Electronic Communications of the EASST
Volume 66 (2013)

Proceedings of the
Automated Verification of Critical Systems
(AVoCS 2013)

Evaluation of Graphical Control Flow Management Approaches for Event-B Modelling

Dana Dghaym, Michael Butler and Asieh Salehi Fathabadi

15 pages

Guest Editors: Steve Schneider, Helen Treharne
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Evaluation of Graphical Control Flow Management Approaches for Event-B Modelling

Dana Dghaym[1], **Michael Butler**[2] **and Asieh Salehi Fathabadi**[3]

dd4g12[1], mjb[2], asf08r[3]@ecs.soton.ac.uk
University of Southampton, UK

Abstract: Integrating graphical representations with formal methods can help bridge the gap between requirements and formal modelling. In this paper, we compare and evaluate two graphical approaches aiming at describing control flows and refinement in Event-B, and we use a fire dispatch system case study to perform this evaluation. The fire dispatch system case study provides a good example of a complex workflow through which we try to identify a process that facilitates defining the structural and the behavioural parts of the Event-B model. In our case study, we focus on building the dynamic part of the model to evaluate the two diagrammatic notations: UML Activity Diagrams and Atomicity Decomposition Diagrams. Based on our evaluation, we try to identify the advantages and limitations of both approaches. Finally, we try to compare how both graphical notations can affect the Event-B formal modelling of our case study.

Keywords: Event-B, Atomicity Decomposition, UML Activity Diagram, Control Flow

1 Introduction

From requirements described in free natural language to specifications in mathematical notations with strict semantics, a large gap exists that suggests the need for a semi-formal intermediate layer. That is why many studies and experiments, such as [RSP07], suggest the integration of graphical representations with formal methods to reduce the gap from the requirements document to the formal model on one hand, and facilitate the understanding of the formal models effectively on the other hand.

In this paper we study two graphical techniques that describe control flows and refinement in the Event-B formal method. These modelling techniques are UML activity diagrams (ActD) and atomicity decomposition (AD) diagrams, which have a hierarchical based structure but are not as well known to industry as UML diagrams. We apply these representations to a fire dispatch system case study, that has a complex control flow and thus these techniques can be very useful to simplify the problem.

In this case study, we try to compare and evaluate these two diagrammatic approaches, where both have their advantages and limitations. Atomicity decomposition's strength appears in its natural approach to describe explicit refinement and explicit constructs to describe replications and multiple instances in control flows. Activity diagram's strength lies in the flexibility of its layout supported by directed arrows and the use of guards explicitly in decision-nodes. However, both have some limitations when it comes to multiple instances replicators, atomicity decomposition with the limited behaviour of its all-construct, and activity diagram with the lack of some explicit replicator constructs like the some-construct.

This paper is organised as follows: Section 2 describes some needed background information about Event-B, activity diagrams and atomicity decomposition. Then in Section 3, we describe our case study by presenting the requirements, the entity diagram that presents the structure of the model, and finally to describe the behavioural part of our case study, we use activity and atomicity decomposition diagrams. Later in Section 4, we compare and evaluate the dynamic representations of the fire dispatch system, both graphically and in relation to the Event-B model. Finally, we compare our approach with other related work and conclude in Sections 5 and 6 respectively.

2 Background

2.1 Event-B

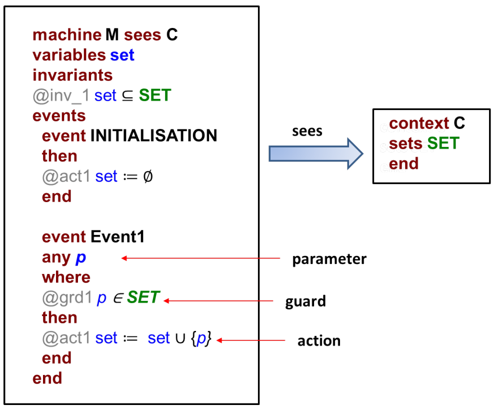
Event-B is a formal method for modelling complex systems [Abr10]. Its language is based on the set theory and first order logic. Correctness and consistency of models, are proved by means of mathematical proofs [ABH+10].

Refinement & Decomposition: Refinement is a key concept in the Event-B modelling, where instead of building a single model, we start with an abstract one, focusing on the main functionalities of the system. Then during refinement levels, details of the model and/or new functionalities are added gradually. Refinement process aids understanding the model and proving its correctness and consistency [AH07]. However, adding lots of details at different refinement levels, sometimes results in complex and long models. In some cases only parts of the model need to be refined, hence the need for model decomposition, which is another key concept in Event-B. In model decomposition, we divide a large model into some sub-models, each can be refined individually [AH07].

Structure: The Event-B model is divided into a static part, the "Context" and a dynamic part, the "Machine". The context consists of the set types, constants and their properties as axioms. The machine consists of variables, invariants describing the constraints applied on the variables, and events that show how the variables' values change provided some guard conditions hold. In addition, theorems can be defined in both context and machine [MAV05]. Figure 1 presents a simple Event-B model showing both the static and the dynamic parts of the model. Machine *M*, on the left, consists of two events: the *INITILIASATION* event and "*Event1*", which adds values to the variable "*set*". In order to define this variable as a subset of "*SET*" in context *C*, using invariant "*inv_1*" enforced by the guard condition "*grd1*", we add the "*sees*" clause to machine *M* to access context *C*.

2.2 UML Activity Diagram

Unified Modelling Language (UML) [RJB98], is a graphical representation for modelling object-oriented systems. UML consists of many graphical types representing objects, behaviours, and states. Activity diagram (ActD) is one of these types, that describes the behavioural, or the



```
machine M sees C
variables set
invariants
@inv_1 set ⊆ SET
events
  event INITIALISATION
  then
  @act1 set ≔ ∅
  end

  event Event1
  any p
  where
  @grd1 p ∈ SET
  then
  @act1 set ≔  set ∪ {p}
  end
end
```

```
context C
sets SET
end
```

Figure 1: Simple Event-B Model

dynamics, part of the system [Obj12].

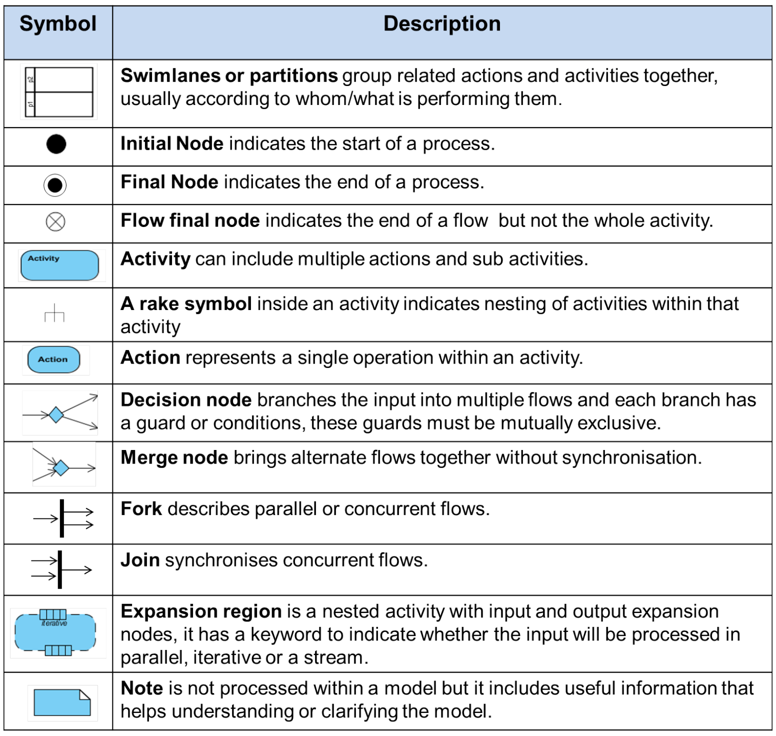
An ActD is a directed graph, in which the direction of the arrows suggests the flow of activities. In older versions of UML, an ActD was considered as a type of state machine diagram, but since the introduction of version 2.0, ActD semantics is widely enriched, based on token flow inspired by Petri Nets [Obj12]. However, it still lacks a formal definition and that is why many researchers have tried to define a formal semantics of UML activity diagrams in different ways [Obj12]. Figure 2 describes some of the notations of UML ActD, that are used in this paper. We would like to focus on expansion region [Obj12], which is a structured activity node that takes collections of elements as input and can return output collections described in expansion nodes. The execution of elements can be described in three different modes according to a keyword at the top of the region, these modes are: *parallel*, where the elements execute concurrently and can overlap in time. *Iterative*, where execution is done in sequence, so an execution does not start until the previous one is completed, and follows the same order if ordering is applied. Lastly in a *stream* mode, there is exactly one execution in the region, where elements are offered to execution in a stream, preserving ordering if applied.

2.3 Atomicity Decomposition

Atomicity Decomposition (AD) diagrams were first introduced by Butler [But09]. AD provides a graphical representation based on Jackson's diagrams (JSD) [Jac83], that helps understanding the relations between the abstract and concrete levels of the Event-B models. AD also helps to explicitly describe the flows of events, which are implicitly described through guarded events in an Event-B model [But09].

Figure 3 shows a basic structure of the atomicity decomposition diagrams, where each node represents an event. The AD diagram indicates that the atomicity of the abstract event, appears in the root node, is decomposed into some concrete sub-events, appear in the leaf nodes. Similar to Jackson's structure diagrams, the leaf events are read from left to right, so *Event1* is executed first, followed by *Event2* and finally *Event3*.

The dashed line represents a newly added event that does not have a counterpart at the abstract level, which we refer to as refining *skip*. The solid line represents a refining event, where exactly



Symbol	Description
	Swimlanes or partitions group related actions and activities together, usually according to whom/what is performing them.
●	**Initial Node** indicates the start of a process.
◉	**Final Node** indicates the end of a process.
⊗	**Flow final node** indicates the end of a flow but not the whole activity.
Activity	**Activity** can include multiple actions and sub activities.
	A rake symbol inside an activity indicates nesting of activities within that activity
Action	**Action** represents a single operation within an activity.
	Decision node branches the input into multiple flows and each branch has a guard or conditions, these guards must be mutually exclusive.
	Merge node brings alternate flows together without synchronisation.
	Fork describes parallel or concurrent flows.
	Join synchronises concurrent flows.
iterative	**Expansion region** is a nested activity with input and output expansion nodes, it has a keyword to indicate whether the input will be processed in parallel, iterative or a stream.
	Note is not processed within a model but it includes useful information that helps understanding or clarifying the model.

Figure 2: Some Notations of UML Activity Diagrams

one event can refine an abstract event. A construct can be one of the following: *and, or, xor, loop, all, some, one*. The construct can be connected to one or more events depending on its type. We can also add parameters to the events to indicate multiple instances modelling [But09, SBR12].

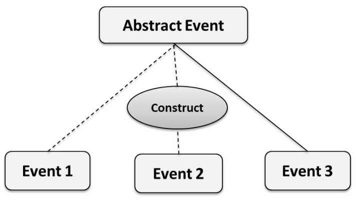


Figure 3: Basic Structure of Atomicity Decomposition Diagram

Semantics is given to AD diagram by transforming it into an Event-B model. Salehi et al. in [SBR12] describe some of the rules of AD transformation into Event-B modelling language. They use the subset relations approach to describe the order between events. They also apply AD to two case studies [SB10, SRB11] to evaluate its role in clarifying the refinement and the explicit ordering of events in the Event-B modelling language. Figure 4 shows the corresponding Event-B translation of *Event2* in Figure 3, provided that there is no construct applied. As shown, the ordering is specified by invariant *inv_e2_seq*, indicating that *Event2* must follow *Event1*. In the event, this is controlled by the guard *grd_seq*, indicating that *Event1* has occurred first, before Event2 is allowed to happen, which has not occurred yet according to *grd_self*.

Single Instance Case	Multiple Instances Case
invariants	**invariants**
@inv_e1_type Event1 ∈ BOOL	@inv_e1_type Event1 ⊆ **P**
@inv_e2_seq Event2 = TRUE ⇒ Event1 = TRUE	@inv_e2_seq Event2 ⊆ Event1
event Event2	**event** Event2
where	**any** *p*
@grd_seq Event1 = TRUE	**where**
@grd_self Event2 = FALSE	@grd_seq *p* ∈ Event*1*
then	@grd_self *p* ∉ Event*2*
@act Event2 ≔ TRUE	**then**
end	@act Event2 ≔ Event2 ∪ { *p* }
	end

Figure 4: Translation of the Sequencing Pattern from AD to an Event-B Model

3 Fire Dispatch System Case Study

3.1 Requirements Overview

In this section, we present the requirements of a fire dispatch system. We only try to model the main operations of the dispatch system and discard some details for simplicity of presentation. So, we do not include call waiting queues and we assume that all calls are served immediately and there are enough resources to cover the incidents. We try to focus on the functional operations that keep track of the status change of the incident and the resources throughout the dispatch process, and also how these resources are assigned to an incident.

Functional Requirements: These can be divided into the following goals: location matching, duplicate calls, action response, resources mobilisation to incident, resources attendance and incident closure. First the control operator tries to get information about the incident, its location, type and its emergency level that can be prioritised into 3 levels; critical, non critical or other. Each incident type has a default priority level, that can be changed manually by the operator. Then, the system tries to identify if the incident is a duplicate or not. If it is duplicate, it alerts the operator who confirms that, resulting in an automatic closure of the incident call. From the type and location (e.g. factory, school ...), the system can identify the suitable predetermined action response plan, which in turn gives information about the type of resources needed.

As a result, the system will suggest the quickest available resources to the incident, that can be allocated by the operator. The allocated resources will be notified automatically, and here we assume an immediate response. At this point the tracking of the incident and resources status change will start. Until the resource arrives to the incident location, the system will keep looking for new quicker resources to become available, and suggest reallocating them to the operator. After attending the incident and all resources get deassigned or deallocated, the incident will be closed and the deallocated resources will either go back to station, or get reassigned to new incidents. The status of the incidents changes from Opened, to In_Progress and finally to Closed, and that of a resource can be any of the following: Station_Available, Mobile_Incident, In_Attendance, Mobile_Station or Unavailable. We use detailed states of the resources and the incidents to help us keep track of the events later.

To summarise: The case study is focused on solving the following problems: how the action plan (PDA or Pre-Determined Actions) determines the resources needed for the incident. This depends on the type, location and the priority level of the incident[1]. We also focus on how the status of the resources and the incident is changed throughout the dispatch process, which mainly depends on the location of the resources compared to the incident and the stations.

Figure 5 describes the class diagram including the main entities and relationships used in our model, which helps us in modelling the structural part of the Event-B model. For example, entities *INCIDENTS* and *RESOURCES* are defined as sets in the context. The “*Assigned to*” relationship, the arrow between these entities, is described as a relation between *incidents* and *Resources* using an invariant in the Event-B model.

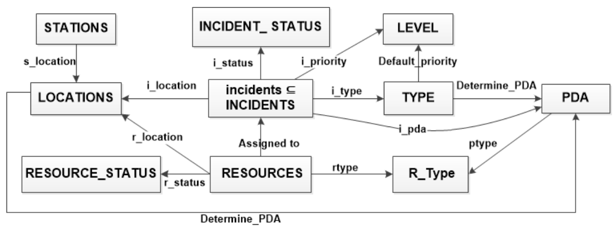


Figure 5: Fire Dispatch System Entities and Relationships

3.2 UML Activity Diagram Representation

After using the class diagram to identify the main sets and relationships needed, we represent the fire dispatch system using UML activity diagrams, which mainly help in identifying the required events and their sequence. We start by making a detailed graph of all the operations derived from the requirements, because we find it easier to visualise the overall sequence of the events than trying to figure out the relationship between the different sub-problems. Then we group the related events into nested activities and this helps in abstracting the system, and later elaborating these activities as steps, we do not show all these steps due to space limitation.

Figure 6 shows the ActD of the fire dispatch system after applying abstraction. The approach we follow in grouping the events is that whenever there is a decision-node, we apply grouping. Because, decision-nodes usually describe different scenarios of the events flow, before merging into the next event; which is similar in a way to different solutions of the same problem before moving into the next one. However, in the case of a duplicate location which is represented as a decision-node (*Is_Duplicate*), we leave it at the abstract level due to the presence of the activity final node, where we want to indicate the end of the whole dispatch activity of the incident, and not only one sub-activity like in the case of nesting.

Another grouping is used, when we need multiple instances in a sequence of events, or in other words when we need expansion regions, and obviously these events are related by the same objects, and consequently they represent the same sub-problem. At this point we end up with three activity diagrams, but not shown for space limitation. The first one is an overall diagram representing one instance, one resource, in the action plan. Then we apply grouping according to the decision nodes, and finally add the second grouping of multiple instances, in this case the resources in the action plan, to end up with Figure 6.

[1] In our model, the effect of priority on PDA is implemented indirectly through its relation with the incident type “*TYPE*”, and we restrict its change to before setting the action plan, but the direct effect of priority changes throughout the workflow and its effect on resource allocation is left for future work.

As you can see in Figure 6, we represent the multiple instances, the resources, in the *Resource Allocation* using an expansion region with the *parallel* keyword to indicate that all these events should run in parallel or concurrently for each resource, and this is the reason why, we do not just use a loop and a decision node to decide when to stop. The rake symbol indicates nesting or further refinement, for example *Resource Allocation* can be further refined, at the right, to include the sub-events *Find Quickest Resource, Allocate Resource* etc.

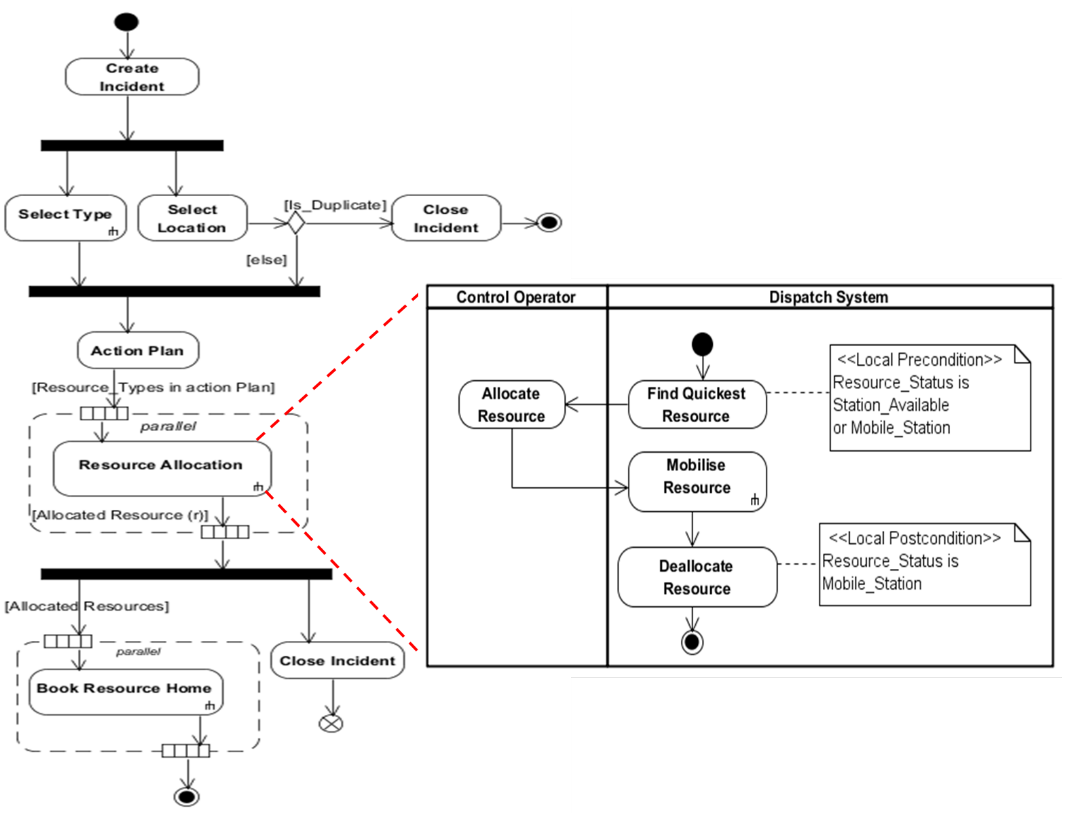
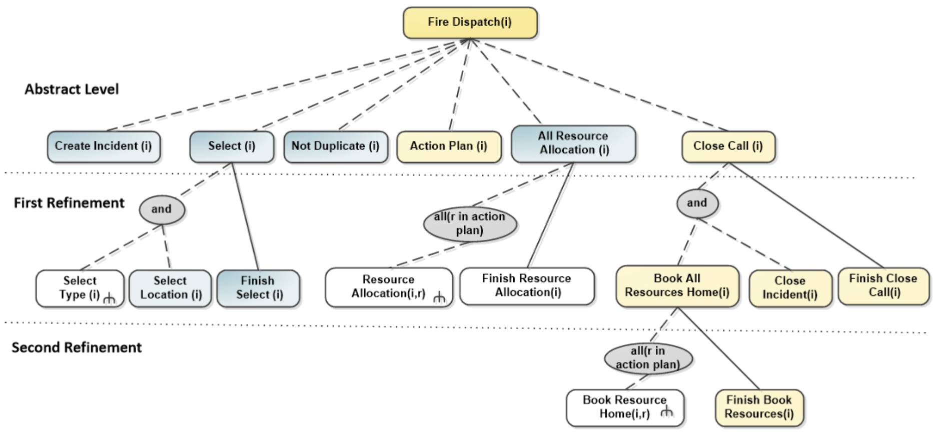


Figure 6: Abstract Level of the Fire Dispatch System ActD, with Focus on Resource Allocation

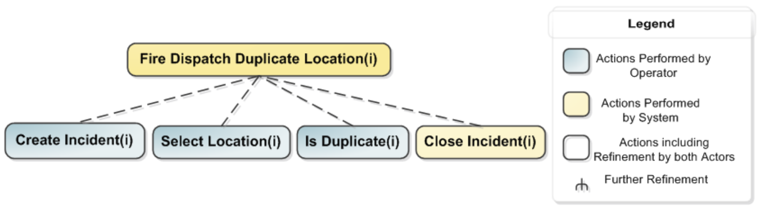
3.3 Atomicity Decomposition Representation

As the next step, we represent the fire dispatch system using AD diagrams (Figure 7), which focus on both control flows and refinement relationships between concrete events and the corresponding abstract events. As explained earlier in ActD, Figure 6; there are two activity final nodes, which means two scenarios can close an incident call. That is why we split the AD into two cases, one representing the normal case i.e. non-duplicate case, Figure 7(a); and the other is when the incident is duplicate and hence it is closed immediately, Figure 7(b). The rake symbol is used to indicate further refinements, which are omitted in this paper. The colour codes are used to describe who/what performs the operations, like the swimlanes in ActD that are shown only in the refinement of *Resource Allocation*, due to space limitation, in Figure 6.

In Figure 7(a), the root node of the tree represents the process name of the system, and it is distinguished from other event nodes by being connected to the abstract level events using dashed lines only. Considering the event sequencing from left to right, in the abstract level, first event *Create_Incident* executes, followed by *Select* event, which is in turn followed by *Not_Duplicate* and so on. As you can see, event *Select* is further decomposed into two branches in the first refinement level. The and-branch that has two events (*Select_Type* and *Select_Location*), and *Finish_Select*, which refines the abstract *Select* event. The and-constructor sub-events, *Select_Type* and *Select_Location*, must execute in any order, before the execution of *Finish_Select*.



(a) Atomicity Decomposition Diagram of Fire Dispatch System - Non-Duplicate Call Case


Fire Dispatch Duplicate Location(i)
Create Incident(i)
Select Location(i)
Is Duplicate(i)
Close Incident(i)
Legend
Actions Performed by Operator
Actions Performed by System
Actions including Refinement by both Actors
Further Refinement

(b) Atomicity Decomposition Diagram of Fire Dispatch System - Duplicate Call Case

Figure 7: Fire Dispatch System Atomicity Decomposition Diagrams

4 Evaluation

4.1 Graphical Comparison

We start with the AD constructs and finding their counterparts in UML ActD. We do not find counterparts the other way round, because the notation of UML ActD is very rich especially after the introduction of UML 2.0; and in general the constructs used in AD are more focused on the Event-B modelling language.

Figure 8 shows some of the AD constructs and their counterparts in UML ActD. The and-construct in AD, top left, means both *Eve1* and *Eve2*, should be executed before *Eve3* is enabled. In ActD this is translated as a fork to describe the concurrent execution of *Eve1* and *Eve2*, and then they are joined or synchronised into one flow before *Eve3* can be executed. The xor-construct, top right, means only one of the sub-events can be executed, similar to a decision-node with mutually exclusive guards in ActD. The guards (*condition1, condition2*) are not shown explicitly in the AD diagram, they can be manually added to the event definition in the Event-B model, but here they are shown for comparison purposes; the same applies to the loop conditions. The or-construct means at least one of the sub-events is executed before the next event (*Eve3*) can be enabled, that is why we use a merge-node in the ActD which requires at least one flow to proceed and does not synchronise the flows like the join-node. While this case is not acceptable in UML 1.x, UML 2.0 has removed the restriction of matching forks with joins, which makes our comparison valid. Sequencing in AD is read from left to right, and in ActD the direction of the arrow describes the order of events. Lastly, the star symbol indicates looping, zero or more iterations of the event, which can be described by using a decision-node and a merge-node that bring all flows together without synchronisation. Here the decision-node is added before the loop event (*Eve1*) to describe the zero iteration, but if we need one or more iterations, we place it after the loop event. UML 2.0 has also introduced a structured loop-node which is equivalent to the use of the decision and merge nodes as shown in Figure 8.

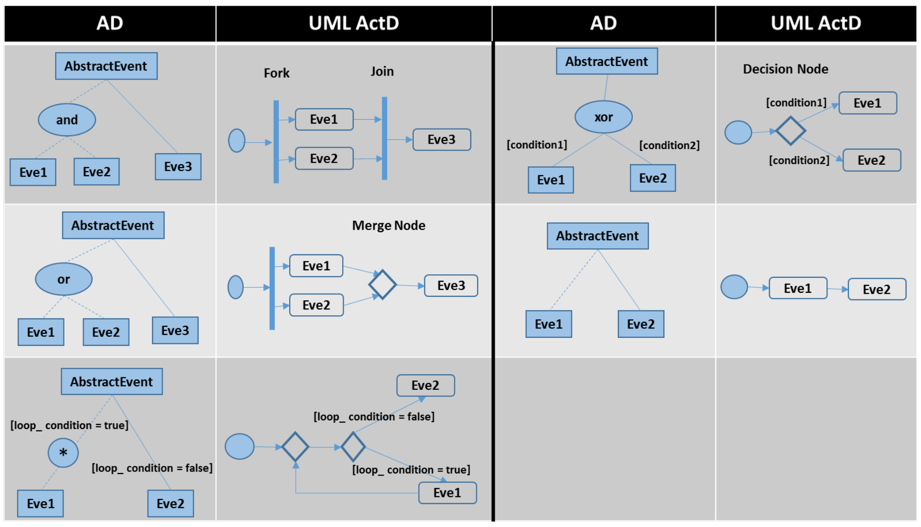


Figure 8: Comparison between AD Non-Replicator Constructs and UML ActD

Replicator Patterns AD describes another three replicator constructors, which introduce a new parameter to the contruct's sub-events. These replicators are: all-replicator, which is a generalisation of the and-construct; some-replicator, a generalisation of the or-construct, and lastly the one-replicator, which is a generalisation of the xor-construct, that describes exactly one iteration of an instance. To find an equivalence of the all-replicator in ActD, we distinguish between two cases; one where you know how many instances are in your collection, in this case you can use expansion regions. The fire dispatch case study in Figure 6, uses an expansion region with parallel execution in *Resource Allocation*. Since the collection of resources needed is determined at runtime, during the *Action Plan* event, and in this case you can decide how the events can be executed using a keyword in the expansion region, this can be either parallel, iterative or stream execution. In the second case where you can not determine the size of the input collection neither at design time nor during runtime, you can use a loop for the multiple iterations and a decision node to determine whether you need to continue or stop these iterations. Regarding the some-replicator and the one-replicator in AD, they can be represented in the same way as the all-construct, Figure 7(a); but they have no explicit constructs in ActD.

Assessment We managed to represent the fire dispatch system using both AD and ActD, and we found the corresponding construct for all the non-repliactors and some of the replicator patterns of the AD to ActD, which shows that a transformation from AD to ActD should be possible. But, we still have some limitations regarding the multiple instances representations of both diagrammatic notations.

Regarding AD, the all-construct is described as a generalisation of the and-construct, so it only describes parallel execution of events, unlike the expansion regions which are more general that they can describe parallel, iterative and stream executions.

Take the example of a doctor who needs to visit “x” number of patients daily, this can not be described as an all-construct in AD, because if the visit event is refined into further sequence of sub-events, this event will be interleaving with other visits by the same doctor. This could be described using a loop, but it is not so convenient because a loop means a zero or more iterations, but here we have a specific number of iterations. To solve this problem, we need a new description of the all-construct with sequential execution that preserves ordering of instances.

On the other hand, there is no explicit constructs of the some-construct and the one-construct in the activity diagram. Although these can be described using a guard, this can lead to an ambiguity in the representation and each modeller can use his own interpretation.

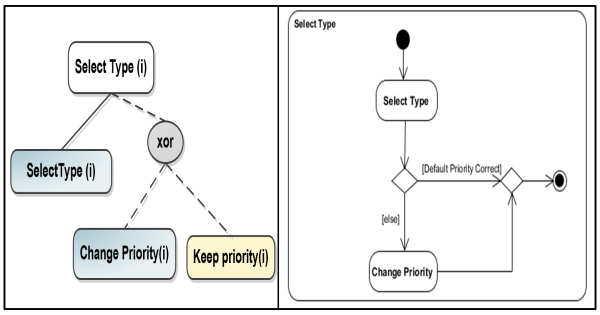


Figure 9: Select_Type Refinement in Atomicity Decomposition and Activity Diagram

Concerning the layout of both diagrams, we find the directed arrows of the actD can be easier to follow than the hierarchical representation of the AD diagrams. Take for example Figure 7(a), if we want to follow the sequencing of the non-refining leaf nodes, we should start from *Create_Incident* at the abstract level, then go to some refining level higher than the first refinement because *Select_Type* will be further refined as indicated by the rake symbol, then back again to the first refinement level for *Select_Location* and *Finish_Select* and so on. On the other hand, the hierarchy and structure of AD restrict the way you build your model in a positive sense. Because the refinement layers come naturally with its structure; whereas with UML ActD, it is up to the modeller, how to apply nesting and refinement. So AD diagrams can provide a guideline for when and how to refine the Event-B model, like the constrain that, the all-construct should be only applied to one event, so in this case we add an extra level of refinement to apply the all-construct to a sequence of events e.g. *Resource_Allocation* in Figure 7(a). You can also see explicitly how the new events and the refining events relate to the abstract event using solid and dashed lines. On the other hand, the rake symbol in ActD can describe that a refinement exists, but does not show how the concrete events are related to the corresponding abstract events.

Regarding details and clarity, UML ActD with its rich notations can help in understanding the model more. By using notes with pre and post conditions, and explicit use of guards in the diagram. These guards can clarify some of the details of the Event-B model, by describing why a flow takes a certain route rather than the other. Like in the decision-node that describes explicitly the guards needed, unlike its AD equivalent, the xor-construct, the guards are left for manual implementation in the Event-B model, as shown in Figure 9.

4.2 Event-B Comparison

For both representations we use the entity diagram (Figure 5) to define the structure of the model, where the entities describe the sets, and the associations between them describe the relations needed. Since there is no available plug-in to translate ActD to Event-B in Rodin, we apply the following approach: all actions become events in Event-B and the event conditions are added as guards; and whenever we have call action behaviour indicated by a rake symbol, we use refinement. In our overall diagram we use notes to help in clarifying the details needed for events, like the local pre and post conditions used in the refinement of *Resource Allocation* in Figure 6, where preconditions become event guards and postconditions become event actions.

The AD translation into Event-B is done both manually and automatically using the AD plug-in [Wik12]. AD uses subset relations to describe sequencing, where each event has a corresponding variable with the same name. Moreover one invariant and one guard are enough to describe the ordering of the event, Figure 4, which facilitates tracking the workflow of events, especially with the naming convention used. On the other hand, as a result of the flexible manual modelling, the sequencing used in the ActD model is more related to the relations described in the entity diagram and the status change of the resources, which sometimes requires more guards to describe sequencing. For example, to describe the sequencing of the *Action_Plan* event in ActD, we need to check that the incident "*i*" is not duplicate and it is in the domain of both functions *Select_Type* and *Select_Location*, whereas for AD just checking if the incident "*i*" belongs to *Non_Duplicate* is enough to describe the sequencing. However, we need to add the application-specific data structures, described in the entity diagram, manually for the AD model as a refinement to describe the details of the model, which can result in some kind of repetition.

AD can sometimes result in having extra events compared to ActD, take for example the refinement of *Select_Type* as shown in Figure 9. In this case we want to say that the operator has to select the incident type first, then if the default priority associated with this type is correct continue to *Action Plan*, else change the priority and then go to the *Action Plan*. In AD, we must have at least two leaves when using the xor-construct, similarly with the decision node in ActD. However, the flexibility of directed arrows can take us directly to the next step, in this case the join-node before the *action_plan* event; whereas in the hierarchical structure, we can not have this flexibility and use the *Finish_Select* as the other leaf because this will change the whole workflow logic. So we need to have a dummy event which is here the *Keep_Priority* that does not make any changes.

Sometimes having the extra events can have advantages, because they can emphasise the behaviour and the logic behind the used constructs. Take for example the *Finish_Select* event in Figure 10, it acts like a join-node in the ActD (Figure 6), and this is clear in the Event-B model through the use of the logical-and between *Select_Type* and *Select_Location* (Figure 10). Also the effect of the xor-construct in changing priority is clear by using the union of *Change_Priority* and *Keep_Priority*. This encourages us to later study the practicality of translating the ActD constructs as separate events or just imply their effects through guards in the existing events.

event Finish_Select **refines** Select
  any *i*
  where
    @grd_seq (*i* ∈ *Select_Type* ∧ *i* ∈ *Select_Location*)
    @grd_self *i* ∉ *Finish_Select*
  then
    @act Finish_Select ≔ Finish_Select ∪ { *i* }
end
	event Finish_Select **refines** Finish_Select
	  any *i*
	  where
	    @grd_seq (*i* ∈ (*Change_Priority* ∪ *Keep_Priority*) ∧ *i* ∈ *Select_Location*)
	    @grd_self *i* ∉ *Finish_Select*
	  then
	    @act Finish_Select ≔ Finish_Select ∪ { *i* }
	end

Figure 10: Finish_Select Event at 1st and 2nd Refinement Levels

Regarding the all-construct of AD, it can only have one branch and is always associated with a parameter. The intent is to execute the all-construct event for all instances of its parameter. In the first refinement level of Figure 7(a), we apply the all-construct to the *Resource_Allocation* event, which will be later refined into some sequence of events. Consequently, we have an extra refinement because we can not apply the all-construct to a sequence of events. We also have the extra event *Finish_Resource_Allocation* to refine the *All_Resource_Allocation* event. This shows how the rules of AD can provide a guideline for how and when to use refinement.

The all-construct mainly affects *Finish_Resource_Allocation*, which is the directly following event. This event can not be executed until all instances of the all-construct parameter finish executing. Figure 11, top part, shows the Event-B translation of *Finish_Resource_Allocation*. The invariant "*inv_seq*" and the guard "*grd_seq*", ensure that *Finish_Resource_Allocation* can not be executed before *Resource_Allocation* finishes executing for all instances of the parameter, $ptype[i\_pda(i)]$. The function *i_pda* describes the relation between *incidents* and *PDA*, and the all-construct parameter, *p_type*, represents the resources types of the incident's PDA.

AD Approach	@inv_seq ∀*i*· *i* ∈ *Finish_Resource_Allocation* ⇒ *All_Resource_Allocation* [{ *i* }] = ***ptype[{i_pda(i)}]***
	event Finish_Resource_Allocation **refines** Resource_Allocation
	  any *i*
	  where
	    @grd1 *i* ∈ *dom(i_pda)* *// manually*
	    @grd_seq All_Resource_Allocation [{ *i* }] = ***ptype[{i_pda(i)}]***
	    @grd_self *i* ∉ *Finish_Resource_Allocation*
	  then
	    @act Finish_Resource_Allocation ≔ Finish_Resource_Allocation ∪ { *i* }
	end

ActD Manual Approach	@inv_Closed_stat ∀*i*·*i* ∈ *dom(i_pda)* ∧ *i_status(i)* = ***Closed*** ⇒ ***r_types[{i}]*** = ***ptype[{i_pda(i)}]***
	event Close_Incident **extends** Close_Incident
	  any *p*
	  where
	    @grd3 *p* = *i_pda(i)*
	    @grd4 r_types[{i}] = **ptype[{*p*}]**
	end

Figure 11: All-Construct Effect on AD and ActD at 2nd Refinement Level

In the ActD manual model, the parallel execution of *Book_Resource_Home* and *Close_Incident* follows the *Resource_Allocation* expansion region. Due to space limitation, we only show the translation of *Close_Incident* in Figure 11, bottom part. In this case we added the relation *r_types* to keep track of the allocated types. Similar to *Finish_Resource_Allocation*, we managed to describe the effect of the expansion region using the invariant *inv_Closed* and the guard *grd_4*. However for *Book_Resource_Home* we only used *grd_4* but could not enforce the sequencing using an invariant due to the complexity of the relations used to describe the *Book_Resource_Home* especially after refinement. This shows how having an extra event, *Finish_Resource_Allocation*, and using the subset relations in AD, in this case both *Close_Incident* and *Book_Resource_Home* will be subset of Finish_Resource_Allocation, can simplify the sequencing invariants and make the sequencing guard stronger by always being enforced by a sequencing invariant.

5 Related Work

Many researchers have studied the integration of formal methods and graphical representations in an attempt to reduce the gap between requirements and formal language. A good example of such approach is the translation of behaviour trees into CSP, that is extended to include state based constructs [CH11]. Unlike our approach, behaviour trees as first introduced by Dromey [Dro03], define each functional requirement as a separate tree and later all requirements' trees are integrated into one. This integration is not simple and requires interference of the modellers to solve integration problems.

Both Matoussi et al. [MGL11] and Laleau et al. [LSM$^+$10] describe a formal translation of the KAOS goal model. The first into Event-B and the latter use the KAOS goal model to extend the SysML requirements model and then map it into the B method. Regarding refinement they both follow the approach described in [DL96], where parent goals are refined into AND/OR refined sub-goals, and they also use sequencing of sub-goals. Ben Younes et al. also use this approach to describe refinement patterns of UML activity diagrams [YAH12], and they also describe the translation of activity diagrams into Event-B in [BB08]. In both cases they do not describe, how they determine the static part of the model.

Regarding UML activity diagrams de Sousa et al. [SSS11] also describe the translation of IOD, which is a variant of activity diagrams, into Event-B, but they include boundary-control-entity class diagram to extend UML-B class diagram and describe the static part of the formal model. However, when it comes to refinement in Event-B, they use the ICONICX process, that includes four types of UML diagrams [SSS12].

In all the previously described approaches, there is no explicit notion of replicators like all-construct and some-construct, even the ones that use activity diagrams, do not describe the use of expansion regions or the loop case. When it comes to refinement, all these techniques do not show how the new events relate to the abstract ones, the way AD explicitly describes.

6 Conclusion and Future Work

In this paper, we present an evaluation of two graphical approaches that show the behaviour of a system and aim at facilitating the control flow of the Event-B model. Through the evaluation of our case study, we would like to end up with a process to facilitate the Event-B formal modelling of complex workflow systems. This process starts by deriving an entity diagram from the requirements to model the structural part, and then using an activity diagram to better understand the workflow and the details of the system through guards, pre and post conditions etc. and finally using atomicity decomposition as a guideline for refinement and follow its translation rules to formally model the system using Event-B, which can be further enriched with the help of the details described in the activity diagram. Based on our comparison, we would like to study the possibility of extending the atomicity decomposition patterns and translation rules and add these extensions to the existing atomicity decomposition plug-in. We would also like to study the possibility of integrating AD approach with ActD and the entity diagram. Finally, we intend to enhance our case study to see the effects of the change of an event conditions on the rest of the Event-B model.

Acknowledgements: Butler and Salehi were partly funded by the FP7 ADVANCE Project (287563), www.advance-ict.eu. We would like to thank the reviewers for their helpful comments.

Bibliography

[ABH$^+$10] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer* 12:447–466, 2010.

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[AH07] J.-R. Abrial, S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.* 77(1-2):1–28, 2007.

[BB08] A. Ben Younes, L. J. Ben Ayed. From UML Activity Diagrams to Event B for the Specification and the Verification of Workflow Applications. In *Computer Software and Applications. COMPSAC'08. 32nd Annual IEEE International*. Pp. 643–648. 2008.

[But09] M. Butler. Decomposition Structures for Event-B. In Leuschel and Wehrheim (eds.), *Integrated Formal Methods*. Volume 5423, pp. 20–38. Springer Berlin Heidelberg, 2009.

[CH11] R. J. Colvin, I. J. Hayes. A semantics for Behavior Trees using CSP with specification commands. *SCIENCE OF COMPUTER PROGRAMMING* 76(10, SI):891–914, 2011. 7th International Conference on Integrated Formal Methods, Dusseldorf, GERMANY, FEB 16-19, 2009.

[DL96] R. Darimont, A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. SIGSOFT '96, pp. 179–190. ACM, 1996.

[Dro03] R. Dromey. From requirements to design: formalizing the key steps. In *Software Engineering and Formal Methods, 2003.Proceedings. First International Conference on*. Pp. 2–11. 2003.

[Jac83] M. A. Jackson. *System Development*. Englewood Cliffs, N.J. : Prentice-Hall, 1983.

[LSM$^+$10] R. Laleau, F. Semmak, A. Matoussi, D. Petit, A. Hammad, B. Tatibouet. A first attempt to combine SysML requirements diagrams and B. *Innovations in Systems and Software Engineering* 6(1-2):47–54, 2010.

[MAV05] C. Metayer, J.-R. Abrial, L. Voisin. Event-B language. RODIN Project Deliverable 3.2. 2005. Available online at http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

[MGL11] A. Matoussi, F. Gervais, R. Laleau. A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*. Pp. 139–148. 2011.

[Obj12] Object Management Group. OMG Unified Modeling Language Version 2.5. 2012. http://www.omg.org/spec/UML/2.5/Beta1/PDF [Accessed: 17 Mar 2013].

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1998.

[RSP07] R. Razali, C. F. Snook, M. R. Poppleton. Comprehensibility of UML-based Formal Model: A Series of Controlled Experiments. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech)*. Pp. 25–30. November 2007.

[SB10] A. Salehi Fathabadi, M. Butler. Applying Event-B Atomicity Decomposition to a Multi Media Protocol. In Boer et al. (eds.), *Formal Methods for Components and Objects*. Lecture Notes in Computer Science 6286, pp. 89–104. Springer Berlin Heidelberg, 2010.

[SBR12] A. Salehi Fathabadi, M. Butler, A. Rezazadeh. A systematic approach to atomicity decomposition in Event-B. *Lecture Notes in Computer Science* 7504:78–93, 2012.

[SRB11] A. Salehi Fathabadi, A. Rezazadeh, M. Butler. *Applying Atomicity and Model Decomposition to a Space Craft System in Event-B*. 2011.

[SSS11] T. de Sousa, C. Snook, P. Silva. A proposal for extending UML-B to support a conceptual model. *Innovations and Systems and Software Engineering* 7:293–301, 2011.

[SSS12] T. C. de Sousa, P. S. M. Silva, C. F. Snook. A practical Event-B refinement method based on a UML-Driven development process. In *Abstract State Machines, Alloy, B, VDM, and Z*. Pp. 357–360. Springer, 2012.

[Wik12] Wiki.event-b.org. Atomicity Decomposition Plug-in User Guide - Event-B. 2012. http://wiki.event-b.org/index.php/Atomicity_Decomposition_Plug-in_User_Guide.

[YAH12] A. B. Younes, L. J. B. Ayed, Y. B. Hlaoui. UML AD Refinement Patterns for Modeling Workflow Applications. In *Computer Software and Applications Conference Workshops (COMPSACW), IEEE 36th Annual*. Pp. 236–241. 2012.