A Proposal for Records in Event-B[*]

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 B method is a well known approach to the formal specification and development of sequential computer programs. Inspired by action systems, the B method has evolved to incorporate system modelling and distributed system development. This extension is called Event-B. Even though several of the structuring mechanisms of the original B method are absent from Event-B, the desire to define and maintain structured data persists. We propose the introduction of records to Event-B for this purpose. Our approach upholds the refinement principles of Event-B by allowing the stepwise development of records too.

1 Introduction

The Praxis[1] case study of the RODIN project is a (subset of a) VDM development of an air traffic control display system (CDIS) undertaken by Praxis in 1992. One of the objectives of the case study is to drive the RODIN methodology, including Event-B itself [7]. CDIS is currently being redeveloped using Event-B and existing B tool support. The motivating feature of the case study is its size, and the challenge is to develop techniques for constructing large specifications in general so that the functionality of the overall system can be understood by everyone involved in a project of this kind (a criticism of the original CDIS specification).

Although the case study does not aim to construct a translation from VDM to Event-B, there are several advantages to preserving the VDM record structure. In particular, it serves to organise a vast amount of structured data. So it is worthwhile investigating how records (with arbitrary field types) can be incorporated in Event-B. More generally, however, we have identified the benefits of incorporating additional subtyping/inheritance-like properties of records to enable their stepwise development through refinement, and to allow better conceptual modelling during the early stages of an Event-B development. In order to address the challenges of CDIS, this allows us to start with a very abstract/generic view of the system and, through refinement, introduce airport-specific details later in the development. Hence, the project members can choose a suitable level of abstraction to view the system.

[*] 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.

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

Our proposal does not require any changes to the semantics of Event-B, although we propose an extension to its syntax.

After we have given an introduction to Event-B, we give a brief overview of records (composites) in VDM. We then show how records can be modelled using existing B constructs, namely **SETS**, **CONSTANTS** and **PROPERTIES**. Along the way, we propose some syntactic sugar to make such definitions more succinct. Our intention is to incorporate this syntax into the Event-B language, thereby eliminating the need to define an unsugared version manually. We then introduce two forms of record refinement: record extension and record subtyping. An example is given to illustrate the use of record refinement in a development, which includes a novel use of record refinement to enable the interface extension of an event. Finally, we discuss other issues that arise from our approach. This example demonstrates the refinement techniques currently being used in the CDIS case study.

Note that open source tools supporting Event-B are currently under construction as part of the RODIN project. However, by writing stylised specifications, existing B tools such as Atelier B [3] and the B Toolkit [5] can be applied to Event-B specifications.

2 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)[2]. The guard must hold 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.

[2] The guard is omitted if it is trivially true.

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].

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 concrete event. In addition, Event-B allows refinement of a model by adding new concrete events on the proviso that they cannot diverge (i.e. execute forever). This condition ensures that the abstract events can still occur. Since the concrete events operate on the state variables of the refined model, they must implicitly refine the abstract event *skip*.

3 VDM Composites

A composite type consists of a name followed by a list of component (field) names, each of which is accompanied by its type. In general, this looks like:

$$\begin{array}{rl} \mathit{type\_name} :: & \mathit{component\_name}_1 : \mathit{component\_type}_1 \\ & \vdots \\ & \mathit{component\_name}_n : \mathit{component\_type}_n \end{array}$$

One can see that this resembles record declarations in many programming languages. However, it is possible to constrain the type of a composite further by including an invariant for the values of the components. Note that the nature of invariants in VDM is different from invariants in Event-B: invariants in Event-B have to be proven, whilst in VDM they are enforced. State in VDM is declared as a special kind of record whose components are the state variables which can be accessed and modified via *operations* (functions having side effects on the state).

Even though we have focused on VDM composite types specifically, record-like structures are also present in other formal notations (for example, composite data types in Z schemas [12], or signatures in Alloy [2]).

4 A Set-Based Approach in Event-B

This approach attempts to mimic the record type definitions of VDM by using the **SETS**, **CONSTANTS** and **PROPERTIES** clauses of an Event-B context. One of the motivations of this work is to enable a stepwise development of complex record structures (in the spirit of refinement) by introducing additional

```
CONTEXT Func
SETS R ; A ; B
CONSTANTS r1 , r2
PROPERTIES
  r1 ∈ R → A ∧
  r2 ∈ R → B ∧
  r1 ⊗ r2 ∈ R ↠ A × B
END
```

Fig. 1. A simple record type

fields as and when they become necessary. This is also comparable to inheritance in object-oriented programming in which classes are *restricted* or *specialised* by introducing additional attributes.

Consider the following VDM composite type declaration R

$$\begin{aligned} R :: r1 &: A \\ r2 &: B \end{aligned}$$

That is, R is a record with two fields, named $r1$ and $r2$, of type A and B respectively. In B, we can model this by declaring three deferred sets R, A and B in the **SETS** clause. This is shown in Figure 1 in a context named *Func*. The sets A and B correspond to the types A and B in the declaration and, as such, these could be replaced by specific B types (such as NAT), or could themselves be other record types. Note that recursive record types are not part of this proposal, although we are investigating this for future work.

The set R represents the record type that we are trying to specify. We can think of this set as representing all of the potential models of the record type. Since we are unaware of the appropriate model for the record, because we may want to refine it during later stages, this set remains deferred until we are sure that we do not want to refine it any further. Instead, we can specify properties of the set within the **PROPERTIES** clause.

Two *accessor* functions are declared in the **CONSTANTS** clause to retrieve the fields of an R record instance: *r1* retrieves the value of the field of type A, and *r2* retrieves the value of the field of type B. The properties of these functions are given in the **PROPERTIES** clause. In particular, note that for every pair of values from A and B there is a record instance (i.e. a member of R) whose fields have these values. This is expressed succinctly using Event-B's direct product operator $\otimes$ and the surjective mapping $\twoheadrightarrow$, where

$$r1 \otimes r2 = \{\ (x,(y,z))\ \mid\ (x,y) \in r1\ \wedge\ (x,z) \in r2\ \}$$

This approach to modelling composites is quite verbose for a two-field record. Instead, we propose some syntactic sugar. Within the **SETS** clause, we propose composite-like declarations for records. Hence, for this example, we would allow an equivalent context as shown in Figure 2. We choose to put such definitions in the **SETS** clause because this clause is most closely associated with type definitions.

```
CONTEXT Func
SETS
  R :: r1 : A,
       r2 : B
END
```

Fig. 2. Syntactic sugar for record types

A machine that **SEES** this context may contain state variables of type R. Such variables hold an instance of the record, and events can be defined to update the values of their fields using the accessor functions. The structure of these events follows a definite pattern: non-deterministically choose an instance of the record type such that its fields have certain values. For example, consider the event in Figure 3 that changes the $r1$ field of a variable r of type R with a value x. The new value y is chosen so that its $r1$ value is equal to x, and its $r2$ value remains unchanged. It is important to state explicitly which fields do not change, otherwise they will be assigned non-deterministically.

```
Update_r1_of_r ( x ) ≙
 ANY  y WHERE
   y ∈ R ∧
   r1 ( y ) = x ∧
   r2 ( y ) = r2 ( r )
 THEN
   r := y
 END
```

Fig. 3. A record update operation

Before we proceed to consider refinement, it is worth mentioning an alternative approach which, under suitable conditions, individual state variables are used to model the fields of a record directly. The approach in [4] uses the structuring mechanisms of classical B (in particular, the **INCLUDES** mechanism) and naming conventions to model the record structure. Their approach resulted from an attempt to construct a translation from VDM to B. A shortcoming of their approach is that it would be impossible to perform parallel updates of 'fields' that reside in the same machine (a constraint imposed by **INCLUDES**). Although renaming can be employed to re-use such definitions, we feel that our approach (with its syntactic sugar) gives a representation that is more suitable at an abstract level; and it is also amenable to parallel updates. More fundamentally, however, renaming and machine inclusion are not available in Event-B.

It would be possible to use variables instead of constants to model accessor functions. In some way this would simplify the approach as updates to a field could be specified more succinctly. For example, if $r1$ and $r2$ were specified as variables, then the update in Figure 3 could be specified as

$$\textbf{Update\_r1\_of\_r}(x) \ \widehat{=} \ r1(r) := x$$

The variable $r2$ is not modified by this assignment. The problem with using variables rather than constants for accessor functions is that it does not work in a distributed setting. In a distributed development we wish to avoid designs in which variables are globally available since maintaining a consistent global view of variables is too much effort. Constants, on the other hand, can easily be globally agreed since they never change. Using constants as accessor functions means we specify a fixed way of accessing fields of a record that is globally agreed.

5 Refining Record Types

We now investigate the effect of refining the record type R defined in Section 4 by introducing a new accessor function. There are two ways of doing this: we can either 'extend' R by adding the accessor function directly, or we can declare a new subtype of R (which we call Q), on which the accessor function is declared. Since the latter refinement will add further constraints to R, Q's set of potential models will be a subset of R's. In this example, both kinds of refinement have an additional field *r3* of type C. For a simple record extension, we propose a syntax as follows:

$$\textbf{EXTEND}\ R\ \textbf{WITH}\ r3 : C$$

For subtyping, we propose the following syntax:

$$Q\ \textbf{SUBTYPES}\ R\ \textbf{WITH}\ r3 : C$$

Their verbose definitions are shown in Figures 4 and 5 respectively. The proposed syntax means that the developer does not have to interact with the verbose definitions directly. Notice that these definitions are both refinements of the context machine given in Figure 1. Hence, the properties declared in the refinement are in addition to those of the original machine. The final property in Figure 4 states that all possible field combinations are still available in R, and the corresponding property in Figure 5 states that all possible field combinations are available in Q (without adding any further constraints to R).

Subtyping of this kind can be seen in programming languages such as Niklaus Wirth's Oberon [11], and specification languages such as Alloy [2]. The accessors $r1$ and $r2$ can still be applied to objects of type Q in Figure 5, but $r3$ can *only* be applied to objects of type Q (and any of its subtypes).

```
CONTEXT FuncR
REFINES Func
SETS C
CONSTANTS r3
PROPERTIES
  r3 ∈ R → C ∧
  ( r1 ⊗ r2 ⊗ r3 ) ∈ R ↠ A × B × C
END
```

Fig. 4. An extended record type

CONTEXT *FuncR*
REFINES *Func*
SETS C
CONSTANTS Q , *r3*
PROPERTIES
  $Q \subseteq R \wedge$
  $r3 \in Q \rightarrow C \wedge$
  $(r1 \otimes r2 \otimes r3) \in Q \twoheadrightarrow A \times B \times C$
END

Fig. 5. A record subtype

Depending on whether extension or subtyping is used, a certain amount of care is required when refining the events associated with the records. The event **Update_r1_of_r** shown in Figure 3 is still applicable in the refined context of Figure 4, even though it would assign r's new $r3$ field non-deterministically. Refinement could then be used to assign something meaningful to this field. However, using the refined context of Figure 5, if the model is refined so that r is defined to be of type Q then this event is no longer applicable without modification because the quantified variable y ranges over Q's superset R. The **ANY** clause would need to be strengthened so that it chooses an element of Q (rather than R). Note that the surjectivity constraints of a record extension are consistent with the constraints of the original record definition. Indeed, the original constraints follow from those of the extension, i.e.:

$$(r1 \otimes r2 \otimes r3) \in R \twoheadrightarrow A \times B \times C \;\Rightarrow\; (r1 \otimes r2) \in R \twoheadrightarrow A \times B$$

5.1 Other Possible Refinement Combinations

In addition to a single chain of record refinements, which is most easily achieved by record extension, the subtyping of record types presented above permits other, less restrictive, kinds of development.

The diagrams shown in Figure 6 give two possible extension hierarchies. Each of these is meaningful, and we would like them to be available in Event-B. Hence, records should not constrain the structuring of context machines.

$$\begin{tikzcd}
 & R & \\
Q \arrow[ur] & & Q' \arrow[ul]
\end{tikzcd}$$

(i)

$$\begin{tikzcd}
 & X & \\
Y \arrow[ur] & & Y' \arrow[ul] \\
 & Z \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

(ii)

Fig. 6. Possible Record Hierarchies

In Figure 6(i), two different sets, Q and Q', subtype the same record type R. That is, Q and Q' have the common ancestor R, and both are defined to be subsets of R. (The relationship between Q and Q' in this case is left unspecified, but we can impose an extra property to ensure they are disjoint if necessary.) In (ii), the record type Z combines the record subtypes Y and Y'. In this situation, the model for Z must be a model for both Y and Y', and the accessor functions of both Y and Y' can be applied to objects of type Z. The least constrained set of models that fulfils this relationship is the intersection of Y and Y'. Hence, they should not be disjoint. Syntactically, the fields belonging to a record of type Z (prior to any extensions to Z) are the union of the fields of Y and Y' [3].

6 One-Field Variables

In section 5, we have seen how existing record types can be refined to give new record types with more complex structure. At the most abstract level, the specifier might be unaware that a simple (non-record) state variable requires a record structure at a later stage in the development. For example, we may declare a variable v to be of type $VALUE$, but then decide that for every value we need to associate some other characteristic (say, a format). We would then need to define a record type

$$\begin{aligned} FVALUE :: \; & val : VALUE, \\ & format : FORMAT \end{aligned}$$

and declare a concrete variable fv of type $FVALUE$. In order to link the variables v and fv in a refinement, the gluing invariant must link the val component of fv with v. In this case, we have

$$v = val(fv)$$

Hence, at the most abstract level, we are not expected to identify all record types. These can be introduced during the refinement stages.

7 Extension Example

In order to motivate the use of record types, we present an example to show how a very simple abstract specification can be refined into a model with structured objects. We consider an electronic mail delivery system in which users (with identities) can send and receive messages. We begin with a very abstract view of the system. The context (which we call *Context*) declares two sets *User* and *Message*, and one record type *Send_interface*. This is shown in Figure 7.

The corresponding machine (which we call *Email*) declares a variable *mailbox*, which maps users to their respective set of messages. We specify two events: **send** and **read**. These are shown in Figure 8. At this stage, the **send** event requires two parameters that represent the message to be sent and the intended recipient.

[3] We assume there are no name clashes between the accessors of Y and Y'.

```
CONTEXT Context
SETS
   User ; Message ;
   Send_interface :: dest : User,
                     mess : Message
END
```

Fig. 7. The Abstract Context

However, during the refinement stages **send** will require additional parameters. Interface extension is not possible in the current B tools, but by using record types instead we can extend the interface of **send** via record extension. The record type *Send_interface* is declared for this purpose. Note that this is a very abstract representation of the system because the **send** operation magically deposits the message in the appropriate user's mailbox. Subsequent refinements will model how this is actually achieved. The **read** event non-deterministically retrieves a message from the input user's mailbox and returns it as an output.

As a first refinement we begin to introduce more detail in the form of a more realistic architecture. This is depicted in Figure 9. Each user is associated with a mail server that is responsible for forwarding mail and retrieving mail from the middleware. As part of this refinement, we introduce a record type to structure the data passing from senders to receivers via the communications medium. The record type called *Package* is declared using our proposed syntax in the context

```
MACHINE Email
SEES Context
VARIABLES mailbox
INVARIANT mailbox ∈ User → ℙ ( Message )
INITIALISATION mailbox := User × { ∅ }

OPERATIONS
   send ( ii ) ≙
     PRE  ii ∈ Send_interface THEN
        mailbox ( dest ( ii ) ) := mailbox ( dest ( ii ) ) ∪ { mess ( ii ) }
     END ;

  mm ⟵  read ( uu ) ≙
     PRE  uu ∈ User THEN
        ANY  xx WHERE
           xx ∈ Message ∧ xx ∈ mailbox ( uu )
        THEN
           mm := xx
        END
     END
  END
```

Fig. 8. The Abstract Machine

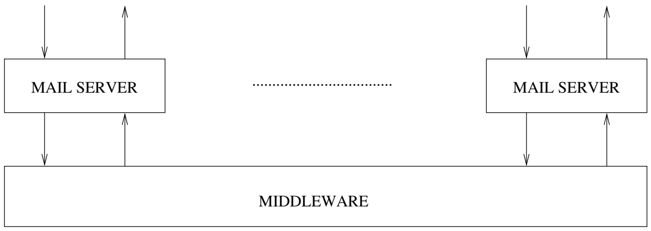

Fig. 9. Architecture for the e-mail system

CONTEXT *Context2*
REFINES *Context*
SETS
  Server **;**
  Package :: *destination* : *Server* ,
    recipient : *User* ,
    contents : *Message* **;**
  EXTEND *Send_interface* **WITH** *source* : *User*
CONSTANTS
  address
PROPERTIES
  $address \in User \rightarrow Server$
END

Fig. 10. First refined context

refinement named *Context*2. This is shown in Figure 10. Note that in addition to *Package* we declare a new set *Server* which represents the different mail servers, and we declare a function *address* that returns the (unique) server hosting a particular user. We also extend *Send_interface* by adding a new field *source* that contains the identities of the senders.

The refined state comprises new variables *sendbuf*, *receivebuf* and *middleware*. The variable *middleware* holds the packages on the communications medium, and each mail server has separate buffers for messages waiting to be sent and messages waiting to be read (i.e. mappings from *Server* to $\mathbb{P}$ (*Package*)). The refined event **send** constructs packages and adds them to the server associated with the sender. The event **read** selects packages from a user's server and output's their contents. These are shown in Figure 11.

As part of this Event B refinement, we introduce two new events **forward** (which passes packages from servers to the middleware) and **deliver** (which takes packages from the middleware and adds them to the appropriate server's receive buffer). Note that these events (also shown in Figure 11) will not collectively diverge because only a finite number of packages will be waiting to be transferred.

The gluing invariant that links the refined state with the abstract state is dictated by the need to preserve outputs. Since we are refining from simple messages to packages, we use the technique given in Section 6. In the abstract model,

```
send ( ii ) ≙
  PRE  ii ∈ Send_interface THEN
    ANY  ss , pp WHERE
      ss ∈ Server ∧ pp ∈ Package ∧
      ss = address ( source ( ii ) ) ∧
      destination ( pp ) = address ( dest ( ii ) ) ∧
      recipient ( pp ) = dest ( ii ) ∧
      contents ( pp ) = mess ( ii )
    THEN
      sendbuf ( ss ) := sendbuf ( ss ) ∪ { pp }
    END
  END ;

mm ⟵  read ( uu ) ≙
  PRE  uu ∈ User THEN
    ANY  ss , pp WHERE
      ss ∈ Server ∧ pp ∈ Package ∧
      ss = address ( uu )
      pp ∈ receivebuf ( ss ) ∧
      recipient ( pp ) = uu
    THEN
      mm := contents ( pp )
    END
  END ;

forward ≙
  ANY  ss , pp WHERE
    ss ∈ Server ∧ pp ∈ Package ∧
    pp ∈ sendbuf ( ss )
  THEN
    sendbuf ( ss ) := sendbuf ( ss ) − { pp }  ||
    middleware := middleware ∪ { pp }
  END ;

deliver ≙
  ANY  ss , pp WHERE
    ss ∈ Server ∧ pp ∈ Package ∧
    pp ∈ middleware ∧
    destination ( pp ) = ss
  THEN
    middleware := middleware − { pp }  ||
    receivebuf ( ss ) := receivebuf ( ss ) ∪ { pp }
  END
```

Fig. 11. First refinement events

the output from **read** is obtained from the input user's mailbox, whereas it is retrieved from *receivebuf* in the refined model. We link the *contents* field of the packages in *receivebuf* with *mailbox*. Hence, we introduce the following invariant

$$\forall s, u, p.(s \in Server \;\wedge\; u \in User \;\wedge\; p \in Package \;\Rightarrow$$
$$p \in receivebuf(s) \;\wedge\; recipient(p) = u \;\Rightarrow\; contents(p) \in mailbox(u))$$

This fulfils the proof obligation derived from the output of **read** but, since the event **deliver** adds packages to *receivebuf*, we must strengthen the invariant as follows

$$\forall u, p.(u \in User \;\wedge\; p \in Package \;\Rightarrow$$
$$p \in middleware \;\wedge\; recipient(p) = u \;\Rightarrow\; contents(p) \in mailbox(u))$$

That is, in addition to the contents of the packages in *receivebuf*, the contents of the packages on the medium must also be elements of *mailbox*. By attempting to discharge the proof obligations once more, we discover that we have to strengthen the invariant further

$$\forall s, u, p.(s \in Server \;\wedge\; u \in User \;\wedge\; p \in Package \;\Rightarrow$$
$$p \in sendbuf(s) \;\wedge\; recipient(p) = u \;\Rightarrow\; contents(p) \in mailbox(u))$$

This is sufficient to discharge all of the proof obligations. Hence, we have shown that the contents of *any* package in transit must be an element of the corresponding abstract mailbox. Of course, it would be possible to strengthen the invariant further by stating other properties of the system, but this is not pursued here.

As a second refinement, we extend *Package* with a priority field. In addition, we extend *Send_interface* with a field *pri*. These refinements are shown (using our proposed notation) in Figure 12.

CONTEXT *Context3*
REFINES *Context2*
SETS
  EXTEND *Package* **WITH** *priority* : *BOOL* **;**
  EXTEND *Send_interface* **WITH** *pri* : *BOOL*
END

Fig. 12. The second context refinement

This refinement specifically affects the order in which packages are moved onto the middleware: packages with priority *TRUE* take precedence over packages with priority *FALSE*. In order to model this, we refine the events **send** and **forward** (as shown in Figure 13). The **send** event is refined because we have extended its interface to incorporate a priority field (named *pri*). Using this extension to *Send_interface*, we can assign priorities to the refined packages. The refinement of **forward** is an example of the refinement of a single event into two events. The first refined event (also called **forward**) only selects packages with high priority (i.e. those packages whose priority field is *TRUE*). The second event, called **forward2** selects low priority packages, but only if there are no high priority packages at the same server. Hence, high priority packages are forwarded before low priority packages. Since this refinement does not introduce any new variables, no gluing invariant is required.

```
send ( ii ) $\widehat{=}$
  PRE  $ii \in Send\_interface$ THEN
    ANY  $ss$ , $pp$ WHERE
      $ss \in Server \wedge pp \in Package$
      $ss = address ( source ( ii ) ) \wedge$
      $destination ( pp ) = address ( dest ( ii ) ) \wedge$
      $recipient ( pp ) = dest ( ii ) \wedge$
      $contents ( pp ) = mess ( ii ) \wedge$
      $priority ( pp ) = pri ( ii )$
    THEN
      $sendbuf ( ss ) := sendbuf ( ss ) \cup \{ pp \}$
    END
  END ;

forward $\widehat{=}$
  ANY  $ss$ , $pp$ WHERE
    $ss \in Server \wedge$
    $pp \in Package \wedge$
    $pp \in sendbuf ( ss ) \wedge$
    $priority ( pp ) = TRUE$
  THEN
    $sendbuf ( ss ) := sendbuf ( ss ) - \{ pp \} \parallel$
    $middleware := middleware \cup \{ pp \}$
  END ;

forward2 $\widehat{=}$
  ANY  $ss$ , $pp$ WHERE
    $ss \in Server \wedge$
    $pp \in Package \wedge$
    $pp \in sendbuf ( ss ) \wedge$
    $\forall qq . ( qq \in sendbuf ( ss ) \Rightarrow priority ( qq ) = FALSE )$
  THEN
    $sendbuf ( ss ) := sendbuf ( ss ) - \{ pp \} \parallel$
    $middleware := middleware \cup \{ pp \}$
  END
```

Fig. 13. The second refinement

8 Subtyping Refinement

The example could be refined further by specialising *Package* using subtyping. Using this technique, it is possible to refine *Package* in more than one way (see Figure 6(i)) so that different kinds of packages are dealt with in different ways. For example, consider the following subtype declarations

AirportPackage **SUBTYPES** *Package* **WITH** ...
RunwayPackage **SUBTYPES** *Package* **WITH** ...

It would then be possible to specialise the servers to meet the needs of the different kinds of package. On the other hand, we would be able to continue to use the middleware unaltered because it would simply treat both subtypes uniformly (i.e. as objects of type *Package*). In the CDIS case study, this technique is being used to model VDM union types in Event-B.

9 Wider Issues

Although we have not set out with the aim of addressing object oriented modelling or programming approaches, there is a link between our work and various formal approaches to object oriented modelling and programming. Directly relevant to our work is the UML-B approach of Snook and Butler [10] which defines a mapping from a UML profile to B. In UML-B, class attributes and associations are modelled in B as accessor functions on object instance identifiers, i.e., if a is an attribute of type A of class C, then a is modelled in the B notation as a function $a \in C \rightarrow A$. UML-B effectively combines our form of subtyping with extension to represent class inheritance. In our approach accessor functions are represented as constants whereas in UML-B attributes and associations can be declared as either constant or variable and the corresponding accessor functions are in turn either constants or variables.

Naumann's work [8] is a good example of a relevant formal framework for reasoning about object oriented programs. This uses records and record subtyping to represent objects in an object oriented programming language. There are two significant differences from our work. Firstly, Naumann allows record fields to be methods thus modelling method overriding and dynamic dispatch of method calls, an important feature of object oriented programming. We do not address overriding of events rather we focus on refinement. Secondly, Naumann uses record constructors and a rich notion of subtyping for record types as is commonly found in formal approaches to object oriented programming. Our notion of subtyping is simply subsetting of the deferred B type used to model records and is independent of any subtyping of the fields. This means we avoid having to address the issue of covariance versus contravariance of method arguments [6]. Naumann's language is influenced by Oberon [11] which provides inheritance through record extension.

10 Conclusion

Without changing its semantics, we have proposed a method of introducing record types in Event-B that is amenable to refinement. Our experience in the redevelopment of CDIS has identified the benefits of such an approach. In particular, it allows us to start with a very abstract model and defer the introduction of airport-specific information until later in the development.

Our example has demonstrated how it is possible to specify an abstract view of a system with a very abstract representation of the data that it handles. In addition to the existing refinement techniques of Event-B, our refinements show how it is possible to introduce structured data in a stepwise manner in order to progress towards the formal design and implementation of the system.

During the implementation phase of a B development, it may be necessary to describe how records are to be implemented. Since a record is defined as a deferred set, a decision must be made to give an explicit representation of the set. In addition, fields of such records are declared as constant functions whose algorithmic behaviour must be given as part of the implementation.

Of course, it is not the case that the records within a B development will necessarily be implemented as records in program code - for example, a record could be used to model structured data such as XML messages. However, there is provision for the implementation of record-like structures using existing B technology: **SYSTEM** definitions of **BASE** machines are macros for the implementation (using B libraries) of database style structures. (See [9] for a detailed description of **BASE** machines and the common B libraries.) The similarities between our proposed syntax and the syntax of **BASE** machines suggest that they provide a natural progression from the specification and refinement of records to their implementation, although this has yet to be investigated.

Acknowledgements

The authors thank Jean-Raymond Abrial and Cliff Jones for many valuable discussions, and are also grateful to Helen Treharne for useful advice and suggestions.

References

1. Abrial J. R.: *The B Book: Assigning Programs to Meanings*, Cambridge University Press (1996).
2. The Alloy Analyzer, `http://alloy.mit.edu`.
3. Atelier B, `http://www.atelierb.societe.com`.
4. Bicarregui J. C.,Matthews B. M., Ritchie B., Agerholm S.: *Investigating the integration of two formal methods*, Proceedings of the 3rd ERCIM Workshop on Formal Methods for Industrial Critical Systems (1998).
5. B Core (U.K.) Ltd, `http://www.b-core.com`.
6. Castagna G.: *Covariance and Contravariance: Conflict without a Cause.*, in ACM Trans. Program. Lang. Syst., volume 17, number 3, pages 431-447 (1995).
7. Métayer C., Abrial J. R., Voisin L.: *Event-B Language*, RODIN deliverable 3.2, `http://rodin.cs.ncl.ac.uk` (2005).
8. Naumann D. A.: *Predicate transformer semantics of a higher-order imperative language with record subtyping*, in Sci. Comput. Program., volume 41, number 1, pages 1-51 (2001).
9. Schneider S.: *The B Method: An Introduction*, Palgrave (2001).
10. Snook C., Butler M. J.: *UML-B: Formal modelling and design aided by UML.*, in ACM Trans. Software Engineering and Methodology, to appear (2006).
11. Wirth N. : *The Programming Language Oberon*, in Softw., Pract. Exper., volume 18, number 7, pages 671-690 (1988).
12. Woodcock J. C. P., Davies J.: *Using Z: Specification, Refinement, and Proof*, Prentice Hall (1996).