Incremental Construction of Large Specifications: Case Study and Techniques*

Neil Evans[1] and Michael Butler[2]

[1] AWE, Aldermaston, U.K.
[2] School of Electronics and Computer Science, University of Southampton, U.K.

Abstract. The RODIN project is an EU-funded project concerned with the provision of methods and tools for rigorous development of complex software-based systems. Ultimately, through the development of open-source tools and techniques, the project aims to make formal methods more appealing and accessible to industry. The project is driven by a number of case studies, each of which is designed to exercise the technology being developed and create methodologies for the future. In this paper we focus on the methodologies being developed in one of the case studies (the CDIS subset). This case study is based on a commercial air traffic information system that was developed using formal methods 14 years ago, and it is still in operation today. The key goals of our approach are to improve the comprehensibility of large specifications and to achieve a complete mechanical proof of consistency.

1 Introduction

The CCF[3] Display and Information System (CDIS) is a computerised system that provides important airport and flight data for the duties of air traffic controllers based at the London Terminal Control Centre. Each user position is a workstation that includes a page selection device (to select CDIS pages) and an electronic display device (to display the selected pages). The original system was developed by Praxis[4] in 1992 and has been operational ever since. This system is an example of an industrial scale system that has been developed using formal methods. In particular, the functional requirements of the system were specified using VVSL [5] — a variant of VDM [4]. The formal development resulted in about 1200 pages of specification documents and about 3000 pages of design documents. The reliability of the delivered system is encouraging for formal methods in large scale system development because the defect rate was a considerable improvement on other similarly sized projects [6]. However, no formal reasoning was applied to the specification.

This paper describes a case study of the RODIN project that is based on CDIS. Contemporary tool support has been used to develop a formal specification. The

* This research was carried out as part of the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) http://rodin.cs.ncl.ac.uk

[3] Central Control Function

[4] Praxis High Integrity Systems Ltd., U.K.

objective of the case study (that we shall refer to as 'the CDIS subset') is to derive a methodology for large scale formal development. Redeveloping an existing system also allows us to reflect on the lessons learned from the original development. Our aim in this paper is to demonstrate how we have attempted to overcome the lack of comprehensibility and formal proof of the original CDIS development by adopting a methodology that makes use of available tool support in an effective way. The rest of the paper is as follows: Section 2 gives an overview of the system on which the case study is based, Section 3 describes Event-B (which is our notation of choice), and Section 4 describes our methodology in detail. After defining an abstract specification of a generic display system, we give some example refinements to demonstrate how more airport-specific features of the CDIS subset are introduced into the specification in a stepwise manner. We discuss the merits of this approach in Section 8.

2 The CDIS Subset

In order to keep the case study manageable in the context of the RODIN project, a subset of the original CDIS has been carefully chosen for redevelopment [7]. However, rather than focusing on individual aspects of CDIS, a 'vertical slice' has been taken so that all of the interesting features of the system are covered (albeit in a lesser form). At the heart of the CDIS subset is the 'core specification' that gives the functional properties of the system, and shall be the focus of this paper. In addition to the core specification, there is a concurrency specification and a description of the user interface.

2.1 The Original Specification

The core specification is only one part of the overall CDIS documentation. It gives an idealised view of the entire functional behaviour of the system. (The design document states how this is actually realised.) In order to avoid ambiguity, in this paper we will often refer to the core specification as 'the original VDM specification'.

The core specification consists of a number of VVSL modules, each of which contains type, constant and state definitions. (The bulk of the specification is made up of Boolean functions that are used in the pre/post conditions of other definitions.) A module can import other modules so that the imported definitions are available in the importing module. This gives a VVSL specification its structure. This approach encourages a bottom-up development in which the overall specification emerges from the way in which its modules are combined.

The core specification of CDIS comprises 15 modules. However, we can identify three main parts (or contexts):

– Airport-related data. This concerns airport-specific values such as weather or runway information. The *Meta_data* module identifies the airport attributes and their value types. Functions are defined to update the values of the attributes. The *Airport_records* module declares a state variable that holds all of the actual values of the attributes.

– Page-related data. This gives a device-independent and data-independent record of the pages that can be displayed by CDIS. Types are declared to model the layouts of pages. Actual pages are held in the state variables declared in the *Pages* module.
– Display-related data. This concerns the physical devices that are used to retrieve and display information.

Other subsidiary modules such as the date/time module are concerned with other important features of CDIS. By far the largest module in the core specification is *EDD_displays* that contains the operations of the system. All of the modules listed above are imported by *EDD_displays* to enable the definition of the operations.

2.2 Conclusions Drawn

It is worth emphasising that the CDIS specification is necessarily complicated. Even though the core specification has been criticised for its complexity, it is unrealistic to expect any significant improvements in the size of a specification that captures all aspects of CDIS, regardless of the notation used. However, the bottom-up construction in VVSL forces a level of specification that is too detailed to get an appreciation of the overall system behaviour.

Too much complexity also precludes formal analysis. In order to reason about a specification formally, it is necessary to keep the level of detail as simple as possible. Otherwise mathematical proof becomes infeasible. Analysing monolithic specifications such as the CDIS core specification would be beyond the capabilities of contemporary formal methods tools without intense human intervention. This was not an issue during the original CDIS development because tool support was largely unavailable, and large-scale formal analysis was out of the question.

2.3 Ideal Specification vs. Reality

Another drawback of the original development is the lack of continuity from the specification to the design. In the *idealised* view of the core specification, updates are performed instantaneously at all user positions, whilst there is an inevitable delay in the actual system because the information must be distributed to the user positions. Hence, there is no natural refinement of the original specification (in the usual sense of the word) to the design. We are investigating more novel notions of refinement in order to find a suitable link between the two viewpoints. In this paper, however, we are specifically interested in the idealised view of the system.

3 Event-B

An abstract Event-B specification comprises a static part called the *context*, and a dynamic part called the *machine*. The machine has access to the context via a **SEES** relationship. All sets, constants, and their properties are defined in the context. The machine contains all of the state variables. The values of the variables are set up using the **INITIALISATION** clause, and values can be changed via the execution of *events*. Ultimately, we aim to prove properties of the specification,

and these properties are made explicit using the **INVARIANT** clause. The tool support generates proof obligations which must be discharged to verify that the invariant is maintained.

Events are specialised B operations [1]. In general, an event E is of the form

$$E \ \widehat{=}\ \textbf{WHEN}\ G(v)\ \textbf{THEN}\ S(v)\ \textbf{END}$$

where $G(v)$ is a Boolean guard and $S(v)$ is a generalised substitution (both of which may be dependent on state variable v)[5]. The guard must be true for the substitution to be performed (otherwise the event is *blocked*). There are three kinds of generalised substitution: *deterministic*, *empty*, and *non-deterministic*. The deterministic substitution of a variable x is an assignment of the form $x := E(v)$, for expression E, and the empty substitution is *skip*. The non-deterministic substitution of x is defined as

$$\textbf{ANY}\ t\ \textbf{WHERE}\ P(t, v)\ \textbf{THEN}\ x := F(t, v)\ \textbf{END}$$

Here, t is a local variable that is assigned non-deterministically according to the predicate P, and its value is used in the assignment of x via the expression F. Note that in this paper we abuse the notation somewhat by allowing events to be decorated with input and output parameters (and preconditions to type the input parameters) in the style of classical B [1]. However, semantically, they can be treated as **ANY** parameters.

In order to refine an abstract Event-B specification, it is possible to refine the model and context separately. Refinement of a context consists of adding additional sets, constants or properties (the sets, constants and properties of the abstract context are retained).

Refinement of existing events in a model is similar to refinement in the B method: a *gluing invariant* in the refined model relates its variables to those of the abstract model. Proof obligations are generated to ensure that this invariant is maintained. In Event-B, abstract events can be refined by more than one event. In addition, Event-B allows refinement of a model by adding new events on the proviso that they cannot diverge (i.e. execute forever). This condition ensures that the abstract events can still occur. Since the new events operate on the state variables of the refined model, they must implicitly refine the abstract event *skip*.

4 A Methodology for CDIS in Event-B

As stated above, we shall be concerned with an idealised view of the system, as modelled in the core specification. Thus, we model a system that has a centralised database from which information can be retrieved. In order to get a better overview of the entire system, we follow a top-down approach. At the top level, we ignore all of the airport-specific features to produce a specification describing a generic display system. Through an iterated refinement process, we introduce more features into the specification until all of the CDIS functionality is specified. This procedure is supported by the tool B4Free. At each step the tool generates a

[5] The guard is omitted if it is always true.

number of proof obligations which must be discharged in order to show that the models are consistent with their invariants. Since each refinement introduces only a small part of the overall functionality, the number of proof obligations at each step is relatively small (approximately less than 20).

4.1 Generic Display Context

The purpose of CDIS is to enable the the storage, maintenance and display of data at user positions. If we ignore specific details about what is stored and displayed then CDIS becomes a 'generic' display system. We begin by constructing a specification for a generic system (which will be, of course, somewhat influenced by the original VDM specification) and, through subsequent refinements, introduce more and more airport-specific details so that we produce a specification of the necessary complexity, and reason about it along the way. By providing a top-down sequence of refinements it is possible to select an appropriate level of abstraction to view the system: an abstract overview can be obtained from higher level specifications whilst specific details can be obtained from lower levels.

Meta Data Context. Rather than specifying individual airport attributes (such as wind speed) as state variables of a particular value type, two abstract types are introduced that correspond to the collection of attribute identifiers and attribute values. This allows us to represent the storage of data more abstractly as a mapping from attribute identifiers to attribute values.

CONTEXT *META_DATA*
SETS *Attr_id* **;** *Attr_value*
END

Pages Context. The pages of CDIS are device-independent representations of what can be displayed on a screen. Each page is associated with a page number, and each page consists of its contents.

CONTEXT *PAGE_CONTEXT*
SETS *Page_number* **;** *Page_contents*
END

Displays Context. At this abstract level, we model the physical devices with which the users interact with the system. However, we only need to acknowledge that each position is uniquely identified (by its *EDD_id*), each user position has a type, and each user position has a physical display. Some user positions are

'editors' which have the capability of manipulating data and pages.

CONTEXT *DISPLAY_CONTEXT*
SETS *EDD_id* ; *EDD_type* ; *EDD_display*
CONSTANTS *EDDs* , *EDIT* , *EDITORS*
PROPERTIES
$EDIT \in EDD\_type \land$
$EDDs \in EDD\_id \rightarrow EDD\_type \land$
$EDITORS \subseteq EDD\_id \land$
$EDITORS = EDDs^{-1}[\{ EDIT \}]$
END

Merge Context. By merging the previous three contexts (via a **SEES** clause), we can declare a function that can determine the actual display, given the appropriate information. In declaring this function, we use an unfamiliar syntax. In [2], we have proposed the introduction of a record-like structure to Event-B. This proposal does not require any changes to the semantics of Event-B, but it gives us a succinct way to define structured data. The declaration of *Disp_interface* in the **SETS** clause of the following context is an example of our proposed syntax

CONTEXT *MERGE_CONTEXT*
SEES *META_DATA* , *DISPLAY_CONTEXT* , *PAGE_CONTEXT*
SETS $Disp\_interface :: data : Attr\_id \rightarrow Attr\_value,$
$contents : Page\_contents$
CONSTANTS *disp_values*
PROPERTIES $disp\_values \in Disp\_interface \rightarrow EDD\_display$

The type *Disp_interface* is a record comprising two fields *data* (of type $Attr\_id \rightarrow Attr\_value$) and *contents* (of type *Page_contents*). This record type defines the interface to the function *disp_values*. The intention is that, given a database of values and the device-independent representation of a display, *disp_values* calculates what is actually displayed (i.e. it returns a value of type *EDD_display*). The benefit of using a record type is that it can be refined by adding extra fields (see [2] for more details). This is necessary because the actual display is dependent on parameters that are introduced during the refinement stages. The extension of record types through refinement allows us to modify the interface accordingly (an example of this is given in Section 5).

As in the original CDIS specification, the fact that we represent *disp_values* so abstractly does not undermine the value of the specification. The dynamic part of the specification (shown below) focuses on updating attributes and pages, and defines the pages selected at user positions.

4.2 The Abstract Model: A Generic Display

The variable *database* represents the stored data, and *page_selections* records the page number currently selected at a user position. Note that this is a partial function which means that user positions are not obliged to display a page. The variable *pages* is a partial function mapping page numbers and page contents. The variable *private_pages* holds the page contents of a page prior to release. This is

intended to model an editor's ability to construct new pages before they are made public. Finally, *trq* models the 'timed release queue' that enables a new version of a page to be stored until a given time is reached, whereupon it is made public.

MACHINE *ABS_DISPLAY*
SEES
META_DATA, *DISPLAY_CONTEXT*, *PAGE_CONTEXT*, *MERGE_CONTEXT*
VARIABLES *database* , *pages* , *page_selections* , *private_pages* , *trq*
DEFINITIONS
$inv \mathrel{\widehat{=}}$
$database : Attr\_id \rightarrow Attr\_value \wedge$
$pages : Page\_number \nrightarrow Page\_contents \wedge$
$page\_selections : EDD\_id \nrightarrow Page\_number \wedge$
$private\_pages : Page\_number \nrightarrow Page\_contents \wedge$
$trq : Page\_number \nrightarrow Page\_contents \wedge$
$\mathsf{ran}(page\_selections) \subseteq \mathsf{dom}(pages)$
INVARIANT *inv*
INITIALISATION *database* , *pages* , *page_selections* , *private_pages* , *trq* : (*inv*)

Note that, in addition to type information, the invariant insists that pages can be selected only if they have contents. We keep the model simple by initialising the system to be any state in which the invariant holds.

Almost all of the operations given below correspond to operations defined in the original VDM specification. One exception is the **VIEW_PAGE** operation that uses the *disp_values* function to output an actual display. This is a departure from the original VDM specification but, since outputs must be preserved during refinement, it forces us to ensure that the appearance of actual displays is preserved.

UPDATE_DATABASE models the automatic update of data via the stream of data coming from the airports (see [7]), and **SET_DATA_VALUE** models the manual update of values (by editors). **DISPLAY_PAGE** enables any user to select a new page to be displayed, and **DISMISS_PAGE** removes a page selection. **RELEASE_PAGE** makes a private page public, and **DELETE_PAGE** enables an editor to delete the contents of a page. In addition to the manual release of pages (via **RELEASE_PAGE**), pages can be released automatically at specific times. **RELEASE_PAGES_FROM_TRQ** models the timed release of pages. However, at this stage no notion of time exists in the specification. Therefore, this operation selects an arbitrary subset of the pages from *trq* to be released. This is refined when we introduce a notion of time (as shown in Section 5.1). The operations use common B operators such as function overriding $\lhd\!\!+$, domain subtraction

$\lhd$, and range subtraction $\rhd$.

UPDATE_DATABASE (*ups*) $\widehat{=}$
  PRE
    ups $\in$ *Attr_id* $\nrightarrow$ *Attr_value*
  THEN
    database := *database* $\Leftarrow$ *ups*
  END ;

SET_DATA_VALUE (*ei* , *ai* , *av*) $\widehat{=}$
  PRE
    ei $\in$ *EDD_id* $\wedge$
    ai $\in$ *Attr_id* $\wedge$ *av* $\in$ *Attr_value*
  THEN
    WHEN *ei* $\in$ *EDITORS* **THEN**
      database (*ai*) := *av*
    END
  END ;

DISPLAY_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    ei $\in$ *EDD_id* $\wedge$ *no* $\in$ *Page_number*
  THEN
    WHEN *no* $\in$ dom (*pages*) **THEN**
      page_selections (*ei*) := *no*
    END
  END ;

DISMISS_PAGE (*ei*) $\widehat{=}$
  PRE *ei* $\in$ *EDD_id* **THEN**
    WHEN
      ei $\in$ dom (*page_selections*)
    THEN
      page_selections :=
        { *ei* } $\lhd$ *page_selections*
    END
  END ;

ed $\longleftarrow$ **VIEW_PAGE** (*ei*) $\widehat{=}$
  PRE *ei* $\in$ *EDD_id* **THEN**
    ANY *di* **WHERE**
      ei $\in$ dom (*page_selections*) $\wedge$
      di $\in$ *Disp_interface* $\wedge$
      data (*di*) = *database* $\wedge$
      contents (*di*) =
        pages (*page_selections* (*ei*))
    THEN
      ed := *disp_values* (*di*)
    END
  END

RELEASE_PAGE (*no*) $\widehat{=}$
  PRE *no* $\in$ *Page_number* **THEN**
    WHEN
      no $\in$ dom (*private_pages*)
    THEN
      pages (*no*) :=
        private_pages (*no*) $\parallel$
      private_pages :=
        { *no* } $\lhd$ *private_pages*
    END
  END ;

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
  ANY *SS* **WHERE**
    SS $\in$
      Page_number $\nrightarrow$ *Page_contents* $\wedge$
    SS $\subseteq$ *trq*
  THEN
    pages := *pages* $\Leftarrow$ *SS* $\parallel$
    trq := *trq* $-$ *SS*
  END ;

DELETE_PAGE (*ei* , *no*) $\widehat{=}$
  PRE
    ei $\in$ *EDD_id* $\wedge$
    no $\in$ *Page_number*
  THEN
    WHEN *ei* $\in$ *EDITORS* **THEN**
      pages := { *no* } $\lhd$ *pages* $\parallel$
      private_pages :=
        { *no* } $\lhd$ *private_pages* $\parallel$
      trq := { *no* } $\lhd$ *trq* $\parallel$
      page_selections :=
        page_selections $\rhd$ { *no* }
    END
  END ;

5 Refinement

The abstract specification described in the previous section omitted many of the features that characterise CDIS. However, this made it possible to give a broad overview of the system, including its state variables and operations, within a few pages. Now we use this specification as a basis for refinement in which the omitted details are introduced. We introduce a notion of time so that we can add age information to attributes, and add creation and release times to pages.

5.1 Adding Time

In terms of the CDIS subset, there are two main reasons for adding time: each piece of airport data has an age which affects how it is displayed, and the version of each page that is displayed is also time-dependent. In this refinement we shall once again use our proposed syntax for record types [2].

Time Context. We begin by introducing a new context to the development. The set *Date_time* represents all of the different points in time. We also include a total ordering relation (*leq*) between these points.

CONTEXT *TIME*
SETS *Date_time*
CONSTANTS *leq*
PROPERTIES
$leq \in Date\_time \leftrightarrow Date\_time \land$
$\forall (a).(a : Date\_time \Rightarrow (a, a) : leq) \land$
$\forall (a, b).(a : Date\_time \land b : Date\_time \Rightarrow$
$((a, b) : leq \land (b, a) : leq \Rightarrow a = b) \land$
$((a, b) : leq \lor (b, a) : leq)) \land$
$\forall (a, b, c).(a : Date\_time \land b : Date\_time \land c : Date\_time \Rightarrow$
$((a, b) : leq \land (b, c) : leq \Rightarrow (a, c) : leq))$
END

Meta Data Context. In order to record the age of a piece of data as well as its value, we refine the *META_DATA* context by defining a record type *Attrs* with two fields *value* and *last_update*.

CONTEXT *META_DATA1*
SEES *META_DATA* , *TIME*
SETS *Attrs* :: *value* : *Attr_value*,
last_update : *Date_time*
END

Note that the range of *value* is of our original value type *Attr_value*. The gluing invariant of the refined model will ensure that the values of the entries in the refined database will match the corresponding entries in the original database (see Section 5.1). The field *last_update* (of type *Date_time*) records the time at which the value of the attribute was last updated.

This technique of 'wrapping' an abstract type in a refined type is a pattern that occurs frequently in our approach. In general, if $f \in I \rightarrow A$ is an abstract

collection formed from abstract type A and in a refinement we wrap A in a record $B :: a : A, \cdots$, then abstract variable f is replaced by $g \in I \rightarrow B$ with gluing invariant $f = g; a$.

Pages Context. We proceed by refining the pages context in a similar manner. We declare a record type *Page* with two fields: *page_contents* holds the structure of a page, and *creation_date* holds the time at which a page was created. Note that this has nothing to do with the time at which the page is released. In order to model the timed release queue faithfully, we must associate a release date with every page on the queue. By using our proposed syntax for record refinement [2], this is achieved by defining a subtype of *Page* (called *Rel_page*) whose elements have an additional field called *release_date*.

CONTEXT *PAGE_CONTEXT1*
SEES *TIME* , *PAGE_CONTEXT*
SETS
Page :: *page_contents* : *Page_contents*,
creation_date : *Date_time* **;**
Rel_page SUBTYPES *Page* WITH *release_date* : *Date_time*
END

Only pages of type *Rel_page* occur on the timed release queue. We shall see how the refinement of the operation **RELEASE_PAGES_FROM_TRQ** uses this additional information.

Merge Context. Now that we have introduced a notion of time, the display function *disp_values* can be augmented so that the ages of the data in the database is taken into account when they are displayed. We change the interface of the function by adding a new field to *Disp_interface* called *time*. The operator 'EXTEND' is similar to the 'SUBTYPES' operator, but it adds fields to *all* elements of the record type.

CONTEXT *MERGE_CONTEXT1*
SEES
META_DATA , *DISPLAY_CONTEXT* , *PAGE_CONTEXT* ,
TIME , *META_DATA1* , *PAGE_CONTEXT1* , *MERGE_CONTEXT*
SETS EXTEND *Disp_interface* WITH *time* : *Date_time*
END

Whenever the function *disp_values* is called, the current time can be passed as a parameter so that the ages of the relevant data can be determined. In CDIS, the colour of a value when displayed indicates its age (although this detail is not included at this level of abstraction).

The Refined Model: A Timed Display. The state variables and the operations of *ABS_DISPLAY* are refined to incorporate the timed context. Four of the variables in the refinement replace those of the abstract model. The invariant gives the relationship between these concrete variables and their abstract counterparts. For example, the abstract variable *database* is refined by *timed_database*, and they

are related because the attribute values held in *database* can be retrieved from the *value* fields in *timed_database*.

REFINEMENT *ABS_DISPLAY1*
REFINES
ABS_DISPLAY
SEES
META_DATA, *DISPLAY_CONTEXT*, *PAGE_CONTEXT*, *MERGE_CONTEXT*, *TIME* , *META_DATA1* , *PAGE_CONTEXT1* , *MERGE_CONTEXT1*
VARIABLES
timed_database ,
page_selections ,
timed_pages ,
private_timed_pages ,
dated_trq ,
time_now
DEFINITIONS
inv1 $\widehat{=}$
$timed\_database \in Attr\_id \rightarrow Attrs \wedge$
$timed\_pages \in Page\_number \nrightarrow Page \wedge$
$private\_timed\_pages \in Page\_number \nrightarrow Page \wedge$
$dated\_trq \in Page\_number \nrightarrow Rel\_Page \wedge$
$time\_now \in Date\_time \wedge$
$database = (timed\_database ; value) \wedge$
$\mathsf{ran} (page\_selections) \subseteq \mathsf{dom} (timed\_pages) \wedge$
$pages = (timed\_pages ; page\_contents) \wedge$
$private\_pages = (private\_timed\_pages ; page\_contents) \wedge$
$trq = (dated\_trq ; page\_contents) \wedge$
$\forall n . (n \in \mathsf{dom} (timed\_pages) \Rightarrow$
$(creation\_date (timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom} (private\_timed\_pages) \Rightarrow$
$(creation\_date (private\_timed\_pages (n)), time\_now) \in leq) \wedge$
$\forall n . (n \in \mathsf{dom} (dated\_trq) \Rightarrow$
$(creation\_date (dated\_trq (n)), time\_now) \in leq)$
INVARIANT *inv1*

Some of the operations affected by the refinement are shown below.

UPDATE_DATABASE (*ups*) $\widehat{=}$
PRE $ups \in Attr\_id \nrightarrow Attr\_value$ **THEN**
ANY *ff* **WHERE**
$ff \in Attr\_id \nrightarrow Attrs \wedge$
$\mathsf{dom} (ff) = \mathsf{dom} (ups) \wedge$
$(ff ; value) = ups \wedge$
$(ff ; last\_update) = \mathsf{dom} (ff) \times \{ time\_now \}$
THEN
$timed\_database := timed\_database \lhd\!\!+ ff$
END
END

The parameter to the **UPDATE_DATABASE** operation maintains its type,

but the **ANY** clause is used to construct a new mapping from *Attr_id* to *Attrs* all of whose *last_update* components are assigned to the current time (to reflect the time of the update). This mapping is used to overwrite the appropriate entities in the timed database. An interesting refinement occurs in the operation **RELEASE_PAGES_FROM_TRQ**. Rather than selecting an arbitrary subset of *trq* to release, *time_now* is used to select those elements whose release date is earlier than the current time. The released pages (held in *timed_pages*) are updated accordingly.

RELEASE_PAGES_FROM_TRQ $\widehat{=}$
LET SS **BE** $SS =$
$dated\_trq \rhd \{\ rp \mid rp \in Rel\_Page \wedge (\ release\_date\ (\ rp\ )\ , time\_now) \in leq\ \}$
IN
$timed\_pages := timed\_pages \lhd\!\!+ SS \parallel$
$dated\_trq := dated\_trq - SS$
END

Next, we introduce a new operation, called **CLOCK** that increases the current time by some unspecified amount. This operation models the passing of time.

CLOCK $\widehat{=}$
ANY $time\_next$ **WHERE**
$time\_next \in Date\_time \wedge$
$(time\_now\ ,\ time\_next) \in leq \wedge$
$time\_next \neq time\_now$
THEN
$time\_now := time\_next$
END

5.2 Another Refinement: Highlighting Manual Interaction

Several other aspects can affect the way values are displayed. One requirement of CDIS is that any manually updated values should be highlighted when they are displayed. Hence, with each attribute value, we need to record whether it was updated manually. Once again, we use our notion of record refinement to achieve this. The Boolean value associated with the new field *manually_updated* indicates whether the attribute's latest recorded value (accessed via the *value* field) has been input manually. In this case, we extend the record type *Attrs* as follows:

EXTEND *Attrs* WITH *manually_updated* : *BOOL*

If left unaltered, the existing B operations **UPDATE_DATABASE** and **SET_DATA_VALUE** would update this field nondeterministically, but we can refine them to assign meaningful values. In this case, the appropriate refinements are:

```
UPDATE_DATABASE ( ups ) ≙
 PRE  ups ∈ Attr_id ⇸ Attr_value THEN
   ANY  ff WHERE
     ff ∈ Attr_id ⇸ Attrs ∧
     dom ( ff ) = dom ( ups ) ∧
     ( ff ; value ) = ups ∧
     ( ff ; last_update ) = dom ( ff ) × { time_now } ∧
     ( ff ; manually_updated ) = dom ( ff ) × { FALSE }
   THEN
     timed_database := timed_database ⩤ ff
   END
 END


SET_DATA_VALUE ( ei , ai , av ) ≙
  PRE  ei ∈ EDD_id ∧ ai ∈ Attr_id ∧ av ∈ Attr_value THEN
    WHEN  ei ∈ EDITORS THEN
      ANY  aa WHERE
        aa ∈ Attrs ∧
        value ( aa ) = av ∧
        last_update ( aa ) = time_now ∧
        manually_updated ( aa ) = TRUE
      THEN
        timed_database ( ai ) := aa
      END
    END
  END
```

Since the operation **UPDATE_DATABASE** models the automatic update of values, all *manually_updated* fields are set to *FALSE*; **SET_DATA_VALUE**, which models a manual update, sets the *manually_updated* field to *TRUE*. Proving consistency of this form of superposition refinement is completely automatic.

6 Introducing Concrete Values

The ultimate aim of the refinement process is to construct a specification in which constants and variables are associated with concrete values and operations are defined to maintain the state accordingly. As part of this process, we have to separate an abstract type into subtypes. In the case of CDIS, this technique is used to introduce concrete attribute identifiers and value types into the specification. For example, the original VDM specification defines *Attr_value* as a union type made up of value types such as *Wind_direction* and *Wind_speed*. Although union types do not exist in B, we employ the separation technique to achieve the same goal. We define a new context in which *Wind_direction* and *Wind_speed* are defined subtypes of *Attr_value*[6].

[6] Even though *Attr_value* is not a record type, deferred sets such as this can be viewed as 'fieldless records'. By subtyping deferred sets, we can incorporate structure.

CONTEXT *META_DATAn*
SEES *META_DATA* , *META_DATA1*, $\cdots$
SETS
Wind_speed SUBTYPES *Attr_value* WITH *speed* : 0..99 **;**
Wind_direction SUBTYPES *Attr_value* WITH *dir* : 0..359 **;**
$\vdots$
END

Note that in this example we have refined *Attr_value* in two different ways. This is a reasonable thing to do (as discussed in [2]). The subtype *Wind_speed* has a single field *speed* which ranges from values 0 to 99. Similarly, *Wind_direction* has a single field *dir* which ranges from 0 to 359.

This is just one of the many refinements needed to introduce concrete types. A further refinement introduces *AV_WIND_SPEED*, *MIN_WIND_SPEED* and *MAX_WIND_SPEED* as concrete attribute identifiers (since they appear in the core specification). From these refinements, it is necessary to specialise the update operations to ensure that only values of the correct type update the database. As stated in Section 3, abstract operations can be refined into one or more concrete operations. Previously, **SET_DATA_VALUE** updated any attribute identifier with any attribute value. Now it must be refined to a collection of operations, each referring to specific attribute identifiers and attribute values.

7 Error Handling

With every operation that assigns a meaningful value to a concrete attribute identifier (such as **SET_WIND_SPEED_VALUE** above), we must also say what happens when an attempt is made to assign an out-of-range value. This situation gives us the opportunity to handle potential errors in the update of CDIS explicitly. We define additional operations to handle updates with such out-of-range values. This approach in Event-B corresponds to the built-in error handling capabilities of VVSL. As an example, consider the following operation fragment (which is another refinement of the **SET_DATA_VALUE** operation) that attempts to assign an out-of-range wind speed.

SET_WIND_SPEED_ERROR (*ei* , *ai* , *av*) $\widehat{=}$
PRE $ei \in EDD\_id \wedge ai \in Attr\_id \wedge av \in Attr\_value$ **THEN**
WHERE
$ei \in EDITORS \wedge$
$ai \in \{\ AV\_WIND\_SPEED, MIN\_WIND\_SPEED, MAX\_WIND\_SPEED\ \} \wedge$
$av \notin Wind\_speed$
THEN
$\vdots$

This operation only considers values outside the subtype *Wind_speed*. The body of the operation should handle this anomaly in an appropriate way (such as by ignoring the update and issuing an error message).

8 Conclusion

This paper represents a methodological contribution to the construction of large formal specifications. Our experience shows that incremental construction through iterative refinement makes it feasible to apply tool-based formal analysis to large specifications. This increases our confidence in the specification greatly and provides the basis for tool-based formal development of a design and implementation. We also believe that this approach makes a large formal specification more accessible and comprehensible both to those constructing the specification and to others.

A key factor in our success was the construction of good initial abstractions capturing the essentials of the system concerned. Such a skill is not easily transferable of course, but by providing good examples, such as the one here, we can help others understand how to construct good abstractions. Beside this, we have provided a number of concrete techniques which are transferable to the construction of other large formal specifications. In particular we made strong use of the developmental pattern of extending records to add additional information to information structures and to extend function signatures in in refinement steps. We identified and made use of a related pattern of wrapping abstract types within record structures in a refinement step, providing a standard pattern for a gluing invariant. We also made use of record subtyping and record extension to differentiate structures in refinements and to add attributes to abstract deferred sets. These techniques allow us to avoid unnecessary clutter at the more abstract levels. The techniques are easily supported by existing B provers and our experience is that the associated proof obligations are mostly automatically discharged.

Acknowledgements

The authors thank Anthony Hall, Cliff Jones and Joey Coleman for their valuable input into this research.

References

1. J. -R. Abrial: *The B Book: Assigning Programs to Meanings*, Cambridge University Press (1996).
2. N. Evans and M. Butler: *Proposal for Records in B*, accepted for publication, FM06.
3. A. Hall: *Using Formal Methods to Develop an ATC Information System*, Software, Vol. 13, No. 2, IEEE, March 1996.
4. C. Jones: *Systematic Software Development using VDM*, Prentice Hall, 1990.
5. C. A. Middleburg: *VVSL: A Language for Structured VDM Specifications*, Formal Aspects of Computing, Vol. 1, No. 1, Springer, 1989.
6. S. Pfleeger and L Hatton: *Investigating the Influence of Formal Methods*, Computer, Vol. 30, No. 2, IEEE, February 1997.
7. RODIN Deliverable D4: *Tracable Requirements Document for Case Studies*, http://rodin.cs.ncl.ac.uk/deliverables/D4.pdf, 2005.