7
The proposed domain specific language: operational level

In our methodology, a *scenario* corresponds to the specification of concrete activities in the pervasive mobile game, including interactions among players, devices, sensors, actuators and their properties, using the elements defined in Chapter 6 and in this chapter. It is the main concept in the DSL operational level.

This chapter details the operational level for the proposed DSL – how the ontological level concepts (defined in Chapter 6) take part in the scenario definitions, as well as defining a visual language for specifying activities.

We had the following goals for proposing scenarios:

- Specify and document *activities*;
- Specify and document *interactions*;
- Specify and document *acknowledgments*;
- Specify and document *events*.

Scenarios can be described textually or visually. The desired abstraction level for specifying activity flows in scenarios is equivalent to the level of *use cases* (Cockburn 2000). That means that we are concerned about *intent*, using an abstraction level that is high enough to specify the activities in terms of the concepts defined in Chapter 6, but not including implementation details. However, a use case is very general in regard to its structure and elements. As our proposal is a DSL, it defines specific rules (Section 6.3), vocabulary (Sections 6.2 and 6.2.10) and language (this chapter) for specifying activities in this domain of pervasive mobile games.

This chapter provides the elements for scenarios in both text (Section 7.1) and visual format (Section 7.2), next chapter provides concrete and more detailed examples on how to use these elements. These examples correspond to the specification of activities for the *Pervasive Word Search* prototype – the final part of our case study.

7.1 Specifying scenarios in text format

The proposed DSL defines an *activity template* to specify scenarios in text format. The elements in this template correspond to the *activity properties* defined in Section 6.2.10. Figure 7.1 illustrates the template.

```
Scope                          The activity scope
Primary Actors                 Actors that start the activity
Secondary Actors               Other actors affected in the activity
Sensors                        List of required sensors
Actuators                      List of required actuators and/or displays
Interaction granularity        "Discrete", "Continuous", "Mixed"
Control                        "Explicit", "Implicit", "Mixed"

Overview
Brief overview if desired.

Default flow
The default flow, with numbered steps.

Alternative flows
The specification of alternative (success) flows, errors, and exceptions.

Mobile device event flows
Alternative flows related to mobile phone events.

Operation parameters
Specification of these requirements if necessary.

Uncertainty handling policy
List of policies and how they should be applied.

Miscellaneous
Specification of miscellaneous issues.
```

Figure 7.1: The complete activity template

This methodology follows general "use case style" conventions for specifying steps in activity flows (default, alternative and mobile device event flows), as Cockburn (2000) describe in detail. For example, actions in activity flows should be numbered, as they occur in the flow. Chapter 8 provides concrete examples on this.

7.1.1 Notation for ending activities in alternative flows

As alternative flows are likely to occur in several cases, we propose the notation **`EOA`** (end of activity) to indicate when an alternative flow ends the whole activity, in order to simplify the writing. When this notation is used, it means that all flows that might be running concurrently are also terminated. Chapter 8 provides concrete examples on using this notation.

7.1.2 Notation for acknowledgments

When specifying activities as text, we propose the following notation to specify acknowledgments (bold font, short form), to simplify writing:

- **ack (T)**: short for "the game triggers *acknowledgment* of technology *T*"

The *technology* parameter *T* corresponds to the technology property[88] of the actuator used to deliver the acknowledgment. The *scope* for the acknowledgment can be specified by associating the acknowledgment with an actor or external actuator. Figure 7.2 illustrates an example from our prototype *The Audio Flashlight 5*:

```
Game ends level for Flashlight 1, Flashlight 2 and Spoiler (Spoiler is the win-
ner)
  ▪  ack (audio): defeat sound (Flashlight X)
  ▪  ack (audio): defeat sound (Flashlight Y)
  ▪  ack (audio): victory sound (Spoiler)
```

Figure 7.2: Example of acknowledgments in text specification

In this example, when the "Spoiler" player wins the game, the game delivers an acknowledgment as audio to all players. However, the contents of the acknowledgments are not the same for all of them, in this example. Chapter 8 provides examples of using this notation.

The acknowledgments in this example occur concurrently (as indicated by the unnumbered list). However, if the acknowledgments should happen in some specific sequence, the specification should use a numbered list that reflects this sequence.

88 Section 6.2.3 discusses the technology property of actuators.

7.2
Specifying scenarios in visual format

Since the beginning of this research work we would like to represent activities visually, mainly focused on the flow of actions that compose the activity. To accomplish this goal, we searched for a visual language based on a standard representation that would provide:

- Support for modeling behavior aspects, control and action flows, as the activity specification is mainly concerned with those items;
- Support for specifying parallel activities, due to the (possible) distributed nature of pervasive mobile games;
- Operators and operands to represent the proposed domain concepts, minimizing possible extensions to the standard;
- Adequate abstraction level to represent the domain concepts. Too much detail could hinder using this approach.

We have selected the UML 2.4 Activity Diagrams (OMG 2011) as the basis for the visual language for the domain of pervasive mobile games. The UML 2.4 specification[89] (OMG 2011) describes an "activity" as follows:

> "An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. The subordinate behaviors coordinated by these models may be initiated because other behaviors in the model finish executing, because objects and data become available, or because events occur external to the flow. The flow of execution is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents. Activity nodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions ... "

It is arguable that other representations could be used, as UML Sequence Diagrams and State Machine Diagrams (OMG 2011). However, UML Sequence Diagrams offer a level of detail that is higher than necessary for our purposes. For example, UML Sequence diagrams deals with object instances and messages exchanging between objects.

89 In this work, "UML 2.4 specification" refers to the "UML 2.4 specification, Superstructure".

The focus of the proposed activity specification diagram is on the action flows. Because of this intention, we have not chosen UML State Machine Diagrams, as they emphasize representing the system as a set of states.

Another possibility would be using a Petri net (Peterson 1981). However, the UML 2.4 specification (2011) states that activity diagrams have semantics based on Petri nets: *"Activities are redesigned to use a Petri-like semantics instead of state machines. Among other benefits, this widens the number of flows that can be modeled, especially those that have parallel flows"*. Hence, as UML is a well-known standard, we have opted to use UML Activity Diagrams.

Therefore, the activity concept defined in Section 6.2.10 is represented through an UML Activity. The remainder of this section details how the other concepts are mapped to UML Activity diagrams, along with extensions to characterize some of the domain concepts (presented in Chapter 6). The UML concepts are highlighted with bold font.

7.2.1 Specifying actions

In the UML specification (2011), an "action" is *" ... a named element that is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing in the modeled system"*. Hence, we have chosen to map our *actions* to this element. Actions are represented as **ActionNodes**. Figure 7.3 illustrates the notation.

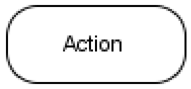

Figure 7.3: Notation for actions

Player *interactions* are specified also as **ActionNodes**. For actions related to interactions with *implicit style*, we define the stereotype «`imp`» to differentiate from actions related to explicit-styled interactions (that do not have a defined stereotype). An UML stereotype (OMG 2011) is a standard way (in UML) to create variations of a modeling element using the same shape, but with different semantics. This style property applies to other concepts, as will be discussed later. In this regard, the «imp» stereotype appears with many elements to denote implicitness.

The style of an interaction can be inferred also by the style of events it generates. Please see Section 7.2.7 for information on this.

7.2.2 Specifying connections

Actions are connected through **ActivityEdges** (OMG 2011), which represent a direct connection between two diagram elements. **ActivityEdges** may have optional **guards**. In the UML 2.4 specification (OMG 2011), a **guard** denotes a condition that must be satisfied for the activity flow to continue through the **ActivityEdge.** Figure 7.4 illustrates the **ActivityEdge** with a **guard.**

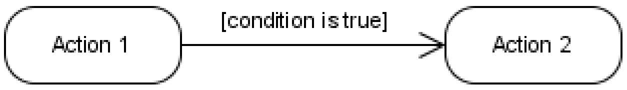

Figure 7.4: Activity edge with a guard (optional)

7.2.3 Specifying start and end points

Figure 7.5 illustrates the notations for **ActivityInitialNode** and **ActivityFinalNode** (OMG 2011), which denotes the starting and ending points for an activity.

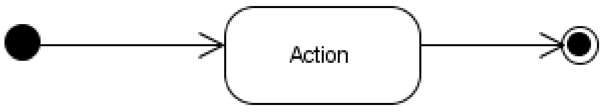

Figure 7.5: Starting point (left circle) and end point (right circle) for an activity

ActivityInitialNodes have one outgoing **ActivityEdge** (and no incoming ones), while **ActivityFinalNodes** have one incoming activity edge (and no outgoing ones). Activities can have several **ActivityFinalNodes**. In this case, the first one reached stops all execution tasks in the activity, thus terminating it.

7.2.4
Specifying distributed activities

Distributed actions can be grouped in **ActivityPartitions** (OMG 2011). Figure 7.6 illustrates the notation, also known colloquially as the “swim lane” notation.

Partitions can be drawn either horizontally (as in Figure 7.6), or vertically. For example, Figure 7.6 could illustrate an example of a multi-player activity involving two players. The partition “Player 1” would represent actions that are executed in first player's device, while the partition “Player 2” would represent actions that are executed in second player's device. The communication between the devices is represented as flows (Action 2 to Action 3, and Action 4 to Action 5).

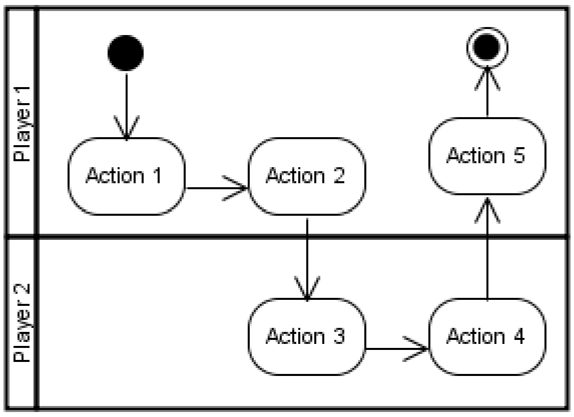

Figure 7.6: Partition notation

For completeness, we present another way the UML specification defines to represent distributed activities, which corresponds to using stereotypes for the involved actors. Figure 7.7 illustrates the example of Figure 7.6 using this notation.

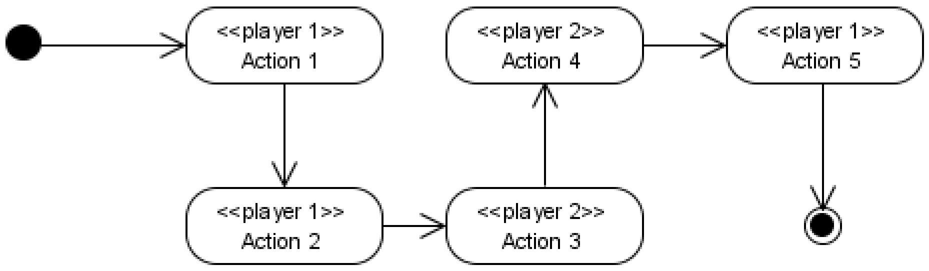

Figure 7.7: Distributed activities using stereotypes

In our methodology, the preferred notation for distributed activities is grouping actions in activity partitions.

7.2.5 Specifying branching in activities

Branching in activities is specified through **DecisionNodes** (OMG 2011). **DecisionNodes** have one (or more) incoming **ActivityEdges** and a set of outgoing **ActivityEdges** (at least one edge). All outgoing **ActivityEdges** have **guards** to specify branching conditions. Those guards must cover all possible conditions, and they must be mutually exclusive. An alternative to specify a "catch-all" guard for conditions other than the primarily evaluated one is using `[else]`.

When an activity flow reaches a **DecisionNode**, the guards are evaluated and the flow continues through the branch that satisfies the condition, while the other alternatives are abandoned. The **DecisionNode** does not specify the ordering the guards are evaluated, only the alternatives.

Branched flows can be merged with **MergeNodes**, which have the same notation as **DecisionNodes**. **MergeNode** have a set of incoming edges and one outgoing edge. However, merge nodes do not provide means for synchronization of alternative flows. Figure 7.8 illustrates an example with a **DecisionNode** and a **MergeNode**.

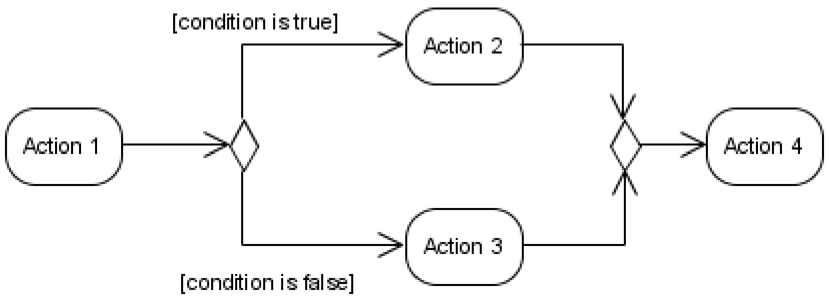

Figure 7.8: Notation for DecisionNodes and MergeNodes

Alternative flows may reach an end point, while the activity is not over yet. To represent this situation the UML defines a **FlowFinalNode** (OMG 2011), illustrated in Figure 7.9.

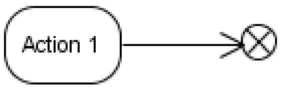

Figure 7.9: Notation for FlowFinalNode

The **FlowFinalNode** only terminates the referenced alternative flow, while **ActivityFinalNodes**[90] terminate the whole activity (including all flows that might be running concurrently).

7.2.6 Specifying concurrency in activities

Concurrent actions are specified with **ForkNodes** and **JoinNodes** (OMG 2011). When the activity flow reaches a **ForkNode**, parallel flows start, which can be of any number. **ForkNodes** have one incoming activity edge.

Later, alternative flows can be synchronized with the **JoinNode**, which receives multiple activity edges and has only one outgoing edge. **ForkNodes** and **JoinNodes** share the same notation, as Figure 7.10 illustrates.

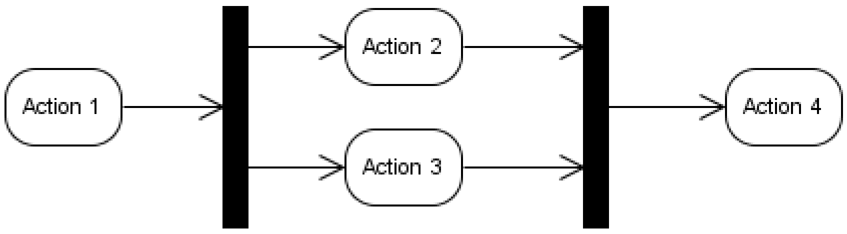

Figure 7.10: Notation for ForkNodes and JoinNodes

The bars can be vertical or horizontal depending on the direction the flow is laid out.

7.2.7 Specifying events

This section discusses the specification of *interaction events*, generic events, and time events.

90 Please see Section 7.2.3.

7.2.7.1
Interaction events

To represent interaction events, we have chosen the UML element **SendSignalAction** (OMG 2011). Figure 7.11 illustrates the notation.

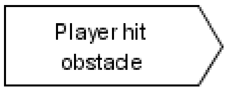

Figure 7.11: A send signal action generating the event "Player hit obstacle"

The UML 2.4 specification (2011) defines **SendSignalAction** as *"an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity"*. Regarding UML Activity Diagrams, this concept was the one we considered the closest to our concept of *interaction event*. In the **SendSignalAction** definition, a "signal" is defined as *"the type of signal transmitted to the target object"*, with no stricter meaning. Also, although the definition refers to a *"target object"*, it seems the specification does not enforce that the receiving side of **SendSignalAction** must be an object (as instances of some class)[91]. Some authors of UML books have also interpreted "signals" as concepts related to "events", as (Larman 2004; Ambler 2005; Pilone and Pitman 2005; Graessle *et al.* 2005).

Interaction and *interaction events* are closely related. The style property of both interaction and interaction events should match, as of Consistency Rule 8[92]. Hence, implicit-styled interactions can be identified by the style of events it generates. To differentiate between explicit and implicit style, we define the «imp» stereotype for interaction events with implicit style. Figure 7.12 illustrates the notation.

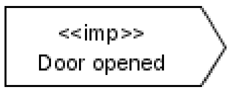

Figure 7.12: Notation for interaction events with implicit style

91 As an example, please refer to Figure 12.132 of the UML 2.4 Superstructure specification: Page 422, Section "12.3.46 SendSignalAction (as specialized)".

92 Section 6.3. This rule says: "(mandatory) the style property of interactions, generated events and associated acknowledgments must match".

Interaction events are closely related to *acknowledgments.* An interaction event *implies* the existence of a related acknowledgment.

7.2.7.2
Generic events

Generic events correspond to other types of events that may happen in a game. The UML 2.4 specification (2011) provides the **AcceptEventAction** (OMG 2011) to represent an action that waits for an event to occur. The semantics for this element defined in the UML 2.4 specification (2011) declare that when an **AcceptEventAction** does not have incoming activity edges, it starts when the activity starts and keeps on waiting for the specified event to occur. In this case, it can receive several events while the activity is happening. If there are incoming edges, the control flow has to reach it like normal action nodes. When the control flow reaches the **AcceptEventAction**, it stops until the node receives the specified event. Figure 7.13 illustrates the notation for **AcceptEventAction**.

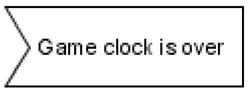

Figure 7.13: Notation for generic events

Events may also interrupt actions or activities. In this case, the UML 2.4 specification provides the element **InterruptibleActivityRegion** (OMG 2011) and a special connector ("zigzag like") to indicate that the flow has been interrupted. When the interruptions occur, all actions inside the interruptible region are aborted. Figure 7.14 illustrates the notation.

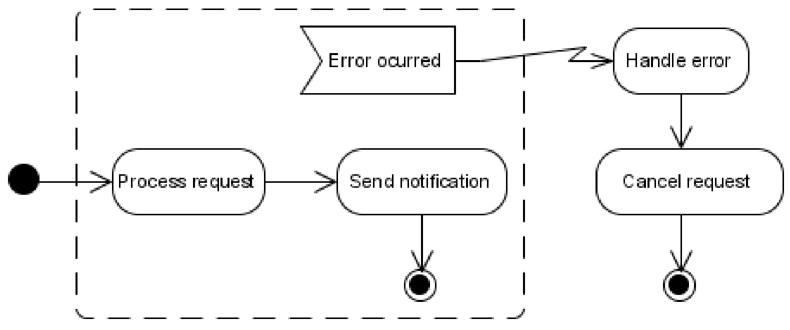

Figure 7.14: Notation for interruptible activity regions

7.2.7.3 Repetitive time events

There is also a notation for timing (or periodic) events the UML 2.4 specification also defines as an **AcceptEventAction**. Figure 7.15 illustrates the notation for repetitive time events (OMG 2011):

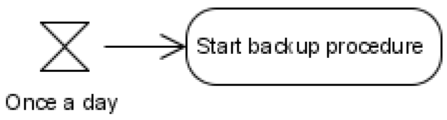

Figure 7.15: Notation for repetitive time events

7.2.8 Specifying acknowledgments

The acknowledgment is an important concept in this methodology, and thus receives special treatment. An acknowledgment is a special type of action, and is represented with the **ActionNode** extended with UML stereotypes. We have decided to use stereotypes to highlight acknowledgments in the diagram. The stereotype for acknowledgments is «ack». Also, we have defined another stereotype to differentiate the acknowledgment *style*. For the *implicit* style the stereotype is «ack, imp», while for the *explicit* style is just «ack». Hence, by default acknowledgments are considered as having explicit style. Figure 7.16 illustrates the notation.

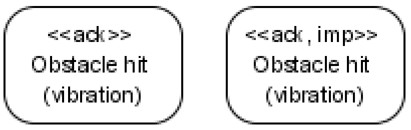

Figure 7.16: Notation for explicit acks (left) and implicit acks (right)

The acknowledgment node informs the subject of the notification (*e.g.* “obstacle hit”) and the technology of the actuator that is going to transmit this information (*e.g.* “vibration” in Figure 7.16).

When associated with *interaction events*, the acknowledgments should match the interaction event *style*.

Acknowledgments are closely related to events. Figure 7.17 illustrates an example flow with acknowledgments and an event.

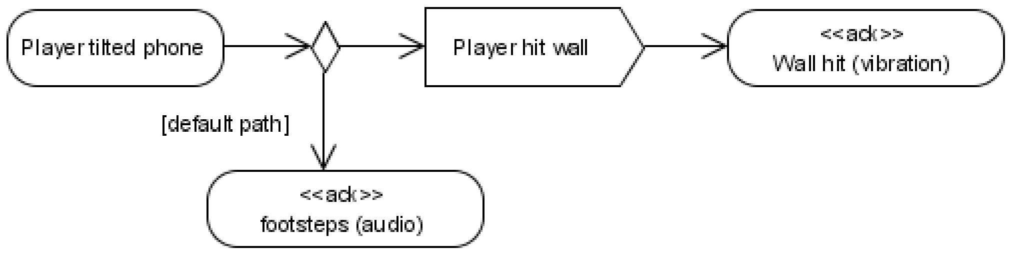

Figure 7.17: Acknowledgments and events

The example in Figure 7.17 starts when the player tilts a mobile phone (equipped with an accelerometer), and the game detects the required gesture. In the virtual world, there may be obstacles in the player's way. If this is the case, the event “player has collided against wall” occurs, triggering an acknowledgment as vibration. If there are no obstacles, the game triggers an acknowledgment as audio. This latter alternative can be considered as the common path, while the path generated from the collision is the alternative one.

The acknowledgment *scope* (local, remote, or full) is specified using the partition notation discussed in Section 7.2.4. Local and remote acknowledgments appear only in the partition of the related actor. Full acknowledgments appear in all partitions (repeated).

7.2.9 Specifying sub-activities

The UML 2.4 specification (2011) defines **CallBehaviorAction** nodes (OMG 2011) that can be used in a main diagram to invoke other activities. These nodes are represented as an **ActionNode** augmented with a rake-style symbol. Figure 7.18 illustrates an example of the UML notation.

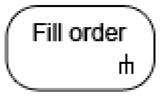

Figure 7.18: Notation for invoking sub-activities

In this example, when the control flow reaches the node, the “Fill order” activity is invoked. Using this notation is helpful for managing complex activities, when it is desirable to divide the activity into smaller sub-activities.

7.3 Summary

In this chapter, we have discussed the operational level for the DSL of our methodology. The operational level provides elements to specify scenarios in pervasive mobile games, in text and graphics form. A scenario corresponds to the specification of an activity – the interaction among players, devices, sensors and actuators in the pervasive mobile game. Scenarios are the last step in our methodology.

When specifying scenarios, we are concerned with intent – this is the desired abstraction level for this level. In this regard, it is possible to compare the abstraction level of scenarios with the abstract level of use cases, when it comes to specifying the activity flows. The same idea applies to the visual format.

However, the proposed DSL operational level is very different from ordinary use cases as the DSL provides specific rules, vocabulary and language for specifying activities in this domain (pervasive mobile games).

Apart from those differences, the specification of concrete activity flows in scenarios using text format shares ideas as the ones found in use case specifications.

The language for specifying activity flows visually in scenarios is based UML Activity Diagrams, with some extensions. We have chosen this type of diagram over other possibilities due to the level of abstraction being adequate for our purposes. Too much detailing could hinder using the diagram.

Next chapter provides concrete examples of applying notations described in this chapter.