The original publication is available at http://doi.org/10.1109/ICECCS.2017.29
In ICECCS: Proceedings of 22nd International Conference on Engineering of Complex Computer Systems © IEEE 2017.

Extending ERS for Modelling Dynamic Workflows in Event-B

Dana Dghaym, Michael Butler and Asieh Salehi Fathabadi
Electronics and Computer Science
University of Southampton, UK
Email: {dd4g12, mjb, asf08r}@ecs.soton.ac.uk

***Abstract*—Event-B is a state-based formal method for modelling and verifying the consistency of discrete systems. Event refinement structures (ERS) augment Event-B with hierarchical diagrams, providing explicit support for workflows and refinement relationships. Despite the variety of ERS combinators, ERS still lacks the flexibility to model dynamic workflows that support dynamic changes in the degree of concurrency. Specifically in the cases where the degree of parallelism is data dependent and data values can change during execution. In this paper, we propose two types of extensions in ERS to support dynamic modelling using Event-B. The first extension is supporting data-dependent workflows where data changes are possible. The second extension improves ERS by providing exception handling support. Semantics are given to an ERS diagram by generating an Event-B model from it. We demonstrate the Event-B encodings of the proposed ERS extensions by modelling a concurrent emergency dispatch case study.**

***Keywords*—Event-B; ERS; Refinement; Dynamic Workflows; Emergency Dispatch**

I. Introduction

An Event-B model consists of a collection of guarded atomic actions (i.e. events). In Event-B, event execution is interleaved and there is no support for explicit workflows. ERS (Event Refinement Structures) [1] provides additional structure for Event-B refinements and explicitly describes the ordering of events. The coordination of events is done implicitly in Event-B using guards, which are event enabling conditions. ERS provides various combinators that support sequencing, iteration, choice, synchronisation and several forms of non-deterministic interleaving.

ERS was originally introduced in [1], [2]. However, the original approach was restricted in modelling dynamic changes in workflows. In this paper, we extend ERS with new combinators to support modelling dynamic changes in the degree of concurrency, in addition to supporting interruptions and exception handling mechanisms. Using the new combinators, we demonstrate some dynamic modelling patterns that were not possible in the original ERS approach.

Our focus is to show how the new ERS combinators can facilitate modelling in Event-B and give ERS the flexibility to adapt to changes especially in the case of complex concurrent systems. We demonstrate our modelling approach by applying it to an emergency dispatch system, showing how the ERS extensions are encoded in Event-B.

When modelling, we first use the ERS diagrams as blueprints to define a refinement strategy. Once refinement is decided, we start the Event-B modelling using the ERS tool[1] [3] alongside the Event-B editor within the Rodin tool platform [4]. We adopt the separation of workflows from data handling. We model workflows using ERS and leave the data handling and the state-based variables to be done textually using Event-B. However, separation does not mean the independence of the control variables and the state-based data variables; this is because the ERS semantics is defined by transforming the diagrams into Event-B. Moreover, defining explicit control variables in Event-B, makes it possible to explicitly relate the textually added state-based variables to the control variables using invariants. The model is verified using the Rodin platform and validated using the ProB [5] model checker.

The rest of the paper is organised as follows: Sect. II gives background information about Event-B and the ERS approach. Sect. III presents ERS extensions to support dynamic workflows and exception handling. In Sect. IV, we present the emergency dispatch case study, showing how to apply the new ERS extensions and their Event-B encodings, then we do the model analysis and discussion in Sect V. Finally in Sect. VI, we present our conclusions, related work and future directions.

II. Background

A. Event-B

Event-B is a formal method for modelling and verifying discrete systems [6]. Its language, based on the set theory and first order logic, is mainly influenced by the B-Method [7] and Action Systems [8].

1) Structure: An Event-B model is divided into two parts, the *Context* and the *Machine*. The context is the static part of the model, where the set types and the constants are defined. The machine, is the dynamic part of the model, where the variables, invariants and the events are defined. A machine can access contexts to define its variables, using invariants which describe properties that events are expected to maintain. An event in Event-B can update variables atomically using actions, provided all its guards are satisfied. In general an event e in Event-B with variables v has the following format, where p

[1]An interim update site containing only the first set of extensions, is available on http://users.ecs.soton.ac.uk/dd4g12/ERSUpdateSite/

are the event parameters, $G(p, v)$ are the event guards and $v \ := \ A(p, v)$ are the event actions.

$$e \mathrel{\widehat{=}} \textbf{any } p \textbf{ where } G(p,v) \textbf{ then } v \ := \ A(p,v)$$

2) Refinement: Refinement is a key concept in Event-B that aims at simplifying the complexity of modelling and verification. Using Event-B refinement, a model is built gradually starting with an abstract model. Then details and/or new functionalities of the model are added at further refinement levels. Each event of a refined model either refines some event of the abstract model or is a new event with no corresponding abstract event. New events are required to refine *skip*, an implicit event that has no effect at the abstract level.

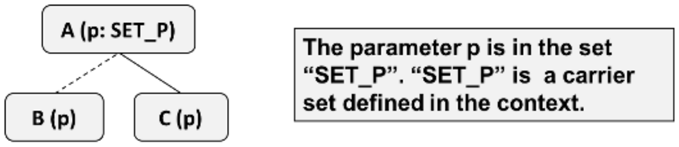
In Event-B, proof obligations are used to prove invariant preservation and refinement correctness. Proof obligations are automatically generated, and can be proved both automatically and interactively using the Event-B tool, Rodin [4].

B. Event Refinement Structures (ERS)

In Event-B, it is common to specify behaviour abstractly as an atomic event and to decompose that behaviour into multiple atomic events via refinement, which are a combination of new events and refining events. Syntactically these events are separated without any explicit links between them. The ERS approach provides a graphical extension of Event-B to represent event decomposition explicitly. In addition to specifying the events that decompose an abstract event, an ERS diagram specifies the control flow amongst the decomposed events. ERS supports a range of workflow combinators including sequencing, looping and several forms of non-deterministic interleaving. ERS was first introduced by Butler [2] and later enhanced in [9] by defining different refinement patterns and formal translation rules to Event-B.

ERS has a tree structure inspired by Jackson Structure Diagrams (JSD) [10]. JSD provides a graphical representation of structured sequential programs with sequential behaviour indicated by left-to-right placement of nodes and additional annotations to represent choice and iteration.

Fig. 1 presents a simple ERS example specifying that event A is decomposed to two atomic events (B and C). Similar to JSD diagrams, the leaf nodes in ERS are ordered sequentially from left to right, so Fig. 1 specifies that B should be executed before C. The dashed line indicates that B is a new event (i.e., refines $skip$) while C refines A. In ERS exactly one of the children should be connected to its parent with a solid line, while the others should be connected using dashed lines. In ERS this is referred to as the "*single solid line rule*".

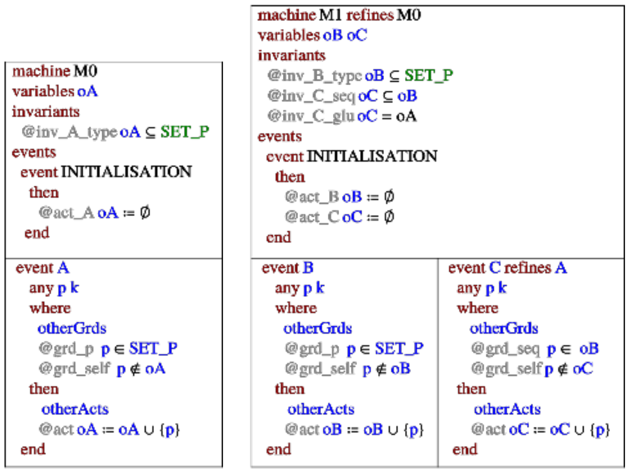


The parameter p is in the set "SET_P". "SET_P" is a carrier set defined in the context.

Fig. 1. A Simple ERS Diagram in the Multiple Instances Case

The addition of the instance parameter p, in Fig. 1 is optional and its presence indicates multiple instance modelling. Multiple instances can be used to model concurrency, where leaf events of different instances may interleave.

1) ERS Semantics: Semantics is given to an ERS diagram by transforming it into an Event-B model. Fig. 2 presents the Event-B representation of the ERS diagram in Fig. 1. The Event-B model in Fig. 2 is generated automatically using the ERS tool [3]. The transformation from ERS to Event-B results in variables, typing invariants, sequencing invariants, and gluing invariants to relate the abstract variables to the refining variables. The ERS transformation also results in events for each leaf, with guards describing the enabling conditions, and actions disabling the event after its execution as appropriate. Fig 1 represents two levels of abstraction: a machine containing event A and a refinement where A is decomposed to events B and C.



```
machine M0
variables oA
invariants
  @inv_A_type oA ⊆ SET_P
events
  event INITIALISATION
    then
      @act_A oA ≔ ∅
  end

 event A
   any p k
   where
     otherGrds
     @grd_p  p ∈ SET_P
     @grd_self  p ∉ oA
   then
     otherActs
     @act oA ≔ oA ∪ {p}
  end
```

(a) Abstract Level (*M0*)

```
machine M1 refines M0
variables oB oC
invariants
  @inv_B_type oB ⊆ SET_P
  @inv_C_seq oC ⊆ oB
  @inv_C_glu oC = oA
events
  event INITIALISATION
    then
      @act_B oB ≔ ∅
      @act_C oC ≔ ∅
  end

 event B
   any p k
   where
     otherGrds
     @grd_p  p ∈ SET_P
     @grd_self  p ∉ oB
   then
     otherActs
     @act oB ≔ oB ∪ {p}
  end

 event C refines A
   any p k
   where
     otherGrds
     @grd_seq  p ∈  oB
     @grd_self p ∉ oC
   then
     otherActs
     @act oC ≔ oC ∪ {p}
  end
```

(b) Refinement Level (*M1*)

Fig. 2. Event-B Specifications of Fig. 1

The Event-B model in Fig. 2(a) represents the ERS diagram before decomposing event *A*, while the refined Event-B model in Fig. 2(b) represents the decomposed ERS diagram, where the leaf nodes of the ERS diagram (*B* and *C*) are mapped to events in Event-B. Each leaf in the parametrised flow has the event parameter *p*, and in the Event-B model, the control variables for the leaf events, *oB* and *oC*, indicate the occurrence of the event for each instance: *oB* and *oC* are subsets of SET_P, the instance set of parameter *p*, and $p \in oB$ indicates that event oB has occurred for instance p. Regarding the ordering of events, the guard $p \in oB$ in event C of Fig. 2(b) ensures that B has occurred for the parameter value p, while events for different values of p may interleave. Adding the parameter value to the control variable of the event indicates that the event has executed for that instance value, preventing the event from executing again for the same instance as indicated by $p \ \notin \ oB$ and $p \ \notin \ oC$ in both events.

The ordering of events is not only ensured by guards, but also using invariants which are proved to be maintained after execution of each event. In the case of the *B* event, no ordering constraints are required so a typing invariant is generated. The solid line is interpreted in Event-B by the gluing invariant, *inv_C_glu*, which relates oC to oA, and the addition of the *refines* keyword to event C indicating that C refines A. In Event-B the relation between B and A is not explicit, while the ERS diagram explicitly shows that the atomicity of A is decomposed into two sub-events, B followed by C. In the case of single instance modelling, where no parameter is added, the occurrence of events is indicated using boolean flags rather than instance sets.

2) *ERS Combinators:* ERS has four groups of combinators:

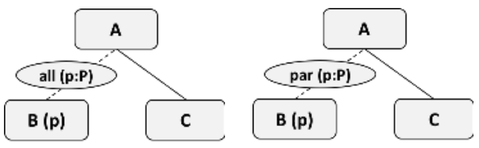
1) Sequencing: represented by the left-to-right arrangement of events.
2) Logical combinators: *and*, *or* (multiple choice), *xor* (exclusive choice).
3) *Loop* indicating zero or more executions and represented by the star symbol.
4) Replicators: *all*, *some*, *one* (generalisations of *and*, *or*, *xor* respectively) are quantified combinators.

The different combinators of ERS affect the ordering constraints of the events. For example, consider the single instance flow: (*E1 and E2*) followed by *E3*. The sequencing guard of *E3* is $oE1 \wedge oE2$, while in the case of *E1 or E2* followed by *E3*, the sequencing guard of *E3* is $oE1 \vee oE2$. Using *xor* results in the same sequencing guard as *or*, with the difference that *E1* and *E2* are mutually exclusive, so a guard is added to *E1* that *E2* has not occurred, and vice versa. In case of a multiple instance flows, sequencing is represented using the intersection and union of sets. The previously described constraints are also represented with invariants.

III. Extending ERS

A. Dynamic Extension

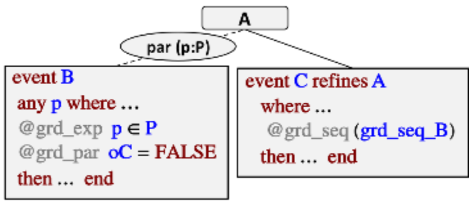
The ERS replicators (*all*, *some*, *one*) introduce a new parameter, adding a new dimension to their events. The behaviour of the replicator events is replicated for different instance values of the parameter set. All the values of the ERS replicator sets must be predetermined before executing their events, which means that ERS lacks the flexibility to model dynamic changes in workflows. For this we introduce the *par* replicator which accepts parameter sets that are not predetermined and can change during the execution of its events.



(a) *all* Replicator (b) *par* Replicator

Fig. 3. ERS Replicators

Fig. 3, illustrates how the ERS replicators introduce a new dimension p to their single instance event (B). In Fig. 3(a), the set P cannot change once B starts executing, and applying *all* will ensure that C can only be enabled after B executes for all instance values (p) in the replicator set P. In Fig. 3(b), it is possible for new values to be added to the replicator set P during the execution of B. Applying $par(p : P)$ does not affect C, but allows zero or more executions of B for different instance values, which is terminated by C. Fig 4 shows the Event-B semantics of applying *par* in Fig. 3(b). Regarding the Event-B semantics of Fig. 3(a), *all* adds only *grd_exp* to B, however the sequencing guard of C is $oB = P$.



```
event B
 any p where ...
 @grd_exp  p ∈ P
 @grd_par  oC = FALSE
then ...  end
```

```
event C refines A
  where ...
   @grd_seq (grd_seq_B)
  then ...  end
```

Fig. 4. *par* Replicator Semantics (Fig. 3(b))

Extending ERS with *par* makes it possible for ERS to model and verify workflows that are dynamic in the degree of concurrency. We can now define a generic structure using ERS combinators (Fig 5) to model the parallel *Producer-Consumer* pattern. Fig 5 illustrates this pattern where it is possible for one event to produce instances ($Produce(p)$) for a set (a), while other instances of a are being consumed ($Consume(p)$). Both events, *End Produce* and *End Consume*, set the stopping criteria for producing and consuming instance values of a.

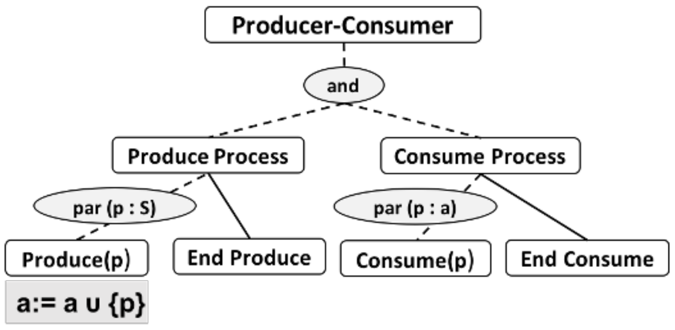


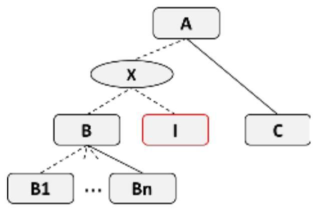
Fig. 5. Parallel *Producer-Consumer* Pattern

The *and* combinator illustrates the possible concurrency between producing and consuming elements. In both the *Produce* and *Consume* events, the number of needed elements cannot be predetermined and depends on the stopping criteria defined by the guards of *EndProduce* and *EndConsume*. This makes *par* a good candidate to represent this behaviour, which cannot be represented by the original ERS replicators (*all*, *some*, *one*).

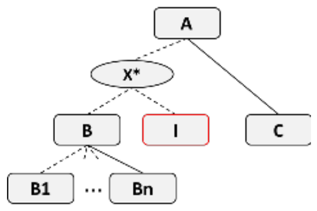
We have also included some natural extensions to the original ERS [9] related to the *parameter* set and *loop* behaviour. The original ERS parameter sets only allowed constant sets, which we generalise to allow any variable expression. We also generalise the *loop* to allow more than one child flow, chosen non-deterministically at each iteration.

B. Exception Handling Extension

In ERS, it is possible to model different cases using the *xor* combinator. However, this could lead to complicating the model with various *xor* branches representing the different alternatives. In order to avoid having complex diagrams and models, we extend ERS with two new combinators (Fig 6) dealing with exceptions, thus separating the normal expected case from the exceptional cases.



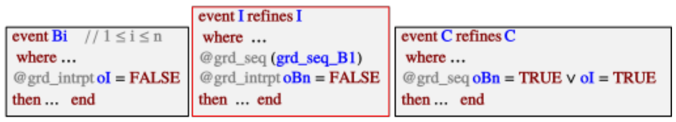
(a) *interrupt* Combinator



(b) *retry* Combinator

Fig. 6. ERS Exception Handling Combinators

The new ERS exception handling combinators (*interrupt*, *retry*) consist of two sub-flows, normal (B) and interrupting (I), where the interrupting sub-flow (I) can stop the execution of the normal flow (B) before its completion. In Fig 6, I can execute and interrupt B at any point between B_1 and B_n. In addition to the interrupting feature, *retry* (Fig 6(b)) allows retrying the normal flow after interruption, by resetting its control variables (oB_1 .. oB_n), hence the additional star symbol of the *loop*. Another difference is in the enabling of the follow-on flow/event (C). In the *interrupt* case, C can be enabled after the completion of either *B* or *I*. In the *retry* case, C can only be enabled after the completion of B, where we usually add *interrupt* at a higher level to avoid retrying forever. The Event-B implication of applying the exception handling combinators to the events at the second refinement level of Fig 6 is illustrated in Fig. 7, where *retry* adds an additional hidden event (*reset_I*) to reset the interrupting flow.

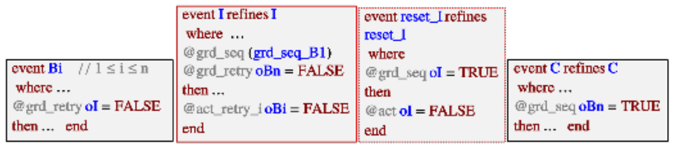


```
event Bi    // 1 ≤ i ≤ n
 where ...
@grd_intrpt oI = FALSE
then ...   end
```

```
event I refines I
 where  ...
@grd_seq (grd_seq_B1)
@grd_intrpt oBn = FALSE
then  ...  end
```

```
event C refines C
 where ...
@grd_seq oBn = TRUE ∨ oI = TRUE
then ...   end
```

(a) *interrupt* Combinator Semantics (Fig. 6(a))



```
event Bi    // 1 ≤ i ≤ n
 where ...
@grd_retry oI = FALSE
then ...   end
```

```
event I refines I
 where  ...
@grd_seq (grd_seq_B1)
@grd_retry oBn = FALSE
then ...
@act_retry_i oBi = FALSE
end
```

```
event reset_I refines
reset_I
 where
@grd_seq oI = TRUE
then
@act oI = FALSE
end
```

```
event C refines C
 where ...
@grd_seq oBn = TRUE
then ...   end
```

(b) *retry* Combinator Semantics (Fig. 6(b))

Fig. 7. ERS Exception Handling Semantics

The new ERS exception handling combinators are the only other ERS combinators besides sequencing where their child events are **not** commutative, because the effect of the normal child events and the interrupting child events are **not** symmetric. In Fig. 6, the last child of the normal flows are solid children and these are the only cases where the solid event must be the last event(s) of a sequential decomposition. The reason for that is related to Event-B refinement. At the abstract level, I is disabled once B completes; if the event refining B was B_i $(i < n)$ rather than B_n, then events $B\_i + 1$ to B_n could not be interrupted by I since I would have to be disabled by B_i.

IV. Emeregency Dispatch Case Study

In this section, we model an emergency dispatch system to illustrate our extensions and show how they can support dynamic behaviour in Event-B. We also demonstrate the Event-B semantics of the ERS extensions.

A. Requirements Overview

The emergency dispatch system is responsible for handling all the incident information, finding the nearest available resources and monitoring their status, and closing incident calls. The system requirements can be summarised as follows:

1) Incident Information: For each incident, the control operator gets information about its type, location and priority. While gathering information, there is a possibility of being a duplicate, which sometimes can be discovered late.

2) Incident Management: From the incident details, an action plan must be determined. The action plan includes the required resources. Initially from the type and location, an action plan can be selected from a set of predetermined plans. Additional resources can be requested and when a stop message is sent, sufficient resources are present and there is no need to send further resources. Proposing resources to incidents should include the quickest available resources.

3) Resource Management: It is possible to reallocate a resource from a lower priority to a higher priority incident. Before a resource attends the incident, the system can try to find quicker resources that become available, but it is up to the operator to reallocate the quicker resource. Resource replacement is required in case a resource breaks down or it is reallocated to another incident.

4) Incident Closure: An incident must not be closed if it has any assigned resources. A non-duplicate incident must not be closed if any of the required action plan items are not completed, nor if a stop message is not received.

B. Refinement Strategy

The model consists of an abstract and nine refinement levels as follows:

- **M0:** Models the creation of an incident and the possibility of duplication.
- **M1:** Refines the non-duplicate case, introducing the information gathering, setting the action plan and managing the incident.
- **M2:** Models the ability of action plan update by requesting new resources.
- **M3:** Introduces further incident details such as incident location and type.

- **M4:** Introduces incident priority and its relation to the incident type.
- **M5:** Refines incident management according to the action plan.
- **M6:** Handles the possible loss of an allocated resource.
- **M7:** Details incident handling and the possibility of a resource loss.
- **M8:** Introduces physical resources and handling resource allocation for duplicate incidents.
- **M9:** Adds further details about the location and duration of a resource.

We adopted this refinement sequence because it is important to show from the beginning that there are cases when an incident deviates from its normal path and does not require the same management process. The non-duplicate incident case is the normal path of an incident and the focus of our model, that is why we introduce the main concepts in the non-duplicate incident case as early as possible, while the model is simple and can be easily verified and validated. Next we introduce the action plan because of its vital role in the incident management course of action. Later we introduce design details that determine the required action plan (location, type, priority). Now, it is possible to relate the action plan to the incident management. Having enough details, we can introduce the possible deviation in handling an incident (loss of resource). We then add more details related to the incident handling and the different related cases of resource loss. In all the previous levels we only dealt with logical resources defined by the action plan, now we introduce the actual physical resources and the exception handling actions which are related to the physical resources. More design details related to physical resources are later added.

The refinement stratgey is also influenced by the ERS restriction that a combinator can only be applied to simple events and no combinator can be the direct parent of another combinator. Such restriction promotes for smaller changes at each refinement level.

In the following sections, we show some of the important features in modelling the emergency dispatch system demonstrating the application of the new ERS combinators and their corresponding Event-B specifications.

C. *Using Interrupt for Duplicate Incidents*

In the emergency dispatch model we apply multiple instances modelling to enable possibly simultaneous dispatch to different incidents. Multiple instance modelling is indicated by the addition of the parameter *i*, representing incidents, to the root[2] of the diagram, Fig. 8.

At the abstract level, Fig. 8, we distinguish between two cases of incidents, *duplicate* and *non-duplicate* incidents. We apply the *interrupt* combinator to the events *NotDuplicate* and *IsDuplicate*. At this stage the application of *interrupt* is similar to *xor*, the difference between the two combinators only appears when the normal child (*NotDuplicate*) is refined.

[2]The root of the tree represents the process name and **not** an event.

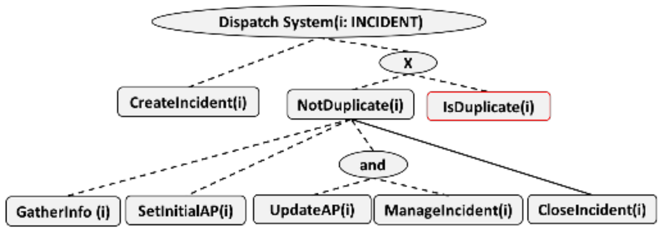
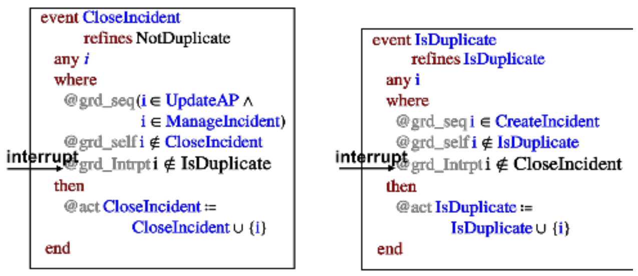


Fig. 8. Distinction Between Duplicate and Non-Duplicate Incidents

The refinement of the non-duplicate case, Fig. 8, shows that an incident requires an initial action plan to be determined (*SetInitialAP*) by the gathered information (*GatherInfo*). The action plan is then followed through during incident management (*ManageIncident*). While managing the incident, the action plan itself may be updated as shown by applying the *and*. After that the incident can be closed (*CloseIncident*). All these events are applied as long as the incident is not interrupted by the *IsDuplicate* event. However, if we use *xor* instead of *interrupt*, once *GatherInfo* executes *IsDuplicate* cannot interrupt the normal flow.


interrupt
interrupt

```
event CloseIncident
      refines NotDuplicate
  any i
  where
   @grd_seq(i ∈ UpdateAP ∧
              i ∈ ManageIncident)
   @grd_self i ∉ CloseIncident
   @grd_Intrpt i ∉ IsDuplicate
  then
   @act CloseIncident :=
             CloseIncident ∪ {i}
 end
```

(a) *CloseIncident* Event

```
event IsDuplicate
      refines IsDuplicate
  any i
  where
   @grd_seq i ∈ CreateIncident
   @grd_self i ∉ IsDuplicate
   @grd_Intrpt i ∉ CloseIncident
  then
   @act IsDuplicate :=
           IsDuplicate ∪ {i}
 end
```

(b) *IsDuplicate* Event

Fig. 9. Event-B Specification of some Events of the Refinement Level (*M1*)

In Fig. 9, we show the Event-B specification of *CloseIncident* and *IsDuplicate* of the first refinement level of the model. All the guards and the actions of these events are the result of the ERS diagram in Fig. 8. The guard $i \notin IsDuplicate$ (Fig. 9(a)) is the result of *interrupt*, and will be also inherited by the events (*GatherInfo*, *SetInitialAP*, *ManageIncident*, *UpdateAP*). Guard $i \notin CloseIncident$ (Fig. 9(b)) is also the result of $interrupt$ to disable *IsDuplicate* after the completion of the normal flow. The *interrupt* combinator behaviour is ensured by the invariant: $CloseIncident \cap IsDuplicate = \phi$.

D. *Parallel Producer-Consumer Pattern for Incident Management*

The initial action plan is followed through during the incident management of non-duplicate incidents. However, in some cases the initial action plan can be updated during the incident management to request additional resources. In Fig. 10, we show how we apply the parallel *producer-consumer* pattern to manage the incident according to a possibly changing action. The *UpdateAp* flow represents the producer part, while *ManageIncident* flow represents the consumer part.

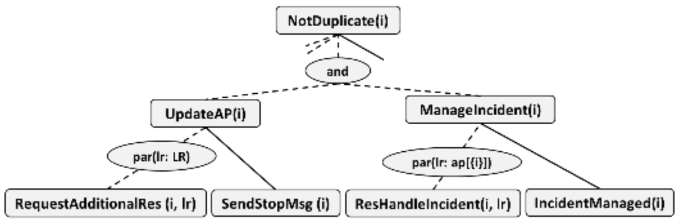


Fig. 10. Parallel *Producer-Consumer* Pattern for Managing Incidents

The new dimension (lr) introduced by *par*, represents the logical resources. The logical resources are introduced by the carrier set (LR) in the extended context. In our model we distinguish between logical resources (*LR*) and physical resources (*PR*). Logical resources are the resources required by the action plan (*ap*) to handle the incident, e.g., an incident may have several different logical resources with the property *class 3 fire engine*. Physical resources, will be introduced in later refinement, are the actual resources to be assigned to the incident, e.g., a specific fire engine. A physical resource, unlike a logical resource, can be only allocated to one incident.

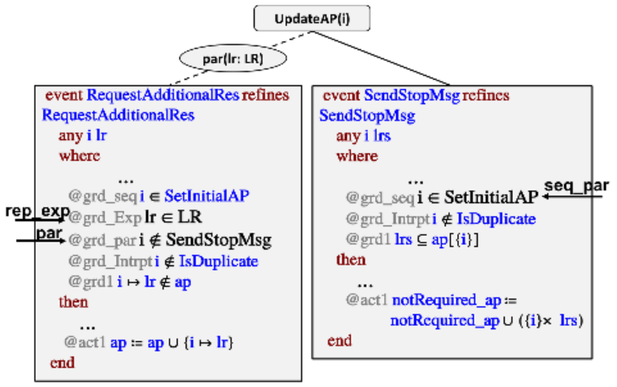
The action plan is represented by the data variable *ap*, which is added manually to the model in textual form and it is defined by the following invariants:

$$\textbf{inv\_ap\_type: } ap \in INCIDENT \leftrightarrow LR$$
$$\textbf{inv\_ap\_seq: } dom(ap) = SetInitialAP$$

The first invariant (*inv_ap_type*) defines the type of the data variable *ap*, which is a relation between incidents and logical resources (LR). The variable *ap* is a relation because an incident can be associated with different logical resources and a logical resource can be associated with different incidents. In the second invariant, we are relating the data variable *ap* to the control variable *SetInitialAP*, this is to ensure that setting the action plan (*ap*) cannot start before the *SetInitialAP* event, and when this event occurs then the incident must have an action plan (*ap*). *inv_ap_seq* shows how we can combine data variables (*ap*) with control variables (*SetInitialAP*) to enforce the requirements using invariants.

The action plan for the incident will first be defined by the event *SetInitialAP*. The action plan can then be updated by adding additional logical resources to the action plan in the event *RequestAdditionalRes*, while the *SendStopMsg* event indicates that the incident manager is satisfied with the resources available at the incident site and no more resources are required to attend. At the same time, the event *ResHandleIncident* will use the logical resources in the action plan (*ap*) to handle the incident, while *IncidentManaged* indicates the completion of the incident handling by completing all the tasks, represented by logical resources, in the action plan (*ap*).

Regarding the producer part of Fig. 10, part of its Event-B specification is shown in Fig. 11. After setting the initial action plan, it is possible to update the action plan (*ap*), by requesting additional resources. This is done textually by updating *ap* in *act1* of *RequestAdditionalRes*. The last guard ($i \mapsto lr \notin ap$) of *RequestAdditionalRes* is also added textually to ensure adding a new fresh resource to incident (i). The rest are all generated by the ERS diagram.



```
event RequestAdditionalRes refines
RequestAdditionalRes
   any i lr
   where
        ...
    @grd_seq i ∈ SetInitialAP
    @grd_Exp lr ∈ LR
    @grd_par i ∉ SendStopMsg
    @grd_Intrpt i ∉ IsDuplicate
    @grd1 i ↦ lr ∉ ap
   then
      ...
   @act1 ap := ap ∪ {i ↦ lr}
  end
```

```
event SendStopMsg refines
SendStopMsg
   any i lrs
   where
        ...
    @grd_seq i ∈ SetInitialAP
    @grd_Intrpt i ∉ IsDuplicate
    @grd1 lrs ⊆ ap[{i}]
   then
      ...
    @act1 notRequired_ap :=
          notRequired_ap ∪ ({i}× lrs)
  end
```

Fig. 11. Event-B Representation of the Incident Management *Producer* Part

The *par* replicator generates two guards in *RequestAdditionalRes*. The first guard ($lr \in LR$) is part of our extension, but associated with any replicator including (*all*, *some*, *one*). This extension is natural to ERS to define the replicator using any expression, unlike the original ERS definition, which only allows constant sets, hence there was no need for such guard. For example, in the consumer event (*ResHandleIncident*) the replicator set is $ap[\{i\}]$, i.e., the relational image of ap with respect to $\{i\}$). The second guard $i \notin SendStopMsg$ will disable requesting additional resources, once a stop message is sent to indicate that there are sufficient resources to deal with the incident. It is not always necessary to request additional resources and change the action plan, this is represented by the possible zero execution case of the *par* replicator. Consequently the sequencing guard of the *par* follow-on event (*SendStopMsg*) and its associated invariant, will **not** depend on the *par* event. Therefore, in Fig. 11, the sequencing guard, $i \in SetInitialAp$, of *SendStopMsg* is the same as that of *RequestAdditionalRes*.

When a stop message is sent, this does not only mean no additional resources are needed but also that the incident has sufficient resources in attendance and no more resources are required even if they are part of the action plan. This behaviour is modelled by introducing the data variable *notRequired_ap* defined as $notRequired\_ap \subseteq ap$. This data variable will be updated with the logical resources of the action plan (*ap*) that are not needed any more to manage the incident, as shown by the manually added action (*act1*) of *SendStopMsg*, which could be also empty. The logical resources that can be added to $notRequired\_ap$, which are defined by *grd1* as any set of logical resources in the action plan of the incident, will be defined more precisely in later refinement as the action plan logical resources that are not in attendance.

Note that both events have inherited the *interrupt* guard ($i \notin IsDuplicate$) of their parent event. In fact every event, having *NotDuplicate* as an ancestor will inherit this guard. Regarding the consumer part all its guards and actions are automatically generated using the ERS Tool [3], in a similar way to the producer part.

We have used a combination of data variables and ERS structures to model the intended behaviour. The parallel *Producer-Consumer* pattern, showed how the action plan can be changed, while the incident is still being managed.

E. Using Retry for Handling Physical Resources

In the previous section, we only dealt with the logical resources of the action plan. In this section we introduce physical resources which are the resources that are actually allocated to an incident and can attend its location. Handling a resource (consumer part) will be refined as shown in Fig. 12, presenting the different stages of incident handling with physical resources. In Fig. 12, we first refine *ResHandleIncident* by applying the *retry* combinator for an individual logical resource *lr* of incident *i* which allows interrupting the *HandleIncident* flow and resetting its control variables in an attempt to retry the *HandleIncident* flow. The incident handling interruption occurs in case a physical resource is removed from the incident due to reallocation to a higher priority incident or due to a breakdown. In this case we need to replace the physical resource with another one to fulfil the tasks of the required logical resource in the action plan. In the second refinement we decompose the atomicity of *HandleIncident* showing the different stages of a logical resource incident. Once a physical resource is allocated, it is possible for two changes to occur to the resource any number of times prior to attendance or receiving a stop message: either the physical resource is replaced by a quicker one that becomes available or the physical resource location is updated. Finally after a resource attends or receives a stop message, it will be deallocated from the incident in *DeallocateRes* event. This flow can only complete in case no interruption occurs.

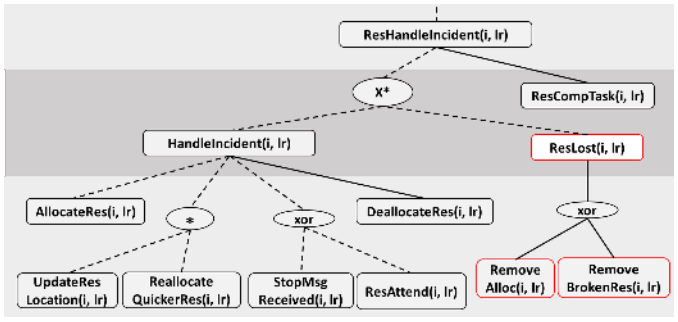


Fig. 12. Introducing Possible Resource Loss and Detailed Incident Handling

As we cannot directly apply a combinator to another in ERS, we introduce *retry* first and then decompose the incident handling. This approach makes the modelling process easier by making smaller changes at each level. For example, applying more than one combinator to several events at the same time can result in more complex sequencing invariants that are harder to read and prove, whereas dividing these changes in separate stages and making use of the ERS refinement gluing invariants will result in simpler sequencing invariants.

The physical resources are introduced using the data variables *alloc* and *alloc_LR*. The following invariants, define *alloc* and *alloc_LR* and their relationship.

inv1: $alloc \in PR \rightarrowtail INCIDENTS$
inv2: $ran(alloc) \subseteq SetInitialAP$
inv3: $alloc\_LR \in PR \rightarrowtail LR$
inv4: $\forall pr, lr . pr \in dom(alloc\_LR) \wedge alloc\_LR(pr) = lr \Rightarrow ptype(pr) = ltype(lr)$
inv5: $dom(alloc) = dom(alloc\_LR)$

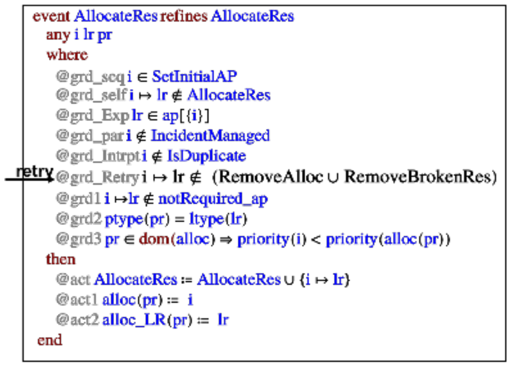
The variable *alloc* maps physical resources to their incident (*inv1*). The variable *alloc* is defined as a partial function so that each physical resource can be assigned to at most one incident. The incident must have an action plan which is why we introduce *inv2* to ensure that incidents are in *SetInitialAP*. Invariant *inv3* defines *alloc_LR*, which assigns a logical resource to a physical resource. This variable will help us in keeping track of which logical resource in the action plan, the physical resource is assigned to. Finally, *ptype* and *ltype* are constants defined in the context to describe the type of the physical and logical resources respectively. The last two invariants ensure that the type of a physical resource must be the same as the type of the logical resource assigned to it, and the equality of the domains ensures that all physical resources allocated to an incident are associated with a logical resource.

After introducing the physical resources, the Event-B specification of *AllocateRes* and *RemoveBrokenRes* is shown in Fig. 13. In Fig. 13(a), the event *AllocateRes* will allocate physical resources to the incident according to the logical resources required by the action plan (*ap*). In Fig. 13, anything related to the physical resource *pr* is modelled textually. The *retry* guard demonstrates the possible interruption by either *RemoveBrokenRes* or *RemoveAlloc*, this guard will be added to every leaf event of *HandleIncident*.

Fig. 13(b) shows one of the *retry* interrupting events in the case that a physical resource (*pr*) breaks down. Similar to *interrupt*, the *retry* guard will disable the interrupting events after incident handling completion. In addition to that, *retry* will reset the control variables of the normal flow (*act_retry1* .. *act_retry4*) to enable retrying the normal flow. An additional resetting event that is not part of the model will also be generated to reset the interrupting events control variables after the interrupting flow completes execution.

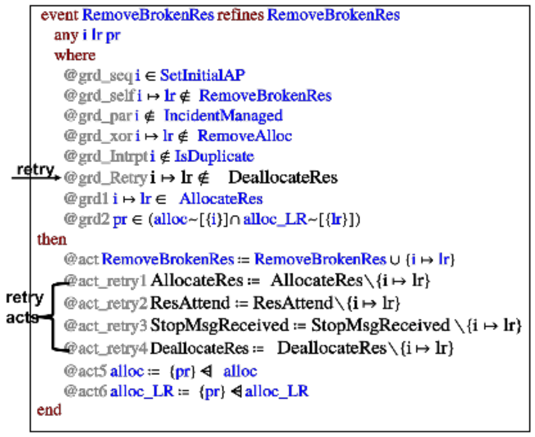
In Fig. 13(b), The textually added actions will deallocate the broken resource from the incident. Therefore, we have modelled the interruption and the handling in the same event. The other interrupting event *RemoveRes* in Fig. 12, is similar to *RemoveBrokenRes*, the only difference is that the physical resource will be already deallocated from the incident due to reallocation to a higher priority incident.

In Fig. 13, both events have the interrupting guard $i \notin IsDuplicate$ because the duplicate interrupt is introduced at a higher level. Therefore, *IsDuplicate* can interrupt the lower level interrupts (*RemoveAlloc*, *RemoveBrokenRes*).



```
event AllocateRes refines AllocateRes
  any i lr pr
  where
   @grd_scq i ∈ SctInitialAP
   @grd_self i ↦ lr ∉ AllocateRes
   @grd_Exp lr ∈ ap[{i}]
   @grd_par i ∉ IncidentManaged
   @grd_Intrpt i ∉ IsDuplicate
   @grd_Retry i ↦ lr ∉  (RemoveAlloc ∪ RemoveBrokenRes)
   @grd1 i ↦lr ∉ notRequired_ap
   @grd2 ptype(pr) = ltype(lr)
   @grd3 pr ∈ dom(alloc) ⇒ priority(i) < priority(alloc(pr))
  then
   @act AllocateRes ≔ AllocateRes ∪ {i ↦ lr}
   @act1 alloc(pr) ≔  i
   @act2 alloc_LR(pr) ≔  lr
 end
```

(a) *AllocateRes* Event



```
event RemoveBrokenRes refines RemoveBrokenRes
  any i lr pr
  where
   @grd_seq i ∈ SetInitialAP
   @grd_self i ↦ lr ∉  RemoveBrokenRes
   @grd_par i ∉  IncidentManaged
   @grd_xor i ↦ lr ∉  RemoveAlloc
   @grd_Intrpt i ∉ IsDuplicate
   @grd_Retry i ↦ lr ∉    DeallocateRes
   @grd1 i ↦ lr ∈   AllocateRes
   @grd2 pr ∈ (alloc∼[{i}]∩alloc_LR∼[{lr}])
then
   @act RemoveBrokenRes ≔ RemoveBrokenRes ∪ {i ↦ lr}
   @act_retry1 AllocateRes ≔  AllocateRes\{i ↦ lr}
   @act_retry2 ResAttend ≔ ResAttend\{i ↦ lr}
   @act_retry3 StopMsgReceived ≔ StopMsgReceived \{i ↦ lr}
   @act_retry4 DeallocateRes ≔  DeallocateRes\{i ↦ lr}
   @act5 alloc ≔  {pr} ⩤  alloc
   @act6 alloc_LR ≔  {pr} ⩤alloc_LR
end
```

(b) *RemoveBrokenRes* Event

Fig. 13. Event-B Specification of some Events of the *retry* combinator

Another feature related to *par* refinement can be observed here. When the *par* event is decomposed, the *par* disabling guard will be added to the first event of the flow, but here since *retry* has two non-commutative flows, the *par* guard is added to the first event of the normal (*AllocateRes*) and Interrupting flow (*RemoveBrokenRes*, *RemoveAlloc*), the same applies for *interrupt*. In addition to that, when a *par* event is decomposed, an additional invariant will be generated, with a corresponding guard added to the *par* follow-on event. In the last refinement level of Fig. 13, the *par* refinement invariant will be:

$$\textbf{inv\_par:}\ \forall i \cdot i \in IncidentManaged \Rightarrow AllocateRes[\{i\}] = ResCompTask[\{i\}]$$

This invariant ensures that the *par* flow must complete for an instance if it already started. When decomposing the *par* event into some sequence of sub-events, we consider the activity of the *par* event is represented by the execution of all the sub-events and not only the refining event. Additionally, the *par* refinement invariant has an important role in verifying the model. For instance in Fig. 12, the solid event, *ResCompTask*, is the refining event of *ResHandleIncident*, hence its guards must be stronger than that of its corresponding abstract event according to the *GRD* Proof obligation. However, *grd_par* of *ResHandleIncident* is only added to the first event of the *par* flow (*AllocateRes*). Having this refinement invariant will ensure satisfying the *GRD* proof obligation and avoid the unnecessary repetition of guards.

This *par* refinement invariant also helps in maintaining the incident closure requirement, that any allocated resources must eventually be deallocated before closing an incident call, this is because the *retry* sequencing invariant of *ResCompTask*: $ResCompTask \subseteq DeallocateRes$.

F. Handling Interrupting Duplicate Incidents Using All

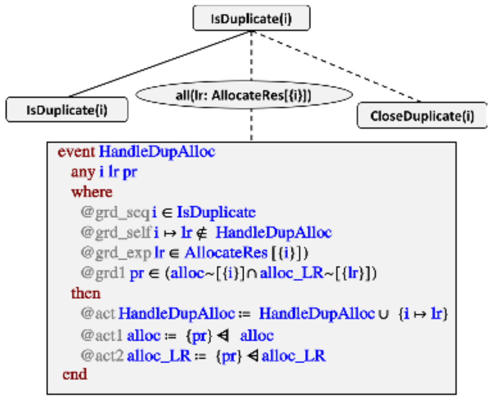
Closing the incident requires that no resources are still assigned to the incident and if the incident is not duplicate, all the required action plan tasks are completed. The following invariants are added manually to ensure that the incident closure requirements are maintained by all the events.

$$\textbf{inv6:}\ \forall i . i \in (CloseIncident \cup CloseDuplicate) \Rightarrow i \notin ran(alloc)$$

$$\textbf{inv7:}\ \forall i . i \in CloseIncident \Rightarrow ap[\{i\}] \setminus notRequired\_ap[\{i\}] \subseteq DeallocateRes[\{i\}]$$

In Fig. 13, *AllocateRes* will add new values to *alloc* and *alloc_LR*, whereas *DeallocateRes* will remove physical resources from them, the same way *RemoveBrokenRes* does. Therefore, *DeallocateRes* will ensure that an incident cannot be closed if it still has allocated physical resources. This is enforced by *inv6* which states that an incident whether duplicate or not can be only closed if it has no physical resources allocated to it. As a result a guard will be added to the incident closure events checking for this property. Invariant *inv7* ensures that all required resources have completed their tasks and have been deallocated before the incident can be closed. This is done by checking that all logical resources in the action plan of the incident *i*, that are not marked as not required as a result of a send stop message, are part of the relational image of *DeallocateRes* with respect to incident *i*.

Regarding *inv6*, *DeallocateRes* will ensure that physical resources are deallocated from non-duplicate incidents, but what about duplicate incidents that are discovered late as duplicate, i.e., after resource allocation is done? We handle this issue by refining the *IsDuplicate* event as shown in Fig. 14.



```
event HandleDupAlloc
  any i lr pr
  where
   @grd_scq i ∈ IsDuplicate
   @grd_self i ↦ lr ∉  HandleDupAlloc
   @grd_exp lr ∈ AllocateRes [{i}])
   @grd1 pr ∈ (alloc∼[{i}]∩alloc_LR∼[{lr}])
  then
   @act HandleDupAlloc ≔  HandleDupAlloc ∪  {i ↦ lr}
   @act1 alloc ≔  {pr} ⩤  alloc
   @act2 alloc_LR ≔  {pr} ⩤alloc_LR
 end
```

Fig. 14. Handling Allocated Resources of a Duplicate Incident

Using *all* in *HandleDupAlloc* ensures that *inv6* is satisfied by removing all the physical allocations before closing the duplicate incident in the textually added actions *act1* and *act2*. This is an example of how we compensated the allocated resources in the case of duplicate exception. In Fig. 14, it was possible to use *all* and benefit from its sequencing constraint (*inv_dup_seq*) that affect *CloseDuplicate* because the control variable *AllocateRes*, which is the parameter set of *all*, cannot be changed once *IsDuplicate* executes.

$$\textbf{inv\_dup\_seq:}\ \forall i.i \in CloseDuplicate \Rightarrow HandleDupAlloc[\{i\}] = AllocateRes[\{i\}]$$

Invariant *inv_dup_seq*, resulting from the *all* replicator, ensures that all allocated resources are handled before closing a duplicate incident. This handling mechanism can be generalised for any multi-instances cases, where we can use *all* in conjunction with *interrupt* to model the exception and the handling mechanism.

V. Model Analysis and Discussion

A. ERS Extensions

ERS hierarchical development and its explicit support for event ordering manage modelling complexity in Event-B. However, the existing ERS combinators are dependent on static data and cannot adapt to data changes during the model execution. In [11], we tried to model the emergency dispatch system using ERS without extensions and UML activity diagrams, however we ended up simplifying the requirements and omitting the design details related to adapting to dynamic behaviour such as:

- Ability to change the action plan and respond to those changes after the action plan starts execution.
- Ability to model one workflow instance impacting another workflow instance, e.g., re-allocation of resources from a low priority incident to a high priority incident.
- Ability to interrupt the execution of the workflow due to an exception, and how to handle such exceptions.

It is possible to model the last two requirements with activity diagrams using interruptible regions. However, modelling the first one is not possible because activity diagrams do not support multi-instance executions without predetermined knowledge [12]. Using ERS without the extensions, it is possible to represent interruption, e.g., duplicate incident interruption, using *xor*. The disadvantage will be having several *xor* branches to represent the different points the interruption can take place. For example considering Fig. 8 without the further refinements and event decomposition, we will need six *xor* branches. Such a large number of *xor* branches will not only make the ERS diagram less readable, but will also result in a complex Event-B model.

The ERS *loop*, similar to *retry*, allows repeating behaviour by resetting the control variables. However, the *loop* allows the normal behaviour to complete more than once, while *retry* allows the completion of the normal behaviour only once and the interrupting event can occur more than once until the normal child completes successfully. The *loop* also does not support the interruption feature, hence it will be difficult to achieve a behaviour similar to *retry* using the *loop*, even if combined with other combinators.

Comparing *par* with the other ERS replicators, it is still difficult to achieve the same behaviour even if their replicator sets were allowed to change during execution. *Par* offers the additional feature of checking a certain milestone is reached, represented by the follow-on event, before stopping the behaviour replication.

B. Model Statistics

For the nine refinement levels of the emergency dispatch system, we have 452 out of 479 proof obligations (94.4%) automatically discharged using the Rodin provers. The rest were proved interactively in Rodin. As discussed earlier ERS can contribute to generating variables, invariants, events, guards, actions. In Table I, we present the percentages of ERS generated guards and actions compared to those added textually using Event-B.

TABLE I
Statistics of ERS generated and textually added Event-B guards and actions

	ERS	**Event-B**	**% grds**	**ERS**	**Event-B**	**% acts**
M0	7	0	100	3	0	100
M1	19	0	100	7	0	100
M2	23	5	82	8	3	73
M3	29	9	76	10	6	62.5
M4	35	14	71	11	8	58
M5	41	14	74.5	12	8	60
M6	51	14	78.5	16	8	67
M7	82	24	77	27	8	77
M8	88	36	71	30	18	62.5
M9	0	10	0	0	3	0
Total	88	46	66	30	21	59

In all the machines ($M0$-$M8$), we use ERS refinement, except in the last refinement level, we only extend the model with some design details using textually added data variables. The first two machines are exclusively generated by the ERS diagrams with no textually added Event-B details. Using the ERS refinement approach, the events are only refined and not extended, that is why we have an increasing number of guards and actions from $M0$ to $M8$. Almost two thirds of the total guards and more than half the total actions are generated by the ERS diagram. The percentage of generated guards is higher than that of actions because the ERS actions are only used to record the occurrence of the event or to reset it (e.g., *retry*) while the data manipulation is done textually using Event-B. The ERS guards not only contribute to the ordering of the events but also the refinement correctness and in several cases we related the ERS control variables to the data variables using invariants, which saved us from having additional guards to model the system behaviour. This shows how ERS can reduce the modelling effort especially if used properly before starting the actual Event-B modelling to make design decisions.

VI. Conclusions and Related Work

In this paper we have presented an extension to ERS that support dynamic behaviours, such as the parallel *producer-consumer* pattern, in addition to supporting interruptions and presenting different exception handling techniques (*retry*, *interrupt + all*), making ERS more adaptable to changes. We have also shown how to translate the ERS extensions to Event-B by applying them to the emergency dispatch system. ERS support for the Event-B stepwise refinement and its combinators facilitate the Event-B modelling and help make refinement decisions before starting the formal modelling using Event-B, that is why it is important to make ERS more dynamic, widening the scope of ERS application.

Various approaches integrate state-based and process-based formalisms in an attempt to explicitly model workflows. For instance, [13] integrates Event-B with CSP, while [14], [15] extends Event-B with special expressions similar to process algebra, called flows. A new language (Circus) combining Z and CSP is defined in [16]. Combination of classical B and CSP is also defined in [17], [18]. ERS gets its semantics by transforming it to Event-B and can contribute to the underlying Event-B model, hence its meaning is given entirely using Event-B. Such strong integration gives the ERS control variables access to the state-based variables, making it possible to relate them using invariants. This is in contrast with the above mentioned approaches where they essentially combine different formalisms. Furthermore, the process algebra based constructors do not support interruptions such as the new ERS *interrupt* and *retry*. Similar to ERS, iUML-B statemachines [19], [20], explicitly model control flow in Event-B and contribute to the underlying Event-B model. However, iUML-B statemachines is state-oriented explicitly modelling the transitions from one state to the other, in contrast to the process-oriented ERS. All the previously mentioned approaches do not provide explicit visualisations of the event refinement relationships the way ERS does.

Comparing the new ERS combinators to popular workflow modelling approaches such as UML [21] activity diagrams and BPMN [22], both do not support the addition of new instances once a task has commenced execution. This is Dynamic replication without a priori knowledge, which is one of the patterns of the workflow patterns initiative [12], Unlike BPMN and activity diagrams, this pattern is supported by the ERS *par* replicator and the parallel *producer-consumer* pattern. On the other hand, both support structured and arbitrary cycles, whereas ERS only supports structured iteration and replication.

Regarding exception handling, [23] and [24] formally define BPEL compensation mechanisms using Event-B, with [23] focusing on the role of Event-B invariants during refinement. Our approach only provides the interrupting mechanism which the user can refine and define their own handling and compensating activities, with *retry* having an additional feature that supports compensation by facilitating redoing the activity.

ERS can result in many refinement levels, hence the need to tackle modularity by model decomposition. In the future, we would like to integrate model decomposition with ERS and show how ERS can benefit the model decomposition strategies.

Acknowledgment

This work is partly supported by the PRiME project (See www.prime-project.org for more information).

References

[1] A. Fathabadi, M. Butler, and A. Rezazadeh, "A Systematic Approach to Atomicity Decomposition in Event-B," vol. 7504, pp. 78–93, 2012.

[2] M. Butler, "Decomposition Structures for Event-B," in *IFM*. Springer, 2009, vol. LNCS 5423, pp. 20–38.

[3] D. Dghaym, M. Trindade, M. Butler, and A. Fathabadi, "A Graphical Tool for Event Refinement Structures in Event-B," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2016, pp. 269–274.

[4] J. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in Event-B," *International Journal on Software Tools for Technology Transfer*, vol. 12, pp. 447–466, 2010.

[5] M. Leuschel and M. Butler, "ProB: A model checker for B," in *FME 2003: Formal Methods*, ser. LNCS 2805, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Springer-Verlag, 2003, pp. 855–874.

[6] J. Abrial, *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[7] ——, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[8] R.-J. Back, "Refinement calculus, part II: Parallel and reactive programs," in *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*. Springer, 1990, pp. 67–93.

[9] A. S. Fathabadi, M. Butler, and A. Rezazadeh, "Language and tool support for event refinement structures in Event-B," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 499–523, May 2015. [Online]. Available: http://eprints.soton.ac.uk/366750/

[10] M. A. Jackson, *System Development*. Englewood Cliffs, N.J. : Prentice-Hall, 1983.

[11] D. Dghaym, M. Butler, and A. Fathabadi, "Evaluation of graphical control flow management approaches for Event-B modelling," *Electronic Communications of the EASST*, 2013.

[12] N. Russell, A. H. Ter Hofstede, and N. Mulyar, "Workflow controlflow patterns: A revised view," 2006.

[13] S. Schneider, H. Treharne, and H. Wehrheim, "A CSP approach to control in Event-B," in *International Conference on Integrated Formal Methods*. Springer, 2010, pp. 260–274.

[14] A. Iliasov, "On Event-B and control flow," 2009.

[15] ——, "Tutorial on the Flow plugin for Event-B," 2010, In: Workshop on B Dissemination [WOBD] Satellite event of SBMF, Natal, Brazil.

[16] J. Woodcock and A. Cavalcanti, "The semantics of circus," in *International Conference of B and Z Users*. Springer, 2002, pp. 184–203.

[17] M. Butler and M. Leuschel, "Combining CSP and B for Specification and Property Verification," January 2005, event Dates: 18-22 July 2005. [Online]. Available: https://eprints.soton.ac.uk/260388/

[18] S. Schneider and H. Treharne, "CSP theorems for communicating B machines," *Formal Aspects of Computing*, 2005.

[19] C. Snook, "iUML-B Statemachines: New Features and Usage Examples," in *Proceedings of the 5th Rodin User and Developer Workshop, 2014*, M. Butler and S. Hallerstede, Eds. University of Southampton.

[20] C. Snook and M. Butler, "UML-B and Event-B: an integration of languages and tools," in *The IASTED International Conference on Software Engineering - SE2008*, 2008.

[21] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1998.

[22] M. Chinosi and A. Trombetta, "BPMN: An introduction to the standard." *COMPUTER STANDARDS & INTERFACES*, vol. 34, no. 1, pp. 124 – 134, 2012.

[23] G. Babin, Y. At-Ameur, and M. Pantel, "Web Service Compensation at Runtime: Formal Modeling and Verification Using the Event-B Refinement and Proof Based Formal Method," *IEEE Transactions on Services Computing*, vol. 10, no. 1, pp. 107–120, Jan 2017.

[24] I. Ait-Sadoune and Y. Ait-Ameur, *Formal Modelling and Verification of Transactional Web Service Composition: A Refinement and Proof Approach with Event-B*. Springer, 2015, pp. 1–27.