→ Rob

Providing Support for the Formal Development of Software

J.C. Bicarregui and B. Ritchie
Rutherford Appleton Laboratory
Chilton, DIDCOT,
Oxon OX11 0QX,
UK.

April 25, 1989

Abstract

We describe a system which is intended to support the mathematical reasoning required in the formal development of software. It is being developed by a team from Manchester University and the Rutherford Appleton Laboratory as part of the Alvey-supported IPSE 2.5 project. The system consists of two parts: an generic environment for interactively carrying out proofs in a modularised hierarchical store of inference rules, and a specialised environment for the construction and refinement of formal specifications using Vienna Development Method (VDM). The emphasis in the VDM support environment is on the verification of the specifications though the generation of proof obligations and their discharge within the generic proof system. The system itself was specified in VDM and is being implemented in Smalltalk-80.

1 Introduction

IPSE 2.5 is an Alvey Software Engineering project to build an integrated project support environment (IPSE) incorporating support for formal methods of software design [19][1]. The project as a whole will be covered elsewhere in this conference: this paper concentrates upon work on "Mural"the formal reasoning subproject being carried out by staff at Manchester University and Rutherford Appleton Laboratory.

The aim in Mural is to produce a support environment for the kinds of formal reasoning that arise in software engineering. Using a carefully-specified and constructed kernel, and through the use of "state-of-the-art" user interface design, we hope to harness the mechanical precision of the computer in combination with the intuitive insight of the user. The main purpose of Mural is the design of a *generic* system capable of providing support for many formal development methods. It is also intended to provide a prototype instantiation to support development using the Vienna Development Method (VDM).

The team members [2] have a brought a wide range of relevant experience to Mural, including structure editor generator design [5], the construction of theorem proving tools [6, 18], and a large body of research and practice in the field of model-oriented specifications.

Section 2 of this paper introduces the motivations behind the construction of Mural and Section 3 describes large scale design of the system. In section 4, we present the generic part of the system

[1]Readers wishing to obtain copies of IPSE 2.5 documents should apply to: Mr. M.K. Tordoff, STL NW, Copthall House, Nelson Place, Newcastle-Under-Lyme, Staffs ST5 1EZ, UK.

[2]Apart from the authors of this paper, those directly involved in Mural are: Neil Dyer, Bob Fields, Jane Gray, Kevin Jones, Cliff Jones, Ralf Kneuper, Peter Lindsay, Richard Moore and Alan Wills. Past members who also deserve credit include: Jen Cheng, Ian Cottam, Mark van Harmelen, Lockwood Morris, Tobias Nipkow, Roy Simpson and Chris Wadsworth. As consultant to the project, Michel Sintzoff has provided useful input on many points. We are also grateful to David Duce for facilitating the MU/RAL collaboration; and for useful contributions from the rest of the IPSE 2.5 project.

and in section 5 we describe the part specific to VDM. Section 6 relates the history and current status of the project.

2 The Use of Formal Methods in Software Development

The argument for the use of formal methods in software development has been well-presented elsewhere (for example, see [9, 10]). By a *formal method* of software development, we mean the use of mathematical specification to capture the requirements of a design, and verification of design steps to yield implementations that are correct with respect to this specification.

2.1 Formal development is more than just specification

It is important to realise that the formal development of software involves more than just formal specification followed by implementation. A development may involve several layers of specification, and the construction of formal proofs of various properties of them and of the relationships between the specifications is a vital part of the method.

2.1.1 Validation

Whatever motivates the design of the initial formal specification, we must satisfy ourselves (or our customers) that this specification does indeed describe the system that is required. For example, we should check that no unexpected behaviour arises, and that every eventuality is catered for. This *validation* process is crucial: we can prove that an implementation is correct with respect to a formal specification, but since validation concerns the "interface" between formality and informality, we will never be able to *prove* that a formal specification agrees with an informal one. The best we can do is to informally increase our confidence that this is so. Thus, in a sense, validation is to specifications as testing is to programs. The use of a formal development method increases confidence in a design by minimising the informal aspects; furthermore, the informal/formal interface occurs earlier in the design process, and involves a higher level of reasoning, using terminology suited to the particular application, rather than implementation-biased representations.

One technique used in validation is to translate informal statements about the behaviour of the system into formal statements in the same language as the specification, and then formally prove that these properties hold. In a similar fashion to program test case generation, we can generate "interesting" test cases for a specification. Unlike program testing, a single "test" of the specification can cover a wide range of possibilities, for example, proof that a specification satisfies some property for all values of some input type. Part of the Mural effort is devoted to investigating techniques for the "animation" of specifications whereby one may perform "trial runs" on a specification in order to obtain formal statements about its behaviour.

2.1.2 Verification

Once we have a formal specification, the remainder of the development process can be made fully formal. This is not to say that the rest of the process is automatic, for the design decisions are not, but that each design step can be formally verified.

In order to claim that a development is fully formal, we must prove the correctness of each of our design steps, with respect to the initial specification. This process is known as *verification*. Each design decision we make – a choice of a particular representation of some abstract data type, or an implementation of an implicit function or operation – gives rise to certain *proof obligations* to justify that decision in terms of the original specification.

There are also feasibility obligations upon the specifications themselves. Simple type-checking can be performed mechanically; however, whenever data type invariants are used, proof obligations must be discharged in order to show that terms in specifications satisfy the invariants.

Often, the form of these obligations is determined by the formal development method, so that once the decision is made, the obligations are dictated. (See §5.1.)

Proving such obligations ensures that one layer of a design is correct with respect to the previous layer.

2.1.3 Further roles of proofs

There is more to be gained from doing proofs than confidence that our design is correct. The effort of building a proof often improves our understanding of a problem and, by making us notice subtle points and unexpected consequences of the specification, may highlight imprecisions or errors in it. Sometimes, we may decide that the proof is too awkward (or "ugly"), and this may lead to us deriving a cleaner specification of the problem.

It is also worth noting in passing that a *constructive* proof that (say) an implicit function specification is implementable can be used to generate an implementation automatically; this is used in the NuPRL system [4].

2.2 Logic, Proof and Theories.

In the above, we have freely used the term "proof". Before we proceed it might be useful to be more specific about what we actually mean by formal proof. First, we must discuss formal logic.

A theory of logic is a formal system of rules for the construction of valid forms of argument. The formal system consists of primitive assumptions and rules of inference in some formal language. Ultimately this language is just a system for manipulating uninterpreted symbols, but when we provide an interpretation, bestowing meaning to certain symbols and their combinations, we have a system by which we can assert the validity of the interpretation of other combinations of these symbols. In short, we build a collection of facts, confident that the validity of these facts relies only on the primitive assumptions, sometimes referred to as axioms, and on the primitive deduction rules of the logic. This process of asserting the truth of one statement relative to others is called *proof.* Thus a proof of a fact is a chain of deductions, each justified by other known facts which themselves are either primitive, i.e. assumed *a priori*, or derived, i.e. proven from other facts. Such proven facts are known as the theorems or derived rules of the logical theory[3].

In this way we can build up a large number of theorems. Some are required to assert a fact that we wish to know for its own sake. Others are general relationships between objects, asserting a fact which will be useful in other proofs. However, not all the results will be useful in all contexts: we may wish to postulate an axiom for use when reasoning about some objects, but which is not applicable in other cases, or we may have a symbol which is used to represent different things in different areas. Hence, in a given context only some of the results are in fact valid. Consequently there is a need to organise the "database" of results in such a way that those results which deal with the same symbols under the same interpretation, and rely on the same axioms, are available together. The collection of those results which are valid in such a context is often called a *theory*; and as we extend the database of results we can build a hierarchy of theories upon theories. The building of such a "Theory Store" allows reuse of results and hence raises the level of reasoning in our proofs beyond the level at which the primitive assumptions are stated, to the level of these previously proven results.

Such "fully formal" proofs, where all results are proven "from first principles" are extremely tiresome to construct. In practice, we may find it useful to omit the full proof of a result which we

[3]For a formal treatment of logic see, for example [16].

are confident we could complete if required. The unjustified step then becomes a further assumption. It should be noted that this assumption is of a different nature to the primitive ones as we believe that it could be proved if required, whereas the axioms are assertions that we believe without proof. Such reasoning is called "rigorous proof", and it also has a place in the formal development of software.

2.3 Why (Semi-) Automate Formal Reasoning?

The process of proof requires a combination of skills. Involved is the formulation of the statement of what is required to be proven, the choice of which rule to apply in any particular situation, and the manipulation of symbols required to do that application. When done by hand on paper, the last of these becomes a long, tedious and error-prone chore. Indeed, the process of fully formal proof is so tiresome that it is almost never attempted. Instead, a sketch of the proof is given with the intention that it could, in theory, be extended to the formal proof. As most mathematicians will admit, this leaves the way prone to any number of errors or oversights. Mathematical proofs in the literature only gain credence as successive generations of scholars fail to find errors in them. In software development such a luxury can ill be afforded; rather we would like to exploit the ability of a computer to do huge numbers of faultless symbolic manipulations and hence allow a far more detailed level of proof than was previously possible. Furthermore, in the development of even a modest program the large number of proofs and the huge amount of fine detail in each would make paper and pencil proofs infeasible.

As we are to use a computer to perform the application of the rules, it would be reasonable to ask if it could also choose which rules to apply and hence automate the entire proof production. However it is well known that this cannot be done in general for any but the most trivial logical theories. Even if fully automatic proof is not possible for arbitrary conjectures, perhaps it is possible to build a system with a set of heuristics that can in most practical cases automatically produce a proof? This has been attempted with some success (e.g. [3]). However, the problem with fully automatic theorem proving is that when the system fails to complete a proof, the uninitiated user is left with no understanding of the state reached and little idea of how to proceed. Hence the user must acquire considerable knowledge of the internal workings of the system in order to be able to present the problem to it in such a way that a complete proof is obtained. The view taken by the Mural team is that a better approach is to use the intuition of the human user, who has an insight into the problem domain and a feeling for *why* the result is true, to guide the proof process along the right lines. This involves providing a high degree of feedback during the proof construction in order to maintain the user's understanding. This approach was also explored in [18].

This interactive approach requires a lot of effort on the user's part. In an attempt to lessen this burden, Mural will allow the "encoding" of certain commonly used proof strategies in a style similar to *tactics* in LCF [6].

In short, the primary reason for mechanisation of the proof process is to increase our confidence in the correctness of the proof by allowing proof in greater detail and with a greater degree of accuracy. We hope that this mechanisation will also be a means of making the proof process easier.

3 The Basic Design of Mural

It is intended that IPSE 2.5 should be a generator of IPSEs, and perforce that Mural should be able to be "instantiated" to provide application-specific support. Thus Mural is intended to be capable of supporting a variety of formal development methods and specification languages, for example: VDM [10], Z [8], Hoare logics [9]. However, different formal methods and languages are based on different logics: VDM uses a three-valued "Logic of Partial Functions" (LPF) [1]; Z uses classical set theory; Hoare logics extend predicate calculi. The large variety of logics arises mainly from the desire to use a logic that is specifically suited to the realm of discourse; as with programming languages, different logics are suited to different applications. This means that Mural must be capable of supporting a wide variety of logics, so that it can be instantiated to cater for a particular formalism.

The approach taken in Mural is to provide a *logical framework* (as done, for example, in the Edinburgh Logical Frame [7]), upon which may be defined the notation, axioms and deduction rules of particular logics. Tools based upon the logical framework can then be used in any instantiated logic. This part of Mural is described in §4.

In addition to logical reasoning support, an environment for formal software development must also provide tools specific to the method:

- tools for building and storing specifications, programs and other method-specific constructs;
- a means of recording the links between specifications and their implementations;
- a means of constructing corresponding theories in the logical system;
- a means of generating proof obligations and tracking their progress.

It is more difficult to envisage how such support could be handled generically, and indeed we have not tackled such issues in the generic Mural. Instead, we have chosen to build a example environment for VDM that demonstrates the feasibility of building such support alongside the generic system. This is described in §5.

How the generic and VDM-specific parts of Mural interact is depicted in figure 1.

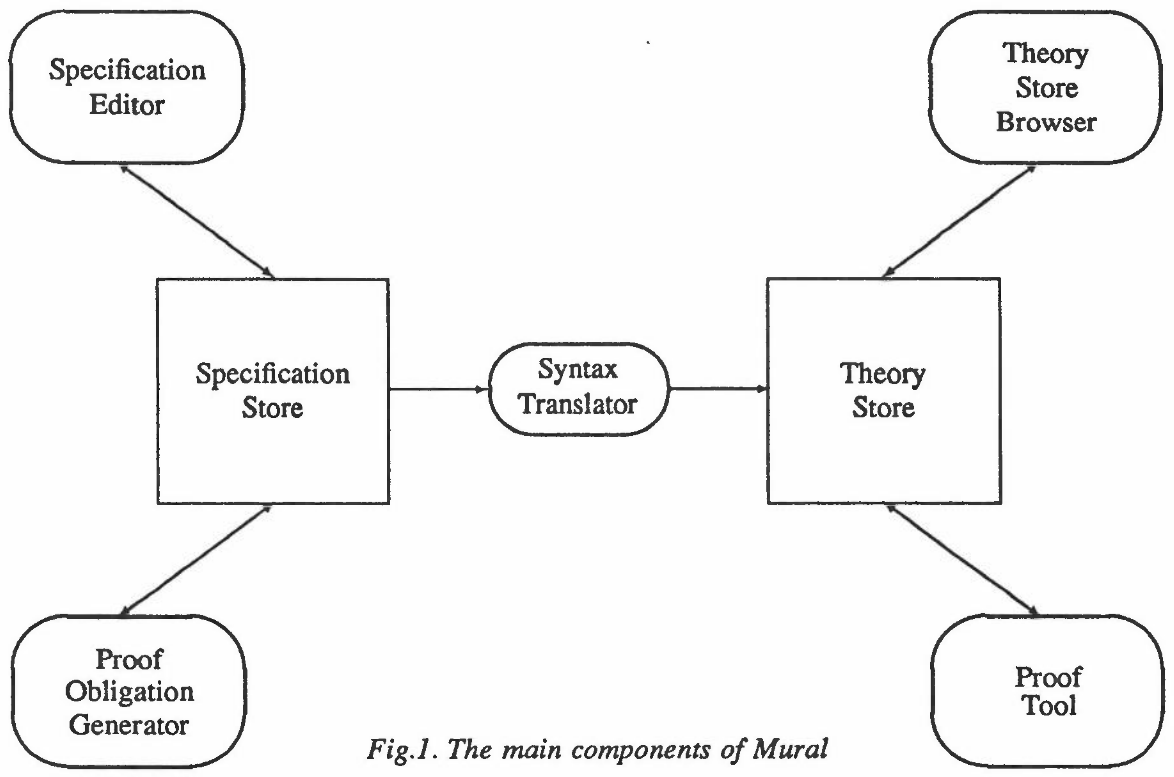

Fig.1. The main components of Mural

Ovals are data manipulation tools, **Boxes** are data stores, **Arrows** are data flow.

Also as part of the Mural project, Ralf Kneuper has proposed a tool for *symbolic execution* of specifications. Here properties of the specification are investigated by calculating the effects of operations on symbolic values. The work is reported in his forthcoming thesis [13]. Here we will just note that these calculations require theorem proving facilities.

Great emphasis has been placed on the design of the user interface of the system which we see as vitally important if the task of interactive proof is to be performed successfully. Using the latest workstation technology we attempt to present sufficient relevant information in such a way that users can maintain their intuition of the problem and hence guide the proof tool in the correct direction.

Input is based on structure editor-style, with a parser to shortcut the entry of low level objects such as expressions, and cut-and-paste facilities that can be used to copy objects between editors.

One of the goals of the proof assistant is to facilitate "proof at the workstation": that is, the construction of proofs on the machine without the prior recourse to a paper and pencil proof. If this is to be achieved it must be easy to try out a particular approach to a proof and to recover as much as possible when an erroneous route is taken. An important aim is to avoid placing too many constraints upon the order in which a user performs tasks. Thus, for example, it should be possible to create derived rules and use them in proofs before they themselves have been proven. Naturally, with such freedom comes the cost that the system should maintain the dependencies between the rules so that it can determine whether a derived rule has been proven from first principles or if it still depends upon some as yet unproven rule.

Another important aim is openness: it should be possible to perform any of the operations applicable to an object, in whatever context the object appears. For example, whether a rule is used in a proof or listed in a theory, it should be possible to look at its proof.

4 The Mural Generic Proof Assistant

4.1 The Logical Frame

Mural provides a Proof Assistant which is generic with respect to the underlying logic. This genericity is achieved by founding the proof system on a logical frame which allows the specific logics to be built up as *theories* within it. A theory contains definitions of types and operators and the axioms concerning them. It also inherits the contents of its parents and so provides the context for proof concerning these objects. Thus the rules that are valid in a theory include those in any of its ancestors. The hierarchical arrangement of theories provides a structuring mechanism for the large number of results that are required to be stored.

The instantiation of the system to a development method involves setting up the theories of the logic and primitive constructs of that method. Further theories are built on these in order to reason about the constructs defined in particular specifications. An example of a possible hierarchy of theories for reasoning about developments in VDM is given in figure 2.

The logical frame, called FSIP (a Formal System with Inclusion Polymorphism), facilitates reasoning about type structures with subtyping in that it allows the definition of a type as the restriction of another type by a predicate. Thus, an element can belong to more than one type. For example, it is possible to define the type of even numbers and the type of prime numbers, both subtypes of the integers, and then the number "2" is a member of all three of these types. This, of course, makes it possible to construct types for which type checking undecidable; however it can still be automated for a subclass of the types called the *patent types*.

As mentioned above the theory structure of FSIP permits a hierarchical construction of theories by the inheritance of the definitions and rules of ancestoral theories. In addition FSIP also makes it possible to construct other links between theories which we call theory morphisms. These allow inheritance of results with a possible renaming of symbols and can be used to record how one theory can be seen as an abstraction of another. For example, if we demonstrate that the integers under addition satisfy the axioms from the theory of groups viewed through a suitable renaming scheme, then we can construct a theory morphism between the two theories which allows the incorporation of all the results from group theory into that of the integers.

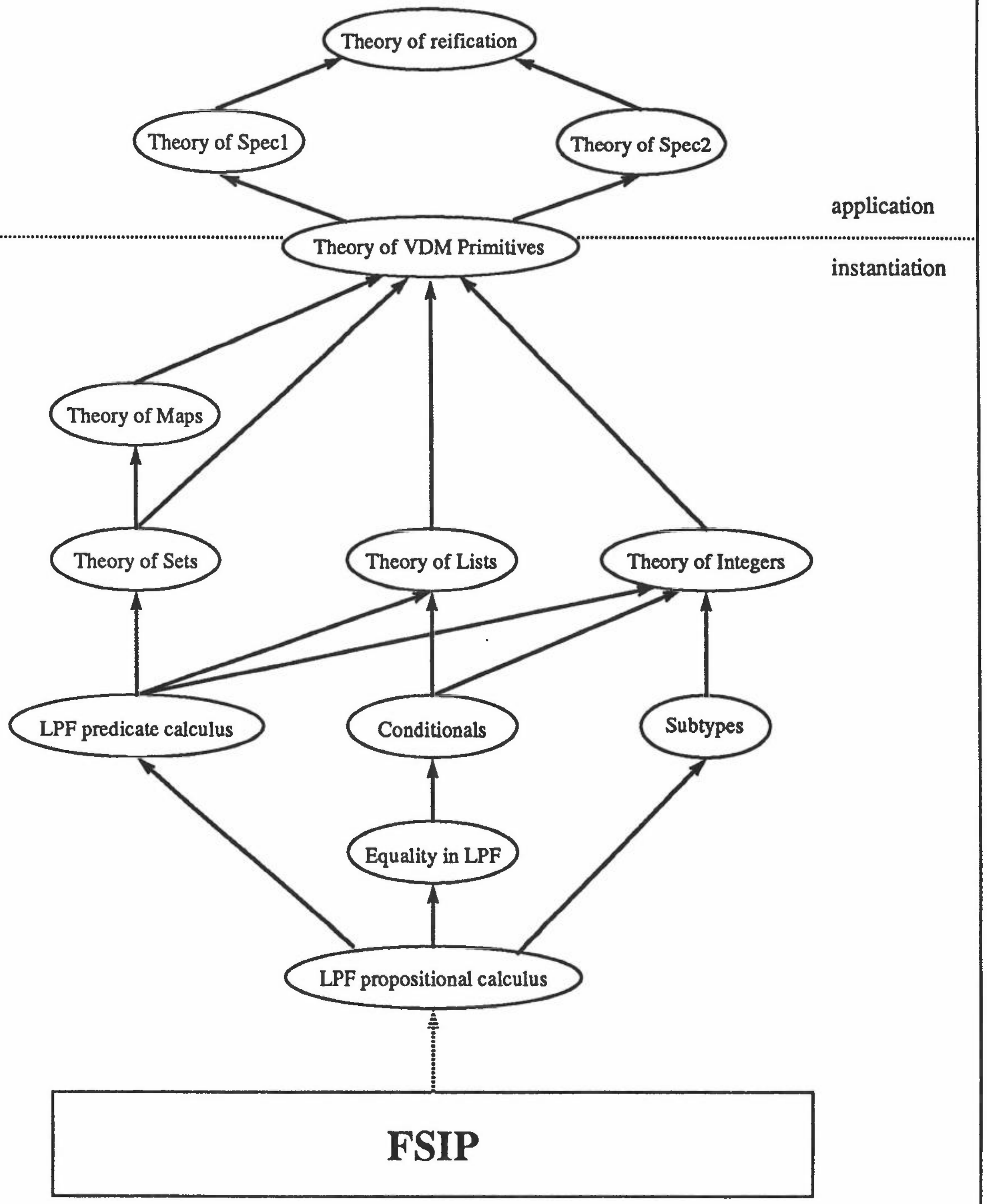

Fig. 2. A possible hierarchy of theories for reasoning about developments in VDM.

The lowest level theory built upon the logical frame FSIP describes the propositional part of LPF. On top of this we define the predicate calculus, equality axioms and subtyping rules. We then define separate theories of the basic type constructions available in VDM; these are brought together in the "VDM Primitives" theory. This theory can then be used in all developments. Theories for individual specifications all inherit from this theory, whilst reasoning about a reification of one specification by another is carried out in a theory that inherits from the theories of the two specifications.

4.2 The Theory Store Browser

The generic Mural provides a tool for the construction and maintenance of this theory store. The theory store browser permits the user to add theories to the theory store, and to edit or delete existing theories. Creating a new theory involves setting up its declarations and rules, and choosing its parental theories (those whose declarations and rules it will inherit). Each component is built using an appropriate structure editor. For example, rules are edited in a theory using a special rule editor. When a new rule is created, the rule editor presents a template; the user can then fill in the hypotheses and conclusion of the rule. One of the options in the rule editor is to view or edit the proof of the rule; selecting this opens an instance of the proof tool (see next section).

The components of a theory can be defined and altered in any order (for example, there is no need to "fix" the parents first). This flexibility makes it easy to experiment with theory construction.

Theory stores can be saved on and restored from disk, in addition to saving the entire Smalltalk image in a 'snapshot'.

4.3 The Proof Tool

Proof Construction

Another major component is the "proof tool", used for editing proofs of derived rules. It is possible to present and manipulate a proof in various ways. One presentation is as a list of numbered lines, each justified by a previous line in a manner similar to that which may be found in a textbook. An alternative view of a partially-completed proof, which we expect to be more useful during proof construction, hides the structure of the proof and shows only the current "knowns" and "goals"[4].

Associated with the proof tool are a variety of means of constructing justifications, including a "rule collector" which searches through the relevant parts of the theory store for rules that could be applied to the current state. The proof tool can be used to construct a proof working forwards from the hypotheses, backwards from the conclusion, starting somewhere in the middle, or any combination of these approaches.

As stated previously, the primary emphasis in the proof tool is in supporting *user-guided* proof. However, it is recognised that a means of capturing commonly-used proof strategies would lessen the user's burden by mechanising repetitive proof tasks. Mural provides a simple *tactic language* in which the user can build proof strategies appropriate to the problem domain. One example is a tactic that will reduce propositional formulae to normal form and hence provide a decision procedure for propositional logic. When applied to a proof (possibly taking additional parameters), a tactic may extend it and possibly even complete it; however it is non-destructive in that it will not undo any of the user's work.

Proof Checking

As stated above the user is given a great degree of freedom during the proof construction process. This laissez-faire approach to proof construction means that there is a need for the proof tool to be able to check the correctness of proofs. The proof tool can check that individual lines of proofs have valid justifications, that whole proofs are justified relative to the rules that are used in them, or even that a proof is valid from first principles.

Ultimately, this is the key component of the whole system, for everything else loses its purpose if we mistakenly admit incorrect proofs. Thus, if any part of the system itself were to be verified, the proof checker would be the prime candidate.

[4]This style was tested in the Muffin proof assistant (see §6).

5 The Mural VDM Support Environment

As stated earlier, a prototype instantiation of Mural to support specification and development in VDM is being built.

The main aims in our design of the Mural VDM support environment are:

- to demonstrate that the generic Mural can be instantiated to support a particular formal method;
- to generate "interesting" proof obligations upon which the theory store and proof assistant can be exercised.

It is not intended to build a "fully-fledged" support environment. In particular, little attention has been paid to tools (such as a parser) which would facilitate the construction of specifications, or enable the transfer of specifications from other systems. Our main interest is in the relationship between the VDM specifications and the hierarchy of theories needed to meaningfully reason about them.

5.1 The Vienna Development Method

VDM is a long-standing formal development method; the specification language which lies at its core has existed in various forms since the late sixties. Of the various dialects of VDM that currently exist we are using a subset of the language defined in the emerging BSI standard [2]. We adopt the design principles and proof obligations from [10].

VDM provides a richly typed specification language, with constructors for set, sequence, map, and compound (record) types, together with subtyping (by giving *invariants* on a type.) With these type constructors, a system designer can define data types suitable for the description of the intended system, and defer until later decisions on how these are to be represented in the implementation language. The language includes a notion of state: *operations* may affect the state and can be nondeterministic; *functions* must be purely applicative and cannot alter the state. Both functions and operations may be specified implicitly, abstracting from any commitment to algorithmic implementation details by giving logical predicates to specify their pre- and post-conditions[5].

As described in [10], starting from an initial (and hopefully "high-level" or abstract) specification, development steps revolve around the choice of "more concrete" representations for the data types of the specification (called *data reification*), and redefinition of the functions and operations to correspond to the new data types (*function/operation modelling*). The relationship between the specifications is formally encapsulated in the *retreive function* which uniquely associates an element of the abstract type to each element of the concrete one. This provides a formal means by which the specifications can be compared.

The method requires the discharging of certain proof obligations arising from these steps which ensure that the reification is correct. One example is the *adequacy* obligation. With each data reification we must show that the concrete data type can indeed be considered to be an "implementation" of the abstract. The adequacy obligation insists that for every element of the abstract type there should be a concrete element that represents it. This is stated formally as:

$$\forall a \in A \cdot \exists r \in R \cdot retr(r) = a$$

where A is the abstract type, R the concrete (or "reified") type, and *retr* the retrieve function.

Similarly, each concrete version of a function or operation must be shown to have behaviour that is "no worse" than its abstract counterpart, as defined by the *domain* and *result* obligations. Satisfaction

[5]To be more specific, the pre-condition of a function is a predicate on its arguments, and the post-condition relates the arguments to the value of the function. An operation's pre-condition may also refer to the "before" state, whilst its post-condition may additionally relate the "before" and "after" states.

of the domain obligation ensures that an operation can be invoked in the concrete environment whenever its abstract version can be invoked in the corresponding abstract environment. Formally, the domain obligation is

$$\forall r \in R \cdot \textit{pre-A}(\textit{retr}(r)) \;\Rightarrow\; \textit{pre-R}(r)$$

where R and *retr* are as before, and *pre-A* and *pre-R* are the preconditions of the abstract and concrete operations respectively. The result obligation is:

$$\forall \overleftarrow{r}, r \in R \cdot \textit{pre-A}(\textit{retr}(\overleftarrow{r})) \wedge \textit{post-R}(\overleftarrow{r}, r) \;\Rightarrow\; \textit{post-A}(\textit{retr}(\overleftarrow{r}), \textit{retr}(r))$$

Here $\overleftarrow{r}$ and r represent the "before" and "after" states of the concrete operation and the obligation states that when $\overleftarrow{r}$ corresponds to a "sensible" initial state for the abstract operation then $\textit{retr}(r)$ is a final state for the abstract operation. Roughly, satisfaction of both these obligations means that anything that can be done within the bounds of the abstract specification will be modelled in the concrete specification.

There are also a set of *operation decomposition rules*, used to justify the (possibly partial) implementation of an operation as the composition of other operations, where "composition" includes a variety of statement-like constructs: sequencing, if-then-else, while-loop, etc.

A reification between two specifications is considered formally verified when all of the associated proof obligations have been discharged.

The development process may involve several layers of specification before an implementation is arrived at, with each layer representing a particular design decision. Compositionality of the method ensures that once the proof obligations of each layer and of the reifications between them have been discharged the final implementation is correct relative to the initial specification.

5.2 A framework for this development process.

The emphasis in our framework is on the development process, not just on the specification language, for the most interesting proof obligations are those that arise from the design steps.

Within our framework:

- A development consists of a set of specifications and "reification relations" between them;
- A single specification consists of a set of data type definitions, a state definition, a set of constant declarations and sets of function and operation specifications;
- A "reification relation" between two specs consists of a set of data reifications and a set of function and operation modellings.

We may claim that one specification is an "implementation" of another. To show that this is so, we must show:

- all abstract data types have concrete counterparts;
- the concrete state has properties corresponding to properties desired of the abstract state;
- each implicit function/operation in the abstract spec has a modelling counterpart in the concrete spec;

where all of these correspondences have precise formal statements.

As part of the instantiation effort, the Mural store must contain some theory structure required by all VDM specifications:

- theories of the logic LPF upon which reasoning about VDM specifications is based.

- theories of the "standard" VDM data types, e.g. sets, sequences, maps, and numerical types;
- definitions of "built-in" functions over these types.

Since a specification may contain new user-defined types (and new information in the form of function and operation specifications), this information has to be translated into new components in the Mural theory store in order to provide the proper theory environment in which to (attempt to) discharge the proof obligations that arise. To establish heuristics for such a translation which take best advantage of sharing of information will require experiment and analysis.

One possibility would be to translate each specification in a development into a theory in the theory store (inheriting information from the "basic VDM" theory), where, for example, an implicit specification of a function f would become:

- a definition of *pre-f* as the pre-condition of f;
- a definition of *post-f* as the post-condition of f;
- an (initially unproven) rule corresponding to the implementability proof obligation for f

A reification of one specification by another could then be translated into a theory which inherits from the theories of both specifications. An operation or function modelling within this reification would generate rules to be proven corresponding to the domain and result proof obligations in that theory.

Naturally, there would be advantages in having "theory libraries" of commonly-used data types, of functions on them and of common data reifications between them. For example, sets are frequently reified by nonrepeating sequences. As we do not intend to support modular specifications, this will not be possible directly as part of specification development; but the sharing of the corresponding formal objects will of course still be possible with good organisation of the theory store.

6 Progress and Status of the Mural Project

We close this paper with a brief report on the history and status of the Mural project at the time of writing (January 1989).

The project began in October 1985 and is due to finish in September 1989. The initial aims for Mural were determined in a "concepts paper" which discussed many issues concerning support for formal reasoning. This was followed by an extensive evaluation of existing support, involving practical experience of many systems as well as a literature survey. The main results were published in [14]. Throughout the project, case studies and scenarios have proven useful for clarifying requirements, particularly those concerning the user interface. Much thought went into the structure editor principles upon which Mural is based, for example see [20].

The formal system which provides the theoretical basis for Mural was developed in a series of discussion papers which combined formal and informal descriptions. The theory store structure and a model of proofs suited to their interactive construction were developed similarly. These ideas were brought together in a detailed formal specification of the generic Mural [15]. The VDM support environment was similarly designed and specified [17]. These specifications enabled extensive debate and consequently went through numerous revisions, both major and minor. Only when we were confident that the specifications correctly captured our requirements did we proceed with the implementation.

It was clear from the start that to achieve the aim of enabling "proof at the workstation" a great deal of effort would have to go into the design of the user interface to the system. Whilst the formal specification proved to be an excellent vehicle for discussion and refinement of our ideas about the core of the system, we were aware that, the formulation of what was required of the user interface for the theorem prover could best be done with a testbed on which to try ideas out. For this reason we built a proof assistant, Muffin. Briefly, Muffin is a highly interactive proof tool for propositional logic with a novel user interface designed to show just the relevant information at each stage in the proof. Muffin and the assessment of its UI is discussed in [11].

Muffin was formally specified before implementation, and its implementation in Smalltalk-80 also served as an assessment of that language for our purposes (see [12]). It is interesting to note that the specification of Muffin took six months but its implementation took only two, even though the implementors were new to the language.

Confident from our experiences in implementing Muffin, we proceeded with a first implementation of Mural from its formal specification in a similar "direct" manner. Many of the interface tools built for the Muffin implementation were adapted and re-used here to provide an initial interface. This implementation provided a basis for a variety of experiments in areas not covered by the formal specification, for example a "rule matching" interface tool for proofs, construction of the theory store of Figure 2, and use of a simple tactics language.

At the time of writing, the experiments upon the first implementation are drawing to a close, and work has begun upon the final implementation, including a more specialised user interface.

References

[1] H. Barringer, J.H. Cheng, C.B. Jones. *A logic covering undefinedness in program proofs.* Acta Informatica **21** (1984) 251-269.

[2] D. Andrews *et al. VDM Specification Language Protostandard.* BSI IST/5/50 document 40, 1989.

[3] R. Boyer and J. Moore. *A Computational Logic.* Academic Press, 1979.

[4] R. Constable *et al. Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, 1986.

[5] I. Cottam, C. Jones, T. Nipkow, A. Wills, M. Wolczko and A. Yaghi. *Mule – an Environment for Rigorous Software Development (Final Report to SERC on Grant Number GR/C/05672).* Department of Computer Science, University of Manchester, 1986.

[6] M. Gordon, R. Milner and C. Wadsworth. *Edinburgh LCF.* Lecture Notes in Computer Science, Vol. 78, Springer-Verlag, 1979.

[7] R. Harper, F. Honsell and G. Plotkin. A framework for defining logics. *Proceedings of Second Symposium on Logic in Computer Science,* 194–204, 1987.

[8] I. Hayes, *ed. Specification Case Studies.* Prentice-Hall International, 1987.

[9] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. In *Communications of the ACM,* Vol. 12 No. 10, 1969.

[10] C. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, 1986.

[11] C. Jones and R. Moore. Muffin: a user interface design experiment for a theorem proving assistant. In: R. Bloomfield *et al.*, eds., *VDM '88: VDM – The Way Ahead.* Lecture Notes in Computer Science Vol. 328, 1988.

[12] K. Jones. *The muffin prototype: experiences with Smalltalk-80.* IPSE 2.5 document 060/00066/1.1, 1987.

[13] R. Kneuper. Forthcoming Ph.D. thesis, title to be announced. University of Manchester Computer Science Department.

[14] P. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal,* 3(1):3–27, 1988.

[15] R. Moore. *The Bumper Mural Spec.* IPSE 2.5 document 060/00143/2.1, 1988.

[16] D. Prawitz. *Natural Deduction.* Almqvist and Wiskell, 1965.

[17] B. Ritchie and J. Bicarregui. *The LHS Spec,* IPSE 2.5 document 060/00144/2.1, 1988.

[18] B. Ritchie. *The Design and Implementation of an Interactive Proof Editor.* Ph.D. thesis, University of Edinburgh report CST-57-88 (LFCS-88-68), 1988.

[19] R. Snowdon. *Scope of the IPSE 2.5 project.* IPSE 2.5 project document 060/00002/4.1.

[20] A. Wills. Structure of interactive environments. In: *Software Engineering Environments, Proceedings of the 3rd Annual Conference in Software Engineering Environments,* 1987.