A Graphical Tool for Event Refinement Structures in Event-B

Dana Dghaym[1], Matheus Garay Trindade[2], Michael Butler[1], and Asieh Salehi Fathabadi[1]

[1] University of Southampton, UK
{dd4g12, mjb, asf08r}@ecs.soton.ac.uk
[2] Federal University of Santa Maria, Brazil
mtrindade@inf.ufsm.br

Abstract. The Event Refinement Structures (ERS) approach provides a graphical extension of the Event-B formal method to represent event decomposition and control-flow explicitly. In this paper we present an improved version of the ERS plug-in, which provides a graphical environment for the ERS approach within the Event-B tool, Rodin. The improved ERS plug-in is based on the available frameworks that are developed to support Event-B with an EMF framework, language extensions and generic diagram extensions.

1 Introduction

The ERS plug-in provides an automatic generation of part of the Event-B model related to the ordering of events and their relationships at different refinement levels. The ERS plug-in can generate additional variables, events, guards, actions and invariants to an Event-B machine.

The ERS language is defined using the Eclipse Modelling Framework (EMF) [1] meta-model, and then transformed into an Event-B EMF meta-model [2]. In the earlier version of the tool [4], an ERS diagram was defined in an EMF tree structure. The transformation from the ERS language to Event-B was performed using the Epsilon Transformation Language (ETL) [3].

In the updated version of the ERS plug-in, we provide a graphical environment for ERS, and we apply a different approach to transform from the ERS language to Event-B. The new approach followed is based on the generic Diagram Extensions framework for Event-B [5]. The framework used, is built specifically to support Event-B providing lots of helpful functionalities. In addition to supporting model transformation to Event-B, the generic Diagram Extensions framework provides graphical and validation support. Unlike, ETL which is a generic model to model transformation language, and supporting a graphical interface and validation requires other tools adding more learning efforts. Moreover, at the time we used ETL, we had technical problems related to debugging and code auto-completion features, while the current approach simply uses Java.

2 Event Refinement Structures (ERS) Approach

The ERS approach provides a tree-like graphical representation of the events, with an explicit representation of the events ordering and the refinement relationships. We illustrate the ERS approach using a small example of the order workflow, Figure 1. The root of the tree, *Order_Workflow(o)*, represents the name of the flow-diagram and the parameter "*o*" indicates multiple instances. Multiple instances means different instances of a workflow may be executed in an interleaved manner. If no parameter was provided, then the ERS diagram will represent a single instance of the flow.

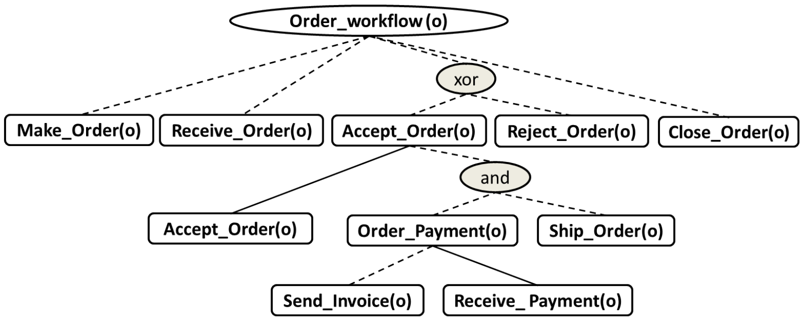


Fig. 1. Overall ERS Diagram of the Order Workflow

The leaves, *Make_Order, Receive_Order* etc., will be transformed into events in the Event-B machine. The ordering of the leaf events is from left-to-right, so *Make_Order* can execute first, followed by *Receive_Order*, etc. To describe the control-flow of the events using Event-B, variables with the same name of the leaf events are generated. We refer to these variables as control variables. The type of the control variables is boolean in the case of single instance modelling, and a set in the case of multiple instances modelling. These control variables are used in the Event-B model to specify the control-flow of the events using invariants and guards.

In ERS, the dashed lines indicate that the events are newly added events, and they are not refining events. Therefore, the abstract level of Event-B, first row, can only have dashed lines. ERS also allows the addition of different combinators between events, represented within an oval shape. In Figure 1, we used two different combinators, the *and-combinator* and the *xor-combinator*. The *xor-combinator* indicates the exclusive choice between events, in this case either *Accept_Order* or *Reject_Order*, but not both, can execute before enabling the event *Close_Order*.

In the first refinement, second row, *Accept_Order* is the only refined event. The solid line means a direct refinement of an event, indicated using the keyword "*refines*" in Event-B. ERS requires that an event can be only refined by one event, this is referred to as "single solid line rule". The only case in ERS, where an event can be refined by more than one event, is by introducing refinement using the *xor-combinator*, which only allows the execution of one event without requiring mutually exclusive guards. The event *Accept_Order* is decomposed into the sequence of events *Accept_Order* followed by the interleaved execution of the events *Order_Payment* and *Ship_Order*, as a result of applying the *and-combinator*. Similarly at the second level of refinement, *Order_Payment* is decomposed into the sequence of events *Send_Invoice* followed by *Receive_Payment*, where *Receive_Payment* is the directly refining event and *Send_Invoice* is a newly added event.

In Figure 2, we present part of the ERS meta-model, showing the *and-combinator*. The meta-model describes the different classes of an ERS diagram, such as the *FlowDiagram*, *Leaf*, *Child*. Each class can have its own attributes, for example a *Child* has the boolean attribute *ref* to determine whether it is refining or not, this is reflected by the solid and dashed lines in the diagram.

Relationships and associations between classes can be represented by the links between them. A link with solid diamond at the end represents a containment association between the two classes, for example a *FlowDiagram* can contain one or more *Children* as indicated by the upper and lower bounds of the association (1..*), and it can refer to a *Child* using the *refine* relation. A link with a triangular end represents a specialisation relation, for example a *Leaf* and a *Constructor* are special types of *Child* and they inherit all its properties.

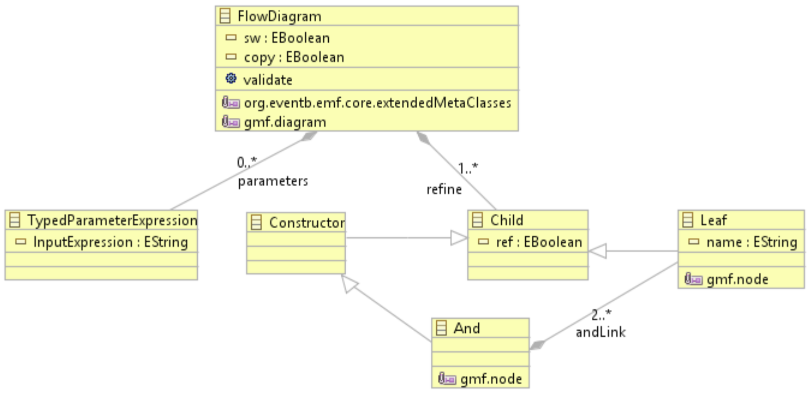
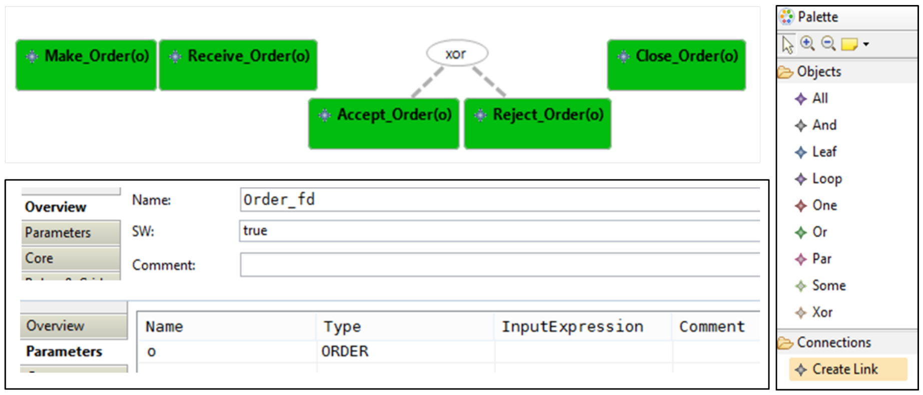


Fig. 2. Part of the ERS Meta-Model

3 The ERS Plug-in

Using the ERS plug-in, we first need an Event-B machine, then we can add an ERS flow-diagram to the machine. We can start from an empty machine, but we need to define the "*sees*" relationship to a context, if parameters are needed. Figure 3 presents an image of the tool interface for the abstract level of the ERS diagram in Figure 1. *Overview* and *Parameters*, panels on the left, are properties of the ERS flow-diagram that can be updated by the user. The palette on the right shows the different combinators available for an ERS diagram such as the *xor*, while the leaf, e.g. *Make_Order(o)*, will be translated to an event in Event-B. Figure 4 shows part of the generated Event-B from the ERS diagram



Name:	Order_fd
SW:	true
Comment:	

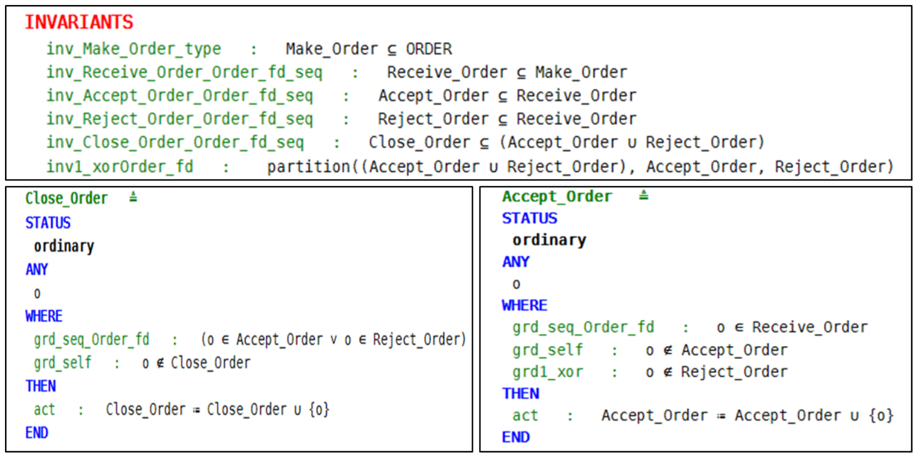
Name	Type	InputExpression	Comment
o	ORDER		

Fig. 3. The ERS Diagram of the Order Workflow at the Abstract Level

of Figure 3. In the ERS plug-in we also support some validation of the diagram, such as the single solid line rule mentioned above. We only show the invariants and some of the events generated, but the complete generated Event-B also includes control variables to support the control-flow of events, initialisations of the control variables and events for every single leaf. In ERS if the leaf does not already exist in the machine, a new event will be generated with the same name of the leaf. Otherwise, we add the generated guards and actions to the existing events.

When refining a machine, the ERS diagram at the previous level will be copied to the refined machine. Then the user can right click and refine the required leaves. After refining an ERS diagram, all the old generated parts of the Event-B will be deleted and updated according to the new refinement. For example after the first refinement, the sequencing guard of the *Close_Order* event will be updated to $((o \in Order\_Payement \wedge o \in Ship\_Order) \vee o \in Reject\_Order)$ as a result of applying *and-refinement* to the preceding event.

Generating the Event-B elements from the ERS diagram is based on the generator framework which is part of the generic Diagram Extension framework [5, 6]. Each rule transforming an ERS element to an Event-B element implements the *Irule* and defines the methods *enabled*, *dependenciesOK*, and *fire*. The *enabled* method checks when the translation rule should be applied, e.g. the rule transforming a leaf to a variable is enabled if the ERS source element is a leaf



```
INVARIANTS
  inv_Make_Order_type   :   Make_Order ⊆ ORDER
  inv_Receive_Order_Order_fd_seq   :   Receive_Order ⊆ Make_Order
  inv_Accept_Order_Order_fd_seq   :   Accept_Order ⊆ Receive_Order
  inv_Reject_Order_Order_fd_seq   :   Reject_Order ⊆ Receive_Order
  inv_Close_Order_Order_fd_seq   :   Close_Order ⊆ (Accept_Order ∪ Reject_Order)
  inv1_xorOrder_fd   :   partition((Accept_Order ∪ Reject_Order), Accept_Order, Reject_Order)
```

```
Close_Order  ≙
STATUS
 ordinary
ANY
 o
WHERE
 grd_seq_Order_fd   :   (o ∈ Accept_Order ∨ o ∈ Reject_Order)
 grd_self   :   o ∉ Close_Order
THEN
 act   :   Close_Order ≔ Close_Order ∪ {o}
END
```

```
Accept_Order  ≙
STATUS
 ordinary
ANY
 o
WHERE
 grd_seq_Order_fd   :   o ∈ Receive_Order
 grd_self   :   o ∉ Accept_Order
 grd1_xor   :   o ∉ Reject_Order
THEN
 act   :   Accept_Order ≔ Accept_Order ∪ {o}
END
```

Fig. 4. Part of the Generated Event-B at the Abstract Level of the Order Workflow

that is not a child of a loop. The *dependenciesOK* method checks if there are any dependencies required like other elements that need to be generated first, e.g. generating a sequencing guard to an event requires the event to be generated first or already existing in the machine, so if dependencies are not satisfied firing the rule will be postponed until all dependencies are satisfied. The *fire* method is the main method where the mapping of ERS elements to Event-B elements takes place. The *fire* method returns a list of *GenerationDescriptors* describing what should be generated, e.g. an ERS *leaf* is mapped to an *event* in Event-B.

Every element generated from the ERS diagram is read-only except for the generated events. They are editable so that users can add application-specific guards and actions using an Event-B editor. The *GenerationDescriptor* has the option whether to mark a generated element as editable or not. In ERS, when refining a leaf, all the elements that are not generated, e.g. the manually added guards and actions, will be passed over to the solid leaf.

4 Conclusions

ERS is a graphical approach that explicitly describe the control-flow of the Event-B events and helps to structure refinement. UML-B [7] is one of the important approaches that provides Event-B with a graphical front-end. UML-B supports class and state-machine diagrams. Similar to ERS, the UML-B state-machine can explicitly describe control-flow in Event-B and supports hierarchical decomposition. The main difference between the two approaches is that UML-B state-machines focus on the state transitions, whereas ERS diagrams focus on the events. depending on the problem one can decide which approach is more appropriate. In many cases both approaches can complement each other, especially after the introduction of the new integrated form of UML-B, iUML-B [8].

The new ERS plug-in provides a graphical environment for the ERS approach, and also supports the validation of the ERS diagrams. Applying the tool in modelling a complex case study resulted in having more consistent and systemic models, faster modelling and the graphical front-end made understanding and validating the model easier.

Using frameworks that directly support Event-B made the development of the plug-in easier and more systematic, by providing different functionalities for defining transformation rules, Event-B generators and graphical editors. In addition to providing various convenience classes and methods for Event-B, like making and finding Event-B elements. In the future, we would like to add more validations to the diagrams, enhance some of the translation rules and allow the users to add application-specific guards, actions and invariants from within the ERS graphical environment without the need to switch to the Event-B editor.

References

1. Steinberg D., Budinsky F., Paternostro M. and Merks E.: EMF: Eclipse Modeling Framework, 2nd Edition. Addison-Wesley Professional, (2008)
2. Snook C., Fritz F. and Illisaov A.: An EMF Framework for Event-B. In: Workshop on Tool Building in Formal Methods, ABZ Conference, Orford, (2010)
3. Kolovos D., Rose, L., Garcia-Dominguez A. and Paige R.: The epsilon Book. Eclipse.org, (2014). http://www.eclipse.org/epsilon/doc/book/
4. Salehi Fathabadi A., Butler M. and Rezazadeh A.: Language and Tool Support for Event Refinement Structures in Event-B. Formal Aspects of Computing, (2015)
5. Savicks V. and Snook C.: A Framework for Diagrammatic Modelling Extensions in Rodin. In: Rodin Workshop, Fontainbleau, (2012)
6. Wiki.event-b.org: Generic Event-B EMF extensions - Event-B. http://wiki.event-b.org/index.php/Generic_Event-B_EMF_extensions, (2016)
7. Snook C. and Butler M.: UML-B and Event-B: an integration of languages and tools. In: The IASTED International Conference on Software Engineering - SE2008, (2008)
8. Snook C.: iUML-B Statemachines: New Features and Usage Examples. In: Proceedings of the 5th Rodin User and Developer Workshop,(2014)