The formal development of a Windows interface

Tim Clement
Adelard

November 15, 1999

1 Introduction

This paper describes an approach to the use of the formal method VDM in the design and implementation of Microsoft Windows^{TM} interfaces. This approach evolved during the development of DUST-EXPERT^{TM}, a Windows-based system for providing design advice on the prevention and control of dust explosions, developed for the Health and Safety Executive (HSE) and now being marketed by the Institution of Chemical Engineers (IChemE).

The approach we have adopted is deliberately conservative: we have aimed to see how we can take guidance in the design of the system from the standard Vienna Development Method rather than inventing new language constructs or new proof obligations. One advantage of this is that we can continue to use the tools that are available for supporting the standard language.

The next section provides some background to the application and some rationale for the use of formal methods in its development. Section 3 describes the specification. Section 4 describes the implementation steps and Section 5 compares our approach with some other formal approaches to interface design. The final section summarizes our experience and gives some metrics for the DUST-EXPERT project. We shall assume some familiarity with VDM, at the level that could be obtained from [Jon90].

2 Background

A dust explosion occurs when a cloud of finely divided particles is ignited. Such clouds, which can be of almost any material capable of oxidation, are created in many industrial processes. Moving grain or powdered sugar into a silo can create one, and may also generate enough static electricity to produce the igniting spark. Dust explosions can produce jets of flame tens of metres long, and are violent enough to destroy factories and kill workers. There is a large body of experimental and theoretical work that describes ways to prevent such explosions, perhaps by filling the vessels with an inert atmosphere, or to control the explosion, typically by providing vents in the plant that will allow it to be directed harmlessly rather than destroying its container. If the latter method is adopted, then the size of the vents must be chosen to suit the expected violence of the explosion (which depends on many factors, particularly the material involved).

The purpose of Dust-Expert is to bring together this widely distributed knowledge and make it available to plant designers and safety inspectors. It is clearly a safety-related application and thus requires a safety case to establish a target for the reliability of the system and provide evidence that it is achieved. Our analysis of the contribution that errors in Dust-Expert could make to explosions showed that if fewer than 1 calculation in 700 led to a dangerous design error, plant designed using it would be safer than the current UK average. This places the application at safety integrity level (SIL) 2 for demand-driven applications [IEC97]. A dangerous error could result from Dust-Expert giving an answer for a vent size that was significantly smaller than it should be (perhaps 10% or more), but not implausibly small. (A suggested size of $1mm^2$ would probably be queried by the user.) Program crashes which do not produce an answer are not dangerous, but would not satisfy other requirements for reliability and usability.

This is a comparatively modest requirement compared with, say, continuously operating avionics systems which must achieve less than one failure in one billion hours of operation. In particular, it makes it feasible to provide statistical evidence that the system achieves its target safety level. [PAM91] shows that 2300 representative tests performed without error provide 90% confidence that the error rate is no more than 1 in 700. However, we need confidence that the development process will produce software that will pass the test: if one error is found it we must complete the 2300 tests and do another 2300 to achieve the same level of confidence in addition to the work of correcting the error [LW97]. Nor can we be satisfied just by achieving a target failure rate. Safety related systems should be developed to achieve failure rates that are as low as reasonably practicable (that is, as low as can be achieved without disproportionate increase in cost). This is usually referred to as the ALARP principle. Also, should a disaster occur due to a failure of the system, it is advisable to have the defence that best practice has been used.

To address these concerns, the development process that we adopted began with the production of a relatively abstract model of the intended system in VDM (the *specification*). The discipline of a formal notation, combined with the consistency checks in VDM, imposes extra costs at this stage compared with an informal design, but returns benefits in uncovering ambiguities in the requirements and loose ends in the design at an early stage, improving software quality and hence raising our confidence that the statistical tests would be passed. We expected that the use of formal methods would at worst cost us no more overall, and so their use is mandated by the ALARP principle. Also, the relevant standards for safety related systems [IEC97, Def97] strongly recommend formal approaches at higher SILs, and it would be difficult to defend not using them at lower levels where they do not add cost.

The next stage of the process was a rigorous development of a VDM model of the implementation. This was then transliterated into Prolog, the target implementation language. Using Prolog had the advantage of keeping the implementation at a fairly high level and hence shortening the development chain. (In fact, there is a single step from original design to implementation, combining a number of simple development steps.) By separating the development from the shift in language, we were able to consider two rather different and potentially tricky steps independently. In the first step we address the question of whether the new definition implements the old within a single notation (and draw on many standard results in that setting) and in the second we consider the equivalence of constructs in two different languages. Throughout the VDM development we used the IFAD VDM toolbox [ELL94] for syntax and type checking, and the Prolog development was supported by static analysis for undefined and unused predicates and variables.

There is no formal proof in the process. To undertake this rather than rigorous argument would have imposed substantial extra cost, which did not seem necessary at our target SIL where the main evidence of safety was to be statistical. (At higher SILs, a fully formal approach demonstrating absence of defects would be the only plausible way of establishing the safety of the system.)

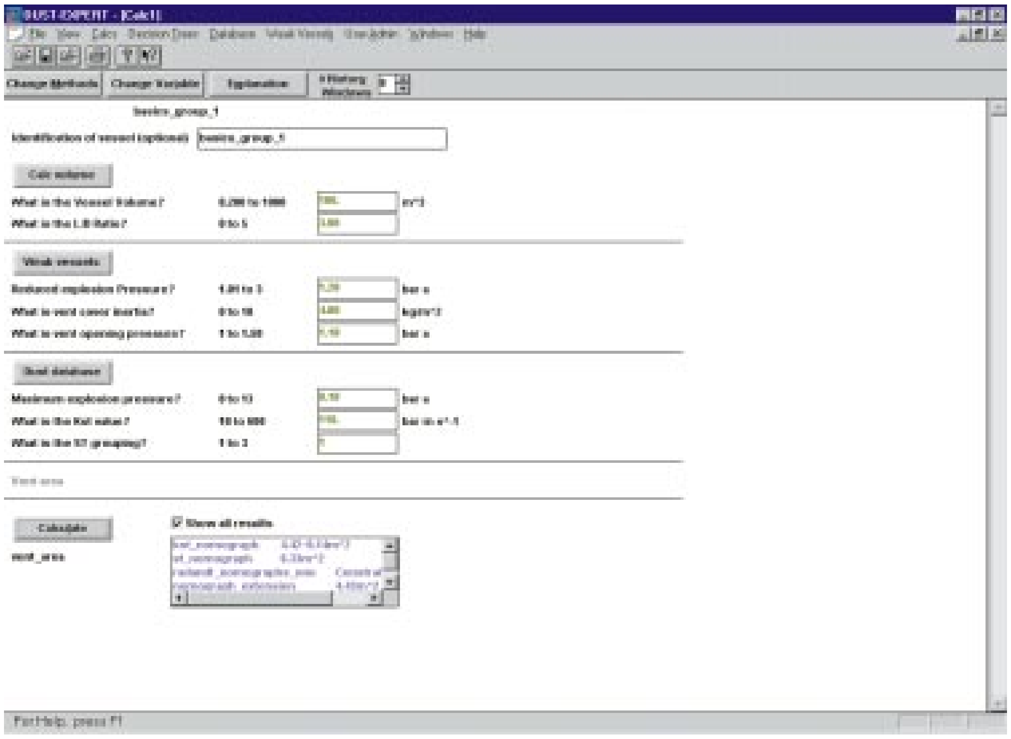
The interesting aspect of the arguments for the use of a formal notation and rigorous development is that they apply just as much to the interface as they do to the core functions of the system. The correctness of the interface in presenting the results is just as important as the correctness of the core code in computing them. It will be exercised by the statistical testing. (One of the attractions of statistical tests is that they include the whole system, including the off-the-shelf components such as the Prolog system and Windows interface, which formal arguments about the code do not address.) The interface is complex enough that we can expect to gain reliability and reduce costs by the detailed consideration of the design at an early stage, and cannot be confident that traditional development techniques would deliver a sufficiently reliable product. The ALARP principle and the need for best practice extend throughout the system. We were thus sceptical of the usual recommendation to restrict the use of formal methods to the core functions of the system. To extend their use, we needed a way to approach interface definition in VDM, and this is what we shall describe in the rest of the paper.

3 Specifying the interface

As a running example, let us consider a simplified version of the DUST-EXPERT interface to calculations of things like vent sizes. In many cases, the calculations are based on the piecewise fitting of functions to experimental curves measured under different conditions. Each piece is represented by an *option*, which defines the function to evaluate and places conditions on its input variables that define the range of values it covers. A *method* combines options to cover an entire set of curves, and may impose further conditions to define its applicability. There are several ways in which the same value can be calculated, so methods are combined into groups. To carry out a calculation, then, we must choose a group and the methods to be used from within it, and provide values for the input variables of their options. Each method then selects its first applicable option and returns the result it gives. If the method conditions are not met or no applicable option can be found, it returns a failure code. The group will return the results for all the chosen methods.

The abstract model of the core of the system defines types for options, methods and groups. The system state includes mappings from the names of groups, methods and options to their definitions. There is a function, `evaluateGroup`, which takes a group name and set of method names, together with a map from variable names to values (a *binding*), and returns a map from methods to results. This function can be seen as a definition of an interface, since it says that the system must have the capability of yielding certain results if given certain inputs.

It is not, however, a *user* interface that this defines. The Windows user interface for Dust-Expert calculations is shown in Figure 1. It presents



basins_group_1

Identification of vessel (optional) basins_group_1

What is the Vessel Volume?	0.200 to 1000	[illegible]	m^3
What is the L/D Ratio?	0 to 5	[illegible]	
Reduced explosion Pressure?	1.01 to 3	[illegible]	bar a
What is vent cover inertia?	0 to 10	[illegible]	kg/m^2
What is vent opening pressure?	1 to 1.20	[illegible]	bar a
Maximum explosion pressure?	0 to 13	[illegible]	bar a
What is the Kst value?	10 to 600	[illegible]	bar m s^-1
What is the St grouping?	1 to 3	[illegible]	

vent_area

```
kst_nomograph              [illegible]
st_nomograph               [illegible]
radandt_nomographs_min     [illegible]
nomograph_extension        [illegible]
```

Figure 1: The Dust-Expert user interface for calculations

users with a representation of the input values and the results of the last calculation on them, if any, and provides operations to edit the variable values one by one and to recalculate the results, as well as to change the set of methods. (The group is fixed when the calculation is created.) Because it allows the input to be built up through a series of actions rather than presented as a single value, it necessarily has a state. This can be modelled in VDM as an element of the following type:

$$\begin{aligned}
\mathit{Feature} :: \quad \mathit{group} &: \mathit{Identifier} \\
\mathit{methods} &: \mathit{Identifier}\text{-}\mathbf{set} \\
\mathit{binding} &: \mathit{Identifier} \xrightarrow{m} \mathit{Value} \\
\mathit{results} &: [\mathit{Identifier} \xrightarrow{m} \mathit{ValueError}]
\end{aligned}$$

with an invariant that the results are those calculated by *evaluateGroup* from the binding when not **nil**.

We would expect eventually to produce another formalisation of the window, based on the concrete data types that the Windows user interface supports. These include character strings in static boxes and edit boxes, and lists of strings in list boxes. (Our choice of controls will be influenced by the operations that the user can perform on them.) Our design for DUST-EXPERT displays each variable that the group may need as a line in the window. The variable value is represented as a string. The internal identifier names are not very informative, but the system provides a function from variable names to suitable prompts. To guide the user further, we also display the bounds on allowable values and the units associated with the variable. The results map is shown as a list displaying each method and its result. The **nil** result is indicated by greying out the display (leaving the previous results visible, which can be useful). This can be formalised as a cross-product of the sequence (*result*) and a Boolean flag (*valid*). The group name appears at the top of the screen. Combining these constructions gives the implementation

$$\begin{aligned}
\mathit{FeatureView} :: \quad \mathit{group} &: \mathit{Identifier} \\
\mathit{display} &: \mathit{DisplayLine}^* \\
\mathit{result} &: \mathit{Answer}^* \\
\mathit{valid} &: \mathbb{B}
\end{aligned}$$

$$\begin{aligned}
&\texttt{inv-}\mathit{FeatureView}(\texttt{mk-}\mathit{FeatureView}(\mathit{group}, \mathit{display}, \mathit{result}, \text{-})) \quad \triangleq \\
&\quad \{d.\mathit{name} \mid d \in \mathbf{elems}\, \mathit{display}\} = \mathit{groupVariables}(\mathit{group}) \wedge \\
&\quad \{r.\mathit{source} \mid r \in \mathbf{elems}\, \mathit{result}\} \subset \mathit{groupMethods}(\mathit{group})
\end{aligned}$$

where we also define

$$\begin{aligned}
\mathit{DisplayLine} :: \quad \mathit{name} &: \mathit{Identifier} \\
\mathit{question} &: \mathit{char}^* \\
\mathit{bounds} &: \mathit{char}^* \\
\mathit{units} &: \mathit{char}^* \\
\mathit{cval} &: \mathit{char}^*
\end{aligned}$$

$$\begin{aligned} \mathit{Answer} :: \ & \mathit{source} : \mathit{Identifier} \\ & \mathit{value} : \mathit{char}^* \end{aligned}$$

This is a slight abstraction of the actual view, where the group is displayed as a character string, and each *Answer* is a string where a representation of the source is concatenated to the value.

Since the user will only see the concrete view, it must accurately represent the underlying calculation state. This can be achieved by making it a data reification of that state. We must then consider the relevance of the standard VDM conditions for data reification to user interfaces. VDM says that there should be an *abstraction function* from the reification to the abstract type. We can define this component by component for the view. To reconstruct the binding, we parse *cval* strings to obtain values, and pair them with the names. The rest of the line is ignored. (Interface design tends to add redundant information in this way.) If the string is empty, the line is ignored. The resulting sequence of pairs is then converted to a map: this is a standard abstraction. The results are abstracted similarly, and abstracting the group name is trivial.

We observe that there is nothing obvious to derive the methods from. Technically, this corresponds to a violation of the VDM adequacy condition for reifications, which requires the abstraction function to be surjective. In a normal data reification, this ensures that every abstract value has a representation. In a user interface, it tells us that the whole state is presented explicitly, a desirable property known as *predictability.* The obvious way to present the active methods as is a list of the group methods with the active ones selected: this is easy for the user to edit to select a new set. This can be formalised as a sequence of identifiers with a subset of selected ones, and the abstraction can be retrieved by throwing away the sequence. (Again, information has been added to support the user.) The window is already rather full, but we can split the presentation across screens (by putting the list of methods in a pop-up dialog, for example) provided we have a way to make all the screens visible when needed (in this case, the `Change methods` button). We can justify not showing the methods to be used on the main screen on the grounds that the methods last used (almost the same thing) are shown in the results. Taken together, the two screens are an adequate representation of the abstract state provided the set of methods in the calculation is restricted to the group methods and the binding should be restricted to the group variables. Both conditions are reasonable. Formally, we should have the invariant

$$\begin{array}{l} \text{inv-}\mathit{Feature}(\text{mk-}\mathit{Feature}(\mathit{group}, \mathit{methods}, \mathit{binding}, \text{-})) \quad \triangleq \\ \qquad \mathit{methods} \subset \mathit{groupMethods}(\mathit{group}) \land \\ \qquad \mathbf{dom}\, \mathit{binding} \subset \mathit{groupVariables}(\mathit{group}) \end{array}$$

VDM also requires the abstraction function to be total. This means that whatever the user sees has an interpretation as a state of the calculation. This too is an important aspect of an interface. In this case it raises the issue of what happens when invalid character strings entered by the user are parsed to obtain new variable values. The obvious choice is to omit the variable from the binding. The results are produced by the program rather than the user, so all strings in this part of the interface should be parseable. Our intuition that this is enough for the abstraction to be defined in every reachable state is justified formally in [Cle94].

Although the formalisation of the user view can be seen as a reification of the calculation state, it is not presented in this way in the DUST-EXPERT development. (This is why we emphasised the relevance of the reification conditions to user interface design above.) Rather, the design maintains both the state of the calculation and the user's view of it. This allows us to close the view while retaining the calculation, and hence to conserve Windows resources in calculations with many subcalculations. There are other advantages, too, in this separation of model from view which is common in object-oriented systems [KP88].

As a consequence, the connection between the internal state and the view is expressed not as an abstraction function connecting two definitions, but as part of a single definition. We can think of it as an invariant, although it is not, technically, an invariant in the strict sense of VDM for reasons we shall see. For now, let us consider how the connection should be defined. Certainly the internal state and view are related by the abstraction function described above, but the link can be tightened. When a view is opened on a calculation, we expect to define exactly what appears: empty strings for unbound variable fields and a particular number of significant figures for a value, for example. Applying the abstraction function to the view thus created must yield the original state if the invariant is to be established. Formally, we have a presentation function that is an injection of calculation states into views, and has the abstraction function as an inverse. We would like the view to show the canonical form of the state defined by the presentation function after every operation, because consistent presentations are easier to read. We thus formalise the connection between calculation state and calculation view in terms of this rather than the abstraction function. (For DUST-EXPERT calculations, the presentation function is only loosely defined, to say that a **nil** result is presented as a false *valid* flag, leaving the results list undefined.) The invariants identified on the internal state and view must also hold.

This should be invariant as far as the user is concerned, in that it should hold between all user operations. It is not an invariant in the VDM sense because it does not hold between VDM statements, or even between all operations. We therefore express it as a predicate and add it explicitly to the preconditions and postconditions of the user interface operations.

This gives us an interesting view of operations. All begin with a user action on a window. Interacting with an edit box, a list box or other controls typically changes the value of the control — that is, the state of the view — before sending a message. This is formalised by defining a parameterless operation which selects a suitable control and makes a suitable change non-deterministically. For example, consider the effect of entering a new value on a calculation screen. The state of the system supports many of these, and many underlying calculations, linking views to their models through a function. (Provided we can open a window on any active calculation, we have presentability as a property of the whole system.)

```
**state** $State$ **of**
    $features : FeatureRef \xrightarrow{m} Feature$
    $fviews : ViewRef \xrightarrow{m} FeatureView$
    $fvmodel : ViewRef \xrightarrow{m} FeatureRef$
```

The interface operation is *ENTER*. We assert that it preserves the invariant. The user input is simulated by choosing an arbitrary window, variable line number, and text to update it with.

```
$ENTER\colon\ () \implies ()$
$ENTER() \equiv$
  let $fview \in$ dom $fviews$ in let
      $active \in variableIndices(fview)$
  in (
    let $text\colon chars^*$ in $fviews(fview).display(active).cval := text$;
    $INVALIDATE\_FEATURE\_ANSWER(fvmodel(fview))$;
    $CHANGE\_VALUE(fview, active)$
  )
pre   $validState()$
post  $validState()$;
```

Once the change to the control has been made, the invariant no longer necessarily holds, so further operations are called, passing the chosen view, control and similar information through the parameters These operations update the internal state and view to restore the invariant. In this case, the result is marked as invalid because it no longer corresponds to the binding.

```
$INVALIDATE\_FEATURE\_ANSWER$: $FeatureRef \implies ()$
$INVALIDATE\_FEATURE\_ANSWER(fref) \equiv$
(
  $features(fref).result$ := nil;
  if $fref \in$ rng $fvmodel$ then let $fview = (fvmodel^{-1})(fref)$ in
    $DISPLAY\_INVALID\_FEATURE\_ANSWER(fref, fview)$
);
```

The answers must be greyed out on the display, which is achieved by *DISPLAY_INVALID_FEATURE_ANSWER*. We also need to restore the connection between the displayed values and the internal binding, which is done by *CHANGE_VALUE*.

```
$CHANGE\_VALUE$: $FViewRef \times [\mathbb{N}_1] \implies ()$
$CHANGE\_VALUE(fview, active) \equiv$
  if $active \neq$ nil then (
    let $fref = fvmodel(fview)$ in let
      $text = fviews(fview).display(active).cval$,
      $name = fviews(fview).display(active).name$
    in (
      if $isWhiteSpace(text)$ then $RETRACT\_BINDING(fref, name)$
      else let $val = parseValue(text)$ in
        if $isError(val)$ then $RETRACT\_BINDING(fref, name)$)
        else $UPDATE\_BINDING(fref, name, val)$;
      $DISPLAY\_FEATURE\_BINDING(fref, fview)$
    )
  );
```

Formally, the invariant provides a strong check on the correctness of the operation definitions through the obligation to prove satisfiability of the specification. (For these explicit operation definitions, we can interpret this as a requirement to show that the explict state change satisfies the postcondition, given the precondition.) Informally, we can use parts of the abstraction function to suggest how to interpret the view while updating the state (suggesting in this case the use of *parseValue*) and parts of the presentation function to define how to reflect changes in the state by updating the view.

Consideration of one user operation requires some rethinking of the invariant. The edit boxes can be altered one character at a time. We do not particularly want to parse the string and update the binding on each change, because the intermediate strings do not represent useful values and will not necessarily be in canonical form. Until we move off the field being edited, then, we do not want the invariant just developed to hold. We could just omit any claim that an invariant is satisfied from the formal definition of the editing operation, but as we have seen the invariant is useful enough to make it worth preserving. We can weaken it until it holds in this case too by adding a *valid* flag to *DisplayLine*. The string in *cval* must then be a reflection of the underlying binding only if *valid* is true. (We expect it to be false for at most one variable at a time.) To preserve the new invariant, fields are marked invalid as they are edited and should be marked valid when the value is accepted and they are redisplayed. We expect components of the view to be visible, and we can make this new one apparent by changing the colour of the text so that the user has a specific warning that a value is not yet accepted by the system.

One final aspect of the view that we have not considered is the buttons. Buttons when pushed send messages, and as we have seen the meanings of messages are defined by parameterless operations. Pushbuttons do nothing but send messages: they return to their original position when released so the push represents an event rather than a change of state. Check boxes, radio buttons and the like do have state that changes and should be modelled in the view, typically as a Boolean or enumerated type. In addition, all buttons, like other controls, can be greyed out or hidden completely. For a pushbutton, this change in state has the effect of making the operation it invokes unavailable. We can model this state as a Boolean in the view, and the invariant should define how it is derived from the rest of the state of the system: the condition should reflect when the associated operation should be available. For example, the *Change methods* button is active only if there is more than one method in the group. In this case, the operation would be defined but pointless for a single method. In other cases, the operation itself is partial. For example, in the database component of DUST-EXPERT, we can only view a selected record if a record has been selected. This can be expressed as a precondition on the *View* operation, and the precondition is then the defining condition for the button in the invariant. Then as long as the invariant holds (which should be whenever a user operation can be performed) the operation can only be invoked if its precondition holds.

4 Development and implementation

The goal of the implementation step was to go from the specification of the system, where we emphasised directness of definition and made full use of the expressive power of VDM, to an equivalent VDM definition whose elements have natural counterparts in Prolog. For the most part, this involved replacing the sets and maps of the specification with lists subject to invariants that elements of sets and members of the domain in maps are not repeated. This is a standard reification which needs little specific justification, although in some cases replacing iteration over a map domain by iteration over the pairs in a sequence made the algorithm look rather different. Sets and maps in the states can normally be left to be implemented as Prolog relations. The translation of the operations is supported by a library of set operations on sequences.

A second component of the implementation step was the replacement of the (relatively few) uses of property-based definition by algorithms. The final component, and the only one specific to the interface, was the splitting of the definition of the view state into two parts, one (the *implicit* state) to be represented by the states of the various controls in the windows, and the other (the *explicit* state) to be maintained by the Prolog.

In the previous section, we took the position that the components of the view state should be visible to the user, since if they carry useful information this should be manifest. However, we can often find exceptions where information must be associated with the view but does not need to be presented because the user can infer it. The *name* field of *FeatureView* is an example. It does not need to appear on the actual view, because the prompt text identifies more clearly what is to be input. In principle it is unnecessary in the state, because we could assume that each variable has its own prompt and retrieve the binding from the prompts and *cval* fields, but having the name in the state makes the abstraction function much easier to define.

We may also have data associated with the buttons to parameterise the operations they invoke. For example, calculation views can contain database lookup buttons (such as *Dust Database* in Figure 1), which provide an alternative to user entry for obtaining values for variables. A particular database lookup needs to know which part of the database to use, which variables to find values for, and which database fields to find them in, as well as a caption to describe the effect of the button. This caption will normally indicate the database used, and the variable prompts in the calculation and the database should provide the user with the link between the sets of variables. Again, then, we can manage without showing these values in the view.

It is very convenient to be able to define a calculation view state in the specification which contains all the data we want, irrespective of whether it is to be presented, and delay the partitioning to the implementation stage. Such a split is easily justifiable as a data reification, though establishing the preservation of presentability needs further arguments like those above. (In essence, we reconsider adequacy for just the implicit part of the state.) This formal treatment assumes that it is possible to read values back from their implicit storage in the view. In most cases this is possible (and efficient enough to be practical), and it is necessary when the user can edit the control. In many cases we do not actually make use of the assumed capability: we do not need to read the calculated results back from the screen, for example. In a few cases where reading back is necessary and not possible or not convenient, we can duplicate the information in the explicit state, tying together the values with an invariant.

Prolog, as a relational language, may seem a strange choice as the target for a translation from VDM, which combines functional and imperative aspects. Functional languages would seem to offer a better technical match, at least to the functional parts, and there are various more or less principled ways in which they can support state. However, we needed a stable (hence commercial) language implementation running under Windows and supporting the Windows interface, and it was easier to find a Prolog implementation that fulfilled these requirements. (We used LPA Prolog.)

In practice, the conversion from VDM to Prolog is relatively straightforward. For example, the VDM shown in Figure 2 is the implementation of the invalidation of the current feature answer, specified earlier in a simplified form. The details of what the operation does are unimportant: the aim is to compare the VDM with the Prolog of Figure 3. For each VDM construct in

```
$INVALIDATE\_FEATURE\_ANSWER$: $FeatureRef \Longrightarrow ()$
$INVALIDATE\_FEATURE\_ANSWER(fref) \equiv$
  let $binding = features(fref).binding$,
      $target = features(fref).target$,
      $answer = features(fref).result$,
      $chpos = features(fref).chpos$
  in (
    if $chpos =$ len $features(fref).history + 1$ then
      $features(fref).history :=$
        $features(fref).history \curvearrowright [mk\_Result(binding, target, answer)]$;
    if $chpos \neq$ nil then (
      $features(fref).result := <UNDEFINED>$;
      $features(fref).chpos :=$ nil;
      if $fref \in$ rng $fvmodel$ then let $fview = (fvmodel^{-1})(fref)$ in
        for $i = 1$ to $historylength$ do
          if $fviewHistoryPosition(fview, i) = chpos$ then
            $DISPLAY\_FEATURE\_HISTORY\_SELECTION(fref, fview, i, chpos)$;
      $RETRACT\_BINDING(fref, target)$;
      $ADJUST\_BINDING\_FOR\_METHODS(fref)$;
      $modified := modified \cup \{featureCalcRef(fref)\}$;
      if $fref \in$ rng $fvmodel$ then let $fview = (fvmodel^{-1})(fref)$ in
        $DISPLAY\_INVALID\_FEATURE\_ANSWER(fref, fview)$
    )
  );
```

Figure 2: The VDM version of *INVALIDATE_FEATURE_ANSWER*

the implementation we can define a standard translation. Functions become predicates with an extra parameter: these will be invoked with this parameter a variable and the others ground terms. Conditionals are translated using the Prolog `P -> Q ; R` construct, and case expressions and the various iterators are defined by subsidiary Prolog predicates of standard forms, using recursion in the case of the iterators. When we come to translate types, the typeless nature of Prolog makes it easier to handle VDM's unconventional union types than it would be in a strongly typed functional language. The main complication is the need to introduce variables to name intermediate values explicitly: this, and the lack of any form of local definition within a

```
methods_view_INVALIDATE_FEATURE_ANSWER(FRef) :-
  methods_view_State_features_fref_binding(FRef, Binding),
  methods_view_State_features_fref_target(FRef, Target),
  methods_view_State_features_fref_result(FRef, Answer),
  methods_view_State_features_fref_chpos(FRef, CHPos),
  methods_view_State_features_fref_history(FRef, History),
  length(History, HHLen),
  ((HHLenP is HHLen + 1, CHPos = HHLen) ->
    (append(History,[mk_methods_view_Result(Binding,Target,Answer)],
            HistoryP),
     modify_methods_view_State_features_fref_at_history(FRef,HistoryP))
  ; true),
  (CHPos \= nil ->
    (modify_methods_view_State_features_fref_at_result(FRef,undefined),
     modify_methods_view_State_features_fref_at_chpos(FRef,nil),
     (methods_view_State_fvmodel(_, FRef) ->
       (methods_view_State_fvmodel(FView, FRef),
        methods_view_historylength(HLen),
        methods_view_INVALIDATE_FEATURE_ANSWER_for(1,HLen,FRef,
                                                  FView,CHPos))
     ; true),
     methods_view_RETRACT_BINDING(FRef, Target),
     methods_view_ADJUST_BINDING_FOR_METHODS(FRef),
     methods_view_featureCalcRef(FRef, CRef),
     modify_methods_view_State_modified_at_calc(CRef),
     (methods_view_State_fvmodel(_, FRef) ->
        (methods_view_State_fvmodel(FView, FRef),
         methods_view_DISPLAY_INVALID_FEATURE_ANSWER(FRef,FView))
     ; true))
  ; true).
```

Figure 3: The Prolog version of *INVALIDATE_FEATURE_ANSWER*

```
methods_view_INVALIDATE_FEATURE_ANSWER_for(I,HLen,_,_,_) :-
  I>HLen.
methods_view_INVALIDATE_FEATURE_ANSWER_for(I,HLen,FRef,
                                           FView,CHPos) :-
  I =< HLen,
  (methods_view_fviewHistoryPosition(FView, I, CHPos) ->
    methods_view_DISPLAY_FEATURE_HISTORY_SELECTION(FRef,FView,
                                                   I,CHPos)
  ; true),
  IP is I+1,
  methods_view_INVALIDATE_FEATURE_ANSWER_for(IP,HLen,FRef,
                                             FView,CHPos).
```

Figure 3: The Prolog version of *INVALIDATE_FEATURE_ANSWER* (cont)

clause, means that care is needed to avoid introducing variable name clashes.

The result is not elegant Prolog: it makes too little use of pattern matching in clauses. It is not always efficient Prolog: observe how the relation implementing *fvmodel* is used four times, once to test membership in the range and once to apply the inverse function each time we determine the view associated with a calculation. However, it does have a direct correspondence with the VDM.

The most interesting feature of the translation is the handling of state. For each field *field* of each composite type *type* we define predicates *type*_*field*`(Record, Value)` and `modify_`*type*`_at_`*field*`(Record, Value, RecordP)` to access and update the record respectively. (These, like all predicates, are prefixed by the module name.) For access to the state, we define predicates `State_`*comp*_*reference*_*field*`(Ref, Value)` and `modify_State_`*comp*_*reference*`_at_`*field*`(Ref, Value)`. These look up and modify the value of the designated field of the calculation or view state mapped to in state component *comp* by reference `Ref` (usually called *reference* in the specification). For the state of the calculation (*methods_view_State_features*) and the explicit view state, the maps are implemented as relations and modification is implemented by retracting the current association between reference and state value, modifying the state value, and re-asserting the connection. This is fast enough for the purpose. The implicit state accesses and updates are implemented by suitable calls to the Windows interface. The exact nature of the state implementation is thus hidden from most of the Prolog code behind the abstraction of the lookup and assignment of structured variables. Using VDM rather than Prolog for the design of the system has the great advantage of a direct treatment of state: the translation of the state definition and state manipulations accounts for most of the increase in code size (about 35%) on going from VDM to Prolog.

When moving from one language to another, there is always a concern that an apparently correct translation is not actually correct because of a subtle difference in the semantics of two apparently equivalent notations. In practice we have encountered relatively few instances of this, perhaps because the translation only involves the simpler features of the two languages. One of the problems we did find arose in this translation. The expression $\mathit{chpos} = \mathbf{len}\,\mathit{features}(\mathit{fref}).\mathit{history} + 1$ is defined and false in VDM when *chpos* is **nil**. The naive translation to Prolog is `CHPos =:= HHLen + 1`, exploiting the Prolog mathematical comparisons, but this fails because Prolog "typechecks" the arguments of the `=:=` relation. The proper typeless translation is `HHLenP is HHLen + 1, CHPos = HLen`.

The Windows interface called by the Prolog is provided by a thin layer of C++ which calls the Microsoft interface library. The VDM provides a specification for what this must do (it must simulate assignment and lookup) but the code was not developed rigorously from this. Our justification is that this property is simple enough to be communicated informally, and that the implementations are all relatively simple straight-line code. The main area of doubt was in the exact specification of the Microsoft library, no formal definition of this was available, and developing one would certainly have imposed a disproportionate cost.

5 Comparison with other work

The tradition of defining only the functional core of interactive systems formally may have begun, inadevertently, with [Suf82]. One of the first papers to address user interfaces directly in a model based formal setting is [Bow92], which provides an abstract model of a small part of the X windows system. This, however, is essentially a functional description of a software system which happens to provide windows, rather than the description of the interface aspects of an application.

[SH90] gave an early formal description of interactors as operations acting on a state. Foley and van Dam categorised aspects of interface description as lexical (to do with the detailed user actions of the interface, such as mouse clicks), syntactic (to do with the allowable sequences of actions), and semantic (to do with the effects of actions): this is described in greater detail in [Sch92]. The remaining work to be cited builds on this.

[HC96] presents an Object-Z specification of a web browser. It illustrates the way in which Z-like languages can construct large specifications from small components. (The module mechanism of VDM gives something of the same effect, but with coarser granularity.) The disadvantage of such specification by construction is that it does not encourage properties of the system as a whole to be made explicit as they are by the invariants in the approach described here. The operations describe the semantics of the interface as they do in the VDM. CSP is used to describe the syntactic aspects of the interface, which are also implicit in the preconditions of the operations. The CSP presentation is more abstract than the implicit definition in the same way that algebraic specifications of data types are more abstract than those of VDM or Z: the definition is in terms of the operations themselves rather than through some model. From the user's perspective, though, the distinction is between conditions in VDM or Z based on the state, which by design is shown in the interface, and conditions in CSP based on the past actions, which are not. The former seem to address more directly the issue of how the user will know which operations should be available.

The paper most directly concerned with the issues explored here is [DH95]. It too addresses the role of a standard notion of reification (in this case, that of Z [WD96]) in the definition of interfaces and the development of implementation. Formally the interface definition is given separately rather than combined with the internal state of the system, but this difference is less significant than it might seem, because the Z notion of reification is expressed as an invariant over a combined specification anyway. More interesting is that it sees the interface as an abstraction of the specification rather than as a reification. This appears to be a technique for framing the state to select just those parts which are to be presented in the interface: in the DUST-EXPERT specification this is implicit in the module structure. Treating the view as an abstraction leads to technical problems when the system and view are separately refined. We have been more explicit about the relation between the requirements on the invariant and properties of the interface, while [DH95] considers the wider issues of the user's view of the system as a tool for performing tasks.

There is a general assumption that the lexical details of the interface are to be abstracted from in the definition. Two aspects of the specification here suggest that this is not always so easy. We found a need to be able to select a subset of elements from a set to select the active methods. The most abstract specification of an interactor to achieve this would take the superset as a parameter and return a subset. If, though, we want to use multiple selection list boxes in the usual way, we should pass the currently selected set too: that is, we must at least decide that the abstract interactor has the ability to show the current selection. We also had a reminder that even primitive events may have a semantics: editing the value field to create the new value for a variable had to invalidate the results on the screen and flag the field to indicate that it no longer corresponded to the binding.

6 Summary

Dust-Expert is a safety-related system, for which a safety case had to be produced containing an assessment of the safety integrity level required (SIL 2) and evidence that this was achieved by the final system. This evidence was principally statistical, from *post hoc* testing, which meant that the development process adopted had to be capable of producing software of this quality. Process quality itself must also be addressed in the safety case. We therefore adopted a path of rigorous development from an initial, relatively abstract formal model (the specification) to a Prolog implementation with a Windows interface (attached via a C++ layer). This paper has described how the specification and development treated the interface to the system.

Our goal in modelling the interface was not to try to deal formally with its usability, but rather to address its correctness as a reflection of the internal state of the system. Some calculations have trees of subcalculations and associated windows five levels deep, with information propagating from the highest to the lowest and *vice versa*, so this correctness was not a trivial issue. The strategy chosen was the usual one for dealing with evolving states: we imposed an invariant that would hold between user operations and define what a correct presentation of an internal state would be. As we have seen, this essentially describes a function from the internal state to the presentation. We could then see the user disturbing the invariant by altering the view and could define operations by asking how the system should act to restore it.

The notion of the view as a reification of the state and the interface-oriented interpretation of the reification came later, and addressed at least the basic issues of the usability of the interface. The abstraction function provided a further guide to how the system should interpret values from the screen when restoring the invariant.

It is important to be specific about what the process did not include. It began with a model where the requirements could be recorded and design issues explored without too much implementation detail, rather than an attempt to give an abstract characterization of expert systems, or anything of that kind. In particular, we (eventually) felt unembarrassed about using explicit function and operation definitions to describe most of the system: at this level there were rarely advantages in a more implicit approach. The development is rigorous rather than fully formal, particularly in the translation from VDM to Prolog, because we believed that this was adequate for the target SIL.

The completed system contains about 16000 lines of Prolog and 17000 lines of C++, excluding comments. (The C++ may be conceptually simple but it is quite bulky.) The initial VDM specification was about 12000 lines. Productivity in terms of lines per day, where the days include all testing, documentation, and project meetings as well as design and coding effort, was substantially above industry norms for safety-related software. (The exact figures are commercially sensitive.) This was despite some reworking of the specification and implementations as our techniques for handling user interfaces developed: next time will be quicker. Our belief that the use of formal methods would be cost-effective was thus vindicated. We note that the productivity in C++ (where no formal specification was involved) was about twice that for the formal specifications plus Prolog. What this means is unclear. The requirements for the C++ were straightforward (even if their achievement in Windows was not) while the specification incorporated a large amount of design work which would have had to be done somehow. We can perhaps put an upper bound of a factor of two on the cost of the formal specification to the project.

Statistical and path coverage testing done systematically and independently after a period of informal testing revealed 31 faults, or less than one per thousand lines of code, none of which caused an invalid answer to be produced. A year of field experience has revealed about 10 more faults. All but two have been classified as minor. About half were in the Prolog and half in the C++, another statistic that is hard to interpret. (Are formal methods useless, or was the Prolog side of the problem harder to start with?) Most of the C++ faults are due to unexpected behaviour of the Windows interface. Half of the Prolog faults were due to mistranslation (including one serious one affecting the security of data). A mechanical translation would thus have achieved a significant gain in reliability, although at a significant cost. About one quarter represent design decisions that the users did not like. This leaves a small residue of problems that more proof at the specification level might have revealed, although the cost would have been disproportionate to the gains.

Overall, we believe that the use of formal definitions and rigorous translations throughout the project provided an appropriate balance of reliability and cost. We would therefore offer this as an addition to the steadily growing list of successful industrial applications of formal methods [FME].

References

[Bow92] J. Bowen. X: Why Z? *Computer Graphics Forum*, 11:221–234, 1992.

[Cle94] T. Clement. Comparing approaches to data reification. In *FME'94: Industrial Benefits of Formal Methods*, pages 118–133. Springer Verlag, 1994. LNCS873.

[Def97] Ministry of Defence. *Requirements for safety related software in defence equipment*, Def Stan 00-55 2nd edition, 1997.

[DH95] D.J. Duke and M. Harrison. Mapping user requirements to implementations. *Software Engineering Journal*, 1:13–20, 1995.

[ELL94] R. Elmstrøm, P. G. Larsen, and P. B. Lassen. The IFAD VDM-SL toolbox: a practical approach to formal specifications. *ACM Sigplan Notices*, 29:77–80, 1994.

[FME] FME applications database. At http://www.csr.ncl.ac.uk/-projects/FME/InfRes/applications.

[HC96] A. Hussey and D. Carrington. Using Object-Z to specify a web browser interface. Technical Report TR96-06, University of Queensland, 1996.

[IEC97] International Electrotechnical Commission. *Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC 61508 draft edition, 1997.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 2nd edition, 1990.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1:26–49, 1988.

[LW97] B. Littlewood and D. Wright. Some conservative stopping rules for the operational testing of safety critical software. *IEEE Transactions on Software Engineering*, 23:673–683, 1997.

[PAM91] D. L. Parnas, C. Asmis, and J. Madely. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32:189–198, 1991.

[Sch92] B. Schneidermann. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, 1992.

[SH90] B. Sufrin and J. He. Specification, analysis and refinement of interactive processes. In *Formal Methods in Human-Computer Interaction*, pages 153–200. Cambridge University Press, 1990.

[Suf82] B. Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*, 1:157–202, 1982.

[WD96] J. Woodcock and J. Davies. *Using Z: specification, refinement and proof*. Prentice Hall, 1996.