoing Events. Any pair of States is connected at most by two Events, one in each direction. Loop Events, i.e. Events that connect a State to itself, are not allowed. The minimum diagram is composed by a StartState connected to a State by an outgoing Event.

5.1.3.2 The VC-t Specification

The VC-t specifications can be used as input to a compiler. Because that compiler is not able to parse mathematical symbols, such as '$\forall$' or '$\Rightarrow$', an alternative notation based in English lexemes must be used. The correspondence between the mathematical notation and the English based one, for the symbols used in this section, is presented in Table 1.

Mathematical Notation	English Based Notation
$\neg$	NOT
$\exists$	EXISTS
$\forall$	FORALL
$\Rightarrow$	IMPLIES
$\Leftrightarrow$	DOUBLEIMPLICATION
$\in$	BELONGING
$\mathbb{P}$	P
$\wedge$	AND
$\vee$	OR
#	CARDINALITY

TABLE 1. Correspondence between the mathematical and English based notations.

The Lexical Analyser of the VC-t compiler accepts, for most lexemes, multiple alternatives. For instance, 'FORALL' may also be written 'UNIVERSAL', 'P' may be written 'POWERSET', 'BELONGING' may be 'MEMBERSHIP', 'IN' or 'E'.

In this section the English based notation is used, so that the obtained VC-t specification may be compiled. In subsequent sections the English based notation is used in situations relating to a VC-t specification which is meant be used as input to the compiler, in all other situations the mathematical notation is used.

The first aspect of the SimpleMT we must specify is its concept structure - the constructs and their properties. For this purpose we will write the 'Preamble' section.

We must declare the MT as a cartesian product of the power sets of its constructs.

```
"SimpleMT" SEMANTICS_SPECIFICATION

PREAMBLE

A.MT_MODEL

    BEGIN

    simpleMT = P StartState x P State x P Event

    END
```

Now, we divide the power sets into the ones that are represented by icons from the ones that are represented by connections and also declare extractors to isolate each one of them from the MT. In section B1 we declare the extractors for the power sets represented by icons.

```
B.SET_EXTRACTORS_DECLARATIONS

    B1.ICONS

        BEGIN

        StartStates : simpleMT -> P StartState
        States : simpleMT -> P State

        END
```

In section B2 we've declared the extractor for the only power set represented by connections.

```
    B2.CONNECTIONS

        BEGIN

        Events : simpleMT -> P Event

        END
```

Here we can declare auxiliary sets and extractors as expressions composed by those declared in section B.

```
C.SETS_DEFINITIONS

    BEGIN

    AnyState == StartState U State
    AnyStates == StartStates(simpleMT) U
                                States(simpleMT)

    END
```

In this section the properties of the MT sets are expressed. It is important to notice that all sets represented by connections must have a property that returns the icon connected at its start, and another returning the icon connected at its end. The equality property is expressed to allow for comparisons during diagram traversal.

```
D.SET_PROPERTIES

    BEGIN

    StartState HAS_PROPERTIES

    name : StartState -> String
    equal : StartState x StartState -> Boolean

    State HAS_PROPERTIES

    name : State -> String
    equal : State x State -> Boolean

    Event HAS_PROPERTIES

    origin : Event -> AnyState
    destination : Event -> AnyState
    equal : Event x Event -> Boolean

    END
```

Now, we must express the semantics already described informally above, using constraints formally written in a logic-based style. For this purpose we will write the 'Semantic Constraints' section. The first constraint (C1) expresses the statement: 'both [StartState and State] have a label with a name, which is unique amongst their instances'. The constraint numbering must be 'C$:' where '$' takes a sequential integer value starting at '1'. 'FORALL' is the Universal Quantifier; it binds the variables s1 and s2 ranging over the elements of the VC type AnyState, which is the union of StartState and State; s1 and s2 belong to the SimpleMT set extracted by the function AnyStates.We are then saying that for all possible pairs of states in the diagram, for instance (s1, s2), if the name of s1 is equal to the name of s2, then s1 and s2 must be the same.

```
SEMANTIC_CONSTRAINTS

    BEGIN

    C1: "Names are unique amongst States and
        StartState"

    FORALL s1, s2 : AnyState • s1, s2 BELONGING
        AnyStates(simpleMT)
        IMPLIES
        (name(s1) = name(s2) IMPLIES s1 = s2)
```

Constraint C2 is an instantiated Predicate Logic (PL) statement. Constraints are divided into two main groups: the quantified PL statements and the instantiated PL statements. C2 belongs to the second group and it is, more specifically, a cardinality constraint. It expresses that 'In a diagram there must be one and only one StartState'.

```
C2: "There is exactly one StartState"

CARDINALITY StartStates(simpleMT) = 1
```

This constraint is an example of a quantified PL statement with nested quantifications. There may be any number of nesting levels and each level can either be an universal quantification or an existential one. C3 expresses: 'the StartState can only be connected to States by outgoing Events'. Note that the PL statement in the constraint also expresses that no loop Events are allowed in the StartState (for the special case of a loop Event we have 'destination(e) = s AND origin(e) = s'), which is also expressed below in C5. But the fact that we have overlapping constraints does not constitute an error in the specification. It would be an error only if they were conflicting constraints (over specification).

```
C3: "The StartState does not have incoming Events"

FORALL s : StartState • s BELONGING
    StartStates(simpleMT)
    IMPLIES
    NOT EXISTS e : Event • e BELONGING
        Events(simpleMT)
        AND
        (destination(e) = s)
```

This constraint reflects the statement 'any pair of States is connected at most by two Events, one in each direction'. In spite of not being defined in the informal specification whether the StartState is included in this constraint, we assumed that it is. Note that constraint C3 forbids incoming Events onto the StartState. However, C3 and C4 are not conflicting; C4 does not impose the existence of incoming Events, it just says that there is at most one incoming Event starting from a particular State. The StartState is conditioned by the two constraints.

```
C4: "There is at most one Event in each direction
    connecting two States or the StartState with a State"

FORALL s1, s2 : AnyState • s1, s2 BELONGING
    AnyStates(simpleMT)
    IMPLIES
    NOT EXISTS e1, e2 : Event • e1, e2 BELONGING
        Events(simpleMT)
        AND
        ((origin(e1) = s1 AND destination(e1) = s2)
            AND
            (origin(e2) = s1 AND destination(e2) = s2))
```

C5 expresses: 'loop Events, i.e. Events that connect a State to itself, are not allowed'.

```
C5: "Each Event must connect a pair of different States
    or the StartState to a State - loop Events are not
    allowed"

FORALL e : Event • e BELONGING
    Events(simpleMT)
    IMPLIES
    (NOT (origin(e) = destination(e)))
```

Every specification must define, either implicitly in several constraints or explicitly in one single constraint, what is the minimum diagram that can be obtained and still makes sense. In our example this is expressed by C6; it relates to the statement: 'the minimum diagram is composed by a StartState connected to a State by an outgoing Event'.

The specification ends with a dot in the beginning of a line after the 'END' of the Constraints Section.

```
C6: "The minimum diagram is composed by a
StartState connected to a State by an outgoing Event"

EXISTS s : StartState • s BELONGING
    StartStates(simpleMT)
    AND
    EXISTS t : State • t BELONGING
        States(simpleMT)
        AND
        EXISTS e : Event • e BELONGING
            Events(simpleMT)
            AND
            (origin(e) = s AND destination(e) = t)

END

.
```

5.2 The Formal Aspects of VC-t Specifications

It is vital that the specifications have a formal basis to make them unambiguous. The formal aspects are presented using the mathematical symbolism proposed by [Woodcock88].

A VC-t specification has two main sections: the **Preamble** where the Modelling Technique (MT) and its VCs are specified; and the **Semantic Constraints** which consists of a sequence of predicate logic sentences. The formal and mathematical details of each section are presented in what follows.

A specification of a MT starts with an identifier (its name) followed by the reserved word 'SEMANTICS_SPECIFICATION'. The complete set of reserved words to be used in a specification can be seen in Appendix D; for simplicity they will be omitted in the remaining sections.

5.2.1 The Preamble Section

The Preamble expresses the MT in terms of its VC types and declares all the sets that will be used in the specification. It comprises four sub-sections: the MT model, set extractors declarations, sets definitions and set properties.

To specify a MT and its VC types we have employed elementary set theory. The **MT model** is the cartesian product of the power sets of all its VC types. A generic MT is expressed as:

'$mtx = \mathbb{P}\ VCt_0 \times \mathbb{P}\ VCt_1 \times .. \times \mathbb{P}\ VCt_n$'

where: mtx is an identifier, called the name of the MT; VCt_0 to VCt_n are the VC types defined for the MT.

As an example, the SimpleMT described in Section 5.1.3.1 has its model specified as:

'simpleMT = P StartState x P State x P Event'

In the **set extractors** sub-section, power sets are divided into the ones that are represented by icons and the ones that are represented by connections. This is not a closed classification; at the moment our diagrams only contain icons and connections, yet, in the future, other diagram constructs may be added, e.g. object inclusion. It is necessary to specify which power sets are represented by icons and which are represented by connections, so that the code generator can establish the mapping from each diagram component (the icons and connections) to the correct underlying abstract graph components (nodes and edges).

For each power set, an extractor is defined. An extractor is a triple $<mt_d, ext, vc_r>$, these being: mt_d the MT for which the extractor is declared, called the domain MT; ext an identifier, called the name of the extractor; vc_r a power set, called the range VC.

So, for the MT named 'mtx' we have:

'$ext_i : mtx \rightarrow \mathbb{P}\ VCt_i, i \in \{0..n\}$'

For the StartState power set, included in the model of the SimpleMT, which elements are represented by icons, the following extractor has been defined under 'B1.ICONS':

'StartStates : simpleMT $\rightarrow$ P StartState'

A VC is a maximal set in the sense that its values may belong to just that VC type. No subtyping is allowed amongst VCs. However, it is possible to specify auxiliary sets in the **sets definitions** sub-section, as a union of VC sets. For instance, for the SimpleMT the following set was defined:

'AnyState == StartState $\cup$ State'

Auxiliary sets can also be specified in extension, which is useful e.g. in the specification of pre-defined label strings. For instance cardinality labels for the ER technique could be specified as:

'Cardinality == {"1, 1", "1,n", "n, m"}'

Auxiliary sets aim at simplifying the constraints in the Semantic Constraints section. See the example of Figures 8 and 10. With other constraints more meaningful simplifications may be obtained.

Properties are declared for each VC in the **set properties** sub-section. Properties are the constrainable components of VCs. They are used in predicate logic statements of the constraints expressed in the Semantic Constraints Section. For instance, the following statement has been used in constraint C5 of the SimpleMT specification:

'$\neg$(origin(e) = destination(e))'

where: 'e' is an instance of 'Event'; 'origin' and 'destination' are properties of 'Event' declared in the set properties sub-section.

Set properties are defined as triples $<vc_d, prop, t_r>$, where: vc_d is a set or a power set, called the domain; prop is an identifier, called the property name; t_r can be a VC type, a predefined type (String, Natural or Boolean) or an auxiliary set, called the range. For example, the equality property used in the predicate logic statement above, has been defined for the SimpleMT 'StartState' as:

'equal : StartState x StartState $\rightarrow$ Boolean'

5.2.2 The Semantic Constraints Section

A semantic constraint is a rule expressed as a sentence in a form of predicate logic with equality.

The constraints can be divided into two groups: the *instantiated predicate logic statements* and the *quantified predicate logic statements.*

An **instantiated predicate logic statement** is simply a constraint where the propositions have no variables, i.e. the propositions are not quantified.

The following is a valid constraint of the kind 'instantiated predicate logic statement', it has been defined for the SimpleMT:

'# StartStates(simpleMT) = 1'

A **quantified predicate logic statement** is a constraint where the propositions are obtained by quantification. This means that all the variables in the statement must be bound by quantifications. Quantifications can be existential or universal. Nested quantifications are allowed with any number of levels.

An existential quantification is specified as:

'$\exists x : T \bullet x \in S \wedge P(x)$'

where: x is a variable bound by the existential quantification; T and S are sets such that S is a subset of T, expressed as $S \subseteq T$; P(x) denotes any boolean expression with one or more predicates on the variable x.

Likewise, a universal quantification is specified as:

'$\forall x : T \bullet x \in S \Rightarrow P(x)$'

N-ary predicates, denoted by $P(x_1, x_2, .. x_n)$ with the variables $x_1, x_2, .. x_n$ ranging over the same or different sets, are also allowed. As for unary predicates, all the variables must be bound by quantifications. E.g. $\forall x : T \bullet x \in S \Rightarrow \exists y : Q \bullet x \in R \wedge P(x, y)$

Below is a constraint of the kind 'quantified predicate logic statement', also defined for the SimpleMT:

'$\forall$ s : StartState $\bullet$ s $\in$ StartStates(simpleMT) $\Rightarrow$
    $\neg\exists$ e : Event $\bullet$ e $\in$ Events(simpleMT) $\wedge$
        (destination(e) = s)'

Please refer to Section 5.1.3.2 for an explanation of this constraint and the one from the previous example.

The variables in the constraints, bound by the quantifications, may range over one VC type or a union of several VC types.

A **predicate logic statement** is a boolean expression including the usual boolean operators. i.e. 'negation' ($\neg$), 'implication' ($\Rightarrow$), 'double implication' ($\Leftrightarrow$), 'and' ($\wedge$), 'or' ($\vee$). A boolean expression can include one or more natural expressions combined by the following natural comparison operators: 'greater' (>), 'less' (<), 'at least' ($\geq$), at most' ($\leq$).

'equality' (=) is allowed over sets and the pre-defined types String and Natural.

The language proved to be able to capture most of the constraints defined by the semantics of several standard modelling techniques covering static aspects of software (data modelling) and dynamic aspects (process modelling). However, we do not claim that the language is able to express all the semantics. For example, a constraint asserting that a diagram must be connected, cannot be expressed in VC-t. In order to achieve that expressiveness, one of the main advantages of the VC-t language, its simplicity, would have to be compromised.

Alternatively, we have provided the possibility of including function calls in the specifications. These functions constitute a library that extends and completes the VC-t language. For the code generation process, implementations of the library functions must be provided in the target language. Two of these functions have already been defined: 'Connect(i)', where 'i' is an icon, and 'uniqueName(extractor)'. The former is applied to the whole diagram, given an icon it returns a set of icons that are connected between them and which include that icon. The latter is a boolean function, also applied to the whole diagram, in the sense that it performs its complete traversal, but it checks only the VCs determined by the extractor given as a parameter; it returns 'true' if all the names are different. While the first function adds expressiveness to the language, as it captures a class of constraints that the VC-t language in unable to capture, the 'uniqueName' function only simplifies the specifications. The uniqueness of names given to VOs in a diagram (for instance, ER entities) can be expressed in the VC-t language, see for example constraint C1 in the SimpleMT specification (Section 5.1.3), but this constraint is so common that a function replacing the full specification in VC-t becomes very useful.

We have tried to reduce the complexity of the formal aspects of the language to the necessary minimum. The language is aimed at users that do not necessarily have expertise in formal methods or a solid mathematical background.

5.3 Type Checking in the VC-t Language

At one stage of implementing the VC-t language, we realised that in some situations, the code generated from correct VC-t specifications would fail the type checking in the target system (Napier88). This results from the fact that type checking is stronger in Napier88 than in VC-t. This problem is further explained in the following and a possible solution is presented.

Type checking a VC-t specification is essential to assert its correctness. The language supports the following types: the built-in types - Natural, String and Boolean, and the user defined types - the VCs.

The Semantic Constraints section is type checked against the information provided in the Preamble. For instance, in the constraint extract:

'FORALL f : FinalState • f BELONGING Finals(std) '

the compiler must check if the extractor 'Finals(std)' has been defined for the VC 'FinalState'.

The example above is fairly straightforward, but the complexity of type checking 'predicate logic statements', which include VC properties, is far greater. Consider the constraint shown in Figure 8 which is included in the SimpleMT specification.

In this example the auxiliary set 'AnyState', defined in the Preamble section is being used. Auxiliary sets defined as unions of VC types can also be included in a specification (for instance 'AnyState', which has been defined as the union between 'StartState' and

```
C4: "There is at most one Event in each direction connecting two States or the
     StartState with a State"

FORALL s1, s2 : AnyState • s1, s2 BELONGING AnyStates(simpleMT)
     IMPLIES
     NOT EXISTS e1, e2 : Event • e1, e2 BELONGING Events(simpleMT)
          AND (
               (origin(e1) = s1 AND destination(e1) = s2)
               AND
               (origin(e2) = s1 AND destination(e2) = s2)
          )

     where AnyState is defined as:
          AnyState == StartState U State
```

FIGURE 8. A semantic constraint defined for the SimpleMT.

'State'). They have been introduced in the VC-t language definition as a way to simplify constraints specification.

In order to type check the constraint in Figure 8, we must type check both the quantification expressions as explained above, and the predicate logic statement:

'(origin(e1) = s1 AND destination(e1) = s2) AND (origin(e2) = s1 AND destination(e2) = s2)'

Type checking the statement above implies checking each of its four components, e.g. 'origin(e1) = s1'. The properties 'origin' and 'destination' have been declared in the Preamble section of the specification; for the former we have:

'origin : Event -> AnyState'

In generic terms, a property declaration has the format:

'propX: A -> B'

where 'propX' is the property name; 'A' is a VC type or a power set of VC types (in this section we will only analyse the case of unary properties, when 'A' is a single VC type; however, the conclusions can easily be generalised for n-ary properties), and 'B' is a type or an auxiliary set. 'A' is called the *domain* of 'propX' and 'B' is its *range*.

In the Semantic Constraints section this property may then be used in a generic expression with the format:

'propX(a) = b'

where ‘a’ is an element of ‘A’ and ‘b’ an element of ‘B’ (for a correctly typed expression).

The process of type checking the above expression involves two actions:

- A1 - type check the argument of the property.
- A2 - type check the values on both sides of the comparison, i.e. does ‘b’ belong to the range of ‘propX’?

The expression ‘origin(e1) = s1’ in the example above is correctly typed because it conforms to the property declaration:

‘origin : Event –> AnyState’

In fact, argument ‘e1’ is an Event and the result is being compared with the value ‘s1’ an element of AnyState.

The possible situations can be seen in Table 2.

DECLARATION / CALL	**Domain VC**	**Domain Aux**	**Range VC**	**Range Aux**
Argument - VC	OK	IMP	IRV	IRV
Argument - Aux	**P1**	IMP	IRV	IRV
Compared Value - VC	IRV	IRV	OK	**P2**
Compared Value - Aux	IRV	IRV	W	OK

TABLE 2. Type checking situations.

Caption:
IMP: Impossible
IRV: Irrelevant
W: Wrong specification
P1, P2: type checking Problems

The columns of Table 2 represent the domain and range of a property declaration; each can either be a single VC type or an auxiliary set. These correspond to ‘A’ and ‘B’, respectively, in the generic property declaration above.

The rows represent the argument and the value that is compared with the value returned; respectively ‘a’ and ‘b’, in the example above.

The darkened cells (marked ‘IRV’), are not relevant because when type checking a property, we are only interested in matching its argument with the domain in the property declaration and the compared value with the range. These two situations are expressed by the white cells.

Action A1 consists of the situations represented by the four top left table cells. Denoting by (l, c) the cell in line 'l' and column 'c', we have:

(1,1) if both domain and argument are VC types, it is only necessary to check if they are the same type;

(1,2), (2,2) the domain cannot be an auxiliary set;

(2,1) in this situation we have a type checking problem - the domain is a VC type but the argument is an auxiliary set, a union of VC types.

Action A2 consists of the situations represented by the four bottom right table cells:

(3,3), (4,4) if range and compared value are both VC types or both auxiliary sets, it is only necessary to check if they are the same;

(4,3) if the range is a VC, the compared value must also be a VC, otherwise it is a specification error;

(3,4) in this situation we have a type checking problem - the range is an auxiliary set but the compared value is a VC.

There are three possible solutions for the above problems:

- the simplest one is to allow the problems to happen in the specification and defer the type checking to the compilation phase of the generated code - the problems would only be detected by the type checker of the target language (Napier88 in our implementation);
- another simple solution would be to eliminate auxiliary sets from the VC-t language;
- a more complex solution is to provide some mechanism at the specification level, to solve the type conflicts.

The first solution has the disadvantage of producing target code that may not compile or behave correctly; for this to happen it is only necessary for the type checking mechanism in the target language to be stronger than the one in VC-t (which is the case in Napier88). This would provoke error messages produced by the target system to be presented to the designer. These messages would make no sense in the context of the specification written by him/her and therefore would be potentially confusing. These situations are common in multi-language systems; we have chosen to avoid them by deliberately not allowing incorrect target code to be produced. The first solution is, therefore, not satisfactory.

The second solution increases the complexity of specifications which use auxiliary sets. If, for example, we eliminate the auxiliary sets used in the constraint of Figure 8, the specification would become the one shown in Figure 9.

Notice that two VC-t constraints are now necessary to express one semantic constraint and in other cases the expansion may be greater. In addition to the obvious loss of simplicity, there is a problem that makes the writing of a specification more complex: the enlargement of the semantic mismatch between the description of the constraint in natural language

```
C4: "There is at most one Event in each direction connecting two States or the
    StartState with a State"

FORALL s1, s2 : State • s1, s2 BELONGING States(simpleMT)
    IMPLIES
    NOT EXISTS e1, e2 : Event • e1, e2 BELONGING Events(simpleMT)
        AND (
            (origin(e1) = s1 AND destination(e1) = s2)
            AND
            (origin(e2) = s1 AND destination(e2) = s2)
        )

FORALL s1 : StartState • s1 BELONGING StartStates(simpleMT)
    IMPLIES
    NOT EXISTS s2 : State • s2 BELONGING States(simpleMT)
        AND
        EXISTS e1, e2 : Event • e1, e2 BELONGING Events(simpleMT)
            AND (
                (origin(e1) = s1 AND destination(e1) = s2)
                AND
                (origin(e2) = s1 AND destination(e2) = s2)
            )
```

FIGURE 9. Specification of the semantic constraint of Figure 8 without auxiliary sets.

and the formal VC-t specification. For these two reasons we have decided to abandon the second solution.

The third solution demands some further research. The obtained results are presented below.

5.3.1 Type Checking Safety Mechanisms

5.3.1.1 The Requirements

The mechanisms to be introduced in the specifications must conform to the following requirements.

Requirement 1

The VC types in the specification must match the MT constructs identified by the designer.

As an example, consider the 'SimpleMT' description: the designer has identified the constructs in the sentence: [the SimpleMT] has only three constructs the 'State', the 'StartState' and the 'Event'. If he/she would be then forced to produce several VC types to specify one construct, the specification would not match what had been expressed. As an example consider the situation of the ‘Event’ construct being specified by two VC types: ‘StateToStateEvent’ and ‘StartStateToStateEvent’.

In conclusion, the mechanisms introduced must not provoke the creation of unreal or artificial VC types.

Requirement 2

The generated code must be correct, i.e. it must be in conformance with the target language syntax and semantics.

This means that a complete type check should be done at specification level by the VC-t compiler.

Requirement 3

The semantic mismatch, that is already present between constraints descriptions in natural language and their specification in VC-t, should not be enlarged.

The language supports a one to one matching from natural language descriptions of MT constraints to VC-t semantic constraints (we are making this claim based on the specifications of MTs already written) - this expressiveness feature should not be compromised.

Requirement 4

The property overloading feature, provided by the VC-t language, must be maintained.

The VC-t language allows for property overloading; for example, in Figure 10 the ‘name’

```
C12: “All elements of the diagram must be named”

FORALL a : DFDelements •
    a BELONGING DFDelementsExtractors IMPLIES

    (NOT (name(a) = “”))

where DFDelements is defined as:
    DFDnodes == Process U Datastore U ExternalEntity U Dataflow
```

FIGURE 10. A semantic constraint defined for DFD.

property has been defined for each one of the VC types ‘Process’, ‘Datastore’, ‘ExternalEntity’ and ‘Dataflow’.

The ‘name’ property will have a different implementation in the target language for each VC type, however this fact is transparent to the designer. The property overloading feature is implemented by generating in the target language, a unique function name for each property. The generated function name is produced in the following way: the property name is prefixed by the construct name and by a letter (I or C) specifying if it applies to an Icon or a Connection, respectively. For the example given in Figure 10, four function names would be generated:

'process_I_name(...)', 'datastore_I_name(...)', 'externalEntity_I_name(...)' and 'dataflow_C_name(...)'

for each value of the variable 'a', the implementation corresponding to its type would be used. Again, we believe that this feature allows us to produce specifications closer to the natural language description of the constraints.

5.3.1.2 The Solutions

Solutions for the problems identified before must be provided; they must conform to the requirements above and, as far as possible, respect the following guidelines:

- The VC-t language syntax should be maintained.
- The specifications should remain simple and readable.

A Solution for the 'Domain/Argument' Problem

As described above ('P1' in Table 2), the following type checking problem may occur:

> the domain of a property is a VC type but the argument is an auxiliary set, i.e. a union of VC types.

The semantic constraint in Figure 11, which has been defined for the SimpleMT, would

```
C1: "Names are unique amongst States and
    StartState"

FORALL s1, s2 : AnyState • s1, s2 BELONGING
    AnyStates(simpleMT)
    IMPLIES
    (name(s1) = name(s2) IMPLIES s1 = s2)

where AnyState is defined as:
    AnyState == StartState U State
```

FIGURE 11. A domain/argument type unsafe semantic constraint.

fail a type check in a strongly typed system. This would happen because 'name', being a property, must have as the argument a value belonging to a single VC type, but it is being applied to an element of an auxiliary set, i.e. the variables 's1' and 's2' belong to 'AnyStates' when they should belong to either 'StartStates' or 'States'.

The proposed solution is to **relax the type information** in the VC-t specification. For this purpose a generalisation mechanism has been provided, through a *meta property*, i.e. a property that applies to any VC types or their unions. This meta property has been named 'generalise' and included in a VC-t library.

For a property PropX: $A \rightarrow B$, the signature of 'generalise' is:

generalise(propertyName: String; extractorName: String; vcVariable: Set $\rightarrow$ B)

The semantics of 'generalise' are:

'Generalise propX(a) defined for Ai(mt) with $1 < i < n$, $a \in A1(mt) \cup A2(mt) \cup .. \cup An(mt)$, $n \in$ Natural'

where 'mt' is a modelling technique and 'Ai(mt)' is an extractor declared for that technique.

The use of the 'generalise' meta property allows a property, that has been overloaded in a number of VC types, to be used without specifying to which of those types it will be applied. Semantically, the difference between this mechanism and property overloading is that, in a specification, the semantics of the generalised properties are expected to be equivalent. We will come back to this point after the presentation of an example.

In this example, 'generalise' is applied to the specification shown in Figure 11. Using the

```
C1: "Names are unique amongst States and
    StartState"

FORALL s1, s2 : AnyState • s1, s2 BELONGING
    AnyStates(simpleMT)
    IMPLIES
    (generalise("name", "AnyStates", s1) = generalise("name", "AnyStates", s2)
        IMPLIES s1 = s2)

where AnyState is defined as:
    AnyState == StartState U State
```

FIGURE 12. The type safe version of the semantic constraint in Figure 11.

meta property, the specification becomes the one in Figure 12. This means that the 'name' property may now be used for values of any of the VC types included in the definition of the auxiliary set 'AnyState'. The constraint is now correctly typed: the variables 's1' and 's2' belong to the auxiliary set 'AnyState', which corresponds to their type when used as arguments of 'generalise'.

Generalising a property named "propX" is only possible if in each VC type within the auxiliary set, used as the argument of 'generalise', a property has been defined with that name. Moreover, all those properties must have the same range.

A constraint that includes the meta property 'generalise', in spite of being correct from a type checking perspective, may have its semantics completely wrong. From the fact that two properties, defined for two VC types, have the same name and the same range, one cannot infer that they have the same semantics. It is the responsibility of the specifier to ensure that this is indeed the case, or, in other words, that the semantics of the generalisation is correct, i.e. that it makes sense.

A Solution for the 'Range/Compared-Value' Problem

The second problem ('P2' in Table 2) that may occur is:

the range of a property is an auxiliary set but the compared value is a VC type.

In the example of Figure 13 the 'destination' property has been declared for the 'Event' VC type as:

'destination : Event -> AnyState'.

```
C3: "The StartState does not have incoming Events"

FORALL s : StartState • s BELONGING
    StartStates(simpleMT)
    IMPLIES
    NOT EXISTS e : Event • e BELONGING
        Events(simpleMT)
        AND
        (destination(e) = s)
```

FIGURE 13. A range/compared value type unsafe semantic constraint.

So, the expression: 'destination(e) = s' does not type check correctly under strong typing rules, 's' is a StartState when it should be an AnyState.

The solution is to **increase the type information** in the VC-t specification. For this purpose, another meta property with the following signature, has been included in the VC-t library:

vcTypeOf(vcVariable: VCtype → String)

The meta property 'vcTypeOf' can be applied to an instance of any VC type, returning a string with its type name.

Using the 'vcTypeOf' meta property, the specification of Figure 13 becomes the one in Figure 14.

```
C3: "The StartState does not have incoming Events"

FORALL s : AnyState • s BELONGING
    AnyStates(simpleMT)
    IMPLIES
    NOT EXISTS e : Event • e BELONGING
        Events(simpleMT)
        AND
        ((destination(e) = s) AND (vcTypeOf(s) = "StartState"))

    where AnyState is defined as:
        AnyState == StartState U State
```

FIGURE 14. The type safe version of the semantic constraint in Figure 13.

5.3.1.3 Conclusions

The VC-t language is deliberately weakly typed to reduce the semantic mismatch between natural language constraints descriptions and formal constraints specifications. This feature, however, may raise some problems in the code generation, namely when the target language provides stronger typing mechanisms. As a result, the generated code may face problems in the target system. The eventual problems will usually produce error feedback to the user (the specifier, in this case), for instance as error messages, which would be meaningless in relation to the specification environment.

To avoid these problems, two meta properties were introduced, as part of a VC-t library. The meta properties, 'vcTypeOf' and 'generalise', may be used to produce type safe specifications; these are guaranteed to be type correct in a strongly typed system.

It is important to notice that a specification using the meta properties could be automatically obtained by transformations applied to a standard VC-t specification. Note that these transformations are done exclusively at the specification level and also that the resulting specification will still conform to the unchanged VC-t syntax.

6. The Generic Visual Language for MT Editors

consistent *adj.* compatible or in harmony
In The Concise Oxford Dictionary of Current English [Allen91]

consistent state: compatible or in harmony with the semantics (our definition).

6.1 Introduction

Current software design tools supporting diagram based modelling techniques (MTs) are not satisfactory. Editors for MTs such as State Transition Diagrams, Dataflow Diagrams or Petri Nets, are always confronted with a problem: how much guidance should be given to the user throughout the editing task? Not enough guidance allows the diagram to evolve to non plausible configurations and may provoke the user to feel lost in the editing process. At the other extreme, if too much guidance is provided, the user feels like being shepherded through the diagram drawing; this results in an obtrusive and unfriendly system. Current tools normally offer a trade-off solution based on the introduction of some semantic constraints in the diagram editor to forbid a number of operations. To assert the correctness of the diagram, the user must explicitly request it to be checked. We believe this solution is not satisfactory. All the semantic constraints should be embedded in the editor in order to allow automatic diagram validation. The challenge is: how to do this without limiting the user's freedom during the editing task? I propose an approach that provides a solution to this problem (this work was introduced in [Serrano94] and further developements were reported in [Serrano95]).

A large number of software design tools have been developed to support MTs. Such tools must provide a visual editor to draw the diagram and some way of evaluating the produced diagrams according to the semantics of the underlying notation. In this chapter we will show how current tools supporting MTs can be improved by introducing a new approach to diagram editing and evaluation.

Depending on how the semantics of the MT are used in the tool, two different ways of performing diagram evaluation can be considered:

- the semantics are not included in the diagram editor - the evaluation is done by an explicit user request;
- the editor uses the semantics at design time - the evaluation is carried out as the diagram is drawn by the user.

The first approach demands a separate validation phase. Normally the diagram is translated into a textual representation which will then be parsed. The problem with this approach is that the editor, having no knowledge about the diagram semantics, is unable to offer guidance to the user during the diagram editing task.

The second approach addresses this problem by including the semantic constraints in the diagram editor, which are then enforced at editing time. As a result, any inconsistency in the diagram will violate semantic constraints and is therefore forbidden. So, with this approach another problem arises: the user is shepherded through the editing process.

Our challenge can, therefore, be expressed as: how to embed the semantics of the MT in the diagram editor without obtrusively limiting the user's freedom? Some systems try to answer to this question by relaxing a set of semantic constraints at design time. This provides some freedom to the user by enabling him/her to produce temporarily inconsistent diagram states. The full set of constraints is only applied when a diagram validation is requested. This approach only partially solves the problem: the user still does not know which operations he/she is allowed to perform on the diagram; there is not a clearly defined method to isolate the set of constraints to be relaxed; a separate validation phase is still required.

We present a solution to the problem, i.e. an approach by which *all* the semantic constraints are embedded in the diagram editor and that still guarantees the user freedom during the editing task.

Constraints are used to represent the MT semantics. The system refers to these constraints in order to continuously monitor the diagram being edited. Every time the system recognises a valid configuration, the diagram is automatically stored in the repository. The user can still edit the diagram until another valid configuration is obtained. At this point, new additions or differences to the previous valid configuration will be stored. This means that *diagram validation is performed incrementally.*

The way this is done will be explained in the next section. We will also show how the system manages inconsistent states between two valid diagram configurations and why this inconsistency management is a key point of the approach and a major advantage to the user.

6.2 The approach

6.2.1 Overview of Visual Objects

In this section we describe an approach for the definition of diagram editors for MTs. As an example a systems designer may want to develop a diagram editor to support the

Entity-Relationship (ER) notation [Chen76] (see Appendix for a short description of ER). With such an editor the user[1] will be assisted during the ER diagram editing task.

In our approach a diagram is composed of *Visual Objects* (VOs) (see Section 4.4 on page 42). VOs include a logical part (the semantics) and a physical part (the properties and visual representation). VOs can either be *icons* [Chang90] or *connections*. Both icons and connections have their semantics expressed by constraints, as we will describe below, but the physical part is expressed differently. The physical part of an icon is a shape (e.g. a labelled square or a circle) while the physical part of a connection is a line style (e.g. an arrowed line or a dotted line).

6.2.2 Visual Objects' Semantics and Constraints

VOs' semantics is expressed by constraints. Each VO has associated with it a set of constraints. Constraints are increasingly being used to specify the graphical layout and behaviour of an application [Zanden91; Borning81]. In this context, a **constraint** is a rule on the properties of VOs that can be checked for validity.

Constraints are used to determine the behaviour of a VO at design time, i.e. how it acts individually and how it interacts with other VOs. This way, a VO is more than just a visual representation; it has a specific behaviour encompassing two aspects:

- **individual** - when it is isolated from other VOs;
- **community** - when it interacts with other VOs.

VO behaviour is always triggered by a user command. The VO reaction to a user command can either be a:

- **direct reaction** if the command is issued over the VO;
- **indirect reaction** if the command is issued over another VO to which it is semantically connected.

The **semantics of a VO** is the specification of its behaviour during the editing task by the use of constraints.

6.2.3 Visual Objects States

Three different states are defined for a VO's life cycle:

- **Complete**: the VO is completely specified.

1. James Odell says: "there is not such a thing as an end-user". In this context the term 'user' refers to the person using the diagram editor.

- **Accepted:** the VO specification is not complete but it is already associated with the diagram.
- **Disconnected**: the VO is incomplete and dissociated from the diagram.

When the user interacts with the diagram the behaviour of the VOs involved will depend on their state. For each VO affected, its constraints will be checked.

6.2.4 Diagram States

Depending on the VOs' states, the diagram may also reach one of three different states:

- **Valid**: all the VOs are in the complete state.
- **Inconsistent**: this is a non valid state but the valid state can be reached solely by adding new VOs or rearranging the existing ones without separation of already associated VOs.
- **Wrong**: the only possibility of reaching the valid state is by removing or dissociating VOs.

Only the third state is forbidden. This is because whenever the user issues a command causing the diagram to enter the third state, it makes no sense to proceed with the current task. In this situation, the user would get feedback from the system indicating an error.

At the beginning of this chapter we defined 'consistent state' as a state that is compatible or in harmony with the semantics, and in our case, of a MT. An inconsistent state is temporarily out of harmony: it is a transition between two consistent states. A 'wrong' state can not be considered a transition to another consistent state because the current flow of interaction must be interrupted.

This way, inconsistent intermediate states are allowed, promoting the user freedom during the design process. In Figure 15, three situations are shown, each one corresponding to one of the three different states. For this example the ER technique was used.

The concept introduced above of 'relaxing constraints' does not apply to this approach. *It is only necessary to identify the constraints that, when violated, lead to diagram configurations corresponding to wrong states.* This is exemplified in the next section.

Diagram validation is performed automatically at design time by constraint checking. With this approach there is no separation between the editing phase and the validation phase; the two are merged together. For each user action the diagram is incrementally parsed. Each time a valid state is reached the diagram is automatically stored.

The next section shows how the approach can be used to produce a diagram editor for the ER technique.

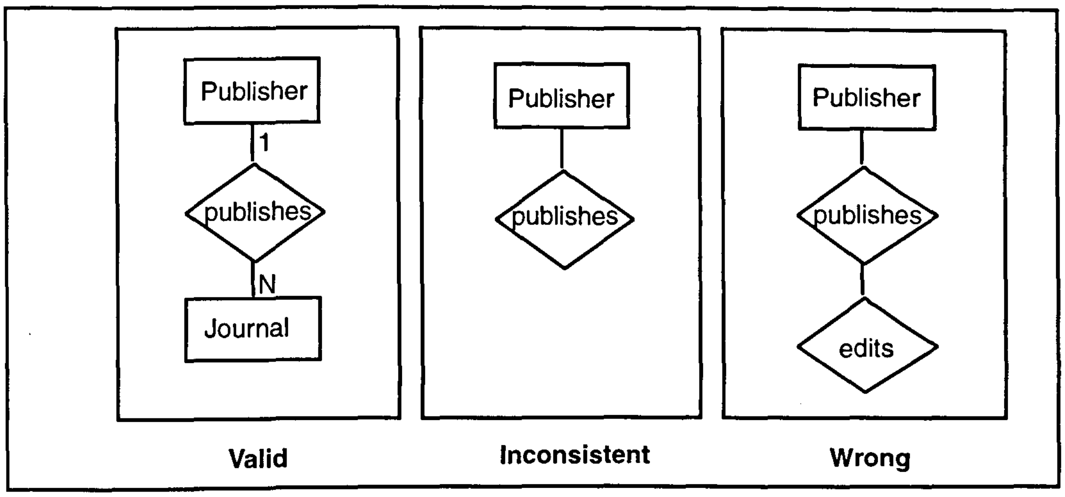


FIGURE 15. Diagram states.

6.3 Using the Approach to Produce an Entity-Relationship Diagram Editor

The editor to be produced supports the ER subset presented below. In the specification we will use natural language to express the constraints.

6.3.1 A Small Subset of the Entity-Relationship Technique

This section gives a short description of the Entity-Relationship (ER) method. Only a subset of the method is presented, concepts like multi-valued attributes, recursive relationships or weak entities are not addressed. A more detailed explanation of the method can be found in [Chen76] or [Sanders95].

The Entity-Relationship (ER) technique is a popular semantic data model that was first published by Peter Chen in 1976. ER is a high-level approach to database design used to understand and simplify data relationships in the real world.

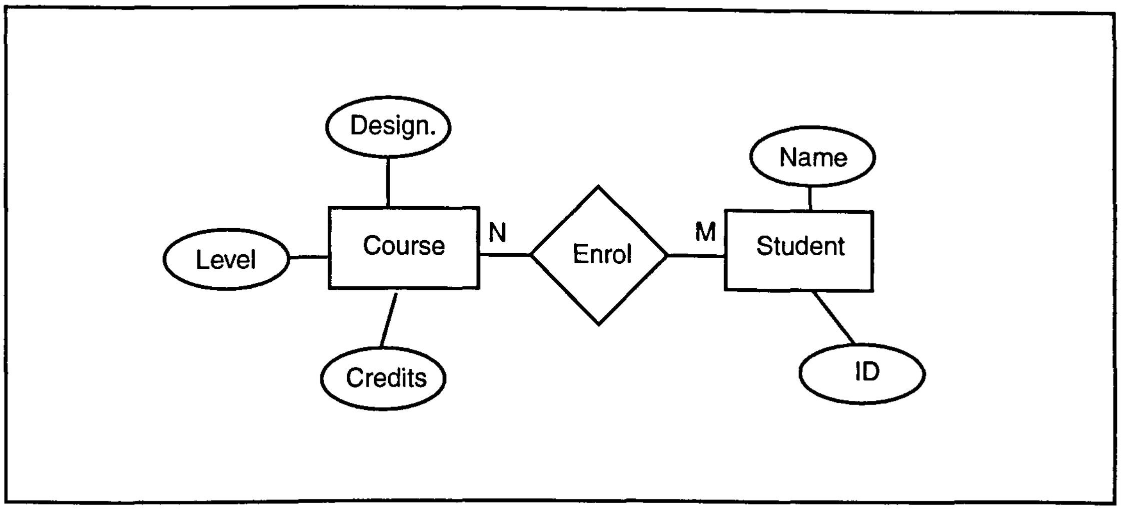


FIGURE 16. An ER diagram.

An ER diagram uses three basic graphic symbols (see Figure 16):

- **entities (depicted by rectangles)**, which are conceptual data units corresponding to objects in the real world;
- **attributes (depicted by ellipses)**, they describe the characteristics or properties of entities;
- **relationships (depicted by diamonds)**, which represent the structural association that exists between entities.

An **entity instance** results of assigning values to one or more attributes of an entity. It is a data object such as the student "Charles Brown".

The **cardinality** of a relationship is a numerical mapping between entity instances. It denotes the maximum number of instances of an entity that may be associated to a single instance of another entity. E.g. in Figure 16 students and courses are associated by a N:M relationship, meaning that each student may be enrolled in one or more courses and each course can have one or more students enrolled in it. A relationship may also have cardinality 1:1 or 1:N.

6.3.2 Designing the Editor

The design of the editor is carried out in four steps:

1 - Identify the VOs

There are three **icons**:

- entity;

- relationship;
- attribute;

and two **connections**:

- attribute-entity;
- relationship-entity.

Figure 17 shows the visual representation of the identified VOs.

2 - Express VOs semantics in terms of constraints

Entities

1. Entities must have a name
2. Entities can only be connected to attributes and relationships

Relationships

3. Relationships must have a name
4. Relationships can only be connected to entities
5. Relationships always connect two entities (as only binary relationships are permitted)
6. Relationships must connect different entities (as recursive relationships are not allowed in this simplified example)

Attributes

7. Attributes must have a name
8. Attributes can only be connected to entities
9. Attributes are always connected to one entity
10. Attributes can only be connected to an entity that is fully specified, i.e. to which a name has already been given .

Entity/Relationship Connections

11. Relationship-entity connections must have a label expressing the cardinality.
12. Cardinality labels on relationship-entity connections are chosen from the following set: {1, M, N}
13. If one of the relationship-entity connections is labelled with M, then the other one must be labelled with N (1:M and M:M cardinality labels are forbidden)
14. If one of the relationship-entity connections is labelled with N, then the other one must not be labelled with N

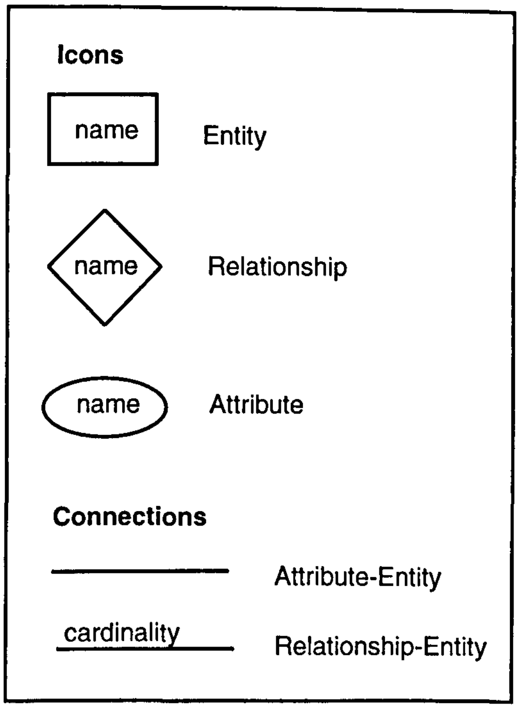


FIGURE 17. VOs for the ER method.

3 - Identify the constraints that when violated lead to wrong states

We are interested in isolating the constraints that prevent the editing task from reaching a point from which it makes no sense to proceed. Those situations must be avoided and therefore the constraints that prevent them are always enforced. The remaining constraints, when violated, only provoke temporary diagram inconsistency but, as explained before, inconsistent states are allowed in order to promote flexibility during the editing task. As we have shown, the violation of a constraint of this group affects VOs by incurring them to enter the *incomplete* or *disconnected* states.

The following constraints, when violated, cause the diagram to enter the *wrong state*:

2, 4, 6, 8, 10, 12, 13 and 14.

4 - Define the compound commands (if any)

A *compound command* is simply a sequence of atomic (pre-defined) commands.

It seems sensible to define a compound command for attribute creation in a way that attributes cannot exist in isolation, not associated with an entity. This command will override the attribute creation command provided by default.

The command 'Attribute Creation' is the sequence of the commands: atomic creation of the attribute; atomic creation of the connection attribute/entity.

We could define another compound command: the creation of a relationship. This compound command would enforce the indication of the connected entities by including the creation of both relationship-entity connections in the atomic command sequence. But we believe that not defining this command helps to promote editing flexibility.

While the definition of constraints is imposed by the underlying diagramming technique semantics, the definition of compound commands consists, essentially, in design options driven by usability concerns.

6.3.3 A note on constraints

Due to the simplicity of the example, the number of constraints is quite low. But for MTs leading to the definition of a larger number of VOs, the increase in the number of constraints is faster than linear. However, a combinatorial explosion does not normally occur. This affirmation is based on the observation that a large number of connectivity combinations between VOs are forbidden in typical MTs. This means that the connectivity matrix is normally very sparse which reduces the risk of combinatorial explosion.

6.3.4 Producing the diagram editor

Based on the specification above, it is possible to undertake the implementation phase of the editor. The natural language constraints specification must first be translated into a VC--t formal specification which will then be used as input to the compiler. Upon successful parsing of the specification, performed by the compiler's front-end, the executable code is generated.

6.4 Conclusions

We have presented a new approach to the design of diagram editors for MTs. A visual language, part of the editor, is obtained with basis on the semantic constraints identified for the MT. This approach solves the fundamental problems found in current diagram editors. The diagram being edited is *incrementally validated* according to the semantic constraints determined by the underlying MT. This means that a separate validation phase is no longer necessary. The main advantage of this approach is that it provides *non-obtrusive orientation* to the user. Allowing him/her to deliberately produce inconsistent diagram states promotes *flexibility* during the editing task.

The approach can be applied to VC-t specifications, it is however independent from the language used to produce the specifications. This means that this approach can be used with any specification language as long as it is based on semantic constraints. We have chosen to use natural language to express the constraints because it makes the example simpler to explain.

Note, however, that natural language is not normally used for software design and code generation due to its inherent ambiguity and complex grammar, characteristics that are not welcome when precise specifications and simple compilers are required. For this reason, natural language is normally only used to produce an initial specification which is then translated into a formal specification that will support the subsequent development cycle.

The visual language obtained with the approach defines solely the basic usage of a MT. This default visual language can be developed to produce a version that better implements the usage defined for the underlying MT and with improved usability. This advanced version can also be tailored to specific types of users. The theory behind that process is presented in Chapter 7 'Specifying Usage by Semantic Constraints'.

7. Specifying Usage by Semantic Constraints

In this chapter a new theory of semantic constraints to express the usage of a MT will be presented.

7.1 Some Notes on Terminology

Semantic constraints are defined both for a VC, which can be an Icon or a Connection, and for the whole diagram. Constraints are checked when they 'become in scope', that is, when a semantics check on the diagram is required for the editing task to proceed. There are four editing situations in which constraints become in scope:

- diagram creation - only for global diagram constraints;
- VC creation;
- VC deletion;
- label update (labels are defined both in icons and connections).

The expression 'constraint becoming in scope' will be used in the following. As in this approach constraints are always 'checked' when they 'become in scope', the two expressions can, and will, be used interchangeably.

Note that 'checking' has a different meaning from 'enforcing' a constraint. **Checking** a constraint determines whether the constraint is satisfied or violated. **Enforcing** a constraint guarantees its satisfaction. A constraint can be checked with, or without, being enforced.

If a check fails, meaning that the constraint was violated, and it is enforced then the user interaction is either coerced to the constraint satisfaction, or stopped and rolled back until the constraint is satisfied.

In the following, the given examples will be based on STD; refer to Appendix B for the complete VC specification of this MT.

7.2 Classification of Semantic Constraints

The semantic constraints included in a VC-t specification will now be labelled according to different classes. The visual language is extended with a new state - the 'unstable' state. It was not included in the default visual language (presented in Chapter 6) as it results from the new specification of usage.

In this approach, constraints are classified according to the following characteristics:

- **Enforcement**: a constraint may be enforced when checked; may never be enforced; enforced within a transaction; or only by a user request. Depending on the situation of its enforcement, a constraint violation leads the diagram to one of the following states: 'complete', 'inconsistent', 'wrong' or 'unstable'.
- **Scope**: it is a *local* constraint when it is defined for the VOs in the diagram, for example the following constraint specified for STD: 'A final state cannot have out-transitions or loop-transitions'; it is a *global diagram* constraint when defined for the diagram itself, for instance, the existence constraint 'A STD must have at least one initial state'.
- **Involvement**: which kind of diagram objects does it apply to (icons, connections, icon labels, connection labels). For instance, this is a *connection label* constraint specified for STD: 'Every transition must have a transition condition'. This characteristic will be further discussed in Section 8.4.

The scope and involvement of a constraint define its **domain**.

It is important to note that the instant in which the constraint is checked is not taken into account for the classification of constraints. Some authors, e.g. [Adreit91], classify constraints into immediate checking and deferred checking. In immediate checking constraints are checked when they become in scope; in deferred checking they are ignored when they become in scope and are only checked by an explicit user request. In our approach, all constraints are of the kind immediate checking. This feature is the vital point to allow for the exclusion of the validation phase in the design editors (detailed discussion in Chapter 6).

7.2.1 The Enforcement

According to their **enforcement**, constraints are classified into:

- **Hard** constraints. They are enforced by the time they are checked. When a violation occurs the diagram would go into the wrong state. In this situation the editing task cannot proceed; an error message stating the violated constraint is given to the user and the user action is rolled-back. The editing task is resumed at the situation immediately before the constraint violation happened.

- **Soft** constraints. As with all other constraints, they are checked as soon as they become in scope but are never enforced. This means that the system gives visual feedback to the user about the constraint violation but no action is required from the user to satisfy the constraint.
- **Hardened** constraints. Their enforcement is carried out within a constraint transaction. As soon as the constraints become in scope a constraint transaction is initiated. The diagram enters the unstable state and remains in it for the duration of the transaction. Any user action not leading to the completion of the transaction is forbidden.
- **Deferred** constraints. The only difference between these and hardened constraints is that their enforcement (and therefore the beginning of the transaction) only happens by explicit user request.

7.2.2 The Scope

In some situations it is not possible to associate a constraint to the VOs. An example is an existence constraint. For instance, the constraint given above 'an STD must have at least one initial state' can not be defined on the 'initial state' itself, because the constraint must be checked even before an initial state is created. Constraints defined for the whole diagram are called **global diagram constraints**. Conversely, constraints associated with the VOs are called **local constraints'**.

Note that only the enforcement classification is used explicitly by the user in the VC specification. The classification according to scope and involvement is automatically done by the system.

7.3 Hardening Constraints

In some situations, it might be interesting to enforce soft constraints. This feature has advantages when producing design tools for novice users. Not allowing the violation of selected soft constraints reduces the *level of inconsistency (LoI)* of the diagram in particular configurations. As an example, during the editing of a STD one might want to ensure that every intermediate state has, at least, one input transition, but output transitions may be left unspecified. This means that the constraint that guarantees an input transition for an intermediate state must never be violated. A possible advantage is the reduction of the number of unconnected states - all intermediate states would be connected to another state. The way to do this and its implications are described below.

The **level of inconsistency** of a diagram configuration is defined as the ratio between the number of different constraints which are violated and the total number of constraints in the specification, i.e.:

level of inconsistency (LoI) = #violated_constraints / #all_constraints * 100%

The **maximum level of inconsistency** is defined as the ratio between the maximum number of different constraints which may be violated in a particular instant or diagram configuration (Max(#violated_constraints)) and the total number of constraints in the specification, i.e.:

LoI_M = Max(#violated_constraints) / #all_constraints * 100%

Hardened constraints are soft constraints that are enforced at the time they are checked.

As an example, the constraint 'Every state must have a name' could be hardened. As a consequence the user would have to give a name for each state at the time of its creation.

The difference between a 'hard constraint' and a 'hardened constraint', is that the violation of the first one always leads the diagram to a wrong state whereas the violation of latter leads it to an inconsistent state (never to a wrong one).

Inconsistent states are necessary as transitions between consistent ones. In order to allow for the occurrence of those states and also to enforce the hardened constraints it becomes necessary to include the satisfaction of these constraints in a transaction. This way the hardened constraints are enforced - and must be satisfied, or otherwise the transaction will not commit. The inconsistent states are allowed within the transaction.

A **constraint transaction** is the mechanism of satisfying all hardened constraints defined for a given VC or for the diagram.

A constraint transaction occurs when the hardened constraints defined for a VC, or for the diagram, are enforced. See Figure 18 for an extract of a natural language specification expressing the semantics of the 'state' icon. In Figure 19 an example of two constraint transactions involving constraints defined for the 'state' icon in the STD specification is given.

Figure 19 shows the violated constraints: C2 and C4 are hardened constraints (represented in underlined style); C3 is a soft constraint. In this example, a constraint transaction corresponds to the satisfaction of C2 and C4. The two possible constraint transactions are shown. The bounding-box with double border enclosing the circle representing a state, indicates that a constraint transaction is taking place

7.3.1 Nested Constraint Transactions

To satisfy the hardened constraints defined in a VC it is sometimes necessary to satisfy constraints defined in other VCs. If those constraints are also hardened, then we have a

Natural Language Specification

Constraint 1:
Names must be unique amongst initialState, states and finalStates.
>Type: hard

Constraint 2:
Every state must have a name
>Type: hardened

Constraint 3:
A state must have at least one out-transition.
>Type: soft

Constraint 4:
A state must have at least one in-transition.
>Type: hardened

FIGURE 18. Extract of the semantics specification for the 'state' icon in STD.

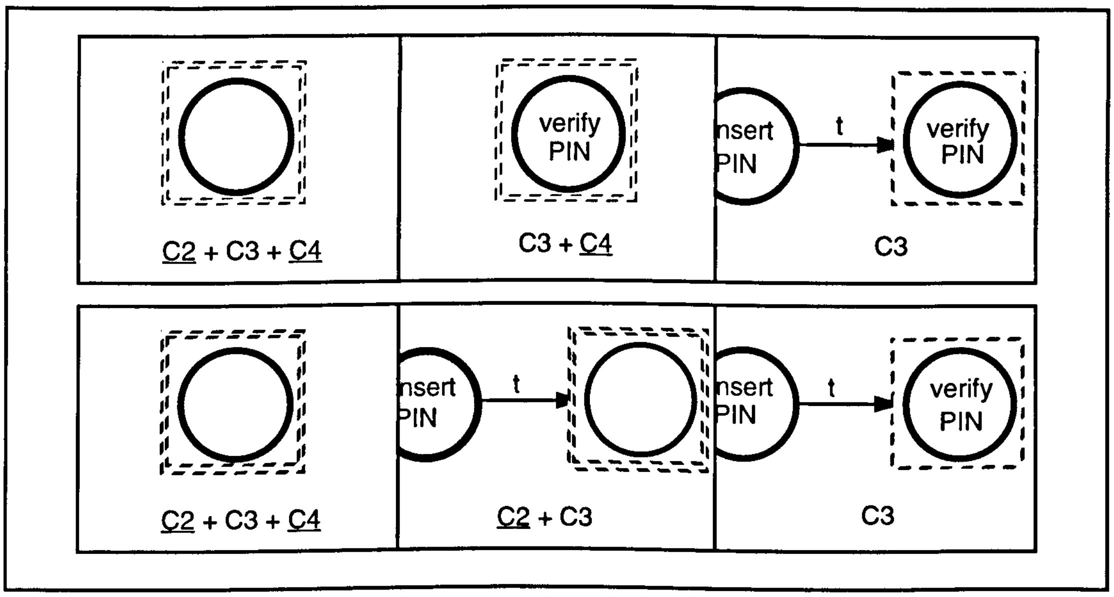


FIGURE 19. The two constraint transactions for the 'state' icon.

nested transaction, as the satisfaction of hardened constraints is always performed within a dedicated constraint transaction.

The example presented in Figure 19 can involve a nested transaction. In fact, to satisfy constraint 4, it is necessary to create a new 'transition'. This will provoke the constraints defined in 'transition' to be checked. If one or more hardened constraints were defined in 'transition', for instance the constraint 'the transitionCondition is mandatory' can be made hardened with possible advantages, then the enforcement of these constraints would start a new constraint transaction. As this new transaction occurs before the completion of the transaction initiated upon the creation of 'state', they become nested transactions.

7.3.2 Automatic Constraint Satisfaction

This design feature is aimed at facilitating the use of the editor in a hardened constraints validation scenario. There are hardened constraints for which it is important to provide an option in the editor to automatically create a diagram element (or editable construct) that satisfies the constraint. For example, if this option was activated for the global diagram constraint 'every STD must have an initial state', the editor would automatically create an unnamed initial state in each new diagram. Another example is to automatically display a name for each newly created state.

This is called **default_constraint_satisfaction**. When this option is activated for a hardened constraint, the editor automatically produces an editable construct that satisfies the constraint being enforced. Constraints with the default_constraint_satisfaction option are completely transparent to the MT user.

7.3.3 Deferring Hardened Constraints

Definition: in the scope of this approach, to **defer a constraint** is to delay its enforcement until a request for global (over the whole diagram) constraint enforcement is made. The request can be explicitly issued by the user or suggested by the system, for instance when the user closes the diagram.

It is important to notice that deferred constraints, as well as all the others, are always checked immediately after they become in scope. A **constraint becomes in scope** by:

- the creation of a new diagram (only for global diagram constraints);
- the creation of a VC for which it has been defined;
- the deletion of a VC for which it has been defined;
- updating a label of a VC for which it has been defined.

The MT designer may mark a constraint as deferred or may only define it as user-deferrable. If a constraint is user-deferrable, the MT user decides (at editor execution time) whether s/he wants it deferred or always enforced. This mechanism is provided to the MT designer to allow the specification of customisable editors for novice and expert users.

Only 'hardened constraints' can be deferred. 'Hard constraints' cannot be deferred because wrong states are never allowed in a diagram; and 'soft constraints' are never enforced, therefore deferring does not apply.

The difference between 'soft constraints' and 'deferred constraints' is then: the former are never enforced while the latter may be enforced by user request. The advantage of this is that the user may decide when to enforce the constraints which has the consequence of reducing the level of inconsistency (LoI) of the diagram.

Who decides which constraints are to be deferred?

Identifying the deferred constraints can be done by:

- The MT designer. Some constraints are marked as deferred at MT editor design time. In this case a number of deferred constraints are included in the editor as a feature. These constraints are called **designer-deferred** or simply **deferred.**
- The MT user. The option to defer some of the constraints included in the editor can be given to the user. These constraints are called **user-deferrable**. This allows the *editor customisation* in accordance with the user's level of expertise, i.e. a naive user might need all the hardened constraints in the default mode (immediate satisfaction) whereas a more experienced user might benefit from a less constrained environment; this increase of semantic freedom can be achieved by deferring a few of the allowed hardened constraints.

Why are hardened constraints deferred?

The process of hardening constraints although very useful reduces the semantic freedom. The advantage of deferring a hardened constraint is to recover the semantic freedom to the same value it was before the hardening process.

7.4 Semantic Freedom

Semantic Freedom is the percentage of unenforced semantics in a semantics specification.

Semantic Freedom (SF) = unenforced_semantics / all_semantics * 100%
or
SF = #soft_constraints / (#soft_constraints + #hard_constraints) * 100% ⇔
SF = #soft_constraints / #all_constraints * 100%

After the process of hardening and deferring constraints (detailed below), the formula becomes:
SF = (#soft_constraints + #deferred_constraints) / (#soft_constraints + #hard_constraints + #hardened_constraints + #deferred_constraints) * 100% ⇔
SF = (#soft_constraints + #deferred_constraints) / #all_constraints * 100%

The 'Semantic Freedom' (SF) concept only applies to editors for which the full semantics of the underlying MT has been expressed in terms of constraints according to the framework described in this report. A special case is a specification with no semantics (e.g. a specification for a drawing package such as MacDraw); as in this case there would be no constraints in the specification, SF would be undefined.

The value of SF is determined by the number of constraints of each enforcement class: soft, hard, hardened and deferred. However, the number of hard constraints is a constant for any given editor; as a result the value of SF cannot be 100% unless no hard constraints were defined. The maximum value of SF for a given editor will therefore be determined by the number of hard constraints (a constant value) and the number of soft constraints before the process of hardening and deferring (written $\#soft\_constraints_{initial}$):

$$SF_M = \#soft\_constraints_{initial} / (\#soft\_constraints_{initial} + \#hard\_constraints) * 100\%$$

This value is shown in Figure 20 left. The minimum value of SF is always zero and it is obtained by hardening all the soft constraints (see Figure 20 right).

The mechanism of hardening constraints gives a full range of possibilities, from a specification where no constraints have been hardened, i.e. the maximum allowed constraints are soft ($SF = SF_M$) to one in which all soft constraints have been hardened, i.e. the constraints are all hard or hardened ($SF = 0$). From the former specification a loose or freehand editor is produced, whereas a confining and restrictive one is obtained from the latter. In practice, the MT designer will locate the editor somewhere in the middle with, possibly, some flexibility to customise the editor for novice and expert users. Editor customisation will be explained in the following section.

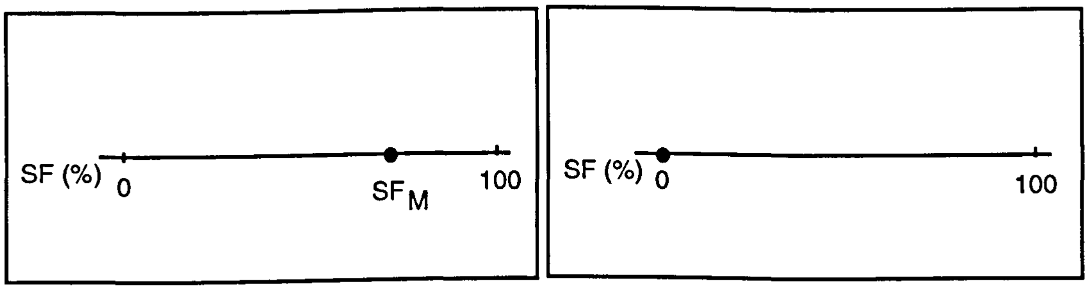


FIGURE 20. Extreme SF values. Left: SF maximum - no hardened constraints; Right: SF minimum (always zero) all soft constraints have been hardened.

7.5 Building a Semantic Specification

A specification of the semantics of a MT is built in five phases: define all constraints; identify the hard constraints; decide which constraints will be hardened; select from the hardened constraints the ones to be deferred; from these isolate the user-deferrable ones. The changes in the SF during these five phases are represented in Figure 21. The arrows show the variations on the value of SF from a phase to the next. The range of customisation is given by the dashed double arrow which will be explained below.

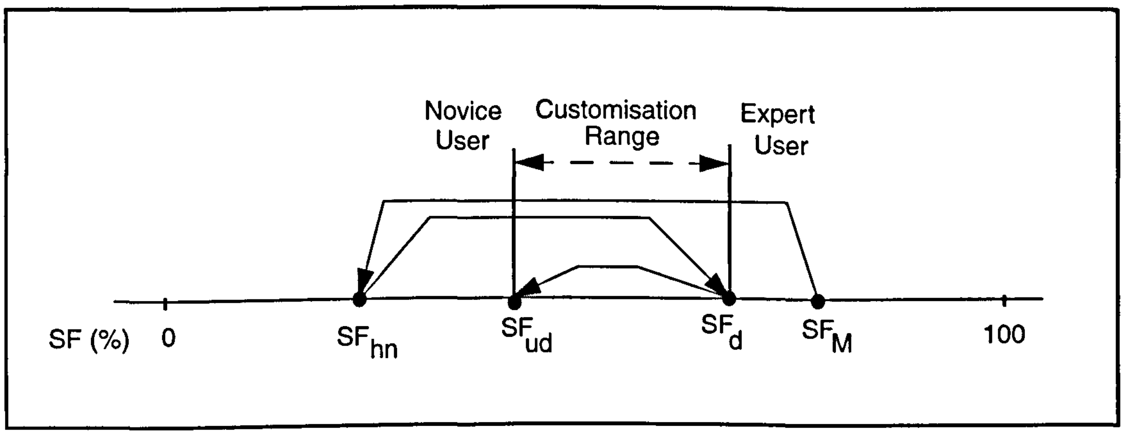


FIGURE 21. SF for different phases of the semantic specification.

First phase: define all constraints

From a description of the semantics of the MT in natural language all constraints are derived. SF cannot yet be determined.

Second phase: identify the hard constraints

After the identification of the hard constraints (the ones that when violated cause the diagram to enter the wrong state), a maximum value for the SF is obtained. The SF is given by:

SF_M = #soft_constr. / #all_constr. * 100%
= (#all_constr. - #hard_constr.) / #all_constr. * 100%

The hard constraints identified for a given MT are a characteristic of the semantics of that MT and therefore it is not a matter of designer's choice.

Third phase: decide the hardened constraints

In this phase the designer transforms some of the soft constraints into hardened constraints. This way the amount of constrained semantics increases and the SF is therefore reduced. This will provoke generating an editor that gives more guidance to the user. The SF is given by:

SF_{hn} = #soft_constr. / #all_constr. * 100%
= (#all_constr. - #hard_constr. - #hardened_constr.) / #all_constr. * 100%

Fourth phase: select constraints to be deferred

The designer introduces some deferred constraints to counterbalance the effects of hardening. This way, the SF is increased which allows the MT user to introduce more inconsistencies in the diagram being edited. The consequence in the value of SF is given by:

SF_d = (#soft_constr. + #deferred_constr.) / #all_constr. * 100%

Through the selection of hardened and deferred constraints, the designer can control the amount of guidance that the editor will provide to the user. In the two extreme situations we have:

- all soft constraints are made hardened, all the MT semantics will be constrained - SF becomes zero ($SF_{hn} = 0$) - and maximum guidance is given to the user.
- no hardened constraints are defined or have been deferred, the MT semantics is only constrained by the hard constraints - in this situation the value of SF is maximum ($SF = SF_M$) - and the maximum level of inconsistency is allowed.

Expert users will usually introduce inconsistencies in the diagram during an editing session. This allows for more flexibility in the editing process. Novice users, conversely, may appreciate being given more guidance by the editor. Hardening and deferring constraints allows the MT designer to specify an editor in accordance to the level of expertise of the future users. However, it might be useful to obtain an editor that allows for customisation, i.e. an editor in which some constraints could be hardened or deferred at execution time. This would allow users with different levels of expertise to use the same editor and also, as users become more acquainted with the editor, they could themselves defer some constraints. This is the purpose of the user-deferrable constraints (see Section 7.3.3). This process occurs in the fifth phase explained in what follows.

Fifth phase: select user-deferrable constraints from the deferred ones

User-deferrable constraints are hardened constraints that can be deferred by the user at editor execution time; i.e. they can interactively be hardened or deferred.

The designer selects a number of constraints, from the deferred ones, to become user-deferrable. The chosen constraints will be in the hardened state by default. The value of SF will be given by:

SF_{ud} = (#soft_constr. + #deferred_constr. - #user-deferrable_constr.) / #all_constr. * 100%

The user-deferrable constraints define a window or range of customisation over the SF values, as depicted in Figure 21, with the minimum in SF_{ud} (all user-deferrable constraints in the hardened state) and maximum in SF_d (all user-deferrable constraints in the deferred state).

The maximum allowed customisation range is obtained when the designer transforms all soft constraints into user-deferrable. This situation is depicted in Figure 22.

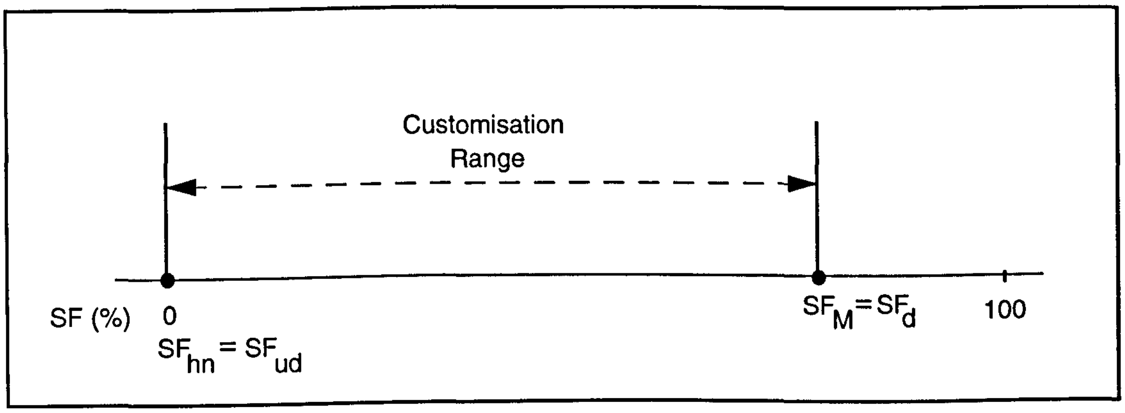


FIGURE 22. Maximum customisation range.

The user can now customise the editor from a situation where all constraints are hardened ($SF_{hn} = 0$) to another where all constraints are deferred ($SF_d = SF_M$) apart from the hard constraints. These two extreme situations are therefore accessible to the user at editor execution time.

In the example below SF values are calculated for all the phases of a semantics specification.

7.5.1 Obtaining SF values for a semantics specification

Phase 1

Total number of constraints: 32

Phase 2

Identified as hard: 6

$SF_M = (32 - 6) / 32 * 100 = 26/32 * 100 = 81.25\%$

Phase 3

Hardened: 11

$SF_{hn} = (32 - 6 - 11) / 32 * 100 = 15 / 32 * 100 = 46.875\%$

Phase 4

Deferred: 8

$SF_d = (15 + 8) / 32 * 100 = 23 / 32 * 100 = 71.875\%$

Phase 5

User-deferrable: 8

$SF_{ud} = (15 + 8 - 8) / 32 * 100 = 15 / 32 * 100 = 46.875\%$

Note that because all deferred constraints (from Phase 4) have been specified as user-deferrable, we have: $SF_{ud} = SF_{hn}$. These values are plotted in Figure 23.

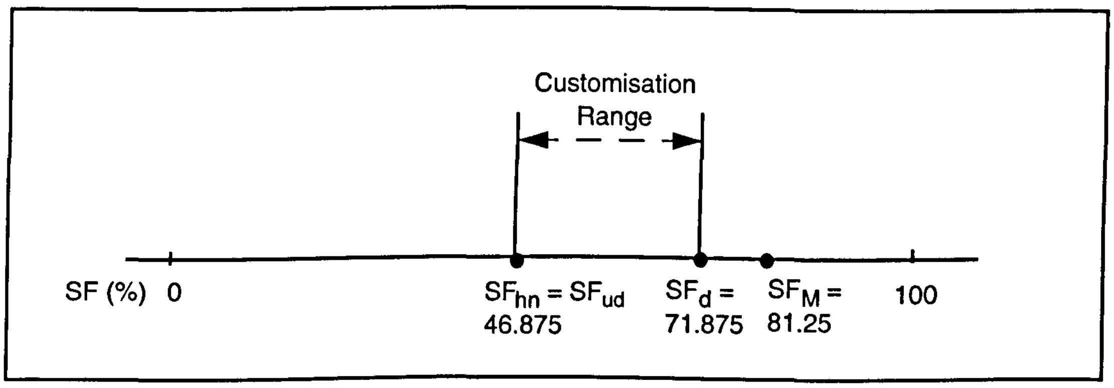


FIGURE 23. SF values for a particular semantics specification.

7.5.2 Example: creation of an initial state in a STD

The **problem**: one of the global diagram constraints for STD is: 'every STD must have an initial state'. This is a not a hard constraint because even if it is violated the diagram does not enter a wrong state, i.e. the user can proceed with the editing process even if an initial state has not been created. However, only if the initial state is present can a valid diagram be obtained.

The solution is to make that global diagram constraint **hardened**. The consequence of this is: when the diagram is created the initial state is compulsorily required. A way to improve the usability in these situations is to provide a **default_constraint_satisfaction** option for the constraint. In this case the initial state (without the name) would be automatically created by the editor.

It must be noted that depending on the specification, the user may be forced to create the initial state during the editing process. For instance, if the following constraint defined for an intermediate state - 'a state must have at least one in-transition' - is specified as hardened, the user will be forced to create the initial state before any other state. Confronted with this situation, the user may not associate the violation of the above constraint, which is defined for 'state' with the necessity of having the initial state in the diagram. Again, hardening the global diagram constraint will solve this problem.

7.6 Potential Problems

In the process of hardening constraints, the MT designer may cause the specification to become too restrictive. A design tool generated from that specification will forbid the user to produce some particular valid diagram configurations.

Because the VC-t specification language is based on a formal logic, it should be possible to develop an automatic theorem prover to detect these situations. However, this subject is outside the scope of our work.

In our approach, the detection of such situations is done by the MT designer during the test phase of the generated design tool. Testing is performed by using the tool in a design session that covers the complete semantics of the underlying MT.

The MT designer should be aware that these situations may happen when hardening constraints so that s/he may try to prevent them.

7.7 Constraint Enforcement Sequences

Depending on the hardening and deferring of constraints, there are some constraint enforcement sequences that are allowed whereas other are forbidden. However it is not up to the MT designer to decide which are which. The determination of allowed and forbidden sequences is a result of the validation process at editing time and cannot be done explicitly. The MT designer can specify that for a given VC the constraint Cx is hardened while constraint Cy is deferred. In this situation it will be impossible to have Cy enforced before Cx. A concrete example is given by Figure 19. Because C3 is a soft constraint - 'a state must have at least one out-transition' - it is the last to be checked in the two possible constraint enforcement sequences.

It is important to note that the specification of constraint enforcement sequences is not done explicitly, i.e. the designer doesn't specify the sequences of constraint enforcement, only classifies the constraints in the VC specification. This approach implicitly defines the allowed sequences.

The explicit specification of constraint enforcement sequences makes it more complex to write a specification. The allowed sequences of constraint enforcement must be identified. Also, this distracts the designer from what should be her/his main concern: expressing the MT semantics.

7.8 Status

The theory presented in this chapter has not been included in the prototype because we had to restrict its implementation to what is fundamental to support the main claim of this dissertation, i.e. that it is possible to generate design tools from high-level semantics specifications of MTs. However, the theoretical foundations have been laid and the concepts introduced, explained and exemplified. An implementation will hopefully be obtained in the near future.

8. Compilation of Specifications and Automatic Code Generation

8.1 Introduction

A VC-t compiler was built with the aid of the software construction tools Lex and YACC [Aho86; Schreiner85]. The implementation ('host') language used by these tools is 'C' [Kernighan88]. The languages used in the compiler are, therefore: 'VC-t' - the source language, 'C' - the implementation language and 'Napier88' - the target language (see Figure 24).

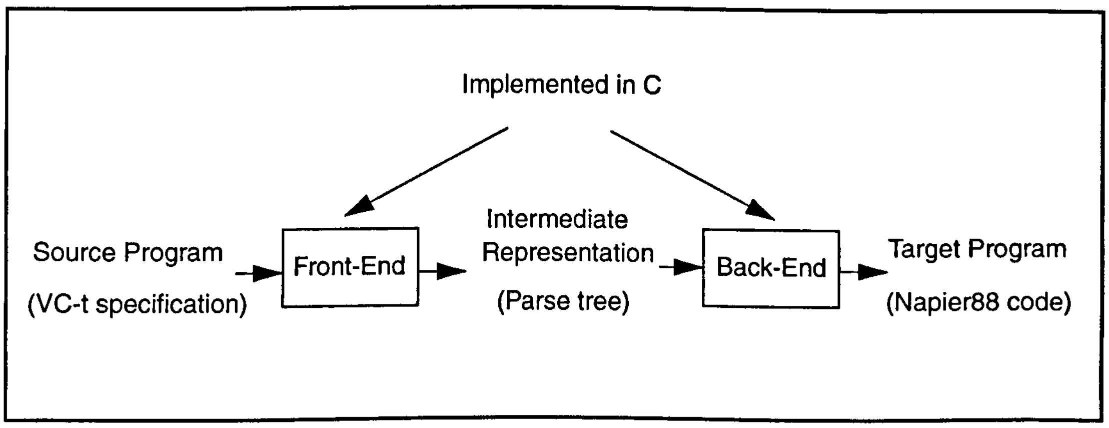


FIGURE 24. The architecture of the VC-t compiler.

The VC-t compiler's model is constituted by a front-end and a back-end. The **front-end** is composed of the lexical analyser and the parser; it performs the analysis phase of the compilation process, i.e. it translates the source program (VC-t specification) into an intermediate representation. The parser also performs semantics analysis consisting in type checking and enforcing scope rules.

The **back-end** is the code generator; it performs the synthesis phase of the compilation, i.e. the target code is generated from the intermediate representation. In this model any component of the compiler related to the target language, which in the implemented prototype is Napier88, is restricted to the back-end.

The main advantage of this model, apart from facilitating testing and maintenance of the compiler, is to produce a *retargetable* compiler. In fact, to transform the compiler to generate a different target language, e.g. Java or C++, it is only required to change the back-end.

8.2 The Compiler's Front-End

The VC-t compiler's construction was performed with well accepted technology; the Lex and YACC software tools were used. For this reason we will focus on the aspects that are particular to this compiler.

The front-end includes a number of data structures which are instantiated by the parser to produce an *intermediate representation* of the VC specification. This intermediate representation (or parse tree) includes references to the entries on the symbol tables which are discussed below.

8.2.1 Symbol Tables

A compiler uses symbol tables to keep track of binding and scope information about names in the source program. A symbol table can be implemented with a number of data structures including linear lists, trees and hash tables. For their efficiency, the hash tables were used in the VC-t compiler.

The lexical analyser creates a new symbol table entry each time it sees a new name in the input. For each name stored in a symbol table a number of attributes must be given values, for instance its type. Additional information discovered about a name, is stored as attribute values in the entry corresponding to that name. Some attributes can be given values by the lexical analyser when the entry is created, the remainder are filled by the syntactical analyser (called parser from now on) when the role played by the name is discovered.

The VC-t compiler uses two symbol tables, one for the 'Preamble' section and another for the 'Semantic Constraints'. The 'Preamble' symbol table is filled during the parsing of that section; the information collected is then used when the 'Semantic Constraints' section is parsed. The 'Semantic Constraints' symbol table is used to keep track of variables used within each constraint. The *symbol table interface* is composed by the following operations:

- 'make_hashtable' creates a new hash table;
- 'lookup' given a name and a hash table, returns the entry containing that name;
- 'insert' given a name and a hash table, if name does not exist in the hash table creates an entry and returns a pointer to it, otherwise returns a pointer to the existent entry;
- 'empty' removes the contents of the hash table.

8.2.2 The Value Stack

YACC maintains a value stack to store both the terminal symbols (called tokens from now on) and non-terminal symbols detected during parsing of the input specification. By default, the value stack elements are integers but in the VC-t compiler an alternative type has been provided. The type of the value stack elements (named YYSTYPE) has been redefined as a union, the structure of which is described below.

The value stack structure is a union with three members:

- *'symbol'* corresponds to tokens such as 'equal', 'greater', 'and'. Its value is the integer identifying the token.
- 'lexitem' corresponds to those tokens that are names in the specification, for instance 'string' or 'natural'.
- 'bucket' which constitute the symbol table entries.

'Symbol' and 'lexitem' instances are created by the lexical analyser and passed on to the parser through the 'yylval' variable defined by YACC.

'Bucket' instances are created by the lexical analyser for each new token seen in the input and inserted in a symbol table. 'Buckets' are then used by the parser which adds extra information, e.g. the type of a variable, and creates references to them during the construction of the parse tree. The parser also passes these references in the productions as the result of reduce operations after acceptance of a non-terminal symbol. For this purpose the variable 'yyval' defined by YACC is used.

8.2.3 Error Handling and Diagnostics

Two types of errors are detected by the compiler: syntax errors and semantic errors.

Syntax errors are the ones detected by the parser built by YACC as a result of grammar violations. A message indicating the nature of the error and its location, token and line number, in the source file is issued to the user. Syntax errors in the constraints section are recovered by the compiler: parsing of the current constraint is interrupted; it is resumed in the beginning of the next constraint.

Semantic errors result of incorrect typing or of violation of scope rules. They are detected within the actions supplied with the productions. As YACC does not give any support to the semantic analysis, this component had to be completely hand built. Functions implementing the actions for the productions detect the semantic errors. As with syntax errors a message is issued to the user with the token and the line number, but stating that it is a semantic error and including a short explanation of why it is an error.

Every error provokes the code generator to stop. This is to prevent erroneous code to be generated, as one of the requirements of the compiler is to always generate correct target code.

8.3 Code Generation - the Compiler's Back-End

The back-end generates the target code (Napier88 in our prototype) from the intermediate representation produced by the front-end. This division of the compiler in two parts makes it easier to port it to a new target system. All the aspects of the compiler that are dependent of the target system are only in the back-end. If the target system changes the front-end can be preserved and only the back-end is modified.

Using traditional technology, the following code would be generated: a constraint base for the semantics, a data dictionary and the necessary code to configure the drawing tool and its user interface. As we are using persistent technology the data dictionary is replaced by persistent data structures from which the data is instantiated at editor execution time. This data will also be persistent.

8.3.1 Template Technology

A **template** is a framework describing the structure of a generated program in the target language. It is implemented as a sequence of components which can be static or dynamic. The **static components** correspond to those constant parts of the generated program that are always present independently of the MT being modelled; they are implemented as arrays of strings. The **dynamic components** are dependent on the particular VC specification; because they do not have a fixed size they are implemented as lists of strings.

Figure 25 shows an extract of a static template component used in the generation of one of the Napier88 output programs. Note that the code in the template looks exactly as it will appear in the generated program; this adds legibility to the template, making this technology very easy to be used.

Dynamic components correspond to blank slots that must be filled to obtain a compilable Napier88 program. The filling of the slots is performed by the code generator. The dynamic components are generated at VC-t compilation time by special purpose functions included in the code generator.

```
'static stc PREAMBLE_S = {
 "!-- Preamble\n",
 "\n",
 "use theStore() with GlasgowLibraries, DesignTools: env in\n",
 "\n",
 "use GlasgowLibraries with BulkTypes: env in\n",
 "  use BulkTypes with Maps: env in\n",
 "      use Maps with\n",
 "            m_find: proc[X, Y]( Map[X, Y], X -> Y ) in\n",
 "\n",
 "use DesignTools with Useful, STDapplication: env in\n',
 "use Useful with\n",
 "  message: proc( string ) in\n",
 "\n",
 "use STDapplication with\n",
 "  STDiconProperties, STDconnectionProperties,\n",
 "  IconSemanticConstraints, ConnectionSemanticConstraints: env in\n",
 [...]
 }'

    where 'stc' stands for static template component.
```

FIGURE 25. Extract of a static template component.

8.3.2 Generic Principles and Problems in Code Generation

Generating executable constraints in Napier88 is the main problem in the code generation process. Each semantic constraint in the VC-t specification must be translated into executable Napier88 code. The output of the translation process must consist of a set of Napier88 constraints (executable code), each one corresponding to a VC-t specification constraint. This translation process is based on a number of rules which cover every aspect of a VC-t specification and produce logically equivalent code.

All the translation rules have been tested by hand, i.e. by applying them to a specification and writing the corresponding Napier88 code. A collection of rules have also been incorporated in the code generator of the prototype (see Section 10.2 on page 140).

Our main goal is to conceive correct and unambiguous translation rules that produce readable, simple and efficient Napier88 code. In a first stage we have decided to favour the former two features of the generated code by sacrificing the latter. By not being too concerned about the efficiency of the generated code we also profit by obtaining simpler translation rules, making it easier to reason about them. Simpler rules implies easier reasoning and, ultimately, a better understanding of the code generation process.

Below we introduce some of the generic principles in the code generation process. We give forward references to the translation rules (which will be presented in the following section) to provide the reader with a global view of the set of rules.

All the specifications of semantic constraints generate code that traverses the diagram (as stated by Rule 3); this is done by generating calls to the special purpose procedure 'traverse' (or 'traverseC' if connections must also be traversed). 'traverse' calls a 'process' procedure for each voIcon in the diagram (or for each voConnection). 'traverse' will return either 'OK' or 'FAIL' depending on the value of the boolean variables 'success' and 'violated' which are set by 'process'. The code for the 'process' procedure is automatically generated (this will be explained in detail in the discussion on quantified predicate logic statements presented below). If 'traverse' returns 'FAIL', this means that the constraint was violated and an error message is issued to the user. The message can be generated because its contents will be given by the semantic constraint description expressed in the specification (Rule 1).

VC semantic constraints are the basic specification constructs that must be dealt with by the translation process. They are classified into instantiated predicate logic statements and quantified predicate logic statements; the latter being either existentially quantified or universally quantified predicate logic statements. We have tried to deal with the three constructs in a consistent way. This is reflected in the similarity of the resulting rules. In what follows we abstract from the individual rules aiming to give an overview of their particularities and of the characteristics of the code. As mentioned above, the code generated for all the constructs will include diagram traversal (Rule 3). This is done differently for each of the constructs:

- in the code generated for **instantiated predicate logic statements**, the 'process' procedure is executed for all the matching icons or connections in the diagram. See Rule 3 and Rule 4.

- in the code generated for **existentially quantified predicate logic statements**, the 'process' procedure sets the boolean variable 'success'. When 'success' becomes true, the diagram traversal stops and the 'traverse' procedure returns 'OK'. This means that diagram traversal is performed until one matching VO (icon or connection) satisfies the predicate. The predicate must be true for at least one matching VO; if this is not the case then 'traverse' returns 'FAIL'. Apropos are Rule 3, Rule 5, Rule 7, Rule 9 and Rule 11.

- in the code for **universally quantified predicate logic statements**, the 'process' procedure sets the boolean variable 'violated'. When 'violated' becomes true, the diagram traversal stops and 'traverse' returns 'FAIL'. This means that diagram traversal is performed until one matching VO (icon or connection) violates the predicate. The predicate must be true for all matching VO's; in this case 'traverse' returns 'OK'. Code generation for universal quantification is given by Rule 3, Rule 6, Rule 8 and Rule 10.

An interesting problem consists in determining if a VC semantic constraint is local to an icon or connection, or on the contrary, concerns the whole diagram. The latter kind generates a global constraint in Napier88. This happens in the cases covered by Rule 4 and Rule 11. It is still unclear whether these rules cover all possible situations.

A special treatment must be given to cardinality constraints (expressed by Rule 12). In fact, an addition to the diagram traversal algorithm explained above must be done. The predicate is no longer evaluated each time the 'process' procedure is called, but instead it is evaluated incrementally. This means that its evaluation is partially performed for each new voIcon (or voConnection) visited. The final evaluation of the predicate is done after completion of diagram traversal inside a 'post-condition' procedure. 'traverse' and 'traverseC' execute a 'post-condition' (a boolean procedure) after completion of diagram traversal. 'traverse' returns either 'OK' or 'FAIL', depending on the result of the evaluation of 'post-condition'. By default 'post-condition' tests 'success' and 'violated' which are boolean variables set by 'process' (this is the situation described above); the 'post-condition' returns 'TRUE' if 'success' is 'TRUE' or 'violated' is 'FALSE', and it returns 'FALSE' otherwise.

8.3.3 Translation Rules for Code Generation

Below is presented the set of rules used to translate VC-t specifications into executable Napier88 code. Short examples are also included.

These rules are the ones we consider more interesting for the code generation process. We did not include rules at such a small level of granularity that become almost straight forward mappings from VC-t constructs to Napier88 constructs.

As we said before, a major goal is to generate code that is readable; but there is a compromise involved in achieving this goal: nicer looking code implies more complex translation rules. In some situations we have sacrificed slightly the readability for rule simplicity. Procedure and variable names could be less complex; however, they are easily generated and because there are no repeated names, we simplify the treatment of scope. Another advantage of unique names regards the automatic code inspection, and versioning. This might be useful in future work on automatic change management.

For all the rules which refer to VO's we assumed that the variable in the specification is an icon. The rules can be applied in the same way to specification with connection variables. There are only two differences in the code generated for connection variables: the diagram traverse procedure 'traverse' becomes 'traverseC', which is a traverse procedure that also visits all connections; and the code fragment 'voIcon(viIcon)(iName)' becomes 'voConnection(vcConnection)(cName)'. This can be understood by looking at the GraphTool datatypes presented in Appendix C.

Rule 1:
Applies to string token included in production rule for non-terminal *constraintDesc*.

The string token in the production rule *constraintnum string*, defined for the non-terminal *constraintDesc*, generates an error message (which will be issued as a result of a constraint violation). This rule is valid for all constraints.

For example the string in:

```
'C3: "Each datastore has a unique name"'
```

generates the procedure call:

```
'constraintFeedback( "Each datastore has a unique name" )
```

Rule 2:
Applies to sets in non-terminals *quantifiedPLstatement* and *instantiatedPLstatement.*

Each set in the specification generates a name (Napier88 string).

Example:

'd: Process'
    where 'Process' is an icon set in the specification

generates:

```
'voIcon( viIcon )( iName ) = "Process"'
```

Rule 3:
Applies to non-terminals *quantifiedPLstatement* and *instantiatedPLstatement.*

Non-terminals *quantifiedPLstatement* and *instantiatedPLstatement* generate code that includes a call to a diagram traversal procedure ('traverse' or 'traverseC').

Note 3.1[1]: diagram traversal is performed by executing the 'traverse' procedure (or 'traverseC' if the non-terminal refers to connections), which itself calls a 'process' procedure for each voIcon in the diagram (and also for each voConnection in the case of 'traverseC').

Rule 4:
Applies to *instantiatedPLstatement* defined for the non-terminal *constraint.*

1. Notes numbering includes the number of the rule for which they are defined, e.g. in Rule 3 the note is numbered 3.1 and in Rule 7 notes are numbered 7.1 and 7.2.

An *instantiatedPLstatement* does not have bound variables, therefore it always generates a global constraint in Napier88.

To a generic piece of specification:

'PREDICATE pred(xs(mt))'

where 'PREDICATE' is a keyword indicating that what comes next is a predicate; 'pred' is the predicate name; and 'xs(mt)' is a set extractor for 'mt', where 'mt' is a modelling technique

will correspond the following pseudo-code:

```
'let processInstantiated
begin
    /* evaluate predicate*/
    if( l_pred( "xs" )) /* 'l_pred' is an implementation of 'pred' and must be pre-defined as a
                            library function */
    do success = TRUE
end

if( traverse( processInstantiated ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

Rule 5:
Applies to non-terminal *existentialQuantification* followed by *plStatement.*

To a generic piece of specification:

'EXISTS x:X • x BELONGING xs(mt) AND predicate(x)'

where X is a set and xs(mt) is a set extractor for 'mt', where 'mt' is a modelling technique.

will correspond the following pseudo-code:

```
'let processExist
begin
    if( voIcon( viIcon )( iName ) = "X" ∧ predicate( x ))
    do success = TRUE
end

if( traverse( processExist ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

Note 5.1: when 'success' becomes true, the diagram traversal stops and the 'traverse' procedure returns 'OK'. This means that the diagram traversal is performed until one matching VO (icon or connection) satisfies the predicate (compare Rule 6). The predicate must be true for at least one matching VO; if this is not the case then the 'traverse' procedure returns 'FAIL'.

Rule 6:
Applies to non-terminal *universalQuantification* followed by *plStatement.*

To a generic piece of specification:

'FORALL x:X • x BELONGING xs(mt) IMPLIES predicate(x)'

will correspond the following pseudo-code:

```
'let processUniversal
begin
    if( voIcon( viIcon )( iName ) = "X" )
    do if ( ~ predicate( x ))
        do violated = TRUE
end

if( traverse( processUniversal ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

Note 6.1: when 'violated' becomes true, the diagram traversal stops and 'traverse' returns 'FAIL'. This means that the diagram traversal is performed until one matching VO (icon or connection) violates the predicate (compare Rule 5). The predicate must be true for all matching VO's; in this case the 'traverse' procedure returns 'OK'.

Rule 7:
Generalization of Rule 5 for multiple bound variables.

For the generic piece of specification:

'EXISTS x1, x2, ... xN:X • x1, x2, ... xN BELONGING xs(mt) AND predicate(x)'

the following pseudo-code fragments will be generated:

Fragment 1:

```
'let processExistentialS1 /*S1 refers to the first variable in the sequence of bound variables*/
begin
    if( voIconS1( viIcon )( iName ) = "X" )
    do if( traverse( processExistentialS2 ) = OK )
        do success = TRUE
end

if( traverse( processExistentialS1 ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

Fragment 2:

```
'let processExistentialS2
```

```
begin
    if( voIconS2( viIcon )( iName ) = "X")
    do if( traverse( processExistentialS3 ) = OK )
        do sucess = TRUE
end'
```

[...]

Fragment N:

```
'let processExistentialSN
begin
    if(  voIconSN( viIcon )( iName ) = "X" ∧
        predicate(  voIconS1, voIconS2, ... voIconSN )) /*see Rule 5*/
    do success = TRUE
end'
```

Produce final code by merging the generated code fragments in a reverse order sequence (see Note 7.1).

Note 7.1: the sequence of the generated code fragments presented above will not compile. To obtain a larger single code fragment that compiles successfully, the individual code fragments must be reordered. To do this, they are stored in separated strings. The final code will result from the concatenation of the strings in the reverse order of generation.

Note 7.2: the stopping condition ('success = TRUE') is firstly triggered by the innermost traverse procedure; i.e. it stops as soon as the predicate is satisfied for the set of matching VO's. This triggers the stopping conditions of the outer traverse procedure until it is propagated to the outermost one.

Rule 8:
Generalization of Rule 6 for multiple bound variables.

For the generic piece of specification:

'FORALL x1, x2, ... xN:X • x1, x2, ... xN BELONGING xs(mt) IMPLIES predicate(x)'

the following pseudo-code fragments will be generated:

Fragment 1:

```
'let processUniversalS1
begin
    if( voIconS1( viIcon )( iName ) = "X" )
    do if( traverse( processUiversalS2 ) = FAIL )
        do violated = TRUE
end

if( traverse( processUniversalS1 ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

```
Fragment 2:
'let processUniversalS2
begin
    if( voIconS2( viIcon )( iName ) = "X" )
    do if( traverse( processUniversalS3 ) = FAIL )
        do violated = TRUE
end'

[...]

Fragment N:
'let processUniversalSN
begin
    if( voIconSN( viIcon )( iName ) = "X" )
    do if( ~predicate( voIconS1, voIconS2, ... voIconSN )) /*see Rule 6*/
        do violated = TRUE
end'
```

Produce final code by merging the generated code fragments in a reverse order sequence (see Note 7.1).

Note 8.1: the stopping condition ('violated = TRUE') is firstly triggered by the innermost traverse procedure; i.e. it stops as soon as a violation occurs. This triggers the stopping conditions of the outer traverse procedure until it is propagated to the outermost one, which provokes the execution of 'constraintFeedback'.

Rule 9:
This rule and the next one apply to non-terminal *quantificationList.* They are, respectively, generalizations of Rule 5 and Rule 6 for nested quantifications.

Note 9.1: in this situation there are three possible locations for the quantification: it can be the outermost one; the innermost; or in between those two. Moreover, the quantification can be either universal or existential. Therefore, we have a total of eight possible combinations. Instead of writing eight different rules we opted for a simpler solution where only two rules are presented: the first rule considers the following combination of quantifications, existential (outermost), universal (in between), existential (innermost); the second rule expresses the inverse combination, universal (outermost), existential (in between), universal (innermost). This way, the two given rules document all locations for both existential and universal quantifications. This simplification is possible because the correlation between the generated code for each one of the quantifications in the three different locations is very weak and easily deduced for the missing combinations.

For the generic piece of specification:

```
'EXISTS x:X • x BELONGING xs( mt ) AND
    FORALL y:Y • y BELONGING ys( mt ) IMPLIES
```

```
        ...
            EXISTS w:W • w BELONGING ws( mt ) AND predicate( x, y, ... w )'
```

the following pseudo-code fragments will be generated:

Fragment 1 - outermost quantification level:

```
'let processExistentialL1
begin
    if( voIconL1( viIcon )( iName ) = "X"  /*see Rule 5*/
        ∧ traverse( processUniversalL2 ) = OK )
    do success = TRUE
end

if( traverse( processExistentialL1 ) = FAIL )
do constraintFeedback ...  /*apply Rule 1*/'
```

Fragment 2 - middle quantification level:

```
'let processUniversalL2
begin
    if( voIconL2( viIcon )( iName ) = "Y" )  /*see Rule 6*/
        do if( ~traverse( processWhateverL3 ) = OK ) /*processWhateverL3 can be either
                                                       processExistentialL3 or processUniversalL3*/
            do violated = TRUE
end'
```

[...]

Fragment N - innermost quantification level:

```
'let processExistentialLN
begin
    if( voIconLN( viIcon )( iName ) = "W"  /*see Rule 5*/
        ∧ predicate( voIconL1, voIconL2, ... voIconLN ))
    do success = TRUE
end'
```

Produce final code by merging the generated code fragments in a reverse order sequence (see Note 7.1).

Rule 10:
See introduction of Rule 9.

For the generic piece of specification:

```
'FORALL x:X • x BELONGING xs( mt ) IMPLIES
    EXISTS y:Y • y BELONGING ys( mt ) AND
        ...
            FORALL w:W • w BELONGING ws( mt ) IMPLIES predicate( x, y, ... w )'
```

the following pseudo-code fragments will be generated:

Fragment 1 - outermost quantification level:

```
'let processUniversalL1
begin
    if( voIconL1( viIcon )( iName ) = "X" )  /*see Rule 6*/
        do if( ~traverse( processExistentialL2 ) = OK )
            do violated = TRUE
end'



if( traverse( processUniversalL1 ) = FAIL )
do constraintFeedback ...  /*apply Rule 1*/'
```

Fragment 2 - middle quantification level:

```
'let processExistentialL2
begin
    if( voIconL2( viIcon )( iName ) = "Y"   /*see Rule 5*/
        ∧ traverse( processWhateverL3 ) = OK ) /*processWhateverL3 can be either
                                                processExistentialL3 or processUniversalL3*/
    do success = TRUE
end'
```

[...]

Fragment N - innermost quantification level:

```
'let processUniversalLN
begin
    if( voIconLN( viIcon )( iName ) = "W" ) /*see Rule 6*/
        do if( ~predicate(  voIconL1, voIconL2, ... voIconLN ))
            do violated = TRUE
end'
```

Produce final code by merging the generated code fragments in a reverse order sequence (see Note 7.1).

See examples at the end of this section.

Rule 11:

Applies to the production rule *existentialQuantification* defined for the non-terminal *quantificationList.*

A semantic constraint beginning with an *existentialQuantification* always generates a global constraint in Napier88.

Note 11.1: this rule does not affect the way the generated code looks. Its purpose is solely to mark the generated Napier88 constraint as a ‘global’ one. Global constraints must be identified and differentiated from local constraints so that constraint management may be optimized.

Note 11.2: an intuitive way of understanding this rule is realizing that a semantic constraint starting with an existential quantification is always referring to the diagram: "there exists in the diagram...". For example, the semantic constraint "There must be at least one external entity (in the diagram) which provides input to the system" is specified by:

```
EXISTS e : ExternalEntity • e BELONGING Externals(dfd) AND
    EXISTS f : Dataflow • f BELONGING Dataflows(dfd) AND
        ( source(f) = e )
```

Rule 12:
Applies to the production rule *naturalExpression naturalComparison naturalExpression* defined for the non-terminal *simpleBooleanExpression.*

Evaluate left *naturalExpression* and right *naturalExpression* inside the 'process' procedure; test *naturalComparison* (Napier88 provides all comparisons defined for VC-t specifications) after execution of the 'traverse' procedure.

In the example below, the left expression is given by a *CARDINALITY set* production rule (see Rule 13) and the right expression is a Natural number.

To a generic piece of specification:

'CARDINALITY xs(mt) ATLEAST n'
    where 'n' is a Natural number.

will correspond the following pseudo-code fragments:

```
'let post-condition /*this procedure is called inside traverse (Rule 4)*/
begin
    if( xs ⩾ n ) /*perform final evaluation of predicate logic statement*/
    do return TRUE
end'

'let processCardinality
begin
    if( voIcon( viIcon )( iName ) = "X" )
    do xs = xs + 1 /*xs has been previously declared as stated in Rule 13*/
end

if( traverse( processCardinality ) = FAIL )
do constraintFeedback ... /*apply Rule 1*/'
```

Note 12.1: 'post-condition' is called inside 'traverse' after completion of the diagram traversal; if it returns 'TRUE', 'traverse' returns 'OK', otherwise 'traverse' returns 'FAIL'.

Note 12.2: the code generator must find out that the range of the set extractor ‘xs’ is ‘P X’ and then generate the string “X” according to Rule 2.

Rule 13:
Applies to the *CARDINALITY set* production rule defined for the non-terminal *naturalExpression*

To a generic piece of specification:

‘CARDINALITY xs(mt)’

will correspond the following pseudo-code:

```
'int xs = 0'
```

8.3.4 Examples on Using the Translation Rules

In this section some concrete examples of code generated from pieces of VC-t specification obtained with the translation rules presented above, are given.The specifications referred to in this section are given in full in Appendix B.

Example 1: generating code for the constraint C5 defined in the VC-t specification of State Transition Diagrams.

```
'C5: "A final state cannot have out-transitions or loop-transitions"

FORALL f : FinalState • f BELONGING FinalStates(std) IMPLIES
    NOT EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( origin(t) = f )'
```

The Napier88 code to be generated is shown below. The translation rules used in the generation process are indicated as comments in the code.

```
'/* Fragment 1*/
/*First level - universal quantification*/ /* Rule 10 */

let processUniversalL1  /* Rule 6 */
begin
    if( voIconL1( viIcon )( iName ) = "FinalState" )  /* Rule 2 */
        do if( ~traverse( processExistentialL2 ) = OK )  /* Rule 3 */
            do violated = TRUE
end

if( traverse( processUniversalL1 ) = FAIL )  /* Rule 3 */
do constraintFeedback(
    "A final state cannot have out-transitions or loop-transitions" ) /* Rule 1*/'

'/* Fragment 2*/
/*Second level - existential quantification*/ /* Rule 9 */

let processExistentialL2 /* Rule 5 */
begin
    if( ~( voConnectionL2( vcConnection )( cName ) = "Transition" and /* Rule 2 */origin(
voConnectionL2 ) = voIconL1 ))
    do success = TRUE
end'
```

To obtain the final code, the fragments must be sequenced in reverse order (see note 7.1 on page 103).

Note: the negation inside the if statement (in 'processExistentialL2') results from having a negation in the existentialQuantification in the specification.

Example 2: generating code for the constraint C4 defined in the VC-t specification of Data Flow Diagrams.

```
'C4: "Each datastore must have at least one input dataflow and one output dataflow
     each of which must be connected to a process"

FORALL d : Datastore • d BELONGING Datastores(dfd) IMPLIES
    EXISTS f1, f2 : Dataflow • f1, f2 BELONGING Dataflows(dfd) AND
        EXISTS p1, p2 : Process • p1, p2 BELONGING Processes(dfd) AND
        (
            ( destination(f1) = s AND source(f1) = p1 ) AND
            ( source(f2) = s AND destination(f2) = p2 )
        )'
```

The Napier88 code to be generated is shown below. As for the previous example, the translation rules used in the generation process are indicated as comments in the code.

```
'/* Fragment 1*/
/*First level - universal quantification*/ /* Rule 10 */

let processUniversalL1 /* Rule 6 */
begin
    if( voIconL1( viIcon )( iName ) = "Datastore" ) /* Rule 2 */
        do if( ~traverseC( processExistentialL2S1 ) = OK ) /* Rule 3 */
            do violated = TRUE
end

if( traverse( processUniversalL1 ) = FAIL ) /* Rule 3 */

do constraintFeedback( "Each datastore must have at least one input dataflow and one output
dataflow each of which must be connected to a process" ) /* Rule 1*/'

'/* Fragment 2*/
```

```
/*Second level - existential quantification; first bound variable in the sequence*/ /*Rule 7 and
Rule 9 */

let processExistentialL2S1  /* Rule 5 */
begin
    if( voConnectionL2S1( vcConnection )( cName ) = "Dataflow" )  /* Rule 2 */
    do if( traverseC( processExistentialL2S2 ) = OK )  /* Rule 3 */
        do success = TRUE
end'

'/* Fragment 3*/
/*Second level; second bound variable in the sequence*/

let processExistentialL2S2  /* Rule 5 */
begin
    if( voConnectionL2S2( vcConnection )( cName ) = "Dataflow"  and  /* Rule 2 */
        traverse( processExistentialL3S1 ) = OK )  /* Rule 3 */
    do success = TRUE
end'

'/* Fragment 4*/
/*Third level - existential quantification; first bound variable in the sequence*/
let processExistentialL3S1
begin
    if( voIconL3S1( viIcon )( iName ) = "Process" )  /* Rule 2 */
    doif( traverseC( processExistentialL3S2 ) = OK )  /* Rule 3 */
        do success = TRUE
end'

'/* Fragment 5*/
/*Third level; second bound variable*/
let processExistentialL3S2
begin
    if(( voIconL3S2( viIcon )( iName ) = "Process" and  /* Rule 2 */  (
            ( destination( voConnectionL2S1 ) = voIconL1 and
            source( voConnectionL2S1 ) =voIconL3S1 ) and
            ( source( voConnectionL2S2 ) = voIconL1 and
            destination( voConnectionL2S2 ) = voIconL3S2 )))
    do success = TRUE
end'
```

To obtain the final code, the fragments must be sequenced in reverse order (see Note 7.1).

8.3.5 Visual Objects Generation

In a MT specification there may exist any number of VC types, which are grouped into icons or connections. When the Napier88 code is generated, the VC types belonging to the icons group and the connections group, will be implemented as the VOicon type and the VOconnection type, respectively (see Appendix C). There is therefore a correspondence of several VC types to just two Napier88 types. For example, in the STD specification there are three VC types belonging to the icons group: 'InitialState', 'State' and 'FinalState'.

These will all be VOicons in Napier88. There must then exist a mechanism to differentiate the various VOs (Napier88 types) according to the different VC types in the specification.

Within the VOicon and VOconnection types, the various VC specification types are differentiated by name (a string). This simplifies the code generator as it does not need to generate a new type for each VC type included in the specification.

For example, in the code below 'voa' is an instance of 'VOicon':

```
'[...] if (voa(voIcon)(iName) = "Process") do [...]'
```

Note that the correctness of the generation process is not affected by this simplification because the code that is generated is already correct. The method of guaranteeing the correctness of the generated code is based on the following principles:

- the type checker in the VC-t compiler guarantees correctly and strongly typed specifications;
- the generation process is transactional, i.e. for each constraint, the generating functions are executed within a transaction, if an error occurs, the parsing of that constraint is rolled-back and no code is written to the output.

8.4 Constraint Management

8.4.1 Introduction - Constraint Checking and Referral

In this section a study of constraint checking for diagram operations is presented. Diagram operations are performed over the following diagram objects: icons, connections and labels. The term 'icon' means an instance of an Icon type, we use it to designate an instance of an Icon type mapped onto the diagram. That is in fact a 'voIcon', but we use the two terms interchangeably. The same argument applies to 'connection' and 'voConnection'.

Every diagram obtained with a MT editor has an underlying graph structure. A graph is an abstract data structure composed of nodes connected by edges. Both nodes and edges hold data objects.

A diagram operation is a transformation of the presentation (the visual component of the user interface) resulting from a user command. There are two kinds of diagram operations available to the user: *graph preserving operations* and *graph altering operations*. In **graph preserving** operations only the layout of the graph is changed; for instance to move an icon or change the font of a label. In **graph altering** operations both the layout and the structure of the graph are changed; for instance, to delete or create a connection.

We have considered the following diagram operations:

Graph preserving operations:

- move - can be applied to icons and connections;
- change label font;
- change representation - applies to icons and connections when the underlying MT offers multiple representations for the VCs, i.e. more than one visual representation for the same concept.

Graph altering operations:

- create - applies to the diagram, icons, connections and labels;
- delete - same as create except for labels;
- update label;

A possible addition to the list of graph altering operations is *change type*, which consists in replacing the Icon or Connection type of an icon or connection, respectively, in a diagram. For example, a connection with type 'partial participation' in an ER diagram can be changed to a 'total participation'. This operation is done without deleting the diagram object.

Constraint Classes (Involvement and Scope)/ Diagram Operations	**Icon**		**Connection**		**Icon & Conn.**		**Icon Labels**		**Conn. Labels**		**Icon Labels & Conn.**		**Icon & Conn. Labels**		**Icon Lbls& Conn. Lbls**	
	L	**D**	**L**	**D**	**L**	**D**	**L**	**D**	**L**	**D**	**L**	**D**	**L**	**D**	**L**	**D**
Icon Creation	√	√			√	√	√	√			√	√	√	√	√	√
Icon Deletion		√			√	√		√			√	√	√	√	√	√
Connection Creation			√	√	√	√			√	√	√	√	√	√	√	√
Connection Deletion				√	√	√				√	√	√	√	√	√	√
Icon Label Creation							√	√			√	√			√	√
Icon Label Deletion							√	√			√	√			√	√
Icon Label Updating							√	√			√	√			√	√
Conn. Label Creation									√	√			√	√	√	√
Conn. Label Deletion									√	√			√	√	√	√
Conn. Label Updating									√	√			√	√	√	√

TABLE 3. Constraints checked for operations performed on diagram objects.

Caption:

L - Local constraints; D - diagram constraints
√ - constraints of the classes on column are checked for the operation on row.

Constraints are checked after any graph altering operation on a diagram object; so their satisfaction can be seen as a post-condition on the operation. If the checked constraints (one or more) are satisfied, meaning that the post-condition was met, then the operation is accepted; otherwise, if any constraint is violated, the post-condition fails, and in this case the operation is undone (rolled-back) and a message referring to the violated constraint(s) is given.

As we have seen in Section 7.2 on page 80, constraints can be classified according to the kind of diagram objects they apply to (*involvement* classes):

- Icon constraints apply to icons; for instance, 'a State Transition Diagram must have an initial state';
- Icon&Connection constraints apply to both icons and connections; for instance, 'on a State Transition Diagram a final state cannot have out-going transitions';
- Icon Labels&Connections apply both to the labels on icons and to connections; as in, 'on an Entity-Relationship Diagram, attributes linked to the same entity must have different names';

  and so on.

Consider for example the following constraint defined for State Transition Diagrams (STDs): 'a final state cannot have out-transitions' (meaning that a final state cannot have transitions whose origin is that state). The constraint concerns both an Icon - the *Final State* - and a Connection - the *Transition*; therefore the constraint belongs to the Icon&Connection class. There are two questions we want to obtain an answer for: when to check that constraint and where should it be referred to - in *Final State* or in *Transition*?

So, there are two problems we must solve:

- When a diagram operation is performed which constraints must be checked?
- Where should constraints be referred to - in icons or connections?

Table 3 shows the constraint *domain* on its columns, which is defined by the involvement and scope classes, and graph altering operations on its rows. The Table indicates which constraint classes are to be checked for each diagram operation. For each involvement class there are two columns corresponding to the classification of the constraint according to *scope*: one for *local constraints*, meaning that they apply to icons or connections; another for *diagram constraints*, which apply to the diagram itself. If we want to maintain updated information on the diagram state, *diagram constraints* must be checked for operations marked '√'. Otherwise they only have to be checked by an explicit *diagram check* operation.

To answer the first question we must look at the table information by rows. This is discussed in Section 8.4.2 and Section 8.4.3. To answer the second question we must look at the information by columns. Knowing which diagram operations induce the checking of each constraint class, we may be able to determine whether icons or connections should refer to that constraint. This is presented in Section 8.4.4.

8.4.2 Simplifying Constraint Checking

The information displayed by Table 3 is very comprehensive; however, as we will see in this section, not all the constraints checks shown in the table are necessary. We will present a study based on assumptions about the drawing method which introduces simplifications in constraint checking.

Diagram Drawing Method Assumptions

When a user edits a diagram, there are some assumptions which determine the way drawing operations are performed. For instance, a connection between two icons is created by first selecting the origin icon and then the destination icon; which means that the user cannot create a connection before the origin and destination icons have been created. Such assumptions will be used to make several reductions on the constraint checks identified on Table 3.

Assumption 1: the operation *Connection Creation* is performed by clicking first on a voIcon, which will become the origin of the newly created voConnection, followed by clicking on a second voIcon (which may be the same), which will become the voConnection's destination.

Assumption 2: the operation *Icon Deletion* provokes the deletion of all voConnections glued to the deleted voIcon.

Table 4 shows the simplifications obtained through these assumptions. To make the table less cluttered we didn't include label constraints and operations on them.

Constraint Classes (Involvement and Scope)/ Diagram Operations	**Icon**		**Connection**		**Icon& Connection**	
	L	**D**	**L**	**D**	**L**	**D**
Icon Creation	√	√			ØA1	ØA1
Icon Deletion		√			ØA2	ØA2
Connection Creation			√	√	√	√
Connection Deletion				√	√	√

TABLE 4. Considering assumptions on Table 3.

Caption:
√ - constraints of the classes on column are checked for the operation on row.

ØAn - constraints of the classes on column would normally be checked for the operation on row; however, due to Assumption n, constraints are not checked.

Empty cell means that constraints of the classes on column are not checked for the operation on row.

In Table 3 we can see that both for icon and connection operations, Icon&Connection constraints must be checked. Similar cases can be observed for operations on labels. In Table 4 constraint checking has been simplified in that Icon&Connection constraints no longer have to be checked for icon operations.

In what follows further simplifications on Table 3 will be done by extending the assumptions to labels and by the introduction of a law on the composition of diagram operations.

8.4.3 Simplifying Constraint Checking - a Formal Approach

Constraint Classes (Involvement and Scope)/ Diagram Operations	Icon		Connection		Icon& Conn.		Icon Labels		Conn. Labels		Icon Labels & Con.		Icon& Conn. Labels		Icon Lbls& Conn. Lbls	
	L	D	L	D	L	D	L	D	L	D	L	D	L	D	L	D
Icon Creation	√	√			Ø A1	Ø A1	*√	*√			*Ø A1	*Ø A1	Ø A1	Ø A1	*Ø A1	*Ø A1
Icon Deletion		√			Ø A2	Ø A2	Ø A4	*√			*Ø A4 A2	*Ø A2	Ø A2	Ø A2	*Ø A4 A2	*Ø A2
Connection Creation			√	√	√	√			*√	*√	√	√	*√	*√	*√	*√
Connection Deletion				√	√	√			*Ø A4	*√	√	√	*√	*√	*√	*√
Icon Label Creation							Ø*	Ø*			Ø*	Ø*			Ø*	Ø*
Icon Label Deletion							Ø*	Ø*			Ø*	Ø*			Ø*	Ø*
Icon Label Updating							√	√			√	√			√	√
Con. Label Creation									Ø*	Ø*			Ø*	Ø*	Ø*	Ø*
Con. Label Deletion									Ø*	Ø*			Ø*	Ø*	Ø*	Ø*
Con. Label Updating									√	√			√	√	√	√

TABLE 5. Formal approach to Table 3.

Caption:

√ - constraints of the classes on column are checked for the operation on row.

ØAn - constraints of the classes on column would normally be checked for the operation on row; however, due to Assumption n (see below), constraints are not checked.

Ø* - constraints of the classes on column would normally be checked for the operation on row; however, due to Law 1, constraints are delegated to another cell in the same column.

*√ - constraints have been delegated to this cell by Law 1 applied elsewhere in the same column.

*ØAn - constraints have been delegated to this cell by Law 1 applied elsewhere in the same column, but due to Assumption n (see below), these constraints (and also the ones originally in this cell) are not checked.

Empty cell mean that constraints of the classes on column are not checked for the operation on row.

Table 5 presents a formal approach to constraint checking based on an extension of the diagram operation assumptions given above. A law on diagram operations is also introduced.

Diagram Drawing Method Assumptions

Assumptions for VOs

Assumption 1: the operation *Connection Creation* is performed by clicking first on a voIcon, which will become the origin of the newly created voConnection, followed by clicking on a second voIcon (which may be the same), which will become the voConnection's destination.

Assumption 2: the operation *Icon Deletion* provokes the deletion of all voConnections glued to the deleted voIcon.

Assumptions for Labels

Assumption 3: the operation *Label Creation* is performed within the *Icon* or *Connection Creation* operation; *Icon* or *Connection Creation* includes creation of all the labels defined for the Icon or Connection.

Assumption 4: the *Label Deletion* operation is performed within the *Icon* or *Connection Deletion* operation; *Icon* or *Connection Deletion* includes deletion of all the labels defined for the Icon or Connection.

The Assumption 4 is more than obvious; Assumption 3 says that the creation of a VO in the diagram will automatically prompt the user to initialise the labels defined for that VO, we will come back to this in the paragraph below. The only label operation that exists in isolation is to update the label value.

As we have said, creation or deletion of a label cannot be performed in isolation, instead they are performed in the operation for the icon or connection to which they belong. We say that label operations (creation and deletion) are bundled with the corresponding icon or connection operations. As a result, constraints which apply to labels are now checked only once, for the operation on the icon or connection. Looking at Table 3 we see that, e.g., Icon Label constraints were checked both for Icon Creation and Icon Label Creation operations.

As we have seen before (see Section 7.1 on page 79) checking a constraint does not imply enforcing it; that only happens if the constraint is hard or has been hardened. So, Icon Label constraints are simply checked when the icon is created; if a constraint (for example 'a STD state must have a name') is hard or hardened then it is also enforced; otherwise, if it is soft or deferred then it is only checked and not enforced. For the example given, if it is hard then the user must give a valid name to the *state* when the icon is created, but if it is soft then the user may leave the label blank.

Law 1: for any diagram operations *opA* and *opB*, if *opB* is performed within *opA* and *opB* is only performed within *opA*, then constraints checked for *opB* can be checked for *opA* instead.

By Assumptions 3 to 5 we can apply Law 1 to operations on VOs and labels. For example, if opA is Icon Creation and opB is Icon Label Creation, then by Law 1 we can state that the constraints checked for Icon Label Creation can now be checked for Icon Creation instead.

8.4.4 Referring to Constraints

When a diagram operation is performed, the subset of constraints to be checked are the ones defined in the icon or connection involved in the operation. The semantics component of a VO (icon or connection) consists of a list of references to constraints. The Constraint Manager uses those references to check the necessary constraints. So, to define which constraints are checked, we must determine where to refer to them.

Consider the following constraint defined for State Transition Diagrams (STDs): 'a final state cannot have out-transitions'. As it belongs to the class Icon&Connection, and according to Table 3, it must be checked for both icon and connection operations. As a result, a reference to the constraint would have to be defined both in Icon *Final State* and in Connection *Transition*. However, by analysis of Table 5, and looking at the rows concerning Icon operations, we notice that constraints of class Icon&Connection no longer have to be checked for icon operations. Therefore the constraint above, and any constraint of the class Icon&Connection, no longer has to be referred to by icons.

Similar reasoning allows us to deduce a number of rules, included below, which determine where constraints must be referenced. These rules were applied in the design of the *Constraint Manager.*

To simplify the analysis process, we obtained Table 6 from Table 5. Cells without constraint checks have been cleared and empty rows have been deleted.

Constraint Classes (Involvement and Scope)/ Diagram Operations	Icon		Conne ction		Icon & Conn.		Icon Labels		Conn. Labels		Icon Labels & Conn.		Icon & Conn. Labels		Icon Lbls& Conn. Lbls	
	L	D	L	D	L	D	L	D	L	D	L	D	L	D	L	D
Icon Creation	√	√					√	√								
Icon Deletion		√						√								
Connection Creation			√	√	√	√			√	√	√	√	√	√	√	√
Connection Deletion				√	√	√				√	√	√	√	√	√	√
Icon Label Updating							√	√			√	√			√	√
Conn. Label Updating									√	√			√	√	√	√

TABLE 6. Final version of Table 3 after assumptions have been used.

Caption:
√ - constraints of the classes on column are checked for the operation on row.
Empty cell mean that constraints of the classes on column are not checked for the operation on row.

Deducing Rules From Table 6

From the information on Table 6 we have deduced a number of rules that determine where to refer constraints.

The first three rules apply only to local constraints (L).

From rows 1 and 2 we see that for all icon operations, only Icon and Icon Label constraints are checked. Hence, we have the rule below.

Rule 1:
Only Icon and Icon Label constraints must be referred to by icons[1]

From rows 3 and 4 we see that for all connection operations, only Icon and Icon Label constraints are not checked. The following rule is deduced.

Rule 2:
All constraints apart from Icon and Icon Label ones, have to be referred to by connections.

1. After creation of the icon not all constraints concerning the icon semantics are guaranteed to have been checked, in fact if the icon is mentioned by any Icon&Connection constraints then they will only be checked when a connection to that icon is created (see Rule 2). Until then the icon will be in the *disconnected* state.

Constraints which are to be checked for label updating operations are the ones defined in the icon or connection to which the label belongs, and within these only the ones that refer to labels.

From row 5 we see that for Icon Label Updating, Icon Label constraints are checked, which are referred to by icons according to Rule 1, but Icon Label&Connection and Icon Label&Connection Label are also checked, which are referred to by connections according to Rule 2. So, we deduce the rule below.

Rule 3:
Icon Label&Connection and Icon Label&Connection Label constraints must be referred to by connections as stated by Rule 2 and must also be referred to by icons.

Row 6, concerning Connection Label Updating, confirms Rule 2 in that for this operation only constraints relating to connection labels are to be checked. As stated in Rule 2 these constraints must be referred to by connections.

Diagram constraints must be checked even when the objects they concern have not yet been created; for instance, the constraint 'a STD must have an initial state' must be checked even before the diagram has an *initial state*. This reasoning adds an extra rule for Diagram constraints.

Rule 4:
Diagram constraints must be referred to by the diagram itself.

8.4.5 Conclusion

First we determined which constraints are checked for each diagram operation. This was displayed in the form of a table relating diagram operations with constraint classes (Table 3). The diagram operations are performed by the user according to a pre-defined diagram drawing method, which is based on a number of assumptions on diagram operations. After analysing these assumptions it was possible to produce a number of simplifications on constraint checking. Some of the simplifications were shown in Table 4.

We then formalised the study by extending the set of assumptions to cover label operations and we gave a law on diagram operations and related constraints; from this we produced Table 5. We then obtained a simplified version of Table 5. The information conveyed by this new table (Table 6) allowed the deduction of a number of rules to determine which VOs should refer to which constraints.

The results obtained from this study simplify the constraint management problem.

9. The GraphTool

9.1 Introduction

Applications making use of visual representations of graphs are becoming increasingly popular. The development effort for such applications can be substantially reduced if a tool incorporating generic facilities to manage graphs and their visual representations is provided to the user[1] (see Figure 26).

The tool described herein, designated by GraphTool, includes a standard graph library, a configurable graph representation manager, a constraint manager and repositories for object representations and constraints.

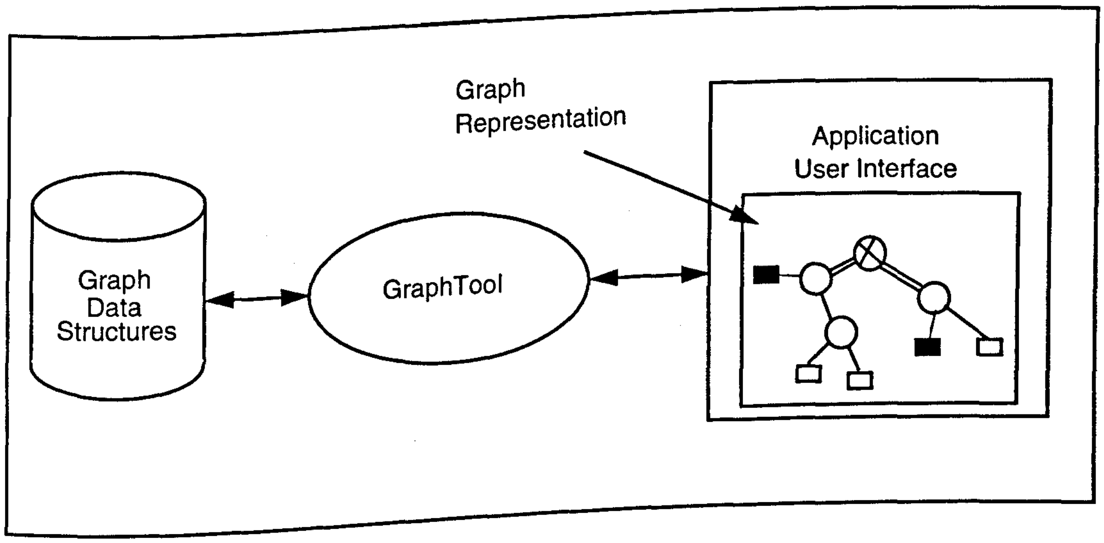


FIGURE 26. The GraphTool.

As an example of a GraphTool application we could have a diagram editor for the ER data modelling technique. A simple version of this editor could have the following components: a drawing area (canvas) where the ER schema is built; a menu bar with options to create a new schema, check the correctness of the schema, save, print, etc.; and a palette allowing the selection of the visual objects, components of a schema, such as *entity, attribute, relationship*, etc..

1. In the following the term 'user' refers to the application designer.

The GraphTool supports the common facilities that are characteristic of such graph-based applications. It includes support for persistence, direct manipulation of diagrams and a number of standard graph operations.

9.2 The Architecture

Herein, the architecture of the GraphTool is described. The tool provides to an application a customisable environment to support graph structures which can be persistent, have a visual presentation and incorporate semantics. The GraphTool also defines a protocol for the communication with the application.

The architecture is shown in Figure 27. In the architecture we can identify three levels: the abstract graph level, the representation level and the application level.

The **abstract graph level** describes standard graph structures which can hold data both on nodes and edges. It also includes the usual graph operations such as insert a node, insert an edge, delete a node, update the data element of an edge, graph traversal algorithms, etc.

The **representation level** holds visual representations for nodes and edges; it includes generic visual operations, such as draw, erase or move. One or more representations may be associated with the same graph. The representation level defines: a mapping with the abstract graph level and a protocol to communicate with the application.

The **application level** corresponds to the client of the GraphTool. The communication with the representation level is determined by a well defined protocol.

The GraphTool covers the abstract graph level and the representation level. The abstract graph level consists of a Graphs Library and a repository for Abstract Graphs which are parameterized with the application types. The representation level is implemented by a Graphs Representation Manager, responsible for the functionality of the GraphTool, the connection to the abstract graph level and the communication with the application; a Visual Objects Manager, which communicates with an User Interface Management System (UIMS); and a Constraint Manager, for the checking, validation and enforcement of the constraints on the Visual Objects. The two later components each have an associated repository. A mapping mechanism is used to attach a representation to the abstract graph; this mechanism will be explained below.

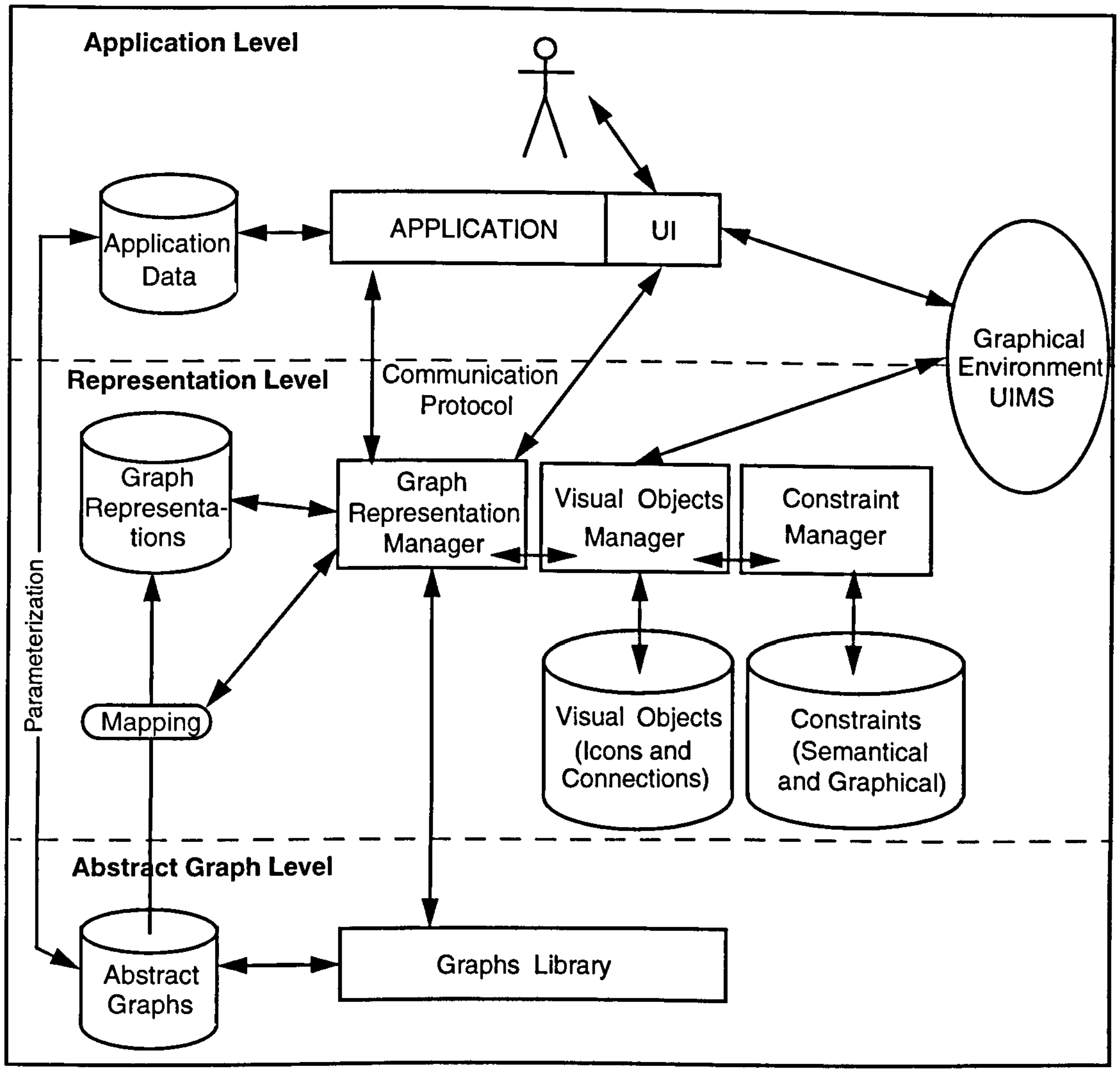


FIGURE 27. Architecture (detailed view).

Considering the ER editor example given above, the application level would include, amongst others, the following types: Entity, a structure composed by a name, attributes and a key; Relationship, a structure including a label and cardinality. The application level would also define procedures to manipulate these types, e.g. instantiate a type or print its values. The representation level would include the EntityIcon type, a structure composed by an image (a rectangle), a label (for the name) and a list of constraints (the semantics); the RelationshipIcon type, the same structure in which the image is a diamond, instead of one there would be three labels (the name and the two cardinality values) and with other constraints. The abstract graph level would include nodes and edges parameterised with the application types.

Mappings are used to attach the representation to the abstract graph; for each attachment between a representation and an abstract graph two mappings are instantiated: one for the icons and one for the connections. In the example there is only one representation for the abstract graph, but we could have for instance another representation in which, not only the name, but also attributes of entities were shown. In this case a second pair of mappings would be used. A single abstract graph structure can be attached to several representations; a connections mapping and an icons mapping are defined for each attachment.

The abstract graph level is supported by a standard graph library. The implementation of this library was simply done by resorting to the established literature [Stubs89] and [Kruse87]. The graphs supported by the abstract graph level only differ from the standard ones in that it is possible to store data on the edges. This design option was taken based on the fact that a large number of applications do use data directly related to graph edges as well as data related to graph nodes.

The main contribution of our work is the design and implementation of the representation level and the interface with both the graph library and the application. It offers mechanisms allowing applications to manage persistent graph data structures through one or more visual representations.

The GraphTool also includes a Visual Objects repository. These Visual Objects are accessed by the application at run-time through the representation level, as described below. The abstract graph level and the representation level are presented below; the discussion will be focused on the latter because this is the innovative aspect of the architecture.

9.3 Abstract Graph Level

A **graph** is a generalisation of a hierarchical data structure. It has nodes containing some information and edges connecting the nodes. At this level any node can be connected to any other node (although the representation level will normally enforce some constraints on the types of nodes that can be connected by which types of edges, we will come back to this in a later section).

A graph is used to provide a structure for other data types in the same way as a list, for instance we can have lists of integers, strings, trains or rabbits. The nodes in a graph also held objects of different data types, the node elements. It is called an **abstract graph** because it abstracts the common concepts of different graphs, hiding the unnecessary details. In order to implement abstract graphs it is necessary to parameterise them with the data types of the objects held by the graph nodes.

This conforms with the standard definition of a graph [Stubs89] and [Kruse87]. But in a large number of application domains it is very useful to have at hand information stored in the edges. For instance, the distance between two cities or the throughput of a connection in a computer network. This is expressed by edge data types. So, in our graphs, both edges and nodes will be parameterised.

In conclusion, in the context of this work, a graph is a double parametric data structure.

The graph library supports any kind of graph, namely: connected/not connected, directed/undirected, weighted/unweighted, cyclic/acyclic[1]. It also provides **standard graph operations**, such as: insert a node, insert an edge, delete a node, update the element of a node, retrieve the element of an edge, graph traversal and shortest paths algorithms.

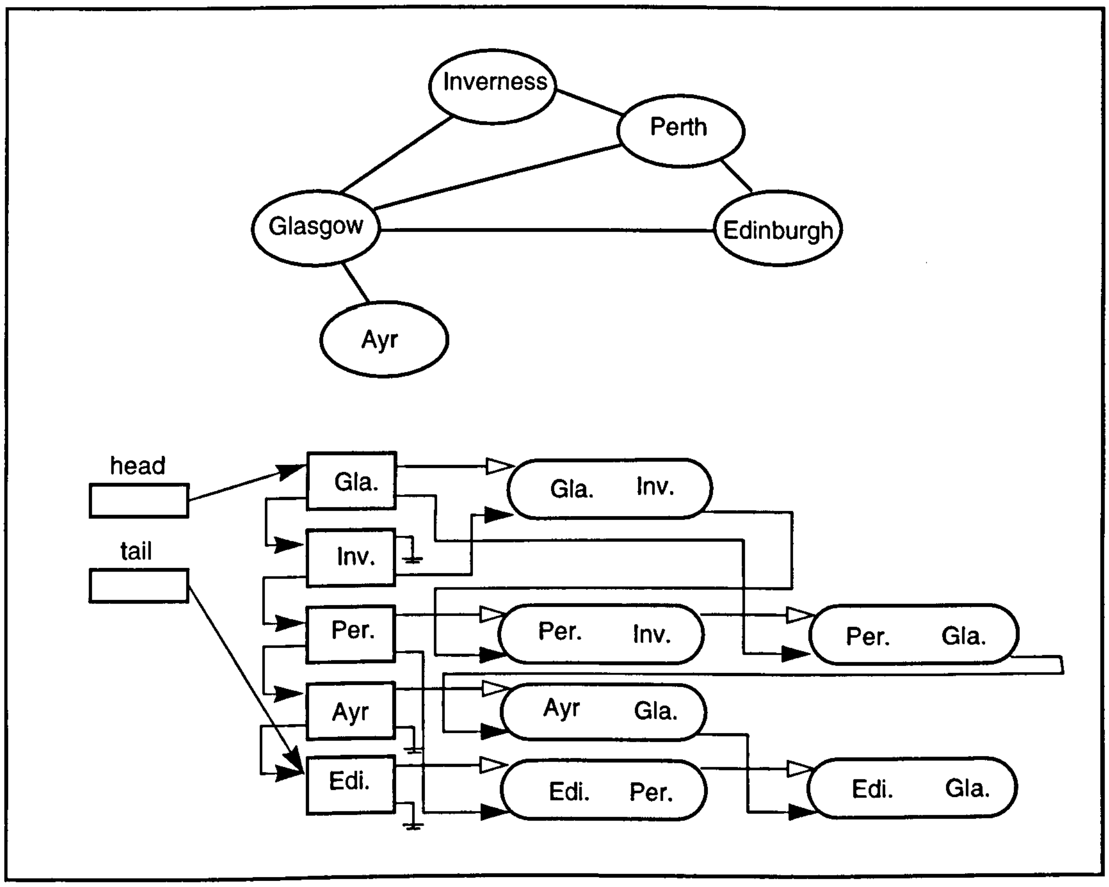


FIGURE 28. Adjacency list representation.

An **adjacency list** was chosen to represent a graph. This representation is specially suitable for not very dense graphs, which is normally the case of those associated with diagrams obtained with modelling techniques. In the future, an alternative adjacency matrix representation will be implemented, in order to support a wider range of application domains. An adjacency list can be viewed as a list of lists. The basic list contains one entry for each node and is called the **node list**. For each node in the node list there are two lists (depicted by

1. At this moment it is not possible to determine if a graph is planar.

different arrow tips), call them **edge lists**, of the neighbours of that node. In the first of these, the node's key value is the first of the pair of values. In the second, the node's key value is the second of the pair of values. In the example of Figure 28 a graph has been used to represent road connections between Scottish cities.

9.4 The Representation Level

The representation level provides a visual and interactive interface between the application and the graph data structures. The Representation Manager provides a default interface style; however, it can be configured and different interface styles can be obtained.

The Representation level includes:

- visual objects which can be of type Icon or Connection; it also provides semantics and interaction capabilities;
- a number of procedure generators which provide a default functionality (these can be overridden by the application);
- connection to the abstract graph level: a mapping mechanism to connect the representation level to the abstract graph level;
- connection to the application level: a run-time communications protocol to bind the GraphTool to the application;
- a visual objects manager and an associated repository;
- a constraint manager and repository to implement the visual objects semantics.

These topics are discussed below.

9.4.1 The Visual Objects

As Visual Objects (VOs) have already been discussed in Section 4.4 on page 42, we redirect the reader to that section.

9.4.2 The Graph Representation Manager

The Generators and Default Functionality

When the GraphTool is in use by a client application, a number of procedures implementing operations to manipulate visual graphs are available, allowing for instance to drag a node or create a new edge. These procedures are parameterised with nodes types and edges types which are provided by the application in hand.

The procedures are obtained by activating the corresponding *procedure generators* provided by the GraphTool. The procedures must be generated, rather than provided in a library, because they are parametric and therefore can only be generated once the application types are known. The generators can be modified in order to customise the default functionality. Hence, as the functionality of the GraphTool is dictated by the generators used, different functionalities can be obtained if different generators are used.

```
!*****************************************************************************
!** Data structure to bind the application to the kernel at compile time

type GraphToolFunctionality[N, E] is structure(
  drawNodeProcGen: proc[N, E](
          CallData[N, E], CurrentGraph[N, E], CurrentIcon,
          WindowManager, callbackList[N, E]
          -> proc( Event ));
  drawEdgeProcGen: proc[N, E](
          CallData[N, E], CurrentGraph[N, E],
          CurrentIcon, WindowManager, callbackList[N, E]
          -> proc());
  chooseIconSetProcGen: proc(
          IconSet, CurrentIcon -> proc( string ));
  drawGraphProcGen: proc[N, E](
          CurrentGraphWrapper[N, E], proc[N, E]() -> proc())
  ......
  )
```

FIGURE 29. Some default generators.

A default set of generators is provided by the tool. It's the application designer's responsibility to choose which generators to use. He/she can use only the default ones but will probably want to define some specific functionality. If this is the case, then he/she must provide one or more customised generators. For instance, if the application needs a customised way of drawing a node, then a generator for the drawNode procedure must be provided. This can be done using the default generator as a template. In conclusion, the tool provides a default functionality but this can be customised by the application designer.

The complete set of default generators produce the following procedures: selectNode; selectEdge; showNode; showEdge; hideNode; hideEdge; drawNode; drawEdge; eraseNode; eraseEdge; dragNode; printNode; printEdge; chooseIcon; chooseConnection; chooseIconSet; chooseConnectionSet; drawGraph; printGraph.

Some procedures only affect the representation, such as dragNode; others affect also the underlying graph, for instance eraseNode. The former are called graph preserving operations and the latter are called graph altering operations (see Section 8.4 on page 111). Having several representations for the same graph raises consistency and data protection problems when a graph changing operation is performed. For instance, if a node is deleted from a graph then all representations of that graph must reflect the change, otherwise they will become inconsistent. As an example of data protection, for a given graph some application may have permission to perform delete operations whereas others don't. The GraphTool does not provide any automatic mechanisms to enforce consistency or data protection policies. The applications are responsible for the definition of those mechanisms.

Connection to the abstract graph level

A number of different representations can be produced for a graph stored in the repository (see Figure 30). New representations can be associated to an already existent graph.

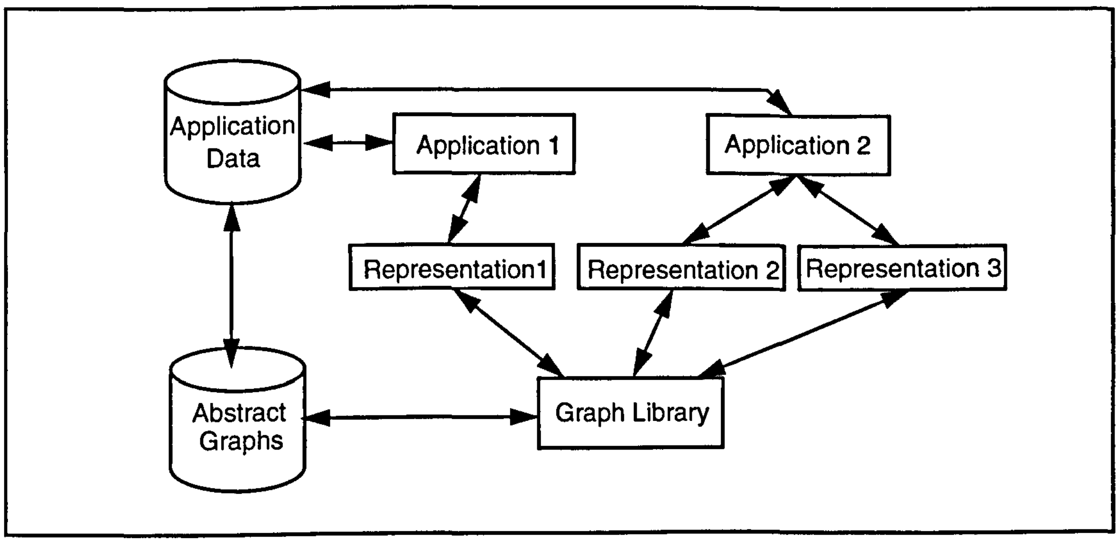


FIGURE 30. Multiple representations.

Each application must be associated with one *Graph Representation Manager* (GRM) for each graph displaying window (see Figure 31). The GRM executes the operations provided by the Graph Library. There is a repository associated with each GRM and just one associated with the Graph Library. The first stores the representations of graphs and the second stores the graph structures. Each GRM repository is attached to the graph repository by means of map structures [Atkinson90].

For each GRM two maps are defined: an icons map and a connections map. The icons map associates each voIcon (called icon from now on) in the diagram to a node in the underlying graph structure; the connections map associates each voConnection (called connection from now on) to an edge in the graph.

There are however auxiliary nodes and edges in the graph that do not have a direct correspondence to VOs in the diagram. Here is the reason why. The standard graph library included in the GraphTool only allows for a single edge between two nodes. However, some applications require multiple connections for a pair of icons. For instance in the STD technique, several transitions can be drawn between the same two states. Even in the example of Figure 28 we could have several road connections between a pair of cities. To cover these situations, without changing the standard graph library, a mechanism was devised: for each edge associated with a connection by the connections map, there will

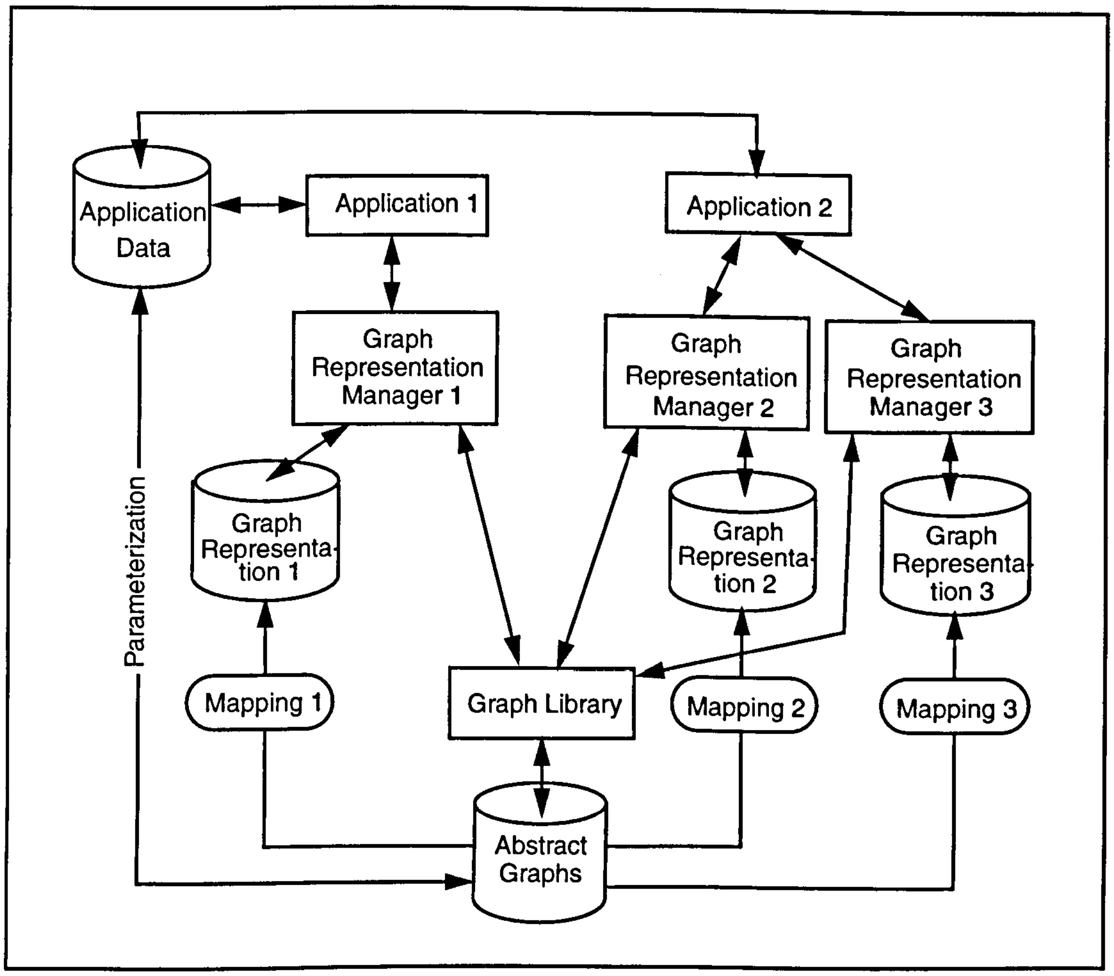


FIGURE 31. The representation managers.

also be an auxiliary node and an auxiliary edge. These auxiliary components are not included in the maps. So, to each connection in the diagram there will be a corresponding sequence 'auxiliary edge, auxiliary node, edge'.

In the example shown in Figure 32, the STD extract represents the text editor 'vi' of the Unix operating system, the transition between the 'command mode' state to the 'insert mode' state can happen by either pressing the 'a' key or the 'i' key. The corresponding graph structure is composed by node 'N1', which is associated with the icon 'command mode'; node 'N2', associated with icon 'insert mode'; and the sequences 'Ea1, Na1, E1' for connection 'press key <a>', 'Ea2, Na2, E2' for connection 'press key <i>'.

9.4.3 Visual Objects Management

The application must choose the VOs it will use; it selects the relevant VOs from a repository. In order to permit user interaction with the diagram, VOs hold interaction capabili-

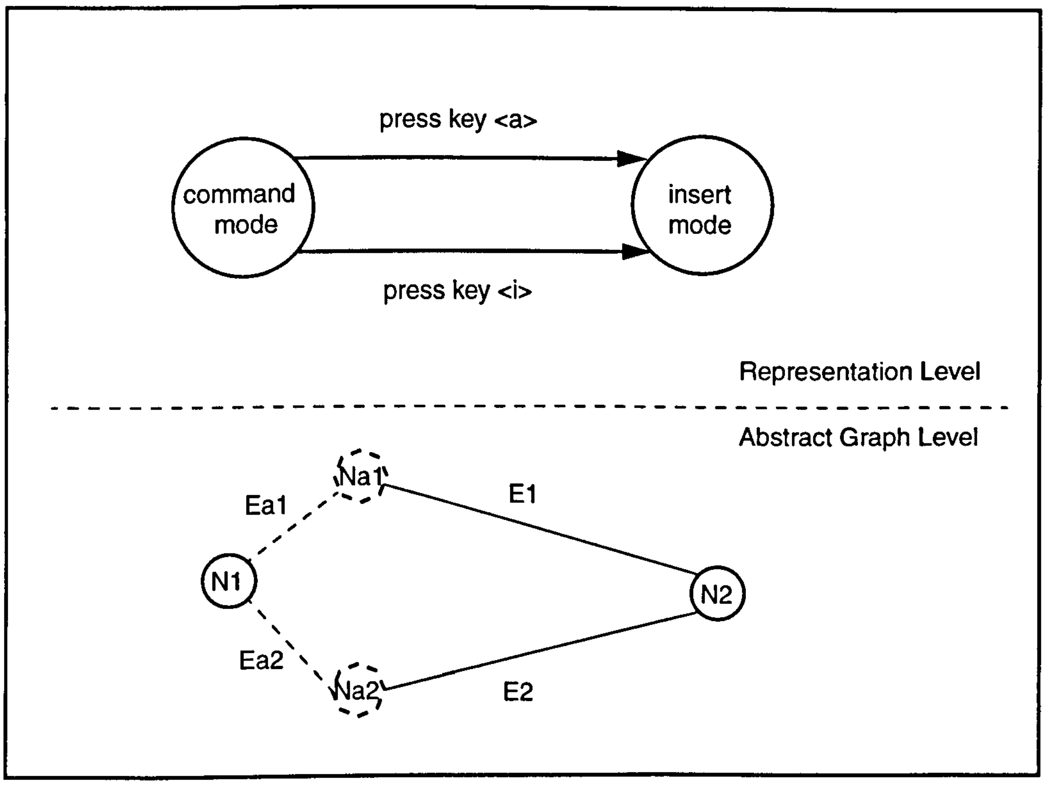


FIGURE 32. Attaching the Abstract Graph Level to the Representation Level.

ties: they include a selectable area and are able to handle user events. These features are supported by an UIMS.

9.4.4 Constraint Management

The GraphTool includes a constraint manager and a constraint repository in the representation level. This allows the semantics of diagram based techniques (e.g the ER data modelling technique) to be supported by the GraphTool. Each diagram based technique defines a number of rules on their concepts or constructs as a result of their semantics; these rules can be implemented as constraints inside the GraphTool repository, which are then applied to the visual objects and checked during the diagram drawing process. Constraint management is presented in Section 8.4 on page 111.

9.5 Connecting an Application to the GraphTool

It is necessary to provide the means to connect the application level to the GraphTool; for that purpose a communication protocol was defined.

9.5.1 Interfacing between the GraphTool and the application

The communication protocol establishes:

- how the application accesses the functionality of the tool;
- how the tool can display the visual graph on a canvas window;
- how the tool can access information provided by the application (for this a callback mechanism is used).

This three element protocol is depicted in Figure 33.

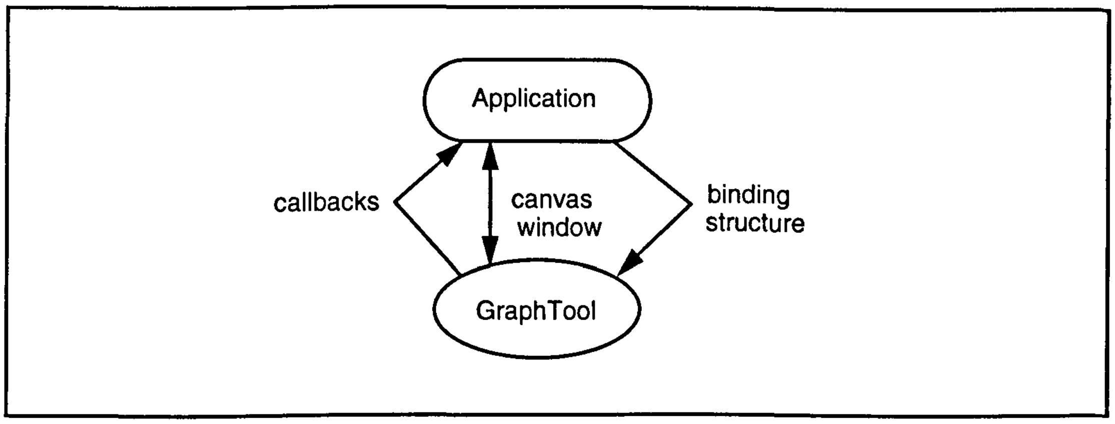


FIGURE 33. The communication protocol.

The protocol regulates the run-time bi-directional communication between the application layer and the GraphTool. We will first discuss the direction from the application level to the GraphTool, i.e. the way the application layer can access the GraphTool.

9.5.2 Accessing the GraphTool functionality

An application is bound to the GraphTool through the *GraphToolFunctionality* structure as described above (see Figure 29). This structure holds the set of generators chosen by the application. Upon compilation, the *GraphToolFunctionality generators* produce a number of procedures to manipulate the visual graphs which implement the functionality. The application uses those procedures as connectors to the GraphTool.

9.5.3 The Canvas Window

In order to display a graph, the application must provide a window to the GraphTool. This window is called the **canvas,** it receives user events and displays visual objects depicting the underlying graph.

The application sketched in Figure 34 presents a canvas window displaying a graph structure.

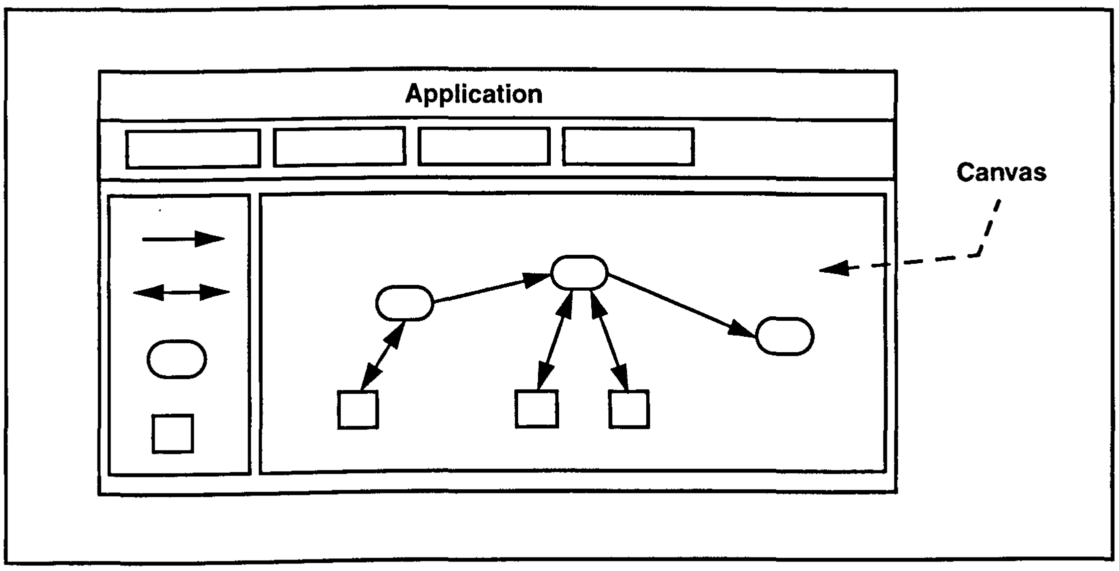


FIGURE 34. The canvas window.

9.5.4 Callback mechanism

```
!*****************************************************************************
!** Data structure to get data from the application to the kernel at design
!** time

type CallbackList[N, E] is structure(
  drawNodeCallback: proc( -> N );
   drawEdgeCallback: proc( -> E );
  selectNodeCallback: proc( Node[N, E]);
  eraseEdgeCallback: proc();
  printNodeCallback: proc( N );
  .....
  )
```

FIGURE 35. Callbacks.

As we mentioned above, the communications protocol must also establish the way the GraphTool can access the application at design-time. The necessity for this arises, for instance, in the following situation: a hypothetical application provides a user command to display the information stored in a node. In this case, the application asks the tool to display that information. However, this operation depends on the type of the data stored in the node, so the application must tell the tool how to do it. This is done by a callback mechanism. Hence, the callback mechanism is used to allow the tool to execute application specific operations.

A callback is a procedure in the application code. It is executed by the GraphTool at diagram drawing-time to perform some application specific operation. The GraphTool defines a number of callbacks in the type *CallbackList* (see Figure 35). For each representation operation specified in the structure GraphToolFunctionality there is a corresponding callback. Callbacks are linked to representation operations by means of post-conditions. There is a post-condition for each representation operation. The callback procedures, provided by the application, are then associated with the corresponding post-condition. Each time an operation is executed over a visual graph the corresponding post-condition can be fired and the corresponding callback executed.

For instance, if a showNode operation is to be executed the application may use the post-condition defined for that operation to display the information stored in the node. The application must provide a callback procedure to print the data held by a node; for this purpose the application associates that procedure to the printNodeCallback specified in the type *CallbackList.* At run-time the application invokes a printNode operation in the tool; this will fire the operation post-condition; the corresponding callback procedure defined by the application is executed; this will print the node data and finally the tool passes the control back to the application level.

A complete example on how to use VOs in an application based on visual graphs is presented in the next section. It will also be explained how to develop a complete application using the GraphTool.

9.6 Building an Application

In this example the application must define the menu bar, the palette and also provide the GraphTool with a canvas window and a window manager for the canvas window. The GraphTool will then generate a number of procedures to edit and manipulate a graph in the canvas window.

The GraphTool is parameterised with the node type and edge type. These types will typically be variants (unions) of a number of types, each of which corresponds to one icon type or connection type used by the application. The application must provide the operations to manipulate those types in the form of callback procedures.

The procedure *generateGraphRepresentationManager* must be executed. It is given the following parameters:

- The canvas window, the window where the graphs will be displayed.
- A callback list, a structure with the callback procedures supplied by the application.

- A structure describing the functionality provided by the Graph Representation Manager, consisting of a number of generators which implement the functionality. If the application requires specific functionality, it may customise the corresponding generator(s).
- The set of icons and the set of connections to be used in the graph representation.

The procedure generateGraphRepresentationManager generates a structure with the procedures to bind the tool to the application at run-time, e.g. create a node, create an edge, drag a node, etc..

Using the GraphTool Step-by-step

1. Create the canvas window and an associated window manager.

2. Provide the callback procedures.

3. Create a structure with the functionality generators.

4. Define the node types and, if necessary, the edge types; any required procedures to deal with these types must also be provided.

5. Select the set of icons and the set of connections to be used.

6. Call the procedure *generateGraphRepresentationManager* with the parameters specified above. Name the return value by assigning it to a variable (e.g. GraphToolBindings).

7. Use *GraphToolBindings* to execute the representation operations. For instance, if the user issues an application command to show a node, then the GraphTool procedure pointed to by 'GraphToolBindings(showNode)' is executed.

9.7 Conclusions

The GraphTool simplifies the development of visual and interactive applications based on graph structures which may be persistent and have intrinsic semantics.

The GraphTool has the following features:

- the graphs are double-parameterised by node types and edge types; this allows data to be stored both in nodes and edges;
- provides support for multiple visual representations of the same graph structure;
- implements most of the common operations amongst graph-based applications;

- its functionality is completely configurable; the provided default functionality can be customised by the application;
- a callback mechanism is used to allow the execution of application specific operations;
- icons and connections can be chosen dynamically from a repository;
- searches both at the representation level and at the data level can be easily implemented using the provided traversal routines;
- support for the use of constraints to express the semantics of Visual Objects (topic covered in Chapter 6) is also provided;
- because it has been implemented on top of a persistent system [Atkinson95], namely the Napier88 system [Morrison94], graphs and their representations can be made persistent;
- supported by a consistent architecture; clear separation between graph and representation.

From the use of the GraphTool in concrete practical situations, such as on the implementation of the prototype for this thesis, we are able to state that: an application based on visual representations of graph data structures can be implemented in a much easier way when a tool with the above features is provided. The GraphTool has been implemented on the Napier88 system and is, therefore, non-portable to other architectures. However, its principles and design can be used in implementations using different systems which offer some kind of support for persistence, e.g. Fibbonaci [Albano95], Tycoon [Matthes94] or O_2 [Bancilhon92].

10. A Prototype

10.1 A Brief Summary of the Technology Used

10.1.1 Using the Napier88 System

Napier88 is a persistent programming language, meaning that it is aimed at building systems supporting long-lived data, which may include procedures for they are higher-order citizens in the language. These systems are termed Persistent Application Systems (PAS), e.g. in [Kuo95] a geographical information system implemented in Napier88 is described.

The Napier88 system has been developed at the University of St. Andrews, and consists of the language and its persistent environment. A description of the language can be found in [Morrison94]. A persistent programming environment [Waite95a] and several libraries [Waite95b] for the Napier88 system have been produced at the University of Glasgow.

We will not present the Napier88 system or discuss the principles of orthogonal persistence. The research on this field is extensively described by a large number of publications. A definitive review on orthogonally persistent object systems is given in [Atkinson95]. Instead, we will focus on how the system was used to build the prototype and in what way its characteristics influenced the design, implementation and execution of the latter.

Napier88 is the target system of the prototype. This means that the VC compiler generates Napier88 code which implements the design tools for the modelling techniques (MT). In addition, the following components were programmed by hand in Napier88:

- the templates, used in the code generation process;
- the GraphTool;
- the Configurable MT Design Tool;
- the Visual Objects for the generic visual language;
- the constraint manager;
- all storage mechanisms.

Below we list the characteristics of Napier88 we have found most important to our work:

- strong typing;
- parametric polymorphism;
- higher-order procedures;
- orthogonal persistence;

- support for dynamic binding.

How these characteristics were reflected in the prototype is shown in what follows.

The user of our prototype uses the VC-t language to write specifications from which Napier88 code is automatically generated. So, type checking is seen by the user only at the VC-t level, not at the Napier88 level. However, we found that the implementation of the prototype itself was made much easier by the strongly-typed programming environment.

Parametric polymorphism has been used in the implementation of the GraphTool. The graph objects (nodes and edges) and procedures can be parameterized with application specific types. This provides genericity to the tool. The GraphTool does not provide a fixed functionality; in fact, the functionality is provided as a set of procedure generators (i.e. procedures that return other procedures) which are defined by the client application (although a default set of generators is provided by the tool). The implementation of this mechanism was possible because procedures are higher-order data objects.

The kind of persistence supported by the Napier88 system is orthogonal to the model of the data. This feature aided in the modelling and implementation of the components of the generic visual language. The visual representation layer was organised into different object classes. These objects have different life spans. The *Graphical Objects Definitions* (shapes and line styles) have the greatest longevity; in effect, they are created before any MT design tool is built. Shapes and line styles are defined in advance. For example, to create a visual representation of the 'process' construct defined for Dataflow Diagrams (DFD), it is necessary to have previously defined its corresponding shape, i.e. a circle. The *Visual Objects Definitions* become alive when the code for a particular MT is generated. These are icons and connections (which include constraints). For example, the DFD construct 'process' mentioned above is defined as an icon. Finally, we have the *Visual Objects*, which are the components of the diagrams. Consider for instance a DFD representing a bank autoteller machine; in this example, a process could be 'request PIN'. The life of Visual Objects is conditioned by the diagram in which they have been created. If the diagram is deleted they are also destroyed. To implement the storage we have simply placed objects having the same longevity in a separate environment. Without an orthogonally persistent system we would have created different databases to store information on the objects grouped according to their longevity. The objects would also have to be translated into the model used by the database. This would have certainly increased the complexity of the system.

The benefits of persistence to software engineering are already well demonstrated [Cooper94; Morrison94]. The main purpose of the Napier88 system is to provide mechanisms to prove the feasibility of theoretical principles. Performance and usability have often been compromised to make it simpler to achieve that goal. The P-Java [Atkinson96] project is currently under way at the University of Glasgow; it is more geared towards industry applications and aims at a larger community of users than Napier88. All aspects of performance will now be a priority.

10.1.2 The TkWin Graphical User Interface Management System

TkWin is a graphical User Interface Management System (UIMS) for the Napier88 system. It was developed as an MSc project [Larsson96], with the goal of connecting non persistent graphical user interface technology to a persistent application system. This project replaced the previous UIMS, called WIN [Cutts89; Kirby94].

WIN was almost entirely implemented inside the persistent environment. For that fact it acquired all the advantages of a persistent system, some of which we already described in Section 10.1.1, but it had two major drawbacks. First, if any new functionality was to be given to the window system it had to be implemented inside the persistent environment. Therefore, it was not possible to exploit new developments in state-of-the-art non persistent window systems. Second, the implementation of WIN inside the persistent environment represented a heavy load to the system both on space and computational power.

The non persistent user interface technology chosen was Tcl/Tk [Ousterhout94]. It comprises two software packages, called Tcl and Tk. Tcl is a simple scripting language for controlling and extending applications. Tk is a toolkit for the X Window System implemented as an extension of Tcl.

Two main requirements for TkWin were:

- the programming interface must hide Tcl/Tk and use only the Napier88 language, i.e. all code for a TkWin application must be written solely in Napier88;
- TkWin applications must be given persistent behaviour.

A library of Napier88 procedures to create and manipulate Tk widgets has been implemented; the first requirement is thus satisfied. To implement persistence, the second requirement, the state of all widgets in a TkWin application is kept on the server side. The complete TkWin application is made persistent by bringing to the Napier88 client the information about all widgets' state. This information is then saved in the persistent stable store.

TkWin is implemented as two separate processes, a Tcl/Tk server and a Napier88 client, communicating via a socket connection (see Figure 36). The server accepts commands from the client and executes them in an integrated Tcl interpreter. 'The Tcl interpreter deals with the low level connection to the X-server and provides (via Tk) a set of functions for creation and manipulation of Motif-like widgets. The Tcl/Tk server also provides a mechanism for registering X events to be redirected to the Napier88 client. This allows for callback functions written in Napier88 to be executed as the result of events occurring in Tk widgets' [Larsson96].

Due to clear advantages of TkWin and also because of our involvement in the project, which gave us an insight to the system, we decided to adopt it in favour of WIN. The prototype had already been started to be developed using WIN, so it had to be changed to accommodate the new UIMS. The following components were affected: the Visual

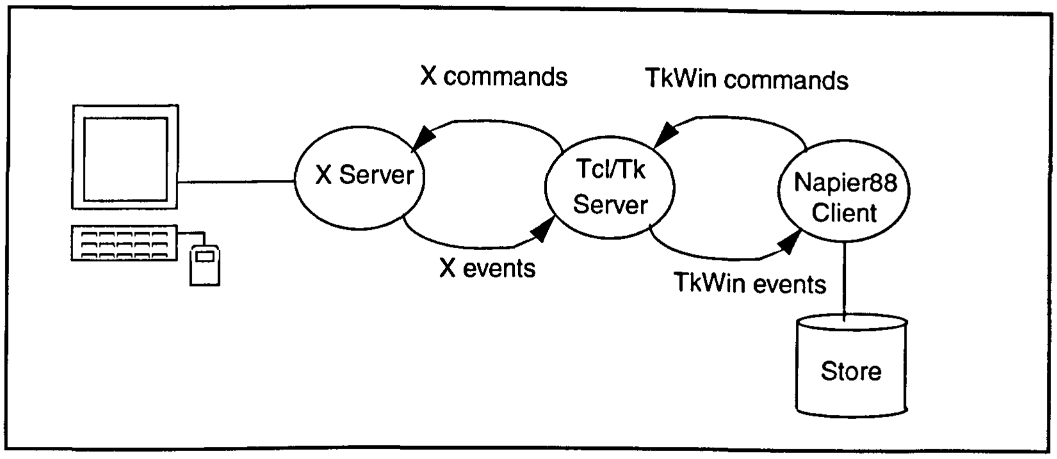


FIGURE 36. The TkWin architecture.

Objects structure, the representation level of the GraphTool and the configurable design tool. The changes in the representation level of the GraphTool consisted in the development of a new drawing canvas. The configurable design tool was completely redesigned and new features added. This work was carried out in the scope of another two MSc projects which are presented, respectively, in [Meng96] and [Fun96].

However, the current version of TkWin presents some tough problems. The widget set provided is very limited. If new widgets are needed they must be imported from the Tk widget set to the persistent environment. At the moment this is done by hand and is a time consuming, non creative and repetitive task. Tool support to automatically port widgets can be accomplished, but it is not provided in the current version. Widget hierarchies inside the canvas widget (which implements the drawing area of the design tools) cannot be made persistent; as a consequence it is not possible to make persistent the diagrams edited.

In spite of those problems, we believe TkWin is a step in the right direction. The look-and-feel of the prototype's graphical user interface was greatly improved which contributed for the overall usability of the prototype.

10.1.3 The Lex and YACC Tools for Compiler Construction

'It is much easier to produce a correct parser using a grammatical description of the language and a parser generator, than implementing a parser directly by hand' [Aho86].

The compiler's front-end was built with the aid of the Lex and YACC tools [Aho86; Schreiner85]. Lex and YACC are program generators that produce programs in a host language. The C programming language was used as the host language.

Lex generates a program which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The generated program reads an input stream, copies it to an output stream, and partitions the input into strings (tokens) that match the given regular expressions.

The Lex program generator was used with the YACC parser generator. A parser is a program that imposes a structure to tokens according to grammar rules. The grammar rules describe the syntax of the source language. The syntax specification is provided to YACC by the user. For specifying the syntax the BNF (Backus-Naur Form) notation is used. Lex recognises the input tokens which are passed on to YACC which in turn organises them into grammatical phrases. These phrases are described by an intermediate representation called a parse tree which is then used by the compiler during the code generation phase.

10.2 Current Status of Implementation

In this section we specify what has been achieved in the implementation at the time of submission of the thesis.

The *compiler's front-end*, which includes the lexical analyser and the parser, have been built. An intermediate representation, which is semantically checked for scope rules and type information, is generated by the parser. The compiler's front-end can therefore be used to test the formal correctness of specifications, in accordance to the VC-t language.

The compiler's design and implementation was done with the forethought of maintaining its front-end independent from its back-end. This allows for the front-end to be used, as it is, with different target languages and systems.

The *compiler's back-end* is a Napier88 code generator which is able to produce a visual and interactive design tool supporting the semantics of an underlying MT.

As part of the back-end an automatic translator of semantic constraints (VC-t to Napier88) has been built. Although some translation rules have not yet been automated (used in the automatic process of translation), amongst these are some of the generalization rules, all of them have been hand-tested (hand translated constraints) which allowed us to assert the correctness of the produced code.

The *constraint manager* is working. The checking and enforcement of constraints is correctly done from a semantic point of view. Some optimizations could still be done to minimise the number of constraints checked. However, for the tested MTs, the system's response times are adequate. We therefore consider that future enhancements on the constraint manager are not critical for demonstrational purposes.

The *configurable GraphTool*, which architecture, principles and design are presented in Chapter 9 has been completely implemented. Its data structures are based on VOs (Visual Objects - a realization of the VC formalism). It may be configured in both the graphical objects used as icons and connections representations, and its functionality. A default functionality is provided but it may be overridden by application specific code.

Although the GraphTool is being used within the prototype of our system, it has been developed as a generic tool, in the sense that it can be used by any interactive application based on graphs with visual representations of the type ‘icon and connection’ diagrams.

The GraphTool has already been used in two MSc projects. In one of the projects, described in [Meng96], the UIMS provided by the Napier88 system, called WIN, which was used in the implementation of the GraphTool's User Interface on version 2.0, was replaced by TkWin, a Tcl/Tk based UIMS [Larsson96]. In the other project a *front-end* for the GraphTool was produced; this consists in a tailorable diagram editor that can be used by an application to produce a dedicated diagram editor. From a straightforward adaptation of this project (only paths and labels were changed) we produced the Configurable MT Design Tool used in our prototype. The project is reported in [Fun96].

The aspects related to the *generic visual language* and the *refinement of specifications to configure the MT usage* were not fully implemented due to time limitations. The fact that the UIMS supporting the visual language has been developed as part of a MSc project, which was described in Section 10.1.2 on page 138, and that this happened simultaneously with the implementation of the prototype, caused the coding phase to take longer than expected. In order to produce a complete implementation some further developments in the UIMS, namely the implementation of a more advanced widget to support the drawing of graphical objects (‘canvas’) would be necessary. What has been implemented is however sufficient to demonstrate the approach.

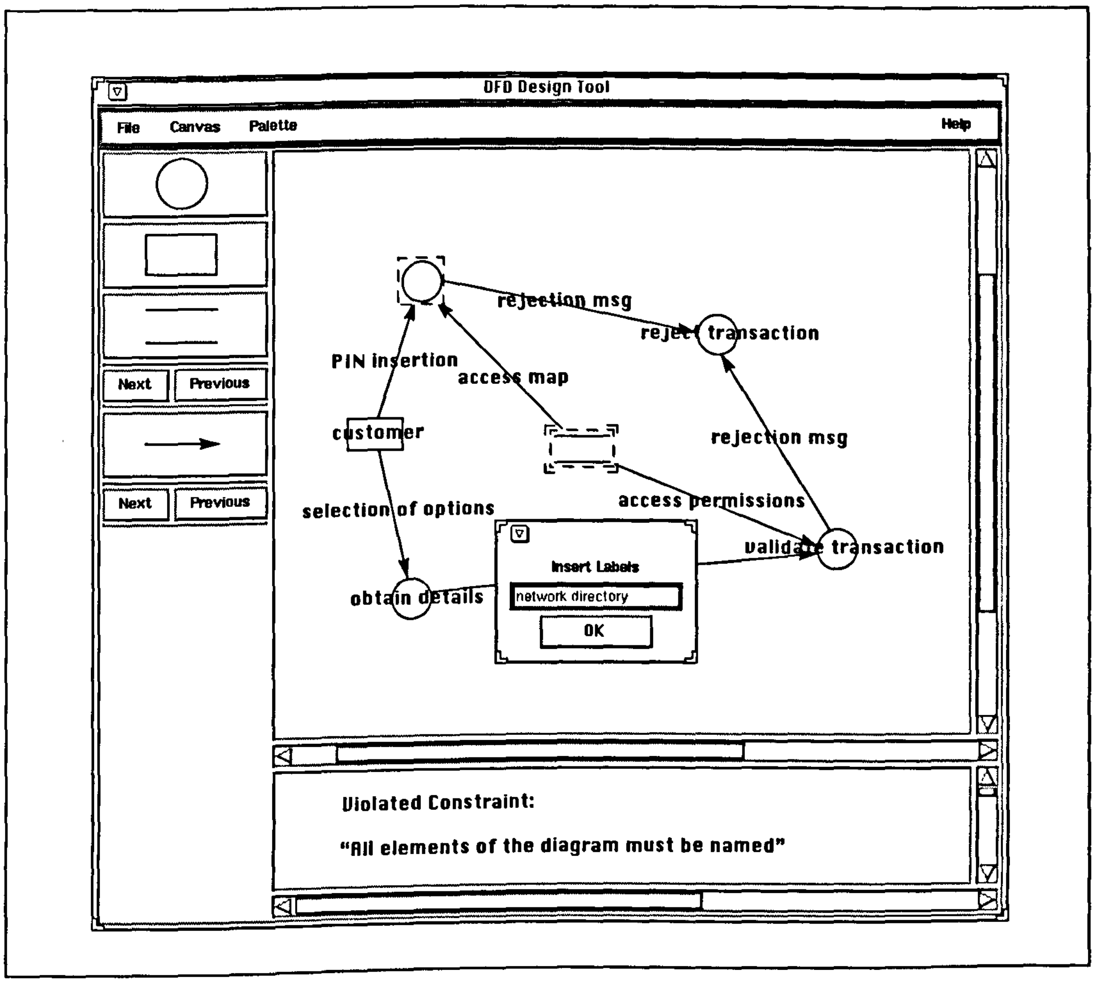
10.3 Using The Complete System for Dataflow Diagrams - An Example of a Design Session

The method of using the implemented prototype, which includes the components described above, starting from writing a VC-t specification to the automatic generation of a supporting design tool, is presented below.

We will demonstrate how to use the complete system to obtain a design tool for Dataflow Diagrams (DFD).

It is necessary to produce a VC-t specification for the DFD modelling technique, we call it ‘DFD.spec’. The obtained specification can then be compiled using the VC-t compiler. For this the following Unix command line is used:

‘vc DFD.spec’.



Violated Constraint:

"All elements of the diagram must be named"

FIGURE 37. Design Tool for Dataflow Diagrams.

A possible specification for DFDs is included in Appendix B. In this particular case the specification is already correct with respect to the VC-t syntax and semantics. But if the specification was being developed from scratch some errors would be expected to arise during the process of writing it, which could be corrected with the help of the error messages issued by the compiler. The result of the compilation phase is a Napier88 program, composed of a number of files, and a compilation script.

In the following step the output script must be compiled by the Napier88 system. This is performed writing the command:

'npc DFD.N'.

This compiles the Napier88 files and generates a file called 'DFD.out' which can be executed, to start the design tool, with the command:

'npr DFD.out'

The phases making use of the Napier system are known in advance to be successful, for the code generator only produces correct code.

In Figure 37 a screen shot of the design tool generated by the system for the DFD specification included in Appendix B is shown.

At this stage the designer can experiment with the tool. As a result s/he may decide to change the specification and generate a new design tool. This iterative development process continues until a suitable tool is produced.

10.4 Conclusion

The achieved implementation demonstrates the feasibility of the approach, namely that it is possible to automatically generate a constraint-based design tool for a specific Modelling Technique (MT) from a high level specification of that MT's semantics.

11. Conclusions

11.1 Contributions

The overall contribution of this thesis was to provide a complete path from the high-level formal specification of Modelling Techniques (MTs) to the generation of design tools. However, along this path we can identify a number of important topics of research where local contributions were made. The following text summarises these contributions.

The Visual Concepts Formalism

This new formalism has been created specifically to express MTs. We didn't find any existing formalism suitable for this task: some are able to express the semantics of MTs but not their visual representation; others, for instance those based on the Object-Oriented paradigm, are unable to capture the semantics of a MT. The Visual Concepts formalism includes mechanisms to capture the physical component of a MT, including the visual representation, the semantic component, by the use of constraints, and its usage.

The VC-t Language and Compiler

A formal language for the specification of MTs, named VC-t, has been designed based on the Visual Concepts formalism. The language is formal, and therefore unambiguous, but also simple to use and from it easily readable specifications can be obtained. Formality was employed only as a tool; the main goal was to obtain a language that could be used by someone who does not have to be knowledgeable on formal notations.

The production of a MT specification is an iterative process in which each new specification obtained can be compiled into a working prototype. The high-level specifications obtained with the language can be translated into executable code by means of a specially built compiler.

To produce the compiler we obtained a formal description of the language in BNF. The parser, the front-end of the compiler, is separated from the back-end, the code generator. This way, the parser is fully portable to other target platforms. The current implementation uses the Napier88 persistent programming system as the target platform. If a new language was to be used, e.g. Java or C++, only the back-end of the compiler would have to be replaced.

The Generic Visual Language

We propose a generic visual language that can be instantiated for a particular MT. It is based on direct manipulation of Visual Objects (VOs), which correspond to the conceptual constructs of the underlying MT. From a VC-t specification of a MT, the generic visual language is instantiated and a design tool is generated to support that MT. The instantiated visual language is then used within the design tool.

Although the visual language supports the semantics of the underlying MT, it is still flexible in that it permits the user to draw inconsistencies in the diagram being edited. By assigning a state to each VO in a diagram and to the diagram itself, it is possible to perform the continuous validation of the diagram during the editing process.

Because we are using semantic constraints both in the specification and in the visual language, it is possible to give semantic feedback to the user during the editing task. The semantic information is not lost during the automatic generation process.

Usage Specification

A first specification for a MT can be obtained with VC-t, however it does not include information regarding the way of using the MT - only a default usage is implicit in this specification. To allow the designer to specify the usage of a MT, a novel theory of semantic constraints has been developed for the refinement of a MT specification. This gives control to the designer to establish the possible levels of inconsistency of the diagrams produced by a given MT during an editing session.

The refined specification defines the possible patterns of user interaction when editing a diagram. The constraints are classified according to their enforcement as hard, hardened, soft and deferred. Using the mechanism of constraint classification on a specification the designer can set the value of the 'semantic freedom', which will be reflected in the level of flexibility of the produced design tools. This way, the design tools may be tailored to match the users' level of expertise.

Constraint Management

Diagram operations are the result of user actions which cause a transformation at the presentation level. If the underlying graph is also changed the operation is called 'graph altering', otherwise it is called 'graph preserving'. Graph altering operations cause constraints to be checked. Each VO in the diagram refers to a subset of constraints so that the constraint manager may check them by knowing which VOs were changed. A problem in constraint management concerns determining the constraints that each VO must refer to.

We undertook a formal study on this subject. The problem was substantially reduced by taking into account a number of assumptions about the diagram drawing method. It was possible to deduce a number of rules determining which VOs should refer to which constraints. Constraint management was simplified with the outcome of this study.

Automatic Generation of MT Design Tools

The VC-t specification of a MT constitutes the input for the automatic code generation process. The current prototype generates code in the persistent language Napier88. The generated code is human-readable. i.e. it can be easily understood by a Napier88 programmer. To aid the generation process two mechanisms are used: templates, structures that are filled with the generated code, and generic design time tools, which are discussed below. The whole code generation process is done in a fully automatic way.

The generation of code follows a set of translation rules from VC-t into Napier88. The rules implement generic principles of translating a VC-t specification into executable code. These principles can therefore be used to produce rules for any other programming language if the prototype is to be ported to a different implementation platform.

Generic Design Time Tools

Two generic tools have been built. Although intentionally made for this approach, they were designed to be generic, i.e. they can be used in other application domains.

The *GraphTool* is a generic tool to be used with visual and interactive applications which are based on graphs. It is used as a kernel providing the necessary capabilities. These include a standard graph library with the usual operations, a mapping mechanism to support multiple visual representations, and user interaction facilities for diagram editing.

It also allows the specification of constraints which will be checked during the process of drawing a diagram. The tool is linked to the application by a pre-defined communications protocol.

A *generic graphical User Interface (UI)* for diagram drawing is the second tool provided by our system. It can act as a front-end to the GraphTool. The tool is composed of: an interactive drawing canvas, a generic palette for Visual Objects (VOs), a messages window for textual feedback and a menu bar. The palette has an import mechanism with which a list of VOs can be loaded. The VOs can be organised into several pages. A modular approach was adopted for the design of its structure: any of its components can be used in isolation.

This tool is used in our system as a *Configurable MT Design Tool.* The palette displays the visual representations of the constructs of the MT in hand; the canvas hosts the visual language described above (which is itself supported by the GraphTool); and the message window shows messages to the user regarding violated semantic constraints. However, this tool can be used as the UI of any application based on interactive diagram editing.

Because of their genericity, these tools contribute to fill the gap existing in the area of providing generic support to applications where diagrams are interactively edited.

11.2 Limitations and Improvements

Extend the VC Formalism

The current version of the VC Formalism is expressive enough to capture most of the widely used MTs. We have obtained specifications for a number of MTs that fall into that category.

However the formalism is unable to capture the following concepts: inclusion, abstraction and specialization. The visual representations of these concepts are, respectively: component relation, i.e. a VO is represented inside another VO; iconification, when a group of VOs are reduced to a single icon; explosion, a given VO corresponds to another full diagram.

Implement MT Usage

The specification of MT usage as described in this thesis has not been implemented in the prototype. In the syntax of the VC-t specification language one new production rule regarding the classification of constraints would have to be written. The main amount of work would be done with the code generator for it would then have to generate the code to implement the various constraint classes.

Optimization

'The First Rule of Program Optimization:
Don't do it.
The Second Rule of Program Optimization (for experts only!):
Don't do it yet.'
-- Michael Jackson (quote seen on Unix's 'fortune', based on [Jackson75])

The code generated by the prototype is not optimized. We were concerned more with demonstrating the feasibility of the approach than with obtaining a high-performance prototype. However, in our tests the response times of the system during interaction with the design tools were good enough (always less than one second). There are several possible optimizations. For instance, in the current version, for every constraint, full diagram traversal is always executed. After optimization only the necessary icons/connections will be checked, not all the diagram. For example, 'in a STD transitions with the same origin and destination cannot have the same transition condition', to check this constraint only those connections which have the same origin and destination icons as the connection for which the constraint is being checked, have to be visited.

Also the constraint manager could be optimized. For instance, when checking constraints in a sequence of implied operations, i.e. when deleting an icon with several connections attached, only check the constraints after the transaction is completed (delete icon, delete connection 1, delete connection 2, ...) and not after each one of the operations.

Final Note

Within the time allowed for a PhD, we have demonstrated with the implemented prototype that it is possible to automatically generate constraint-based design tools from high level specifications of MT's semantics. It was nevertheless impossible to achieve everything we would like to. Unfortunately, the effort required to implement the full theory presented in this thesis amounts to at least an extra one man-year. We believe that the implementation already obtained is enough to demonstrate the feasibility of the ideas proposed in the thesis. It would be, however, very interesting to produce a complete working system which fully implements the approach with its whole theoretical body. This will have to be left for further work, hopefully in the near future.

11.3 Future Research Directions

11.3.1 Visualizing Semantic Constraints

Visual Specification of Semantics

Constraints are used in a VC specification to express the semantics of the underlying MT. At the moment we use a textual representation to specify the constraints: the VC-t specification language. Some kind of computational support should be provided to the designer for the interactive specification of semantic constraints.

It would be interesting to conceive a system for the visual and interactive definition of VCs semantics using visual constraint specification. For this purpose the Programming By Demonstration (PBD) paradigm [Cypher93] could be used. In such a paradigm, the designer shows the system examples of applying the semantics of the VCs in a diagram; from these examples the system is able to deduce the semantic constraints. As a result, the semantic constraints are automatically generated by the system in a logic based textual notation.

The interactive definition of spatial relationship constraints in a visual way has been done in the Rockit system [Karsenty92]. An inference mechanism has been used. However, this system does not support the kinds of constraints we are interested in, i.e. semantic constraints such as the ones exemplified by the VC-t specifications of MTs we have discussed. Still, this paper can contribute with some interesting ideas for future research on this subject.

Animated Semantics - Visualizing Constraints With Animation

When the design tools are generated, for each VC at the specification level there will be a corresponding VO at the implementation level. VOs are then used in the generated design tools as the components of the diagrams being edited. The semantics of a VC will determine the behaviour of the corresponding VO, i.e. the semantics of a VC are specified with constraints, which will then determine the possible behaviour of a VO when a diagram is being edited.

As we have mentioned before, a problem with constraint based specifications resides in the difficulty of their inspection. A possible solution to this problem is to provide the designer with a mechanism for the demonstration of VCs. Because the behaviour of a VO is determined by the semantics of its corresponding VC, it should then be possible to inspect semantics by visualizing the VO's behaviour. A technique for the animation of VOs could be used for this purpose. In a way, this technique is the inverse of the one used to specify constraints we described above. Instead of using a Programming By Demonstration based technique to specify constraints, we now use an *Inspecting By Demonstration* based technique to analyse the previously specified constraints. This time it is the system that demonstrates the constraints to the user. Both the MT designer and the final user (the user of the produced design tools) could benefit with such an animation based facility: a *VC browser* could be built to be used by the designer and *on-line help* could support the user in the design task. In fact, it would be possible to build two generic tools for this purpose which would then be automatically configured for any given MT.

11.3.2 A New Approach to Code Generation

The LEC Approach

The VC specification is submitted to a parsing process from which code is automatically generated. The generated code is divided into code blocks, designated by Linked Encapsulated Code (LEC) blocks. The blocks are made of several fragments linked to each other (this is done by a linking structure superimposed to the code). Each of these blocks is seen as an unequivocally identified object, i.e. the code is encapsulated.

The linked fragments in a LEC block may be implemented in different programming paradigms, for instance persistent programming to implement data structures to store the schemata produced with the editor and logic programming to implement the constraint base corresponding to the constraints included in the specification.

There will be a LEC block corresponding to each VC in the specification. There is a referencing mechanism between the two abstraction levels (specification-implementation) that links each VC in the specification to each of the LEC blocks and vice-versa, i.e. the specification-implementation links are bidirectional. The code generator automatically creates the links.

This approach constitutes an automatic symmetric bridging of the semantic mismatch existing between the specification and the implementation levels. When the formalisms used for the specification and for the implementation are not the same, there is a semantic mismatch between the two levels. If the structure of the specification is to be preserved across the generation process, or if changes in the specification are to be automatically propagated to the implementation code, then this semantic mismatch must be bridged. The proposed approach is a way of doing it. A reflection on a number of important goals in automatic code generation is given below.

Goals That Can Be Achieved With the LEC Approach

Using the LEC approach to automatic code generation is a way of achieving the following goals:

- The specification structure defined by the designer is preserved across the automatic code generation. The well defined specification-implementation linking maps the structure defined by the designer at the specification level to the implementation level.
- Semantic feedback can be provided to the user. Semantic feedback given to the user of a generated application must be meaningful in the particular interaction context. Meaningful in the sense that it conforms with the semantics of the application (expressed in the specification). Semantic feedback may be implemented, for instance, as a message in the screen (English text provided in the specification of a VC which will then be shown as a message in a dialogue window), but also graphically or sonically. Because semantic information is related to the VCs, to implement semantic feedback, the code generator must know which part of the automatically generated code implements each VC in the specification. This is virtually impossible if the relevant code is scattered. With the LEC approach, implementing semantic feedback is simplified: the code is structured into blocks (linked); the code corresponding to a VC is unequivocally identified, i.e. the identity of the VC is preserved (encapsulation principle).
- Program code can be provided along with the specification as a way to enhance the expressive power of the specification formalism. Some parts of the specification may be implemented by hand with code that will be directly used at the implementation level. The code generator links the automatically generated code with the provided hand written code.
- Direct and reverse automatic change propagation. Changes in the specification can be mapped down incrementally and automatically to the implementation code and, conversely, changes in the code can be mapped up to the specification. For this, the links corresponding to the changes are followed in one or the other direction in order to undertake the necessary repairs. While the automatic and incremental changes in the code as a result of changes in the specification seems feasible, the reverse direction - changes in the implementation code being retro-propagated to the specification - is probably more complex. However, in a first stage, it is not as difficult to follow the links starting from the changed LEC blocks in order to tag the corresponding specification fragments as inconsistent. The tagged fragments are treated by the code generator as provided code (as explained above).

11.3.3 Visual Concepts as Components

Visual Concepts can become small, reusable units of specification that will be used to compose specifications of MTs.

In [Jazayeri95] component programming is defined as 'a software development paradigm based strictly on the use of standard software components'. The paper presents an approach for the development of applications using components which are organised in catalogues. The components in each catalogue must support a related set of concepts. The author states that components development must be completely separated from application development.

The reuse of components seems to be more promising than the traditional Object-Oriented (OO) vision. Commercial systems supporting languages based on the traditional OO approach have not been successful in promoting reuse [Udell94]. As pointed out in the paper 'an alphabet soup of standards, including [...] COM, DSOM, CORBA, will provide the mechanisms for component exchange that pure OO Programming has failed to deliver'. One of the reasons for the failure of reuse in the OO approaches was that the techniques used for component development were the same used for application development and the two could be performed simultaneously. This contradicts what is stated by Jazayeri.

Catalogues of VCs and Reuse

In our view, components are used at the specification level; they are used to compose specifications rather than applications. A specification language is provided to the components designer. By adding extra constraints to a VC, it is possible to extend its semantics. This opens a good perspective for the reuse of VCs.

There are advantages in defining the components at the specification level rather than at the implementation level. If a simple specification formalism is used, it is typically easier to understand what a component does by looking at its specification than by looking at its implementation code (a parallel reasoning can be made with digital component catalogues and their implementation in hardware using transistors). It is also much easier to change or extend a specification than the correspondent program code.

Existing catalogues for various MTs could be easily used to produce specifications for other MTs. For instance, a catalogue of VCs produced for Entity Relationship diagrams (ER) could be used to specify the VCs for a new extension of ER.

11.3.4 Conclusion

The three research directions presented above are not isolated, and if pursued simultaneously they act in a synergetic way. For instance, the production of reusable VCs can be aided by visual techniques, and browsers based on animated semantics can be made for the inspection of catalogues of VCs. The goal is to push forward the whole research work, starting at the higher abstraction level of the specifications based on the VC formalism and ending at the generated design tools.

The future will see definite improvements in application modelling supported by advanced diagrammatic design tools. With diagram based MTs being competently supported by design tools, the necessity to use text will weaken. Text based programming languages will be preferably used in local optimization strategies. The work will progressively be done at the specification level rather than at the implementation level. The programming tasks will become lighter as conceptual modelling supported by advanced frameworks and tools, allows users to build programs at higher abstraction levels.

As the demand for conceptual modelling increases, software providers will need to effectively and efficiently design new MTs, or adapt existent ones, and produce suitable supporting tools. We have opened up and illuminated several synergic research directions, giving clear contributions for that aim.

Appendix A - Guide to the VC-t Syntax

A.1 Introduction

The syntax description given herein is meant to help understanding and writing VC-t specifications. For a complete BNF specification please refer to Appendix D.

The following conventions have been used in the syntax description:

- reserved words are in bold upper case, e.g. **EXISTS**. A number of keywords have several alternatives. A list of all keywords and available alternatives is shown in Section A.4;
- reserved characters are enclosed, e.g. "->"
- non-terminal symbols are in lower case, e.g. properties;
- terminal symbols which are not keywords have the first letter capitalised, e.g. Natural (see Section A.2);
- a non-terminal symbol that matches the empty string (empty rule) is indicated as the comment /* empty */.

A.2 Terminal Symbols Which Are Not Keywords

The following terminal symbols are used in the syntax description:

Natural	- a natural number.
String	- a sequence of letters and digits enclosed in single or double quotes.
Lowercase_identifier	- a sequence of letters and digits. The first character must be a lower case letter.
Uppercase_identifier	- a sequence of letters and digits. The first character must be an upper case letter.

A.3 Syntax Description

A.3.1 Overall Structure

vcSemanticsSpecification ::= String **SEMANTICS_SPECIFICATION** preamble
semanticConstraints "."

A.3.2 Preamble

preamble ::= **PREAMBLE** mtModel setExtractors setDefinitions setProperties

mtModel ::= **A.MT_MODEL _BEGIN** variableName "=" powerSetCartesianProduct
END

Extractors

setExtractors ::= **B.SET_EXTRACTORS_DECLARATIONS** iconPowersets
connPowersets

iconPowersets ::= **B1.ICONS _BEGIN** iconExtractorList **END**

iconExtractorList ::= extractor
| iconExtractorList extractor

connPowersets ::= **B2.CONNECTIONS _BEGIN** connExtractorList **END**

connExtractorList ::= extractor
| connExtractorList extractor

extractor ::= setName ":" variableName "->" powerSetCartesianProduct

Auxiliary Sets and Extractors

setDefinitions ::= **C.SETS_DEFINITIONS _BEGIN** setDefinitionList **END**

setDefinitionList ::= /* empty */
| setDefinition
| setDefinitionList setDefinition

setDefinition ::= setName "==" setSpecification

setSpecification ::= set
| "{" elementList "}"

elementList ::= stringList

| naturalList

Set Properties

setProperties ::= **D.SET_PROPERTIES _BEGIN** propertiesList **END**

propertiesList ::= /* empty */
| properties
| propertiesList properties

properties ::= setName **HAS_PROPERTIES** propertyList

propertyList ::= property
| propertyList property

property ::= variableName ":" setCartesianProduct "->" range

setCartesianProduct ::= setName
| setCartesianProduct "x" setName

range ::= set /* the following types are allowed: Boolean, Natural, String */

powerSetCartesianProduct ::= powerSet
| powerSetCartesianProduct "x" powerSet

powerSet ::= "P" setName

A.3.3 Semantic Constraints

semanticConstraints ::= **SEMANTIC_CONSTRAINTS _BEGIN** constraintList
END

constraintList ::= /* empty */
| constraintList constraint

constraint ::= constraintDesc quantifiedPLstatement
| constraintDesc instantiatedPLstatement

constraintDesc ::= "C" Natural ":" /* natural values must be in a sequential order */
| "C" Natural ":" String

Predicate Logic Statements

quantifiedPLstatement ::= quantificationList "(" plStatement ")"

quantificationList ::= universalQuantification

```
                  | existentialQuantification
                  | quantificationList universalQuantification
                  | quantificationList existentialQuantification

universalQuantification ::= NOT universalQuantification
                  | FORALL variableList ":" set "." variableList BELONGING set
                        IMPLIES

existentialQuantification ::= NOT existentialQuantification
                  | EXISTS variableList ":" set "." variableList BELONGING set AND

instantiatedPLstatement ::= plStatement

plStatement ::= NOT plStatement
                  | "(" plStatement ")"
                  | plStatement plConnective plStatement
                  | equality
                  | simpleBooleanExpression

plConnective ::= DOUBLEIMPLICATION
                  | IMPLIES
                  | AND
                  | OR

equality ::= set "=" set   /* An infix inequality operator is not provided. Instead, the
                        symbol 'NOT' should be used at the beginning of the expression, as
                        in 'NOT( i = j )' */
                  | otherObject "=" otherObject

simpleBooleanExpression ::= naturalExpression naturalComparison
                        naturalExpression
                  | predicate
                  | TRUE
                  | FALSE

set ::= setName
                  | setApplication
                  | set setOperation setName
                  | set setOperation setApplication

setOperation ::= INTERSECTION
                  | UNION
                  | SETDIFFERENCE

naturalComparison ::= ">"   /* note: equality is covered by the rule '"(" otherObject
                        "=" otherObject ")"' expressed above */
                  | "<"
```

```
                | ">="
                | "<="

predicate ::= PREDICATE variableName "(" setApplication ")"

setApplication ::= setName "(" ")"
                | setName "(" argList ")"

application ::= variableName "(" ")"
                | variableName "(" argList ")"

argList ::= variableList
                | setList
                | stringList
                | argList "," setList
                | argList "," variableList
                | argList "," stringList

setList ::= setName
                | setList "," setName

variableList ::= variableName
                | variableList "," variableName

otherObject ::= String
                | naturalExpression

naturalExpression ::= CARDINALITY set
                | application
                | variableName
                | Natural

stringList ::= String
                | stringList "," String

naturalList ::= Natural
                | naturalList "," naturalList

variableName ::= Lowercase_identifier

setName ::= Uppercase_identifier  /* the convention that set names begin with a
                capital letter is recommended by [Woodcock88] */
```

A.4 Keywords and Alternatives

SEMANTICS_SPECIFICATION
A.MT_MODEL
B.SET_EXTRACTORS_DECLARATIONS
B1.ICON_POWER_SETS
B2.CONNECTION_POWER_SETS
C.SETS_DEFINITIONS
D.SET_PROPERTIES
SEMANTIC_CONSTRAINTS
HAS_PROPERTIES
BEGIN
END
PREAMBLE
NOT | "~"
UNIVERSAL | FORALL
EXISTENTIAL | EXISTS
AND | "&"
OR | "V"
IMPLICATION | IMPLIES
DOUBLEIMPLICATION
PREDICATE
TRUE
FALSE
CARTESIANPRODUCT | "x"
POWERSET | "P"
DEFINEDAS | "=="
MEMBERSHIP | BELONGING | IN | "E"
UNION | "U"
INTERSECTION
SETDIFFERENCE | "\\"
MAPSTO | "->"
EQUAL | "="
GREATER | ">"
LESS | "<"
ATLEAST | GREATEROREQUAL | ">="
ATMOST | LESSOREQUAL | "<="
CARDINALITY | "#"

Valid characters with no verbose alternatives provided

"." ":" "(" ")"

Appendix B - Complete VC-t Specifications of Modelling Techniques

The modelling techniques (MTs) specified below were chosen for the following reasons:

- they are widely known;
- they model different views (static and dynamic) of the universe of discourse;
- their semantics is well defined;
- their graphical representation is based on icons and connections.

For each MT the following is shown:

- the graphical representation;
- the semantics definition in natural language;
- an example;
- the semantics specification in VC-t.

B.1 Dataflow Diagrams

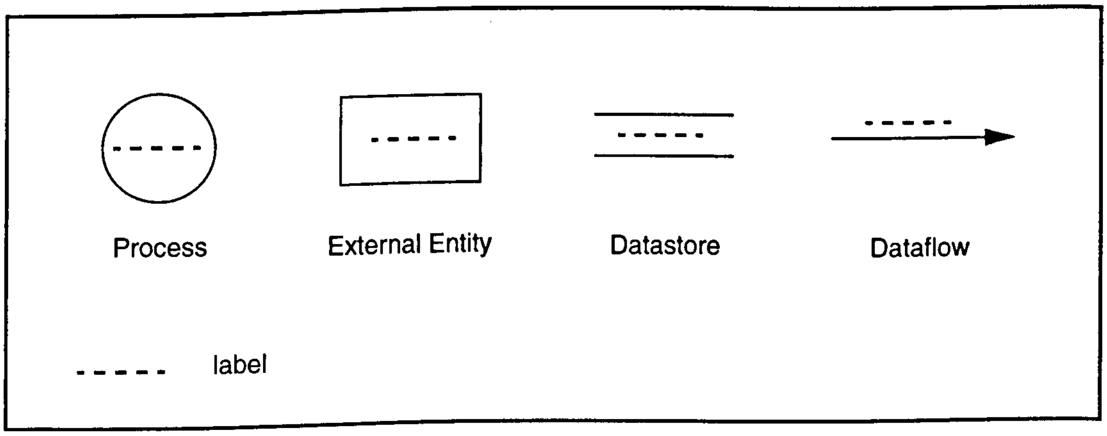


FIGURE 38. DFD graphical representation.

B.1.1 Semantics Definition in Natural Language

A dataflow diagram (DFD) includes four distinct elements: process (or transform), datastore, external entity and dataflow. The first three are icons and the last is a connection.

Each *process* has a unique name and must have at least one input dataflow and one output dataflow which are different.

Each *datastore* has a unique name and must have at least one input dataflow and one output dataflow; all dataflows connected to a datastore at one end must be connected to a process at the other.

Each *external entity* has a unique name and must have at least one dataflow, either input or output; all dataflows connected to an external entity at one end must be connected to a process at the other

A *dataflow* is named, not necessarily uniquely, and connects two different icons of the diagram, no self-loops are allowed. If two icons are connected by two or more dataflows in the same direction then these dataflows must have distinct names.

All elements on a DFD must be named; no anonymous elements are allowed.

A legal DFD must include at least one process. Every DFD must include at least one external entity which provides input to the system and at least one external entity to which output is directed; these external entities need not be distinct. A DFD must be connected; i.e. there should be no isolated icons or disconnected partitions of the diagram.

Assumptions

The following simplifications to the notation have been made:

- diagrams are assumed to be single level without any expansion of processes;
- the identifier of a process is a single string; usually it is split into a serial number and a description;
- the convention that allows duplicate copies of either external entities or datastores to be drawn for topological reasons, has not been considered.

For a complete definition see [Budgen94].

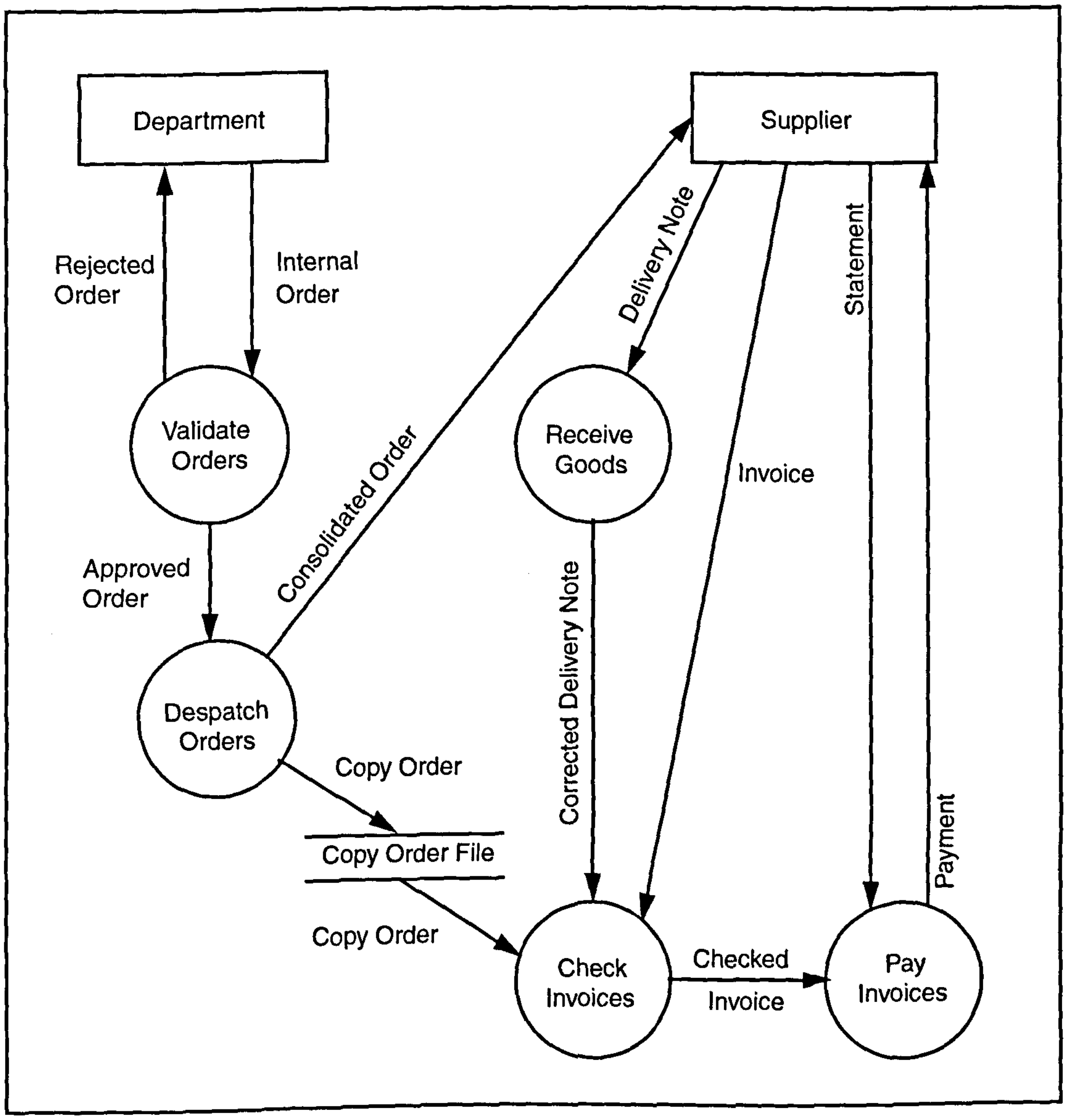


FIGURE 39. DFD example.

B.1.2 Semantics Specification in VC-t

"Dataflow Diagrams" SEMANTICS_SPECIFICATION

PREAMBLE

A.MT_MODEL

BEGIN

```
    dfd = P Process x P Datastore x P ExternalEntity x P Dataflow

    END

B.SET_EXTRACTORS_DECLARATIONS

    B1.ICONS

        BEGIN

        Processes : dfd -> P Process
        Datastores : dfd -> P Datastore
        Externals : dfd -> P ExternalEntity

        END

    B2.CONNECTIONS

        BEGIN

        Dataflows : dfd -> P Dataflow

        END

C.SETS_DEFINITIONS

    BEGIN

    DFDelement == Process U Datastore U ExternalEntity U Dataflow
    DFDelements ==
        Processes(dfd) U Datastores(dfd) U Externals(dfd) U Dataflows(dfd)
    DFDicon == Process U Datastore U ExternalEntity
    DFDicons == Processes(dfd) U Datastores(dfd) U Externals(dfd)

    END

D.SET_PROPERTIES

    BEGIN

    Process HAS_PROPERTIES

    name : Process -> String
```

```
equal : Process x Process -> Boolean

Datastore HAS_PROPERTIES

name : Datastore -> String
equal : Datastore x Datastore -> Boolean

ExternalEntity HAS_PROPERTIES

name : ExternalEntity -> String
equal : ExternalEntity x ExternalEntity -> Boolean

Dataflow HAS_PROPERTIES

name : Dataflow -> String
source : Dataflow -> DFDicon
destination : Dataflow -> DFDicon
equal : Dataflow x Dataflow -> Boolean

END
```

SEMANTIC_CONSTRAINTS

```
BEGIN

C1: “Each process has a unique name”

FORALL p1, p2 : Process • p1, p2 BELONGING Processes(dfd) IMPLIES
    ( name(p1) = name(p2) IMPLIES p1 = p2 )

(* This constraint is so common in MTs that we have defined a predicate to
    simplify it: uniqueName( extractor ). This predicate will be used from now on.
    Hence, alternatively, the following constraint specification could have been used:
    ‘PREDICATE uniqueName( Processes(dfd) )’ *)


C2: “Each process must have at least one input dataflow and one output dataflow
    which are different”

FORALL p : Process • p BELONGING Processes(dfd) IMPLIES
    EXISTS f1, f2 : Dataflow • f1, f2 BELONGING Dataflows(dfd) AND
        ( destination(f1) = p AND
        source(f2) = p AND
        NOT ( f1 = f2 )) (*different flows*)
```

C3: “Each datastore has a unique name”

```
PREDICATE uniqueName( Datastores(dfd) )
```

C4: “Each datastore must have at least one input dataflow and one output dataflow each of which must be connected to a process”

```
FORALL d : Datastore • d BELONGING Datastores(dfd) IMPLIES
    EXISTS f1, f2 : Dataflow • f1, f2 BELONGING Dataflows(dfd) AND
        EXISTS p1, p2 : Process • p1, p2 BELONGING Processes(dfd) AND
        (
            ( destination(f1) = d AND source(f1) = p1 ) AND
            ( source(f2) = d AND destination(f2) = p2 )
        )
```

C5: “Each external entity has a unique name”

```
PREDICATE uniqueName( ExternalEntities(dfd) )
```

C6: “Each external entity must have at least one dataflow, either input or output, which is connected to a process”

```
FORALL e : ExternalEntity • e BELONGING Externals(dfd) IMPLIES
    EXISTS f : Dataflow • f BELONGING Dataflows(dfd) AND
        EXISTS p : Process • p BELONGING Processes(dfd) AND
        (
            ( destination(f) = e AND source(f) = p ) OR
            ( source(f) = e AND destination(f) = p )
        )
```

C7: “Each dataflow must connect two different icons (no self-loops)”

```
FORALL f : Dataflow • f BELONGING Dataflows(dfd) IMPLIES
    ( NOT ( source(f) = destination(f) ))
```

(* the 3 following constraints determine the minimum dfd *)

C8: “There must be at least one external entity which provides input to the system”

```
EXISTS e : ExternalEntity • e BELONGING Externals(dfd) AND
    EXISTS f : Dataflow • f BELONGING Dataflows(dfd) AND
        ( source(f) = e )
```

C9: “There must be at least one external entity which takes output from the system”

```
EXISTS e : ExternalEntity • e BELONGING Externals(dfd) AND
    EXISTS f : Dataflow • f BELONGING Dataflows(dfd) AND
        ( destination(f) = e )
```

C10: “There must be at least one process”

```
CARDINALITY Processes(dfd) >= 1
```

C11: “All icons in a DFD must be connected”

```
EXISTS p : Process • p BELONGING Processes(dfd) AND
    ( Connect(p) = DFDicons(dfd) )
```

(* ‘Connect’ is a pre-defined function that given a DFDicon returns a set of DFDicons that are connected between them and which include that DFDicon. The algorithmic definition of this function is given in B.4. *)

C12: “All elements of the diagram must be named”

```
FORALL a : DFDelement •
    a BELONGING DFDelements(dfd) IMPLIES
        ( NOT ( name(a) = "" ))
```

C13: “The same pair of icons in a DFD cannot be connected by two or more dataflows with the same direction and the same name”

```
FORALL f1, f2: Dataflow • f1, f2 BELONGING Dataflows(dfd) IMPLIES
    (( source(f1) = source(f2) AND
    destination(f1) = destination(f2) AND
    name(f1) = name(f2) ) IMPLIES
        f1 = f2 )

END
```

B.2 State Transition Diagrams

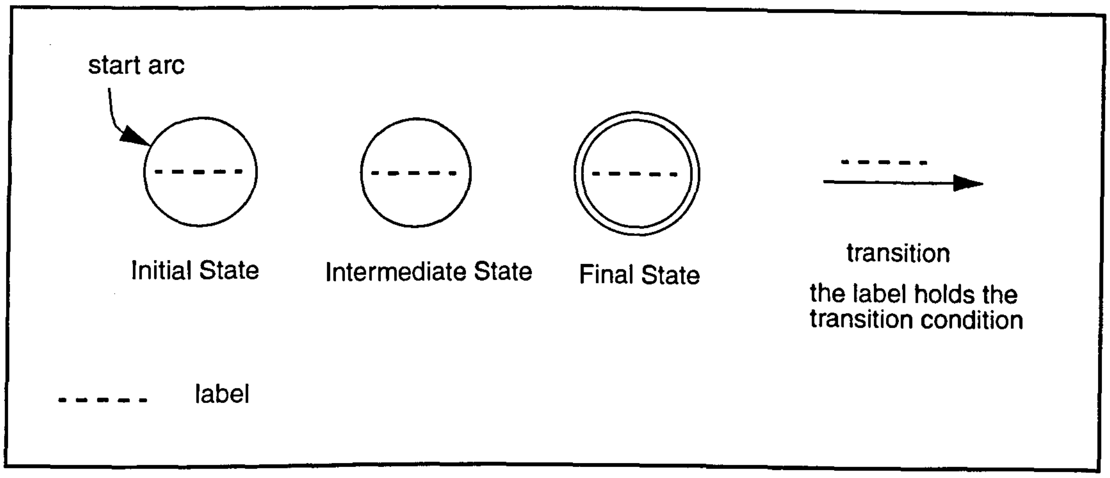


FIGURE 40. STD graphical representation.

B.2.1 Semantics Definition in Natural Language

A *state* represents an externally observable mode of behaviour. Every state is identified by a name that describes its behaviour. Names must be unique.

The *initial state* is the state entered by the system when it is started. Any STD must have exactly one initial state. The initial state must have at least one out-transition (out-transition defined below).

The *final state* is a state that the system cannot leave. Therefore, a final state cannot have out-transitions or loop-transitions (loop-transition defined below). A final state must have at least one in-transition (in-transition defined below). Any STD must have at least one final state.

The *intermediate state* is any state which is neither an initial or a final state. An intermediate state must have at least one in-transition and one out-transition.

A *transition* is a legal change of state. A *transition condition* is associated with every transition; when it evaluates to true the transition occurs. A transition always connects two states together. Any number of transitions can exist between any pair of states. But only one transition with the same transition condition and same direction can exist between a given pair of states. Transitions attached to a state are classified into three types: in-transition, which has its destination attached to the state; out-transition, which has its origin attached to the state; loop-transition, which has both its destination and origin attached to the state.

All states must be reachable from the initial state. This means that starting at the initial state and following transitions in the correct direction it must be possible to reach any other state. Note that we assume that any state is reachable from itself; this will be important when writing the specification of the constraint.

Note: the *start arc* shown in Figure 40 is part of the initial state symbol and has no semantics of its own.

Figure 41 depicts a State Transition Diagram for a fragment of the text editor 'vi' of the Unix operating system.

For additional information see [Budgen94].

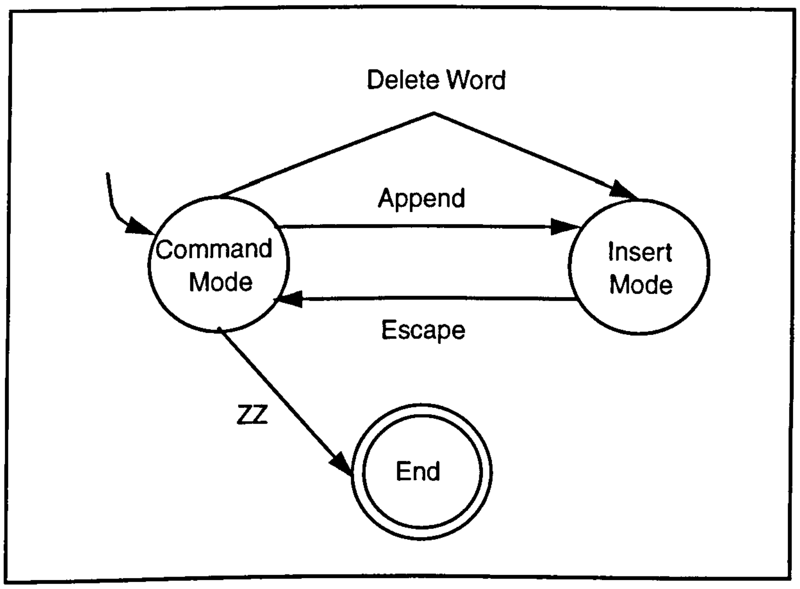


FIGURE 41. STD of a fragment of the 'vi' editor.

B.2.2 Semantics Specification in VC-t

"State Transition Diagrams" SEMANTICS_SPECIFICATION

PREAMBLE

A.MT_MODEL

BEGIN

std = P InitialState x P InterState x P FinalState x P Transition

```
    END

  B.SET_EXTRACTORS_DECLARATIONS

    B1.ICONS

      BEGIN

      InitialStates : std -> P InitialState
      InterStates : std -> P InterState
      FinalStates : std -> P FinalState

      END

    B2.CONNECTIONS

      BEGIN

      Transitions : std -> P Transition

      END

  C.SETS_DEFINITIONS

    BEGIN

    AnyState == InitialState U InterState U FinalState
    AnyStates == InitialStates(std) U InterStates(std) U FinalStates(std)

    END

  D.SET_PROPERTIES

    BEGIN

    InitialState HAS_PROPERTIES

    name : InitialState -> String
    equal : InitialState x InitialState -> Boolean

    InterState HAS_PROPERTIES

    name : InterState -> String
    equal : InterState x InterState -> Boolean
```

```
FinalState HAS_PROPERTIES

name : FinalState -> String
equal : FinalState x FinalState -> Boolean

Transition HAS_PROPERTIES

transitionCondition : Transition -> String
origin : Transition -> AnyState
destination : Transition -> AnyState
equal : Transition x Transition -> Boolean

END
```

SEMANTIC_CONSTRAINTS

BEGIN

C1: "Names must be unique amongst the initial state, intermediate states and final states"

```
PREDICATE uniqueName( AnyStates(std) )
```

C2: "A STD must have at least one initial state"

```
EXISTS i : InitialState • i BELONGING InitialStates(std) AND TRUE
```

C3: "A STD has at most one initial state"

```
CARDINALITY InitialStates(std) <= 1
```

C4: "The initial state must have at least one out-transition"

```
FORALL i : InitialState • i BELONGING InitialStates(std) IMPLIES
    EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( origin(t) = i AND
        NOT ( destination(t) = i ))
```

C5: "A final state cannot have out-transitions or loop-transitions"

```
FORALL f : FinalState • f BELONGING FinalStates(std) IMPLIES
    NOT EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( origin(t) = f )
```

C6: "A final state must have at least one in-transition"

(* note that a loop-transition does not count as an in-transition *)

```
FORALL f : FinalState • f BELONGING FinalStates(std) IMPLIES
    EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( destination(t) = f AND
        NOT ( origin(t) = f ))
```

C7: "A STD must have at least one final state"

```
EXISTS f : FinalState • f BELONGING FinalStates(std) AND TRUE
```

C8: "An intermediate state must have at least one in-transition"

```
FORALL s : InterState • s BELONGING InterStates(std) IMPLIES
    EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( destination(t) = s AND
        NOT ( origin(t) = s ))
```

C9: "An intermediate state must have at least one out-transition"

```
FORALL s : InterState • s BELONGING InterStates(std) IMPLIES
    EXISTS t : Transition • t BELONGING Transitions(std) AND
        ( origin(t) = s AND
        NOT ( destination(t) = s ))
```

C10: "The same pair of states cannot be connected by transitions with
the same direction and the same transition condition"

```
FORALL t1, t2 : Transition • t1, t2 BELONGING Transitions(std) IMPLIES
    (( origin(t1) = origin(t2) AND
    destination(t1) = destination(t2) AND
    transitionCondition(t1) = transitionCondition(t2) ) IMPLIES
        t1 = t2 )
```

C11: "Every state must have a name"

FORALL s : AnyState • s BELONGING AnyStates(std) IMPLIES
(NOT (name(s) = ""))

C12: "Every transition must have a transition condition"

FORALL t : Transition • t BELONGING Transitions(std) IMPLIES
(NOT (transitionCondition(t) = ""))

C13: "Every state in a STD must be reachable from the initial state"

FORALL i: InitialState • i BELONGING InitialStates(std) IMPLIES
(Reach(i) = AnyStates(std))

(* 'Reach' is a pre-defined function that given a state returns all the states reachable from it, including the given state. The algorithmic definition of this function is given in the next section *)

END

B.3 Entity-Relationship (ER) Diagrams

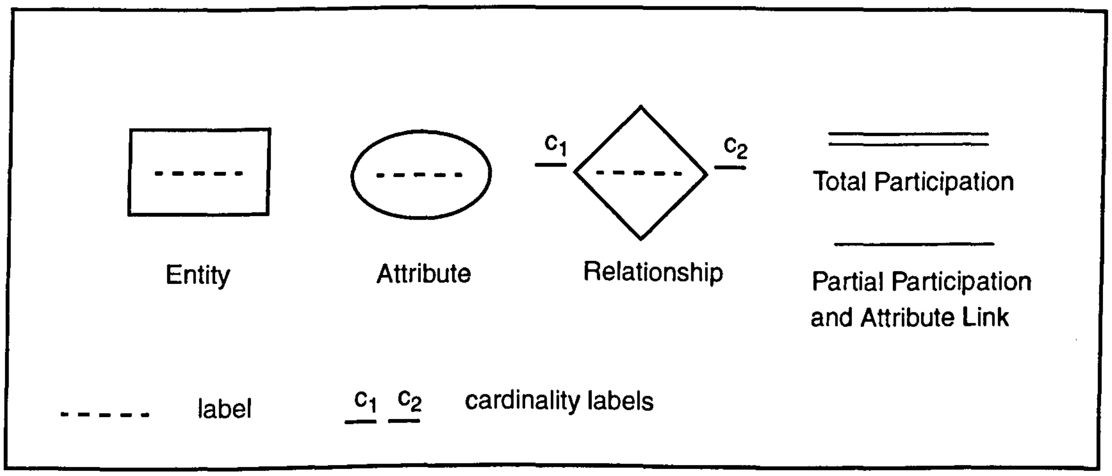


FIGURE 42. ER graphical representation.

B.3.1 Semantics Definition in Natural Language

The definition given below is of a simplified version of the ER modelling technique. For instance, multi-valued attributes and weak entities were not considered. The definition could be extended to cover the whole ER. The complete version can be found in [Chen76] or [Sanders95].

There are three icon types: entity, relationship and attribute. There are three connection types: total participation, partial participation and attribute link. The first two connect entities to relationships, while an attribute link connects an attribute to an entity, to a relationship, or to another attribute (allowing compound attribute structures, see 'Name' of 'Employee' in Figure 43).

Entities must be uniquely named.

Relationships must be uniquely named. A relationship must be annotated with its cardinality. We will assume a *binary* ER model, therefore the cardinality of a relationship is represented by a pair of values. Allowed cardinalities are: one to one, one to many and many to many. Again, because we are assuming a binary model, a relationship has exactly two links to entities.

Attributes are not necessarily uniquely named but within the context of a single entity or relationship the attribute names must be unique.

All icons must be named; no anonymous icons are allowed.

Total participation connects an entity to a relationship.

Partial participation connects an entity to a relationship.

If the two participation connections (total or partial) of a relationship are joined to the same entity, then these connections must be given role names to distinguish them.

An *attribute link* connects an attribute to any other icon type. If an attribute is refined into other (sub-)attributes then there ought to be at least two sub-attributes.

The whole ER diagram should be connected; there can be no isolated icons or distinct partitions of the diagram.

The minimal ER diagram is consists of one entity with one attribute.

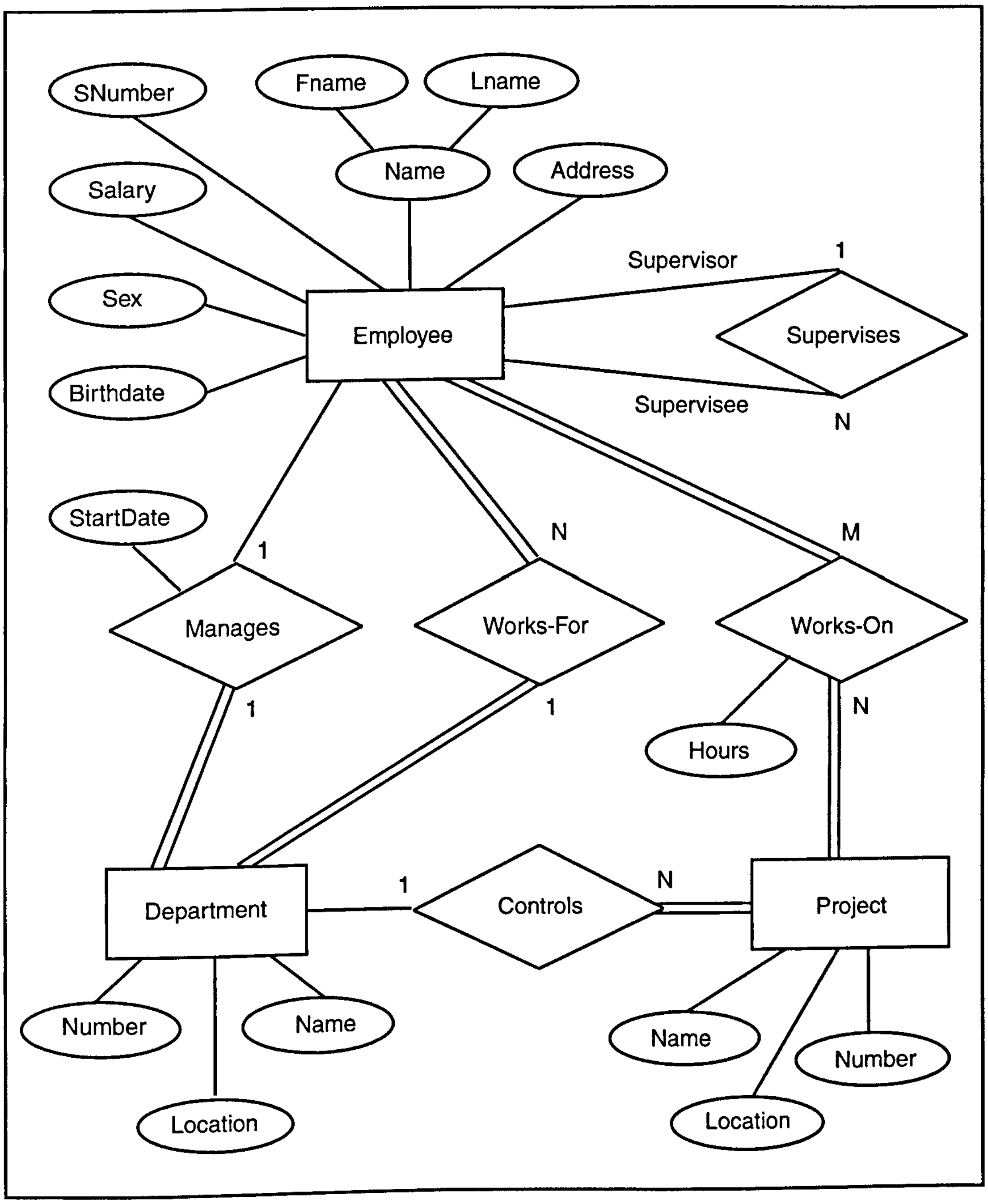


FIGURE 43. ER example.

B.3.2 Semantics Specification in VC-t

"Entity-Relationship Diagrams" SEMANTICS_SPECIFICATION

```
PREAMBLE

A.MT_MODEL

   BEGIN

   erd = P Entity x P Relationship x P Attribute x P TotalPar x P PartialPar x P AttribLink

   END


B.SET_EXTRACTORS_DECLARATIONS

  B1.ICONS

      BEGIN

      Entities : erd -> P Entity
      Relationships : erd -> P Relationship
      Attributes : erd -> P Attribute

      END

  B2.CONNECTIONS

      BEGIN

      TotalPars : erd -> P TotalPar
      PartialPars : erd -> P PartialPar
      AttribLinks : erd -> P AttribLink

      END


C.SETS_DEFINITIONS

  BEGIN

  Cardinality == {"1, 1", "1,n", "n, m"}
  ERicon == Entity U Relationship U Attribute
  ERicons == Entities(erd) U Relationships(erd) U Attributes(erd)
  Part == TotalPar U PartialPar
  Parts == TotalPars(erd) U PartialPars(erd)

  END
```

```
D.SET_PROPERTIES

    BEGIN

    Entity HAS_PROPERTIES

    name : Entity -> String
    equal : Entity x Entity -> Boolean

    Relationship HAS_PROPERTIES

    name : Relationship -> String
    cardinality : Relationship -> Cardinality
    equal : Relationship x Relationship -> Boolean

    Attribute HAS_PROPERTIES

    name : Attribute -> String
    equal : Attribute x Attribute -> Boolean

    TotalPars HAS_PROPERTIES

    roleName : TotalPars -> String
    oneEnd : TotalPars -> Entity
    otherEnd : TotalPars -> Relationship
    equal : TotalPars x TotalPars -> Boolean

    PartialPars HAS_PROPERTIES

    roleName : PartialPars -> String
    oneEnd : PartialPars -> Entity
    otherEnd : PartialPars -> Relationship
    equal : PartialPars x PartialPars -> Boolean

    AttribLink HAS_PROPERTIES

    oneEnd : AttribLink -> Attribute
    otherEnd : AttribLink -> ERicon
    equal : AttribLink x AttribLink -> Boolean

    END

SEMANTIC_CONSTRAINTS

    BEGIN
```

C1: "All icons in an ER diagram must be named"

```
FORALL i : ERicon • i BELONGING ERicons(erd) IMPLIES
    ( NOT ( name(i) = "" )
```

C2: "Each entity has a unique name"

```
PREDICATE uniqueName( Entities(erd) )
```

C3: "Each relationship has a unique name"

```
PREDICATE uniqueName( Relationships(erd) )
```

C4: "All attibute links to the same icon must be uniquely named"

```
FORALL a1, a2 : Attribute • a1, a2 BELONGING Attributes(erd) IMPLIES
    NOT EXISTS al1, al2 : AttribLink • al1, al2 BELONGING AttribLinks(erd) AND
            ( oneEnd(al1) = a1 AND
            oneEnd(al2) = a2 AND
            otherEnd(al1) = otherEnd(al2) AND
            name(a1) = name(a2) )
```

C5: "If the two participation connections of a relationship are joined to the same entity, then these connections must be given role names"

```
FORALL p1, p2 : Part • p1, p2 BELONGING Parts(erd) IMPLIES
    (( oneEnd(p1) = oneEnd(p2) AND
    otherEnd(p1) = otherEnd(p2) ) IMPLIES
    ( NOT ( roleName(p1) = "" ) AND NOT ( roleName(p2) = "" )))
```

C6: "A relationship can only have two participation connections (total or partial)"

```
FORALL p1, p2, p3 : Part • p1, p2, p3 BELONGING Parts(erd) IMPLIES
    (( otherEnd(p1) = otherEnd(p2) AND
    otherEnd(p1) = otherEnd(p3) AND
    NOT ( p1 = p2 )) IMPLIES
    p2 = p3 )
```

C7: "All icons in an ER diagram must be connected"

```
EXISTS e : Entity • e BELONGING Entities(erd) AND
    ( Connect(e) = ERicons(erd) )
```

(* ‘Connect’ is a pre-defined function that given a ERicon returns a set of ERicons that are connected between them and which include that ERicon. The algorithmic definition of this function is given in B.4. *)

C8: “The minimal ER diagram consists of one entity with one attribute”

CARDINALITY Entities(erd) >= 1 AND
CARDINALITY Attributes(erd) >= 1

END

B.4 Algorithmic Function Definitions

B.4.1 Function ‘Connect’ defined for DFD

```
Connect: DFDicon ⇒ ℙ DFDicon
────────────────────────────
Connect($i_0$) ==
    {$i_0$} $\cup$ {Connect(i) | $\exists$ f: Dataflow • f $\in$ Dataflows(dfd) $\wedge$
        (( source(f) = $i_0$ $\wedge$ destination(f) = i ) $\vee$
        ( source(f) = i $\wedge$ destination(f) = $i_0$ )) }
```

B.4.2 Function ‘Reach’ defined for STD

```
Reach: AnyState ⇒ ℙ AnyState
────────────────────────────
Reach($s_0$) ==
    {$s_0$} $\cup$ {Reach(s) | $\exists$ t: Transition • t $\in$ Transitions(std) $\wedge$
        ( origin(t) = $s_0$ $\wedge$ destination(t) = s ) }
```

B.4.3 Function 'Connect' defined for ER Diagrams

$$\text{Connect: ERicon} \Rightarrow \mathbb{P}\ \text{ERicon}$$

$$\begin{aligned}
&\text{Connect}(i_0) == \\
&\quad \{i_0\} \cup \{\text{Connect}(i) \mid \exists\, p\text{: Part} \bullet p \in \text{Parts(erd)} \wedge \\
&\qquad ((\text{oneEnd}(p) = i_0 \wedge \text{otherEnd}(p) = i) \vee \\
&\qquad (\text{oneEnd}(p) = i \wedge \text{destination}(p) = i_0))\}
\end{aligned}$$

B.4.4 Notes

Algorithmic function definitions are not included in the semantic specification. The VC-t specification language is not intended to express those definitions. They must be implemented manually and stored in a repository. The code generator will then use the implemented code.

A library of generic functions could be built to avoid having to write the code. For that purpose, the functions above can easily be made parametric (notice the similarity between the functions given in Sections B.4.1 and B.4.3) so that they may be used with any modelling technique.

Appendix C - Types For The Representation Level - A Generic Lexicon For Interactive Graph Based Systems

Although the data structures below have been defined for the work described on this thesis, we believe that they constitute a generic lexicon that can be used in other interactive graph based systems.

```
!---------- Types for Graphical Objects Definitions (Shape and LineStyle) ---------
!---------- Type LabelSlots ----------

type LabelSlots is structure(
  labelID: int; ! to identify the label in the
  numSlots: int; ! how many labels does the shape/lineStyle have
  posSlots: *Pos ! where are the labels positioned
  )

!---------- Type Shape ----------

type Shape is structure(
  sName: string; ! a name for the shape. E.g. "rectangle"

  ! graphical definition
  sImage: image;
  labelSlots: LabelSlots;

  ! operations
  draw: proc(Pos);
  delete: proc();
  move: proc(Pos);
  insertLabels: proc(int, *string); ! the first argument is the label ID
  deleteLabels: proc(int, *string)
  )

!---------- Type LineStyle ----------

type LineStyle is structure(
  lName: string; ! a name for the line style. E.g. "dashedLine"
  labelSlots: LabelSlots;

  ! operations
  draw: proc(Pos, Pos); ! a procedure to draw a line
  delete: proc();
  move: proc(Pos, Pos);
  insertLabels: proc(int, *string); ! the first argument is the label ID
  deleteLabels: proc(int, *string)
  )

!---------- Types for Visual Objects Definitions (Icon and Connection) ----------
!---------- Type Semantics ----------

type Semantics is Constraint_L  ! Constraint_L is a list of constraints (rules)

!---------- Type Icon_ ----------

type Icon_ is structure(
  iName: string;  ! a name for the icon (given when it is stored in the icon
            ! repository). E.g. "ERentity"
  iPhysical: Shape;
  iLogical: Semantics
  )

!---------- Type Connection ----------

type Connection is structure(
  cName: string;  ! a name for the connection (given when it is stored in the
            ! connection repository). E.g. "OOAgenSpec"
  cPhysical: LineStyle;
  cLogical: Semantics
  )

!---------- Types for Visual Objects (VOicon and VOconnection) --------------
!---------- Type HotArea for VOconnection --------------

type Circular is structure(
  button: ButtonPack;  ! reference to Win button - active area (Win is the Napier88 UIMS)
  position: Pos
  )

type AllLength is structure(
  startPos: Pos;
  endPos: Pos;
  treshold: int
  )

type HotArea is variant(
  circular: Circular;
  allLength: AllLength
  )

!---------- Type VOicon  ----------

type LabelsInfo is structure(  ! label values are provided by the application.
  numLabels: int;
  labels: *string
  )

type VOicon is structure(
  viName: string  ! a name for the VO. A VO is an object to be displayed
            ! on the computer screen. E.g. "Woman"
  viIcon: Icon_;
  viLabels: LabelsInfo;  ! If there is only one label, it may be the same as
              ! the viName. E.g. "Woman". Labels are displayed by the shape
              ! associated with the viIcon.

  ! Screen representation
  viButton: ButtonPack  ! a Win button
  )

!---------- Type VOconnection ----------

type VOconnection is structure(
  vcName: string;  ! a name for the VO. A VO is an object to be displayed
              ! on the computer screen.
  vcConnection: Connection;
  vcLabels: LabelsInfo; ! If there is only one label, it may be the same as
              !  the vcName. As icons, also connections can have more than
              ! one label, e.g. the cardinality in ER: ["1", "m" ]. Labels are displayed by the
              ! line style associated with the vcConnection.

  ! Screen representation
  vcHotArea: HotArea;  ! selectable (or active) area on the connection
  vcOrigin: Pos; ! the position of the VOicon origin
  vcDest: Pos    ! the position of the VOicon destination
  )
```

Appendix D - The Compiler's Front-End: a Parser for the VC-t Specification Language

D.1 Lexical Specification

```
%{

#ifdef DEBUG

main()
{
  char *p;

  while( p = (char*)yylex())
    printf( "%-15.14s is \"%s\"\n", p, yytext);
}

#define token(x) (int)"x"

#else

#include "vcStruct.h"
#include "valueStackUnion.h"
#include "y.tab.h"

#define token(x) (int)x

extern YYSTYPE yylval;

#endif

%}
%A 4000
%o 6000
%p 3000

SEMSPE                  "SEMANTICS_SPECIFICATION"
MTMOD                   "A.MT_MODEL"
EXTDEC                  "B.SET_EXTRACTORS_DECLARATIONS"
ICONPOWERSETS           "B1.ICON_POWER_SETS"
CONNPOWERSETS           "B2.CONNECTION_POWER_SETS"
```

```
SETDEF                  "C.SETS_DEFINITIONS"
SETPRO                  "D.SET_PROPERTIES"
SEMCON                  "SEMANTIC_CONSTRAINTS"
HASPRO                  "HAS_PROPERTIES"
CONSTRAINTNUM           "C"{NATURAL}":"
COMMENT                 ("(*""("*([^*)]|[^*]")"|"*"[^)])*"*"*"*)")
UPPERCASELETTER         [A-Z]
LOWERCASELETTER         [a-z]
DIGIT                   [0-9]
NATURAL                 [1-9]{DIGIT}*
LOWERCASEIDENT          {LOWERCASELETTER}[A-Za-z0-9]*
UPPERCASEIDENT          {UPPERCASELETTER}[A-Za-z0-9]*
  /* strings cannot contain newlines - also in C and Modula-2 */
STRING                  (\'[^"\n]*\')|(\"[^"\n]*\")
SPACE                   [ \t\n]+
OTHERWISE               .

%%
BEGIN                   return token(_BEGIN);
END                     return token(END);
PREAMBLE                return token(PREAMBLE);
NOT                     |
"~"                     return token(NOT);
UNIVERSAL               |
FORALL                  return token(UNIVERSAL);
EXISTENTIAL             |
EXISTS                  return token(EXISTENTIAL);
AND                     |
"&"                     return token(AND);
OR                      |
"V"                     return token(OR);
IMPLICATION             |
IMPLIES                 return token(IMPLICATION);
DOUBLEIMPLICATION       return token(DOUBLEIMPLICATION);
PREDICATE               return token(PREDICATE);
TRUE                    return token(_TRUE);
FALSE                   return token(_FALSE);
CARTESIANPRODUCT        |
"x"                     return token(CARTESIANPRODUCT);
POWERSET                |
"P"                     return token(POWERSET);
DEFINEDAS               |
"=="                    return token(DEFINEDAS);
MEMBERSHIP              |
BELONGING               |
IN                      |
"E"                     return token(MEMBERSHIP);
```

```
UNION                  |
"U"                    return token(UNION);
INTERSECTION           return token(INTERSECTION);
SETDIFFERENCE          |
"\\"                   return token(SETDIFFERENCE);
MAPSTO                 |
"->"                   return token(MAPSTO);
EQUAL                  |
"="                    return token(EQUAL);
GREATER                |
">"                    return token(GREATER);
LESS                   |
"<"                    return token(LESS);
ATLEAST                |
GREATEROREQUAL         |
">="                   return token(ATLEAST);
ATMOST                 |
LESSOREQUAL            |
"<="                   return token(ATMOST);
CARDINALITY            |
"#"                    return token(CARDINALITY);
{SEMSPE}               return token(SEMSPE);
{MTMOD}                return token(MTMOD);
{EXTDEC}               return token(EXTDEC);
{ICONPOWERSETS}        return token(ICONPOWERSETS);
{CONNPOWERSETS}        return token(CONNPOWERSETS);
{SETDEF}               return token(SETDEF);
{SETPRO}               return token(SETPRO);
{SEMCON}               return token(SEMCON);
{HASPRO}               return token(HASPRO);
{NATURAL}              return token(NATURAL);
{CONSTRAINTNUM}        return token(CONSTRAINTNUM);
{LOWERCASEIDENT}       return token(LOWERCASEIDENT);
{UPPERCASEIDENT}       return token(UPPERCASEIDENT);
{STRING}               return token(STRING);

{COMMENT}              ;
{SPACE}                ;
{OTHERWISE}            return yytext[0];

%%

/*
  Valid characters not used in the grammar (the verbose alternatives
  provided are used in the grammar specification - file vcSemantics.y)

  "~" "&" "V" "x" "P" "==" "E" "U" "\" "#" "=" "<" ">"
```

```
   Valid characters used in the grammar (no verbose alternatives provided)

   "." ":" "(" ")"
 */
```

D.2 Syntax Specification

```
/*
 * syntactic analyser for the vcSemantics formal language
 */

%{

/*
 *  EXTERNAL DECLARATIONS
 */

#include <stdio.h>
#include <string.h>
#include "userTypes.h"
#include "vcStruct.h"
#include "valueStackUnion.h"

extern BUCKET *make_bucket();


/* global variables */

  P_DOC *last_p_doc = NULL;
  P_STACK *last_p_stk = NULL;
  P_STACK *current_p_stk = NULL;  /* so' e' usado qd e'
                               diferente de last_p_stk */
  int *neigb_array = NULL;
  boolean IN_LEVEL_0 = TRUE;

/*
 *  END EXTERNAL DECLARATIONS
 */

%}

/*********************************************************************
```

```
 * terminal symbols (tokens)
 ***************************************************************************/

 /*
  * section labels (reserved words)
  */

 %token VCSPEC PREAMBLE MTMOD EXTDEC ICONPOWERSETS
        CONNPOWERSETS SETDEF SETPRO SEMCON _BEGIN END

 /*
  * negation
  */

 %token NOT

 /*
  * quantification
  */

 %token UNIVERSAL EXISTENTIAL

 /*
  * predicate logic
  */

 %token AND OR IMPLICATION DOUBLEIMPLICATION PREDICATE _TRUE
 _FALSE

 /*
  * sets
  */

 %token CARTESIANPRODUCT POWERSET DEFINEDAS MEMBERSHIP UNION
 INTERSECTION SETDIFFERENCE

 %token HASPRO

 /*
  * functions
  */

 %token MAPSTO

 /*
  * equality
  */
```

```
%token EQUAL

/*
 * natural expressions
 */

%token GREATER LESS ATLEAST ATMOST CARDINALITY
%token <vcString> NATURAL

/*
 * identifiers
 */

%token <vcString> LOWERCASEIDENT
%token <vcString> UPPERCASEIDENT

/*
 * other tokens
 */

%token <vcString> CONSTRAINTNUM
%token COMMENT SPACE
%token <vcString> STRING


/*********************************************************************
* nonterminal symbol values
*********************************************************************/

%type <bkt> constraintDesc
%type <bkt> setName
%type <bkt> constraintnum
%type <bkt> string
%type <bkt> natural
%type <bkt> variableName


%start vcSemanticsSpecification

%%

/*
 *  productions of the grammar
 */

vcSemanticsSpecification
```

```
    : string SEMSPE PREAMBLE mtModel setExtractors setDefinitions
        setProperties semanticConstraints "."
    ;

mtModel
    : MTMOD _BEGIN variableName EQUAL powerSetCartesianProduct END
    ;

setExtractors
    : EXTDEC ICONPOWERSETS _BEGIN iconExtractorList END
        CONNPOWERSETS _BEGIN connExtractorList END
    ;

iconExtractorList
    : extractor
    | iconExtractorList extractor
    ;

connExtractorList
    : extractor
    | connExtractorList extractor
    ;

extractor
    : setName ":" variableName MAPSTO powerSetCartesianProduct
    ;

setDefinitions
    : SETDEF _BEGIN setDefinitionList END
    ;

setDefinitionList
    :
    | setDefinition
    | setDefinitionList setDefinition
    ;

setDefinition
    : setName DEFINEDAS setSpecification
    ;

setSpecification
    : set
    | "{" elementList "}"
    ;

elementList
```

```
        : stringList
        | naturalList
        ;

  setProperties
        : SETPRO _BEGIN propertiesList END
        ;

  propertiesList
        :
        | properties
        | propertiesList properties
        ;

  properties
        : setName HASPRO propertyList
        ;

  propertyList
        : property
        | propertyList property
        ;

  property
        : variableName ":" setCartesianProduct MAPSTO range
        ;

  setCartesianProduct
        : setName
        | setCartesianProduct CARTESIANPRODUCT setName
        ;

  range
        : set /* the folowing types are allowed: Boolean, Natural, String */
        ;

  powerSetCartesianProduct
        : powerSet
        | powerSetCartesianProduct CARTESIANPRODUCT powerSet
        ;

  powerSet
        : POWERSET setName
        ;

  semanticConstraints
        : SEMCON _BEGIN constraintList END
```

```
    ;

constraintList
    :
    | constraintList constraint
    | constraintList error
    ;

constraint
    : constraintDesc quantifiedPLstatement
    | constraintDesc instantiatedPLstatement
    ;

constraintDesc
    : constraintnum
    | constraintnum string
    ;

quantifiedPLstatement
    : quantificationList "(" plStatement ")"
    ;

quantificationList
    : universalQuantification
    | existentialQuantification
    | quantificationList universalQuantification
    | quantificationList existentialQuantification
    ;

universalQuantification
    : NOT universalQuantification
    | UNIVERSAL variableList ":" set "." variableList MEMBERSHIP
      set IMPLICATION
    ;

existentialQuantification
    : NOT existentialQuantification
    | EXISTENTIAL variableList ":" set "." variableList MEMBERSHIP
      set AND
    ;

instantiatedPLstatement
    : plStatement
    ;

plStatement
    : NOT plStatement
    | "(" plStatement ")"
    | plStatement plConnective plStatement
    | equality
    | simpleBooleanExpression
    ;

plConnective
    : DOUBLEIMPLICATION
    | IMPLICATION
    | AND
    | OR
    ;

equality
    : set EQUAL set
    | otherObject EQUAL otherObject
    ;

simpleBooleanExpression
    : naturalExpression naturalComparison naturalExpression
    | predicate
    | _TRUE
    | _FALSE
    ;

set
    : setName
    | setApplication
    | set setOperation setName
    | set setOperation setApplication
    ;

setOperation
    : INTERSECTION
    | UNION
    | SETDIFFERENCE
    ;

naturalComparison
    /* note: equality is covered by the rule [ "(" otherObject EQUAL
      otherObject ")" ] expressed above */
    : GREATER
    | LESS
    | ATLEAST
    | ATMOST
    ;
```

```
predicate
    : PREDICATE variableName "(" setApplication ")"
    ;

setApplication
    : setName "(" ")"
    | setName "(" argList ")"
    ;

application
    : variableName "(" ")"
    | variableName "(" argList ")"
    ;

argList
    : variableList
    | setList
    | stringList
    | argList "," setList
    | argList "," variableList
    | argList "," stringList
    ;

setList
    : setName
    | setList "," setName
    ;

variableList
    : variableName
    | variableList "," variableName
    ;

otherObject
    : string
    | naturalExpression
    ;

naturalExpression
    : CARDINALITY set
    | application
    | variableName
    | natural
    ;

stringList
    : string
    | stringList "," string
    ;

naturalList
    : natural
    | naturalList "," naturalList
    ;

variableName
    : LOWERCASEIDENT
    ;

setName
    : UPPERCASEIDENT /* the convention that set names begin with a capital
                letter is recomended by [Woodcock88] page74 */
    ;

natural
    : NATURAL
    ;

string
    : STRING
    ;

constraintnum
    : CONSTRAINTNUM
    ;

%%
```

Appendix E - The Code Generated for State Transition Diagrams

Excerpts of Napier88 code, generated from VC-t specifications of constraints, are given below to exemplify the use of the translation rules (see Section 8.3.3 on page 99). The constraints are extracted from the specification of STD presented in Appendix B.2. The first five constraints provide examples of all the rules except of Rule 7 and Rule 8; these two rules correspond to multiple bound variables. A example of that case is given by constraint C10. Therefore, to make the presentation more concise we restrict it to the constraints C1 to C5 and C10, rather than using the full specification. The translation rules used in the generation process are indicated as comments in the code.

Constraint 1

VC-t specification

```
‘C1: "Names must be unique amongst initialState, states and finalStates"

PREDICATE uniqueName( AnyStates(std) )’
```

Napier88

```
'let processInstantiated  /* Rule 4 */
begin
    /* evaluate predicate*/
    if( l_uniqueName( "AnyStates" )  /* library function */ )
    do success = TRUE
end

if( traverse( processInstantiated ) = FAIL )  /* Rule 3 */
do constraintFeedback(
    "Names must be unique amongst initialState, states and finalStates" )  /* Rule 1*/'
```

Constraint 2

VC-t specification

```
‘C2: “A STD must have at least one initial state”

EXISTS i : InitialState • i BELONGING InitialStates(std) AND TRUE’
```

Napier88

```
‘let processExistential  /* Rule 11 and Rule 5 */
```

```
begin
    if( voIcon( viIcon )( iName ) = "InitialState" )  /* Rule 2 */
    do success = TRUE
end

if( traverse( processExistential ) = FAIL )  /* Rule 3 */
do constraintFeedback( "A STD must have an InitialState" )  /* Rule 1*/'
```

Constraint 3

VC-t specification

```
'C3: "A STD has at most one initial state"

 CARDINALITY InitialStates(dfd) <= 1'
```

Napier88

```
'let pos-condition  /* Rule 12 */
begin
    if(  InitialStates ≤  1 ) /*perform final evaluation of predicate logic statement*/
    do return TRUE
end'

'int InitialStates = 0  /* Rule 13 */

let processCardinality
begin
    if( voIcon( viIcon )( iName ) = "InitialState" )  /* Rule 2 */
    do InitialStates = InitialStates + 1 /*Rule 12*/
end

if( traverse( processCardinality ) = FAIL )  /* Rule 3 */
do constraintFeedback( "A STD has at most one initial state" )  /* Rule 1*/'
```

Constraint 4

VC-t specification

```
'C4: "The initial state must have at least one out-transition"

FORALL i : InitialState • f BELONGING InitialStates(std) IMPLIES
    ( EXISTS t : Transition • t BELONGING Transitions(std) AND
            ( origin(t) = i AND ~( destination(t) = i )))'
```

Napier88

```
'/* Fragment 1*/
/*First level - universal quantification*/ /* Rule 10 */

let processUniversalL1  /* Rule 6 */
```

```
begin
    if( voIconL1( viIcon )( iName ) = "InitialState" )  /* Rule 2 */
            do if( ~traverse( processExistentialL2 ) = OK )  /* Rule 3 */
                    do violated = TRUE
end

if( traverse( processUniversalL1 ) = FAIL )  /* Rule 3 */
do constraintFeedback( "The initial state must have at least one out-transition " )
/* Rule 1 */'

'/* Fragment 2*/
/*Second level - existential quantification*/ /* Rule 9 */

let processExistentialL2  /* Rule 5 */
begin
    if( voConnectionL2( vcConnection )( cName ) = "Transition" and  /* Rule 2 */
            ( origin( voConnectionL2 ) = voIconL1 and
            destination( voConnectionL2 ) ~= voIconL1))
    do success = TRUE
end'
```

To obtain the final code, the fragments must be sequenced in reverse order /*see note 7.1 on page 103*/.

Constraint 5

VC-t specification

```
'C5: "A final state cannot have out-transitions or loop-transitions"

FORALL f : FinalState • f BELONGING FinalStates(std) IMPLIES
    ~( EXISTS t : Transition • t BELONGING Transitions(std) AND
            origin(t) = f )'
```

Napier88

```
'/* Fragment 1*/
/*First level - universal quantification*/ /* Rule 10 */

let processUniversalL1  /* Rule 6 */
begin
    if( voIconL1( viIcon )( iName ) = "FinalState" )  /* Rule 2 */
            do if( ~traverse( processExistentialL2 ) = OK )  /* Rule 3 */
                    do violated = TRUE
end

if( traverse( processUniversalL1 ) = FAIL )  /* Rule 3 */
do constraintFeedback(
    "A final state cannot have out-transitions or loop-transitions" )  /* Rule 1*/'

'/* Fragment 2*/
/*Second level - existential quantification*/ /* Rule 9 */
```

```
let processExistentialL2  /* Rule 5 */
begin
    if( ~( voConnectionL2( vcConnection )( cName ) = "Transition" and  /* Rule 2 */
            origin( voConnectionL2 ) = voIconL1 ))
    do success = TRUE
end'
```

To obtain the final code, the fragments must be sequenced in reverse order /*see note 7.1 on page 103*/.

Note: the negation inside the if statement (in 'processExistentialL2') results of having a negation in the existentialQuantification in the specification.

Constraint 10

VC-t specification

```
'C10: "The same pair of states cannot be connected by transitions with
the same direction and the same transition condition"

FORALL t1, t2 : Transition • f1, f2 BELONGING Transitions(std) IMPLIES
    (( origin(t1) = origin(t2) AND
            destination(t1) = destination(t2) AND
            transitionCondition(t1) = transitionCondition(t2)) IMPLIES
                t1 = t2 )'
```

Napier88

```
'/* Fragment 1*/ /* Rule 8 */

let processUniversalS1  /* Rule 6 */
begin
    if( voIconS1( viIcon )( iName ) = "Transition" )  /* Rule 2 */
    do if( traverse( processUiversalS2 ) = FAIL )  /* Rule 3 */
            do violated = TRUE
end

if( traverse( processUniversalS1 ) = FAIL )  /* Rule 3 */
do constraintFeedback( "The same pair of states cannot be connected by transitions
    with the same direction and the same transition condition"'

'/* Fragment 2*/
let processUniversalS2  /* Rule 6 */
begin
    if( voIconS2( viIcon )( iName ) = "Transition" )  /* Rule 2 */
    do if( ~( ~( origin( voIconS1 ) = origin( voIconS2 ) and
            destination( voIconS1 ) = destination( voIconS2 ) and
            transitionCondition( voIconS1 ) = transitionCondition( voIconS2 ) ) or
            ( voIconS1 = voIconS2 )))
        do violated = TRUE
```

end'

To obtain the final code, the fragments must be sequenced in reverse order (see note 7.1 on page 103).

Bibliography

[Adreit91] Adreit, Françoise and Bonjour, Michel, "EcrinsDesign: a Graphical Editor for Semantic Structures", presented at *Advanced Information Systems Engineering 3rd International Conference CAiSE'91*, Trondheim, Norway, 1991.

[Aho86] Aho, Alfred V.; Sethi, Ravi and Ullman, Jeffrey D., "Compilers Principles, Techniques and Tools", Addison-Wesley Publishing Company, 1986.

[Albano95] Albano, A.; Ghelli, G. and Orsini, R., "An Introduction to Fibonacci: A Programming Language for Object Databases", Technical Report Series FIDE/95/120, 1995.

[Alderson91] Alderson, Albert, "Meta-CASE Technology", presented at *European Symposium on Software Development Environments and CASE Technology*, Konigswinter, Germany, 1991.

[Allen91] Allen, R. E., "The Concise Oxford Dictionary", BCA, 1991.

[Atkinson90] Atkinson, M.P., Richard, P., and Trinder, P.W., "Bulk Types for Large Scale Programming", presented at *First International East/West Database Workshop*, 1990.

[Atkinson95] Atkinson, Malcolm and Morrison, Ronald, "Orthogonally Persistent Object Systems", in journal *VLDB*, vol. 4, pp. 319-401, 1995.

[Atkinson96] Atkinson, M. P.; Daynès, L.; Jordan, M. J.; et al., "An Orthogonally Persistent Java", in journal *ACM Sigmod Record*, vol. 25, 1996.

[Atzeni93] Atzeni, Paolo and Torlone, Riccardo, "A Metamodel Approach for the Management of Multiple Models and the Translation of Schemes", in journal *Information Systems*, vol. 18, pp. 349-362, 1993.

[Bancilhon92] Bancilhon, F.; Delobel, C. and Kanellakis, P., "The Story of O2: Building an Object-Oriented Database System", Morgan Kaufmann, 1992.

[Battista90] Battista, G. Di; Giammarco, A.; Santucci, G.; et al., "The Architecture of Diagram Server", presented at *IEEE Workshop on Visual Languages*, Skokie, Illinois, 1990.

[Beer88] Beer, Stephen John, "Supporting Checking in a Generic, Graphical, Software Design Environment", University of Strathclyde, Glasgow, PhD, 1988.

[Bertolazzi92] Bertolazzi, P.; Battista, G. Di and Liotta, G., "Parametric Graph Drawing", CNR, Roma, Italia, Technical Report 6/67, 1992/7 1992.

[Blaha92] Blaha, Michael, "Models of Models", in journal *JOOP*, pp. 13-18, 1992.

[Borning81] Borning, Alan, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", in journal *ACM Transactions on Programming Languages and Systems*, vol. 3, 1981/10, pp. 353-387, 1981.

[Borning87] Borning, A.; Duisberg, R.; Freeman-Benson, B.; et al., "Constraint Hierarchies", presented at *OOPSLA*, 1987.

[Bott89] Bott, Frank, "ECLIPSE An Integrated Project Support Environment", in *IEE Computing Series*, vol. 14, Peter Peregrinus Ltd., London, United Kingdom, 1989.

[Brinkkemper93] Brinkkemper, S., "Integrating Diagrams in CASE Tools through Modelling Tranparency", in journal *Information and Software Technology*, vol. 35, pp. 101-105, 1993.

[Budgen92] Budgen, David, "Augmenting the Design Process: Transformations from Abstract Design Representations", presented at *Advanced Information Systems Engineering 4th International Conference CAiSE'92*, Manchester, United Kingdom, 1992.

[Budgen94] Budgen, David, "Software Design", Addison-Wesley, 1994.

[Chang90] Chang, Shi-Kuo, "Principles of Visual Programming Systems", Prentice-Hall International, Inc., 1990.

[Chen76] Chen, P., "The Entity Relationship Model - Toward a Unified View of Data", in journal *ACM Transactions on Database Systems*, 1976.

[Cooper90] Cooper, Richard, "Configurable Data Modelling Systems", presented at *9th Entity Relationship*, Lausanne, 1990.

[Cooper94] Cooper, Richard and Welland, Ray, "The Contribution of the Persistent Approach to Software Engineering", presented at *Workshop on the Intersection Between Databases and Software Engineering*, Sorrento, Italy, 1994.

[Cutts89] Cutts, Q. I.; Dearle, A.; Kirby, G. N. C.; et al., "WIN: a Persistent Window

Management System.", Universities of St Andrews and Glasgow, Persistent Programming Research Report 73, 1989.

[Cypher93] Cypher, Allen, "Watch What I Do - Programming by Demonstration", The MIT Press, 1993.

[Edel88] Edel, Mark, "The Tinkertoy Graphical Programming Environment", in journal *IEEE Transactions on Software Engineering*, pp. 1110-1115, 1988.

[Ehrig86] Ehrig, H.; Nagl, M.; Rozemberg, G.; et al., "Graph-Grammars and Their Application to Computer Science. 3rd International Workshop, Warrenton, Virginia, USA", in *Lecture Notes in Computer Science*, vol. 291, Goos, G. and Hartmanis, J., Eds.: Springer-Verlag, 1986.

[Fun96] Fun, Choy Lai, "Development of a Graphical User Interface in TkWin for the GraphTool", University of Glasgow, MSc 1996.

[Green89] Green, T. R. G., "Cognitive Dimensions of Notations", presented at *HCI'89 Conference on People and Computers V,* 1989.

[Harel88] Harel, David, "On Visual Formalisms", in journal *Communications of ACM*, pp. 514-530, 1988.

[Hekmatpour87a] Hekmatpour, Shahram, "Formal Specification Based Prototyping", Open University, PhD, 1987.

[Hekmatpour87b] Hekmatpour, Sharam and Woodman, Mark, "Formal Specification of Graphical Notations and Graphical Software Tools", presented at *1st European Software Engineering Conference*, Strasbourg, France, 1987.

[Hofstede93] Hofstede, A. H. M. ter; Proper, H. A. and Weide, Th. P. van der, "Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models", in journal *Information Systems*, pp. 489-523, 1993.

[Hofstede96] Hofstede, A. H. M. ter and Verhoef, T. F., "Meta-CASE. Is the Game Worth the Candle?", in journal *Information Systems*, pp. 41-68, 1996.

[Jackson75] Jackson, M. A., "Principles of Program Design", vol. 12, Academic Press, 1975.

[Jazayeri95] Jazayeri, Mehdi, "Component Programming", presented at *Software Engineering - ESEC'95 5th European Software Engineering Conference*, Sitges, Spain, 1995.

[Kaplan90] Kaplan, Simon M., "Applying Graph Grammars to Software Engineering", presented at *Graph-Grammars and Their Application to Computer Science. 4th International Workshop*, Bremen, Germany, 1990.

[Karsenty92] Karsenty, Solange and Landay, James A. and Weikart, Chris, "Inferring Graphical Constraints with Rockit", presented at *HCI*, York, United Kingdom, 1992.

[Kelly96] Kelly, Steven; Lyytinen, Kalle and Rossi, Matti, "MetaEdit+ A Fully Configurable Multi-User and Muti-Tool CASE and CAME Environment", presented at *Advanced Information Systems Engineering 8th International Conference, CAiSE'96*, Heraklion, Crete, Greece, 1996.

[Kernighan88] Kernighan, Brian W. and Ritchie, Dennis M., "The C Programming Language", 2 ed, Prentice-Hall, 1988.

[King89] King, Roger, "My Cat is Object-Oriented", in "Object-Oriented Concepts, Databases, and Applications", Kim, Won and Lochovsky, Frederick H., Eds. Reading, Mass., Addison-Wesley Publishing Co., 1989, pp. 23-30.

[Kirby94] Kirby, G. N. C.; Brown, A. L.; Connor, R. C. H.; et al., "The Napier88 Standard Library Reference Manual (Version 2.2)", Department of Computational Science, University of St Andrews, Research Report CS/94/7, 1994.

[Kreowski90] Kreowski, Hans-Jörg, "Applied Graph Transformation", presented at *Graph-Grammars and Their Application to Computer Science. 4th International Workshop*, Bremen, Germany, 1990.

[Kruse87] Kruse, R.L., "Data Structures and Program Design", Prentice/Hall International Inc., 1987.

[Kuo95] Kuo, Ying Jean, "The Design and Implementation of a Truly Integrated GIS Using the Persistent Programming Language Napier88", University of Glasgow, Technical Report Series FIDE/95/131, PhD, 1995.

[Larsson96] Larsson, Peter, "TkWin a Modern GUI for Napier88", University of Glasgow, MSc 1996.

[Levialdi93] Levialdi, Stefano; Mussio, Piero; Protti, Marco; et al., "Reflections on Icons", presented at *VL'93*, 1993.

[Marttiin95] Marttiin, Pentti; Lyytinen, Kalle; Rossi, Matti; et al., "Modeling Requirements for Future CASE: Modeling Issues and Architectural

Considerations", in journal *Information Resources Management Journal*, pp. 15-25, 1995.

[Matthes94] Matthes, Florian; Müßig, Sven and Schmidt, J. W., "Persistent Polymorphic Programming in Tycoon: An Introduction", FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, FIDE Technical Report Series FIDE/94/106, 1994/8 1994.

[Meng96] Meng, Alvin Lam Fai, "Development of a Drawing Canvas in TkWin for the GraphTool", University of Glasgow, MSc 1996.

[Minas95] Minas, Mark and Viehstaedt, Gerhard, "A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams", presented *at 11th International IEEE Symposium on Visual Languages, VL'95*, Darmstadt, Germany, 1995.

[Morrison94] Morrison, Ron; Brown, Fred; Connor, Richard; et al., "The Napier88 Reference Manual Release 2.0", University of St Andrews, Research Report CS/94/8, 1994.

[Myers90] Myers, Brad A.; Giuse, Dario A.; Dannenberg, Roger B.; et al., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces", in journal *Computer*, vol. 23, pp. 71-85, 1990.

[Myers92] Myers, Brad A. and Zanden, Brad Vander, "Environment for Rapidly Creating Interactive Design Tools", in journal *The Visual Computer*, pp. 94-116, 1992.

[Nagl86] Nagl, Manfred, "A Software Development Environment Based on Graph Technology", presented at *Graph-Grammars and Their Application to Computer Science. 3rd International Workshop*, Warrenton, Virginia, USA, 1986.

[Nagl90] Nagl, Manfred, "Graph Grammars which are Suitable for Applications", presented at *Graph-Grammars and Their Application to Computer Science. 4th International Worksho*p, Bremen, Germany, 1990.

[Nuseibeh92] Nuseibeh, Bashar and Finkelstein, Anthony, "ViewPoints: A Vehicle for Method and Tool Integration", presented at *International Workshop on CASE (CASE92)*, Montreal, Canada, 1992.

[Ousterhout94] Ousterhout, J. K., " Tcl and the Tk Toolkit", 4 ed. Reading Massachusetts, Addison Wesley, 1994.

[Paredes93] Paredes, Carlos; Fiadeiro, Josė Luis and Costa, Josė Fėlix, "Object

Specifications: Modeling Behavior through Rules", presented at *OOPSLA'93 Workshop on Specification of Behavioral Semantics in Object-Oriented Information Modeling*, Washington, DC, USA, 1993.

[Poswig92] Poswig, Jörg; Teves, Klaus; Vrankar, Guido; et al., "VisaVis - Contributions to Practice and Theory of Highly Interactive Visual Languages", presented at *IEEE Workshop on Visual Languages*, Seattle, Washington, 1992.

[Rader93] Rader, Jock; Morris, Ed J. and Brown, Alan W., "An Investigation into the State-of-the-Practice of CASE Tool Integration", presented at *Software Engineering Environments*, Reading, United Kingdom, 1993.

[Reeves95] Reeves, Andrew; Marashi, Mustafa and Budgen, David, "A Software Design Framework or How to Support Real Designers", in journal *Software Engineering Journal*, pp. 141-155, 1995.

[Rekers94] Rekers, Jan, "On the Use of Graph Grammars for Defining the Syntax of Graphical Languages", Leiden University, Department of Computer Science, Technical Report 94-11, 1994/11 1994.

[Rumbaugh91] Rumbaugh, James; Blaha, Michael; Premerlani, William; et al., "Object-Oriented Modeling and Design", Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Sanders95] Sanders, G. Lawrence, "Data Modeling", boyd & fraser publishing company, 1995.

[Sarkar92] Sarkar, Manojit and Brown, Marc H., "Graphical Fisheye Views of Graphs", presented at *CHI'92*, 1992.

[Schreiner85] Schreiner, Axel T. and Friedman, H. George, Jr., "Introduction to Compiler Construction with UNIX", Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1985.

[Schürr95] Schürr, Andy; Winter, Andreas J. and Zündorf, Albert, "Graph Grammar Engineering with PROGRES", presented at *Software Engineering - ESEC'95 5th European Software Engineering Conference*, Sitges, Spain, 1995.

[Serrano94] Serrano, J. Artur and Cooper, Richard, "On the Semantics of Visual Objects", presented at *HCI'94 People and Computers*, Glasgow, Scotland, United Kingdom, 1994.

[Serrano95] Serrano, J. Artur, "The Use of Semantic Constraints on Diagram Editors", presented at *11th International IEEE Symposium on Visual Languages, VL'95*, Darmstadt, Germany, 1995.

[Shu88] Shu, Nan C., "Visual Programming", Van Nostrand Reinhold Company, N.Y., 1988.

[Sleep90] Sleep, M. R., "Applications of Graph Grammars and Directions for Research", presented at *Graph-Grammars and Their Application to Computer Science. 4th International Workshop*, Bremen, Germany, 1990.

[Smolander91] Smolander, Kari; Lyytinen, Kalle; Tahavanainen, Veli-Pekka; et al., "MetaEdit - A flexible Graphical Environment for Methodology Modelling", presented at *Advanced Information Systems Engineering 3rd International Conference CAiSE'91*, Trondheim, Norway, 1991.

[Sommerville86] Sommerville, Ian, "Software Engineering Environments". London, Peter Peregrinus Ltd., 1986.

[Stroustrup88] Stroustrup, Bjarne, "What is Object-Oriented Programming?", in journal *IEEE Software*, vol. 5, pp. 10-20, 1988.

[Stubbs89] Stubbs, D.F. and Webre, N.W., "Data Structures With Abstract Data Types and Pascal", second edition, Thompson Information, 1989.

[Tolvanen93] Tolvanen, Juha-Pekka and Lyytinen, Kalle, "Flexible Method Adaptation in CASE - the Metamodeling Approach", in journal *Scandinavian Journal of Information Systems*, vol. 5, pp. 51-77, 1993.

[Udell94] Udell, Jon, "Componentware", in journal *Byte*, pp. 46-56, 1994.

[Viehstaedt95] Viehstaedt, Gerhard and Minas, Mark, "Generating Editors for Direct Manipulation of Diagrams", presented at *Human-Computer Interaction 5th International Conference EWHCI'95*, Moscow, Russia, 1995.

[Waite95a] Waite, Cathy; Philbrow, Paul; Atkinson, Malcolm; et al., "The Glasgow Persistent Workshop: Principles and User Guide", University of Glasgow, Technical Report Series FIDE/95/125, 1995/8/22 1995.

[Waite95b] Waite, Cathy; Welland, Ray; Printezis, Tony; et al., "Glasgow Libraries for Orthogonally Persistent Systems - Principles, Organization and Contents", University of Glasgow, Technical Report Series FIDE/95/132, 1995.

[Welland88] Welland, Ray, "A Toolbuilders Guide to the ECLIPSE Design Editing System", University of Strathclyde, Glasgow, Research Report CS/ST/1/88, 1988.

[Welland90] Welland, Ray; Beer, Stephen and Sommerville, Ian, "Method Rule Checking in a Generic Design Editing System", in journal *Software Engineering Journal*, pp. 105-115, 1990.

[Wijers90] Wijers, G. M. and Heijes, H., "Automated Support of the Modelling Process: A View Based on Experiments with Expert Information Engineers", presented at *CAiSE'90*, 1990.

[Woodcock88] Woodcock, Jim and Loomes, Martin, "Software Engineering Mathematics", Pitman, 1988.

[Wynekoop93] Wynekoop, Judy L. and Russo, Nancy L, "System Development Methodologies: Unanswered Questions and the Research-Practice Gap", presented at *Fourteenth International Conference on Information Systems*, Orlando, Florida, 1993.

[Zanden91] Zanden, Brad Vanden; Myers, Brad A.; Giuse, Dario and Szekely, Pedro, "The Importance of Pointer Variables in Constraint Models", presented at *UIST*, 1991.