Experience with *mural* in Formalising Dust-Expert

Sunil Vadera, Farid Meziane and Mei-Ling Lin Huang

http://www.salford.ac.uk/isrc/vadera
email:S.Vadera@cms.salford.ac.uk
Department of Computer and Mathematical Sciences, University of Salford,
Salford M5 4WT, UK

Stream : Industrial Applications, Experience report
Keywords: Safety, Tools

Abstract. The *mural* system was an outcome of a significant effort to develop a support tool for the effective use of a full formal methods development cycle. Experience with it, however, has been limited to a small number of illustrative examples that have been carried out by those closely associated with its development and implementation. This paper aims to remedy this situation by describing the experience of using *mural* for specifying Dust-Expert, an expert system for the relief venting of dust explosions in chemical processes. The paper begins by summarising the main requirements for Dust-Expert, and then gives a flavour of the VDM specification that was formalised using *mural*. The experience of using *mural* is described with respect to users' expectations that a formal methods tool should: (i) spot any inconsistencies, (ii) help manage and organise the specifications and allow one to easily add, access, update and delete specifications, (iii) help manage and carry out the refinement process,(iv) help manage and organise theories, (v) help manage and carry out proofs. The paper concludes by highlighting the strengths and weaknesses of *mural* that could be of interest to those developing the next generation of formal methods development tools.

1 Introduction

Although there are a number of tools that support the specification phases of the formal development cycle, there are few that support the full development cycle covering specification, refinement, generating proof obligations, and theorem proving. A notable exception to this is the research on the *mural* system [6], which was developed by Manchester University and Rutherford Appleton Laboratory under the Alvey IPSE 2.5 project. The *mural* system aims to support specification, refinement, and verification, all within one user-friendly environment. To achieve these tasks, *mural* has two principal components: a specification support tool, and a generic theorem proving assistant. However, experience with the *mural* system is limited to only a few relatively small case studies such as

a Reactor Watchdog [3], and a simple address book [1] that have been carried out by those who were closely associated with the development of *mural*. This paper takes a step towards remedying this situation by describing an attempt to use *mural* for the specification and design of an expert system known as Dust-Expert [9, 11, 12].

The paper is organised as follows. Section 2, introduces Dust-Expert, its requirements, and outlines an informal specification. Section 3 presents a formalisation of some of the operations using VDM. The specification is then refined in section 4. Section 5 presents the experience of using *mural* for each of the main phases of a formal development cycle: specification, refinement, verification (theorem proving in *mural*). Finally, section 6 summarises the main strengths and weaknesses of *mural* that could be of interest to those designing the next generation of formal methods tools.

2 An Informal Description of Dust-Expert

Dust-Expert aims to help companies that process or manufacture powders and dusts satisfy safety procedures. The main concern with dust handling processes is that of an explosion in a vessel. If a cloud of dust is ignited by a spark, such an explosion could result in a rapid rise in pressure that could destroy a vessel and lead to injuries to employees. The Institute of Chemical Engineers publish a guide [7] that explains how a relief panel can be placed on a vessel to avoid this kind of rise in pressure. The basic idea is that, as the pressure rises, the relief panel will open and release the pressure thereby avoiding an explosion. The guide also includes methods for calculating the size of the relief panel based on the kind of vessel, the kind of dust, the strength of the vessel, etc.

To enable greater utilisation of safety guidelines, the Health and Safety Executive (HSE) led a research project that developed a prototype expert system, called Dust-Expert, that was evaluated by over 16 member companies of the British Materials Handling Board. The promise shown by the prototype system resulted in the development of a commercial version of Dust-Expert by Adelard Ltd using the IFAD VDM toolkit and which is now available commercially from the Institute of Chemical Engineers.

This paper is based on the experiences gained in using *mural* to formalise the original prototype version of Dust-Expert for which one of the authors was responsible.[1]

[1] The specification of the commercial version is confidential, and the paper includes only those aspects of Dust-Expert that are already published elsewhere.

2.1 The Requirements

The primary requirements for Dust-Expert are typical of most expert systems and include:

1. The system should be easy to develop, requiring little programming expertise.
2. The system should be able to explain why a method is applicable or not as well as why it may be recommended.
3. It should be easy to update and add new methods.
4. It should provide enough flexibility for a user to ask question like
   'What range of dusts are acceptable for the given vessel?'

To meet these requirements a shell was designed and implemented. This shell represents the key to meeting the above requirements since it enables domain experts to encode the knowledge and methods without requiring programming skills. The following section summarises some of the key characteristics of the shell. The reader can consult [12] for more information about the shell as well as a comparative evaluation with CRYSTAL, a more conventional expert systems shells.

2.2 The Dust-Expert Shell

The shell enables an expert to encode knowledge as *constrained methods* and consists of three major kinds of methods called *Optional*, *Actual* and *Any*:

Actual methods. The *Actual* methods enable an expert to define individual constrained methods to calculate the value of a variable.

Optional methods. In general, there will be a number of available methods to calculate the value of a variable. The *Optional* methods enable one to express that a group of methods can potentially be used to work out the value of a variable and also that a group of rules can be used to give priorities to methods (i.e. rank the methods).

Any methods These allow an expert to define a sequence of methods, any of which could be tried to calculate a value for a variable. The shell attempts them in sequence and uses the first one that is applicable.

Figure 1(a) shows an actual method, called the kst_nomograph method for calculating the vent area (Av) and states that the calculation in the body can be utilised provided the expressions in the Constraints box can be satisfied. The text in the Assumptions box consists of additional information that is displayed

if the method is used. The variables in the constraints, (e.g., *Density_area*) may themselves be defined by other methods. Figure 1(b) gives an example of an optional method. If the volume of the vessel V is in the range specified in the constraint, and any of the methods listed are successful, then they are (partially) ordered by the ranking rules. Given a knowledge base of such methods, the task

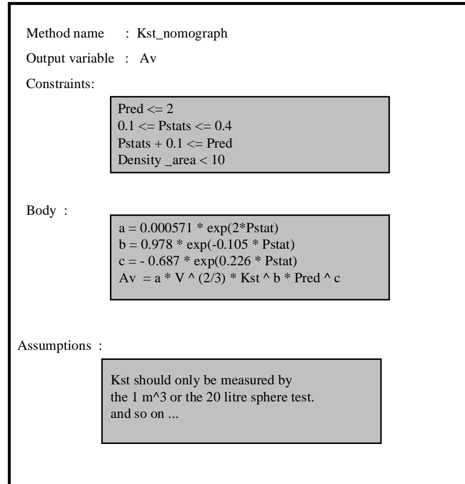

Figure 1(a) Example actual method

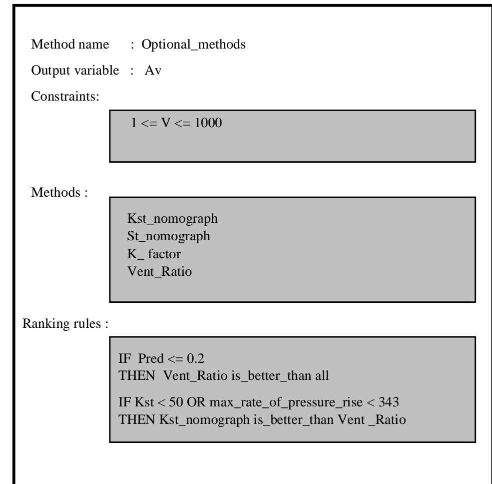

Figure 1(b) Example optional method

of the shell is to begin with a top level optional method and work through the methods in a top-down manner and obtain relevant information from the user so as to reach a recommended value for a variable (e.g. Av above). At any point, a user may ask how a value for a variable was obtained, or why a particular method was utilised. These explanations are provided by displaying a method that was used to obtain a value and displaying the constraints that are satisfied. The user can also understand why a particular method was not used by examining the constraints that were not satisfied.

3 A Flavour of the Formal Specification of Dust-Expert

Given the purpose of the paper, and the limited space available, this section aims to give only a flavour of the formal specification for the *Dust-Expert* shell using VDM. A full specification is available in [4]. Section 3.1 presents the data types, and section 3.2 presents two example specifications.

3.1 Data types

Simple types

The specification uses the following simple data types for the method names, values, and variable names:

$$Method\text{-}name = char^{+}$$
$$Value = \mathbf{R}$$
$$Variable\text{-}name = char^{+}$$

The state of Dust-Expert must consist of the three kinds of methods, and the global variables. The methods are specified using a map from the method name to each type of method and the global variables are specified using a map from the variable name to its information:

$$\begin{array}{l}
\textbf{compose } Dust\text{-}Expert \textbf{ of} \\
\quad opts \;:\; Method\text{-}name \xrightarrow[m]{} Optional \\
\quad anys \;:\; Method\text{-}name \xrightarrow[m]{} Any \\
\quad acts \;:\; Method\text{-}name \xrightarrow[m]{} Actual \\
\quad gts \;:\; G\text{-}Var \xrightarrow[m]{} Gv\text{-}info \\
\textbf{end}
\end{array}$$

For this state, we require that:

- The method names must be unique across the different kinds of methods.
- The variables in the constraints are defined *global* variables.
- The subsidiary methods in an optional or *any* method are defined.
- The last assignment in the body of each *actual* method must aim to assign a value to the output variable which must be a *global* variable. The other variables on the LHS must be local variables.

These requirements can be formalised as the following invariant:

$$\begin{array}{l}
\textbf{inv}(mk\text{-}Dust\text{-}Expert(opts, anys, acts, gts)) \;\; \underline{\triangle} \\
\quad unique\text{-}method(opts, anys, acts) \wedge \\
\quad defined\text{-}global\text{-}in\text{-}cnst(opts, anys, acts, gts) \wedge \\
\quad defined\text{-}subsidiary\text{-}mns(opts, anys, acts) \wedge \\
\quad body\text{-}vars\text{-}in\text{-}actual(acts, gts)
\end{array}$$

The state is initialised as follows:

$$\begin{array}{l}
\textbf{init}(mk\text{-}Dust\text{-}Expert(opts0, anys0, acts0, gts0)) \;\; \underline{\triangle} \\
\quad opts0 = \{\,\} \wedge anys0 = \{\,\} \wedge acts0 = \{\,\} \wedge gts0 = \{\,\}
\end{array}$$

Data types for the methods and variables

The data types for the three methods closely mirror the informal description given in section 2. Each type of method is defined as a composite type with relevant components.

An actual schema has components for the output (*output*), the list of constraints (*cnstl*), and the body (*bdyl*):

$$\begin{aligned} Actual :: \; & output : G\text{-}Var, \\ & cnstl \;\; : Constraint^{*}, \\ & bdyl \;\;\; : Assignment^{+} \end{aligned}$$

The body of an *actual* method has a list of the assignments, where *Assignment* is defined as:

$$\begin{aligned} Assignment :: \; & var \;\; : Variable, \\ & expr : Expression \end{aligned}$$

The type for an *optional* method is defined in a similar way: in addition to the output and the list of constraints, the *optional* method also has the list of methods (*lmns*) and the selection type (*seltp*) :

$$\begin{aligned} Optional :: \; & output : G\text{-}Var, \\ & cnstl \;\; : Constraint^{*}, \\ & lmns \;\; : Method\text{-}name^{+}, \\ & seltp \;\; : Selection\text{-}type \end{aligned}$$

The field *seltp* specifies which of the successful methods should be selected. There are three options: (i) *min* to select the method that returns the smallest value, (ii) *max* to select the method that returns the largest value, and (iii) *ranking-rules* to utilise the associated rules to rank the methods. *Selection-type* is therefore defined by:

$$Selection\text{-}type = \{all, min, max\} \mid Ranking\text{-}Rules$$

$$Ranking\text{-}Rules = Rule^{+}$$

The type $Rule^{+}$ denotes a production rule whose consequent defines one method to be 'better than' others if its antecedent is true. For conciseness, this type is not specified further here (see [12] for details).

The third kind of method, called the *any* method, has the three components of output, a list of constraints, and a list of methods:

$$Any :: output : G\text{-}Var,$$
$$cnstl : Constraint^*,$$
$$lmns : Method\text{-}name^+$$

There are two kinds of variables used in Dust-Expert: the global variables which are defined as output variables and the local variables which are defined as the LHS variables for the local assignments used in *actual* methods. Hence the type *Variable* is defined as:

$$Variable = G\text{-}Var \mid L\text{-}Var$$

$$G\text{-}Var :: gvar : Variable\text{-}name$$

$$L\text{-}Var :: lvar : Variable\text{-}name$$

The value of a global variable can be obtained in three ways: (i) it can be calculated by a method; (ii) a user may provide it; (iii) it can also be a constant such as π or a chemical constant. Hence we define the following types for recording this information for global variables.

$$Gv\text{-}info = Gv\text{-}from\text{-}method \mid Gv\text{-}from\text{-}user \mid Gv\text{-}from\text{-}constant$$

$$Gv\text{-}from\text{-}method :: mn : Method\text{-}name,$$
$$val : [Value]$$

$$Gv\text{-}from\text{-}user :: val : [Value]$$

$$Gv\text{-}from\text{-}constant :: val : Value$$

The types *Expression* and *Constraint*, used above, denote arithmetic expressions and boolean expressions. These are specified in a relatively standard way and are not detailed here.

Data types for the answer

As mentioned earlier, the ability to provide explanations is an important requirement for Dust-Expert. Each type of method can fail or succeed. Hence, the answer must contain enough information to explain why a method failed or how a method succeeded. Thus, the type *Answer* is defined as:

$$Answer = Failure\text{-}ans \mid Success\text{-}ans$$

In general, a method can fail in two ways: it can fail because one or more of its constraints fails; or it can fail because none of its methods succeeds (i.e. when it is an *optional* method or an *any* method). The type *Failure-ans* is therefore

defined as:

$$Failure\text{-}ans = Fail\text{-}cnst \mid Fail\text{-}lmns$$

To enable explanation, when a method fails because of its constraints, we record all the method's constraints together with whether they are satisfied or not. The succeeding constraints are included to provide relevant background information. Hence the type for *Fail-cnst* is:

$$\begin{aligned} Fail\text{-}cnst :: mn &: Method\text{-}name, \\ ans\text{-}cnst &: Cnst^{*} \end{aligned}$$

$$\begin{aligned} Cnst :: cr &: \mathbf{B}, \\ cnst &: Constraint \end{aligned}$$

Where the field *cr* records whether the constraint *cnst* is satisfied or not.

If a method fails because all its subsidiary methods are not applicable, Dust-Expert must be able to explain why each of its methods fails. Hence we define *Fail-lmns* as:

$$\begin{aligned} Fail\text{-}lmns :: mn &: Method\text{-}name, \\ all\text{-}f &: Failure\text{-}ans^{*} \end{aligned}$$

A successful answer will be returned in the type *Success-ans*:

$$Success\text{-}ans = Act\text{-}s \mid Opt\text{-}s \mid Any\text{-}s$$

A successful *actual* method will simply return the answer. We therefore define *Act-s* as:

$$\begin{aligned} Act\text{-}s :: mn &: Method\text{-}name, \\ ans &: Ans \end{aligned}$$

$$\begin{aligned} Ans :: output &: G\text{-}Var, \\ value &: Value \end{aligned}$$

The result returned by a successful *optional* method depends on the kind of priority specified by the selection field: *min*, *max*, or *ranking-rules* as defined earlier. Hence, we define *Opt-s* as follows:

$$\begin{aligned} Opt\text{-}s :: mn &: Method\text{-}name, \\ all\text{-}s &: Success\text{-}ans^{*}, \\ all\text{-}f &: Failure\text{-}ans^{*}, \\ res &: [Success\text{-}ans \mid Ranker] \end{aligned}$$

Note that the fields *all-s* and *all-f* are defined for all succeeding and failing methods so that the information for providing full explanations is available.

The type for an answer returned by a successful *any* method is defined as:

$$
\begin{array}{lll}
Any\text{-}s :: & mn & : Method\text{-}name, \\
 & res & : Success\text{-}ans, \\
 & all\text{-}f & : Failure\text{-}ans^{*}
\end{array}
$$

3.2 Example specifications

The shell for Dust-Expert has a number of operations for: (i) adding, deleting and updating methods, (ii) checking consistency, (iii) processing each type of method, (iv) evaluating constraints and expressions, and (v) explanation generation and user interaction. This section gives two example specifications to give an indication of the kind of specifications that *mural* had to handle.

Example 1: Adding an actual method

The operation for adding new actual methods is specified as follows. The precondition ensures three things: (i) the method name is new, (ii) all variables used in the constraints are defined, (iii) the last LHS variable used in the body of an actual method must be a global variable and the others variables on the LHS must be local variables. In addition, all RHS variables are defined before they are used. The post-condition simply adds the method.

$$
\begin{array}{l}
New\text{-}act\ \ (mn\colon Method\text{-}name, output\colon G\text{-}Var, cnstl\colon Constraint^{*}, \\
\qquad\qquad bdyl\colon Assignment^{+}) \\
\textbf{ext wr}\ acts\ : Method\text{-}name \xrightarrow{m} Actual, \\
\qquad \textbf{rd}\ opts\ : Method\text{-}name \xrightarrow{m} Optional, \\
\qquad \textbf{rd}\ anys\ : Method\text{-}name \xrightarrow{m} Any, \\
\qquad \textbf{rd}\ gts\ \ : G\text{-}Var \xrightarrow{m} Gv\text{-}info
\end{array}
$$

$\textbf{pre}\ mn \notin (\textbf{dom}\ opts \cup \textbf{dom}\ anys \cup \textbf{dom}\ acts)$

$\quad \wedge$

$\quad \textbf{let}\ cnstvars = extract\text{-}cnst\text{-}vars(\textbf{elems}\ cnstl)\ \textbf{in}$

$\quad cnstvars \subseteq \textbf{dom}\ gts$

$\quad \wedge$

$\quad \textbf{let}\ n = \textbf{len}\ bdyl\ \textbf{in}$

$\quad bdyl(n).var \in \textbf{dom}\ gts \wedge$

$\quad \forall i \in \textbf{inds}\ bdyl \cdot i < n \Rightarrow bdyl(i).var \notin \textbf{dom}\ gts \wedge$

$\quad \forall j \in \textbf{inds}\ bdyl \cdot defined\text{-}before\text{-}used(bdyl(j).expr, j, bdyl, \textbf{dom}\ gts)$

$\textbf{post}\ acts = \overleftarrow{acts} \dagger \{mn \mapsto mk\text{-}Actual(output, cnstl, bdyl)\}$

Where the function $extract\text{-}cnst\text{-}vars(cnsts)$ returns all the variables in the set of constraints $cnsts$. The function $defined\text{-}before\text{-}used(expr, i, bdyl, gvars)$ returns true if all the variables in the expression $expr$ are either globally defined (i.e. in $gvars$) or defined earlier (i.e., before the ith) in the list of assignments $bdyl$ and returns false otherwise.

Example 2: Processing an actual method

An *actual* method obtains a value by processing its assignments if its constraints hold. If the constraints fail then the *actual* method must return an answer explaining its failure. If a method succeeds, the answer obtained is recorded in the global variable in the state. Hence the specification takes the form:

$do\text{-}actual\ \ (mn: Method\text{-}name)\ aoc: Answer$

$\textbf{ext rd}\ acts\ : Method\text{-}name \xrightarrow{m} Actual,$

$\quad \textbf{wr}\ gts\ \ : G\text{-}Var \xrightarrow{m} Gv\text{-}info$

$\textbf{pre}\ mn \in \textbf{dom}\ acts$

$\textbf{post}\ \textbf{let}\ acs = process\text{-}cnstl(acts(mn).cnstl)\ \textbf{in}$

$\quad \textbf{if}\ acs.r = \textbf{false}$

$\quad \textbf{then}\ aoc = mk\text{-}Fail\text{-}cnst(mn, acs.ans)$

$\quad \textbf{else let}\ v = do\text{-}body(acts(mn).bdyl, gts)\ \textbf{in}$

$\quad\quad \textbf{let}\ ans = mk\text{-}Ans(acts(mn).output, v)\ \textbf{in}$

$\quad\quad aoc = mk\text{-}Act\text{-}s(mn, ans) \wedge$

$\quad\quad gts = \overleftarrow{gts} \dagger \{acts(mn).output \mapsto mk\text{-}Gv\text{-}from\text{-}method(mn, v)\}$

Where the function $process\text{-}cnstl$ takes a list of constraints as an argument and returns $acs.r$ as true if all the constraints are satisfied and otherwise returns $acs.r$ as false together with the list of constraints, and whether they failed or

not, in *acs.ans*. The function *do-body* evaluates a sequence of assignments and returns the value of the last assignment.

4 Data and Operation Refinement

The process of proceeding from an abstract specification towards a more concrete specification, that is closer to an implementation, can be an important aspect of a formal development. In VDM, this is done by refining the data types and defining specifications on the more concrete types. In order to appreciate the kind of support offered by *mural* for the refinement process, a simple refinement, that is described below, was carried out.

4.1 A Representation

There are many ways in which a specification can be refined in order to bring it closer to an implementation. An abstract specification can be translated into many alternative representations by using different data types.

In the abstract specification, we made extensive use of *maps*. The implementation language for Dust-Expert was intended to be Prolog. Hence, if one is proceeding towards such an implementation language, a refinement of *maps* to *sequences* is appropriate and can be carried out fairly systematically throughout the specification. For example, the abstract state component, *opts*, can be refined to:

$$optsc = Optionalc^*$$

Where *Optionalc* can be specified as:

$$\begin{array}{lll} Optionalc :: & mn & : Method\text{-}name, \\ & output & : G\text{-}Var, \\ & cnstl & : Constraint^*, \\ & lmns & : Method\text{-}name^+, \\ & seltp & : Selection\text{-}type \end{array}$$

The other state components can be refined in a similar way. Given such a refinement, the state representation is:

$$
\begin{array}{l}
\textbf{compose } \textit{Dust-Expert-c} \textbf{ of} \\
\quad \begin{array}{lll}
optsc & : & Optionalc^*, \\
anysc & : & Anyc^*, \\
actsc & : & Actualc^*, \\
gsc & : & Gv\text{-}record^*
\end{array} \\
\textbf{end}
\end{array}
$$

In representing a map by sequences, we may introduce duplicate names unnecessarily. Thus we also have to ensure no duplicate names in the components of the concrete state. Hence the concrete state invariant is defined as:

$$
\begin{array}{l}
inv(mk\text{-}Dust\text{-}Expert\text{-}c(optsc, anysc, actsc, gsc)) \quad \underline{\triangle} \\
\quad unique\text{-}method\text{-}c(optsc, anysc, actsc) \wedge \\
\quad defined\text{-}global\text{-}in\text{-}cnstl\text{-}c(optsc, anysc, actsc, gsc) \wedge \\
\quad body\text{-}vars\text{-}in\text{-}actual\text{-}c(actsc, gsc) \wedge \\
\quad defined\text{-}subsidiary\text{-}mns\text{-}c(optsc, anysc, actsc) \wedge \\
\quad no\text{-}duplicates(optsc, anysc, actsc, gsc)
\end{array}
$$

The functions used in the above invariant are similar to those in the abstract specification. Given the above representation, a suitable retrieve function based on converting sequences to maps was defined and *mural's* theorem proving assistant was used to carry out the usual adequacy proof obligation of VDM.

4.2 Operation Modelling

The main refinement made above was to use *sequences* instead of *maps*. As a consequence, accessing the methods requires different notation. Bearing this in mind, we can produce more concrete specifications to model each of the abstract specifications fairly systematically. The following gives an example of modelling the abstract operation *New-act*:

$$
\begin{array}{l}
New\text{-}actc \;\; (mn\colon Method\text{-}name, output\colon G\text{-}Var, \\
\qquad\qquad cnstl\colon Constraint^*, \\
\qquad\qquad bdyl\colon Assignment^+) \\
\textbf{ext rd } optsc \;:\; Optionalc^*, \\
\quad\;\;\, \textbf{rd } anysc \;:\; Anyc^*, \\
\quad\;\;\, \textbf{wr } actsc \;:\; Actualc^*, \\
\quad\;\;\, \textbf{rd } gsc \;:\; Gv\text{-}record^*
\end{array}
$$

$$
\begin{aligned}
&\textbf{pre } mn \notin method\text{-}names(optsc, anysc, actsc) \wedge \\
&\quad extract\text{-}cnst\text{-}vars(\textbf{elems}\, cnstl) \subseteq \{gsc(l).gvar \mid l \in \textbf{inds}\, gsc\} \\
&\quad \wedge \\
&\quad \textbf{let } n = \textbf{len}\, bdyl \textbf{ in} \\
&\quad bdyl(n).var \in \{gsc(l).gvar \mid l \in \textbf{inds}\, gsc\} \wedge \\
&\quad \forall l \in \textbf{inds}\, bdyl \cdot l < n \Rightarrow bdyl(l).var \notin \{gsc(l).gvar \mid l \in \textbf{inds}\, gsc\} \wedge \\
&\quad \forall m \in \textbf{inds}\, bdyl \cdot m \leq n \\
&\quad \Rightarrow defined\text{-}before\text{-}used(bdyl(m).expr, m, bdyl, \{gsc(l).gvar \mid l \in \textbf{inds}\, gsc\}) \\
&\textbf{post } actsc = \overleftarrow{actsc} \frown [mk\text{-}Actualc(mn, output, cnstl, bdyl)]
\end{aligned}
$$

Where *method-names* is defined as:

$$
\begin{aligned}
&method\text{-}names : Optionalc^* \times Anyc^* \times Actualc^* \rightarrow Method\text{-}name\textbf{-set} \\
&method\text{-}names(optsc, anysc, actsc) \quad \triangleq \\
&\quad \{optsc(i).mnc \mid i \in \textbf{inds}\, optsc\} \cup \{anysc(j).mnc \mid j \in \textbf{inds}\, anysc\} \cup \\
&\quad \{actsc(k).mnc \mid k \in \textbf{inds}\, actsc\}
\end{aligned}
$$

To show that this operation models its abstract version, we needed to use the theorem proving assistant to carry out the domain and result proof obligations [5].

5 Experience with *Mural*

This section reflects upon the experience of using mural to formalise the specification of Dust-Expert that consists of the kind of specifications and refinements given in the above sections.

When using a formal development tool, an analyst will hope or even expect that the tool will:

1. contain most of the formal notation in an easy to use form,
2. spot any mistakes or inconsistencies,
3. help manage and organise the specification and allow one to easily add, store, access, update and delete specifications,
4. help manage and carry out the refinement process,
5. help manage and organise the theory and allow one to easily add, access, update and delete the signatures, axioms and rules,
6. help manage and carry out proof obligations.

The following describes the extent to which these expectations were met. In interpreting our experience, the reader should note that *mural* was in fact a

vehicle for research on formal development tools, and some of the above expectations are for an ideal commercial formal methods development tool. That is, the purpose of reporting our experience is to help identify improvements to the next generation of formal development tools, and not simply to be critical.

5.1 Experience with the VDM Specification Tool

The specification was formalised in three stages in *mural*: first, the data types were created, then second, the specifications were created, and third, the specifications were translated to theories. The following describes each of these steps.

Creating the data types and the invariant

The VDM support tool (VST) of *mural* provides most of the VDM notation for creating data types and invariants. A syntax directed editor can be used to create and edit data types in a relatively straight forward manner. Apart from some minor differences with VDM-SL, the definitions are as one would expect. The *mural* system was being implemented while efforts to standardise VDM were still proceeding, and it is therefore not surprising that some of *mural*'s notation differs from the VDM-SL notation. There are two main differences with VDM-SL that had an adverse effect on creating the data types.

First, VDM-SL's pattern notation allows one to take advantage of pattern matching so that variables can be bound to values. It can be used in many places such as quantifiers, set comprehensions, *let..in* expressions, and *case* expressions. However, *mural* does not allow patterns to be used in such expressions. The only place where this is possible is in an invariant. Even there, the implementation is incomplete. For example, we used a pattern in a state invariant as follows: [2]

$$inv\text{-}Dust\text{-}Expert(mk\text{-}Dust\text{-}Expert(opts, anys, acts, gts) \colon Dust\text{-}Expert) \quad \triangleq$$
$$\ldots .$$

Unfortunately, although this was allowed in the definition, *mural* later failed to translate it to a theory.

Second, the VST does not allow one to create an enumerated type. To achieve the same effect, one can use a suitable invariant. For example, the following data type:

Selection-type= $\{all, min, max\}$ |Ranking-rules

[2] Note that the addition of a type *Dust-Expert* of the parameter is a *mural* requirement.

can be rewritten as follows in *mural*: [3]

$Type$ = is not yet defined

Selection-type= $Selection$ |Ranking-rules

$Selection$ :: sl : $Type$

inv-Selection(sl: $Type$) $\triangleq$ $sl \in \{all, min, max\}$

where 'all', 'min', and 'max' are all defined as $Constant$ of type '$Type$'.

Creating the specifications

Once the data types were created, we proceeded to create the functions and specifications. In *mural*, this is done by adding a specification and filling out a template using a syntax directed editor.

We encountered four problems that we needed to work around when creating the specifications. These are described and illustrated below.

Problem 1: Restricted constraints in quantifiers

Quantifier expressions in *mural* are restricted to have simple constraints of the form:

$variable$: $Type$

Thus, an expression like:

$\forall i \in \mathbf{inds}\, bdyl \cdot i < n \;\Rightarrow\; bdyl(i).var \in L\text{-}Var$

has to be reformulated to: [4]

$\forall i{:}\, \mathbf{N} \cdot i \in \mathbf{inds}\, bdyl \wedge i < n \;\Rightarrow\; s\text{-}var(bdyl(i)) \in L\text{-}Var$

Since quantifiers were heavily used in our specifications, this resulted in an increase in the complexity and length of the specifications.

Problem 2: No let..in expression

The *let..in* notation is not available, forcing the use the existential quantifier. For example, an expression like :

[3] Another alternative, this Selection-type can be re-defined by using the union type:

Selection-type= all | min | max |Ranking-rule

[4] The VST provides a selector function for naming a field from a record so that $bdyl(i).var$ is written as $s\text{-}var(bdyl(i))$.

$$\textbf{let } n = \textbf{len}\, bdyl \textbf{ in}$$
$$bdyl(n).var \in \textbf{dom}\, gts$$

Which is used in the specification of *New-act* has to be reformulated to:

$$\exists n : \mathbf{N} \cdot n = \textbf{len}\, bdyl \wedge s\text{-}var(s\text{-}bdyl(n)) \in \textbf{dom}\, gts$$

This type of reformulation tends to make the specification less readable.

Problem 3: No comprehension notation

Set and *sequence* comprehension expressions are missing in the VST and were frequently used in our specification. Equivalent recursive functions had to be written to work around this omission. For example, an expression used in defining a function *extract-cnst-vars*:

$$\bigcup\{vars\text{-}in\text{-}cnst(c) \mid c \in cs\}$$

was translated to the function:

$$
\begin{aligned}
&extract\text{-}cnst\text{-}vars : Constraint^* \rightarrow Variables\textbf{-set}\\
&extract\text{-}cnst\text{-}vars(cs) \quad \triangleq \quad \textbf{if } cs = [\,]\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \{\,\}\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } vars\text{-}in\text{-}cnst(\textbf{hd}\, cs) \cup extract\text{-}cnst\text{-}vars(\textbf{tl}\, cs)
\end{aligned}
$$

Besides being tedious, this work around made the specification less concise.

Problem 4: Cases cannot be translated

Although the VST provides case expressions, it is unable to translate them into the theories. The developers of *mural* propose two ways of working around this problem [6]:

1. Carry out the translation by hand and leave the specification alone (i.e. with *case* statements).
2. Change the specification so that it no longer uses *case* expressions.

As an experiment, we tried both alternatives and describe the experience below. Consider the function:

$$
\begin{aligned}
&vars\text{-}in\text{-}cnst : Constraint \rightarrow Variable\textbf{-set}\\
&vars\text{-}in\text{-}cnst(cnst) \quad \triangleq\\
&\quad \textbf{cases } cnst \textbf{ of}\\
&\quad mk\text{-}Compare\text{-}expr(cop, cx, cy) \rightarrow var\text{-}in\text{-}expr(cx) \cup var\text{-}in\text{-}expr(cy)\\
&\quad mk\text{-}Boundary\text{-}expr(bop, be, bx, by) \rightarrow var\text{-}in\text{-}expr(be) \cup var\text{-}in\text{-}expr(bx)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \cup\, var\text{-}in\text{-}expr(by)\\
&\quad \textbf{end}
\end{aligned}
$$

The first approach, namely to create the rules by hand, results in:

$$\boxed{\textit{1-vars-in-cnst}}\;\frac{cnst\colon Constraint,\, cnst = mk\text{-}Compare\text{-}expr(cop, cx, cy),\quad (var\text{-}in\text{-}expr(cx) \cup var\text{-}in\text{-}expr(cy))\colon Variable\text{-}\mathbf{set}}{vars\text{-}in\text{-}cnst(cnst) = var\text{-}in\text{-}expr(cx) \cup var\text{-}in\text{-}expr(cy)}$$

$$\boxed{\textit{2-vars-in-cnst}}\;\frac{cnst\colon Constraint,\, cnst = mk\text{-}Boundary\text{-}expr(bop, be, bx, by),\quad (var\text{-}in\text{-}expr(be) \cup var\text{-}in\text{-}expr(bx) \cup var\text{-}in\text{-}expr(by))\colon Variable\text{-}\mathbf{set}}{vars\text{-}in\text{-}cnst(cnst) = var\text{-}in\text{-}expr(be) \cup var\text{-}in\text{-}expr(bx) \cup var\text{-}in\text{-}expr(by)}$$

The second approach, to translate the cases to if's results in:

$$\begin{array}{l}
\mathbf{if}\ (\exists cop\colon Compare\text{-}op \cdot \exists bx\colon Expression \cdot \\
\quad \exists by\colon Expression \cdot cnst = mk\text{-}Compare\text{-}expr(cop, cx, cy)) \\
\mathbf{then}\ var\text{-}in\text{-}expr(s\text{-}cx(cnst)) \cup var\text{-}in\text{-}expr(s\text{-}cy(cnst)) \\
\mathbf{else\ if}\ (\exists bop\colon Boundary\text{-}op \cdot \exists be\colon Expression \cdot \exists bx\colon Expression \cdot \\
\qquad \exists by\colon Expression \cdot cnst = mk\text{-}Boundary\text{-}expr(bop, be, bx, by)) \\
\qquad \mathbf{then}\ var\text{-}in\text{-}expr(s\text{-}be(cnst)) \cup var\text{-}in\text{-}expr(s\text{-}bx(cnst)) \cup var\text{-}in\text{-}expr(s\text{-}by(cnst)) \\
\qquad \mathbf{else}\ \{\}
\end{array}$$

Both approaches proved to be tedious. The first requires going through each and every case expression. It has the disadvantage of the possibility of introducing a mis-match between the case expression and the theory. The second requires translating the case expression into a conditional expression and has the disadvantage that the specification is made more complex. Our own preference was to trade off some readability for knowing that the theory was consistent with the specification, and we therefore opted for the second option of translating cases to if-then expressions.

At this stage of creating the specification, the VST wasn't much help in spotting mistakes. Some significant type mistakes were spotted at the translation stage, however, some mistakes such as logic errors, inconsistencies, or some simple type errors were not revealed until proofs were attempted. For example, we made the following mistake in a function called *defined-before-used*:

$$bdyl(k).var = lvar$$

This was an error, because the data type for *bdyl(k).var* was defined as *Variable* and differed from the type for *lvar* which was *Variable-name*. We did not find this mistake earlier and the VST did not find it at the translation stage. It was only discovered when attempting to prove the well-formedness proof obligation. To be fair, research on *mural* may have sacrificed type checking in order to focus on other research problems.

Translation to theories

Once the specifications are created, we need to obtain the theories before carrying out any proof obligations. The VST provides facilities that allow us to

translate the data types and specifications into their associated theory and also generates proof obligations. The translation stage aims to achieve two goals [6]:

- To reveal errors in the specifications. When this occurs warning messages are given by the system.
- To generate a theory including any proof obligations. This theory can be opened, and proofs can be carried out supported by the TPA.

When we translated our specifications, *mural* found spelling errors and undeclared data type errors. Some undeclared type errors were not real errors in that the type was declared, but after it was used (i.e. a forward reference). This was easily fixed by following the instructions given by *mural*. A more significant problem was that the process was not incremental. If an error was detected or we wanted to change the specification, we first had to remove the signature, axioms and rules related to the changes, second we had to go back to the specification tool to change it, and then retranslate the relevant components. Specifications change and improve, making the lack of help for such changes a significant omission.

5.2 Experience with the theorem proving assistant

One of the most tedious and costly aspects of using formal methods fully is carrying out proofs. Indeed, many of the reported uses of formal methods avoid this phase of formal methods on the grounds that it is uneconomical.

If the role of proofs in real uses of formal methods is to increase then formal methods development tools must help analysts to manage and carry out proofs as painlessly as possible. As well as the usual requirements of theorem proving tools, proof tools must provide an environment that:

1. enables a user to easily construct a proof in a backwards, forwards, or mixed mode,
2. allows users to manage different attempts,
3. provides facilities for using proof strategies.

This subsection summarises the extent to which *mural*'s thorem proving assistant (TPA) provides these features. To do this, the TPA was used to carry out the three proof obligations mentioned in section 4: the adequacy, domain and result proof obligations. Based on the experiences of carrying out these proofs, together with previous experience with the TPA ([10]), the authors believe that the proof tool provides good support for carrying out proofs from first principles:

- One can use the proof tool and work in a backwards, forwards, or mixed directions.
- Theories can be searched for appropriate rules.
- Versions of proofs can be maintained.
- Tactics can be developed and used to carry out proofs or subproofs.

The TPA is undoubtedly very good, and it may be possible to build upon the strengths of such proof tools and develop tactics (e.g. Bundy's explicit proof plans to guide inductive proofs [2]) or even use proofs by analogy [10] to reduce the cost of carrying out proofs.

6 Conclusions

The *mural* system provides a friendly, modern interface for developing formal specifications. Based on this study, its main strengths were:

1. The specification tool provided good support for developing, managing and maintaining the specification.
2. Although not fully implemented, specifications and their refinements could be translated to corresponding theories. This is particularly useful, since other studies have shown that hand translations of specifications can introduce errors (e.g. [8]).
3. The theorem proving assistant provided very good support for managing, organising and maintaining theories, as well as support for carrying out proofs.

There were several problems encountered in using *mural* in this study. The most significant ones are:

- It wasn't an incremental system. When a change is made, a user has to manually trace the consequences of the change in order to ensure consistency. Ideally *mural* should provide some guidance about what is effected.
- Type checking was not carried out as well as in the IFAD Toolbox. This meant that some type errors were detected only when carrying out proofs.
- It wasn't fully implemented. Only a subset of the VDM-SL notation was available in the specification tool, and the operation modelling proof obligations were not generated by the translation process. This meant that alternative, less natural notation was used for some of the specification and proof obligations were created manually.
- As the size of the specification grew, the size of the image (*mural* was implemented in smalltalk) grew very large and the system got slower and slower.

When the findings of this study are brought together with other studies (e.g. [8]), the following main conclusions can be drawn about formal methods tools:

1. They need to be more incremental. Specifications change, either because we are unsure about the requirements or because we make mistakes. If changes are made, then the tools should offer some guidance on what else is effected and help make the changes. For example in *mural*, if a proof can not be completed because of a mistake, and that leads to a specification being corrected. Then new versions of the proof obligations are generated, which then requires a new proof, even for those sub-parts that may have been correct.
2. They should provide better support for the proof process. Providing the basic functionality for carrying out proofs is not enough. Much more needs to be done to aid reusability of proofs and developing proof strategies and tactics.

To conclude, this paper has presented our experience of using *mural* that may be worth considering when developing the next generation of formal methods development tools. In the future, we intend to repeat the exercise with other tools.

7 Acknowledgements

The authors are grateful to the developers of *mural* for allowing the use of the system. The first author is grateful to all those involved in the development of Dust-Expert, particularly Roger Santon, Alan Postill, and the member companies of the British Materials Handling Board (BMHB) who tested the research version of Dust-Expert. Dust-Expert is a trademark of the HSE and BMHB.

References

1. Juan Bicarregui and Brian Ritchie. Reasoning about VDM developments using the VDM support tool in mural. In S. Prehn and W.J. Toetenel, editors, *VDM '91 – Formal Software Development Methods*, pages 371–388. Springer-Verlag, October 1991.
2. A. Bundy, S. Andrew, F. van Harmelen, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artificial Intelligence*, 88(1-2):39–67, 1993.
3. B. Fields and M. Elvang-Gøransson. A VDM case study in mural. *IEEE Transactions on Software Engineering*, 18(4):279–295, 1992.
4. Mei-Ling L. Huang. Expereince with mural in formalising TheMD shell. Master's thesis, University of Salford, UK, 1997.

5. C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, London, 1990.
6. C.B. Jones, P.A. Lindsay, and Moore R. *Mural: A Formal Development Support System.* Springer-Verlag, London, 1991.
7. G. Lunn. *Guide to dust explosion prevention and protections.* Institute of Chemical Engineers, UK, 1992.
8. P. Mukherjee. Computer-aided validation of formal specifications,. *Software Engineering Journal*, 10:133–140, 1995.
9. R. Santon, A. Postill, T T Furman, S. Vadera, W Byers Brown, and K. Palmer. A feasibility study into the use of expert systems for explosion relief. In *HAZARDS XI: New Directions in Process Safety*, pages 317–328, 1991.
10. Sunil Vadera. Proof by analogy in mural. *Formal Aspects of Computing*, 7(2):183–206, 1995.
11. Sunil Vadera and Said Nechab. The development of Dust-Expert. In I. M. Graham, editor, *Proc. BCS Expert Systems Conference*, pages 189–202, 1993.
12. Sunil Vadera and Said Nechab. TheMD shell and its use to develop Dust-Expert. *Expert Systems: The International Journal of Knowledge Engineering and Neural Networks*, 12(3):231–237, 1995.

This article was processed using the LaTeX macro package with LLNCS style