From Activity Diagrams to Class Diagrams

João-Paulo Barros[1], Luis Gomes[2]

[1] Instituto Politécnico de Beja ,Escola Superior de Tecnologia e Gestão, Área Departamental de Engenharia 7800-050 Beja, Portugal, jpb@estig.ipbeja.pt
[2] Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Department of Electrical Engineering, 2825-114 Monte de Caparica, Portugal, lugo@uninova.pt

Abstract. A translation from activity diagrams to class diagrams, with executable code, is presented. The translation is amenable to be made with or without automating tools. An illustrative application example is also briefly presented: Activity diagrams are used in the modelling and implementation of graphical user interfaces, more precisely in the controller part of the Model-View-Controller pattern.

1 Introduction

It is well know the difficulty in maintaining consistency across different views of the same system. The use of the Unified Modelling Language (UML), with their multiple diagrams, can easily give origin to inconsistencies. The UML is more easily perceived as a union of modelling languages than as unification. As most systems revolve around the class diagram, its generation is usually given high priority. Nevertheless, many times, and particularly in event driven systems, a behavioural model is a much more easy and intuitive starting point. Among the behavioural models, the activity diagram is usually not given much attention. This is unfair as the activity diagram is actually very versatile and can be easily used by users with very different backgrounds, namely in, procedural programming languages, flowcharts, state machines, Petri nets and workflow systems. Note that this list includes almost all kinds of designers with some previous experience in the modelling of dynamic systems.

This paper presents an easily applicable translation of activity diagrams to class diagrams. This can be applied to any kind of system. One case study is briefly presented for modelling of the controller part of a Model-View-Controller pattern through the use of an activity diagram.

2 Activity Diagrams

As stated in the UML specification [1], an activity diagram is a variation of a state machine. Depending on the background of the user, an activity diagram can seem like a flowchart, a Petri net or some kind of workflow modelling notation. It allows a very

readable modelling of concurrency and of all elementary programming concepts, namely: sequence, branch, loop, fork and join [2].

Activity diagrams are usually associated to a class and, as such, they model the operations flow inside the class. This flow can depend on internal or external events. Nevertheless, the activity diagram also allows a hierarchical decomposition, through the use of *subactivity states* [1], and so it can model several classes related by class aggregation. Through the use of external events we can even synchronise several activity diagrams.

3 Translation of Activity Diagrams to Class Diagrams

The proposed translation has the following objectives:

1. To be sufficiently simple so as to be routinely and consistently used, even without supporting tools
2. To be sufficiently rigorous so as to amenable to automatic translation by supporting tools

The emphasis on this paper is on the manual translation of activity diagrams to class diagrams. The result of this translation process will be presented by the description of the resulting class diagrams representation with an associated code skeleton. The simplicity of the generated translation, namely its external interface, allows an easy integration with code generated from other models by other methods.

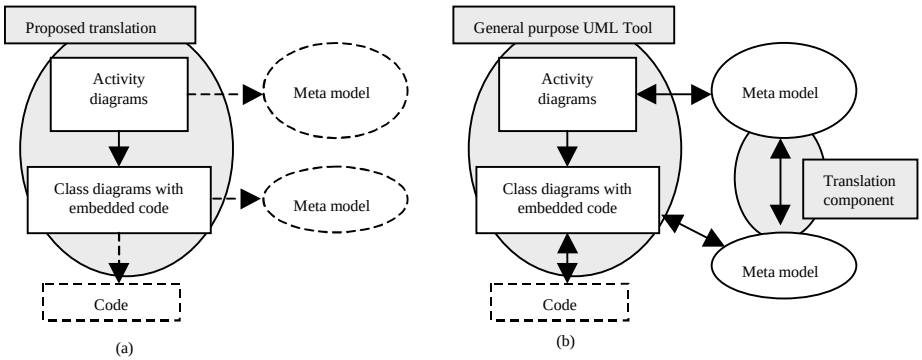


Fig. 1. Proposed translation (a) and ultimate goal (b)

Figure 1(a) shows the referred process where UML meta models are implicitly considered for the translation support (although not generated as they are not amenable for manual editing). The ultimate goal of the present proposals is to support the development of a software component, which will implement the proposed translation process. This component, based on UML meta models, will be amenable to be used inside a autonomous translation tool or integrated with a general purpose UML based development tool. This is illustrated in Figure 1(b). In the next sections, we present the translation details.

3.1 General Procedures

When translating activity diagrams to class diagrams each activity diagram will map to one class in the class diagram. Activity diagram in subactivity states are translated to aggregated classes. This is in accordance with the subactivity specified semantics which states that single activity graph may be invoked by many subactivity states. All activity diagram elements are translated to class elements in the following way:

- Each event maps to an event object. Each event object has a state (active or inactive), a list of associated transitions and a method that calls all the associated transition methods. All event objects are made elements of an event vector which is a class attribute.
- Each action state maps to an action state object. All action state objects are made elements of an action state vector, which is a class attribute.
- Transitions, decisions, merges, forks and joins, map to class methods as presented in the following sections.
- Subactivity states map to attributes. Each of these attributes belongs to a class translated from the activity diagram associated to the subactivity state.

The action state objects have four methods and one state attribute: `IsActive(); SetActive(); SetInactive(); Action(); state`.

The first method is a Boolean function that returns the object state (active or inactive). The second and third ones change the state of the object and the fourth one executes the associated action specified in the action state.

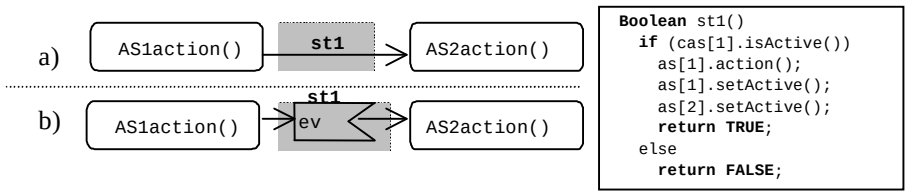


Fig. 2 - Transition translation. One implicit event (a) and one explicitly specified event (b), between two action states give origin to the same transition method.

Each transition is mapped to a Boolean method that returns true if the transition can fire and false otherwise (Figure 2). Here the term transition is considered in a much more broader sense than in the activity diagrams specification, as all the set of all arcs, control icons, decisions and merges between two or more action states are considered part of a transition. In other words, all inscriptions, arcs and nodes between action states are considered part of the same transition and are translated to only one method. We will call this broader concept of transition, super-transition so as to distinguish it from the transitions in the activity diagrams definition [1].

As the transition method is called by the activating event (ev in Figure 2), the transition can fire depending of its previous states as well as of eventual guards (Figure 2). If input states to the super-transition are active the method does the following: executes the action associated with the input action state (AS1action() in

Figure 2); inactivates the input actions states, activates the output action states; returns true to signal success.

Note that the testing of the action states activation is made based on a copy of the action states objects (as will be detailed in §4).

The translation of both super-transitions is the same. In case (a) the implicit event calls the method, in case (b) the event ev calls the method. The implicit event is always active so as to support this behaviour.

The name st1 is only for illustrative purposes as it is not part of the activity diagrams, instead it is generated as part of the translation process.

The situation where an input event occurs but some or all associated transitions are not enabled can have two disjoint interpretations: it is an exceptional but allowed occurrence; it exposes a design error that should be detected during test or verification phase. If that is the case, the action states activity or inactivity act like pre-conditions that if not met will imply a design error.

It is also interesting to consider the possibility of event methods with parameters, which are passed to transition methods.

3.2 Translation of Decisions and Merges

As already stated, decisions and merges are considered part of super-transitions. They typically have several possible inputs and/or outputs. The mapping follows the same basic structure of the simple transitions and adds decision structures to support the multiple possible flows. Figure 3 presents an illustrative example.

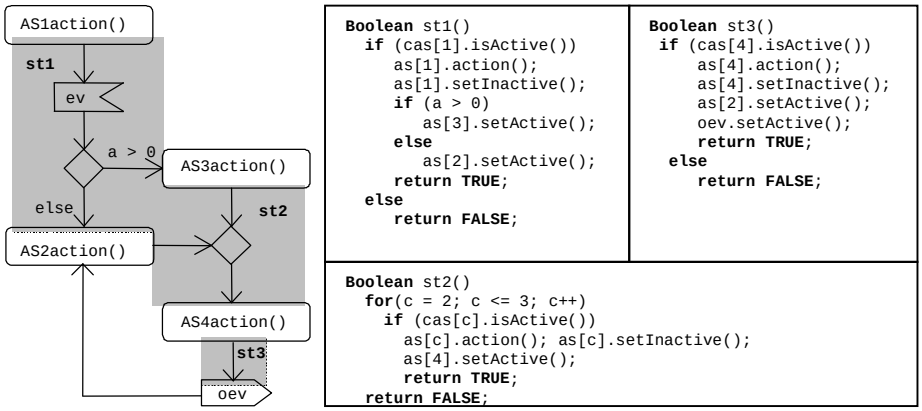
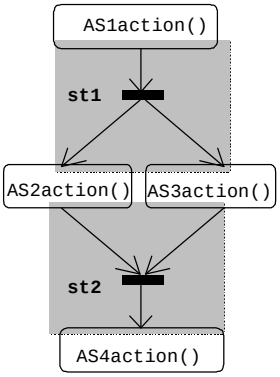


Fig. 3. – Decision and Merge translation

The super-transitions named st1, st2 and st3 are highlighted. In st3, note the activation of the event named oev. This event will be active in the next execution step (§4).

3.3 Translation of Forks and Joins

A fork is a special kind of transition with two or more output arcs or flows. In programming languages terms it translates to the concurrent execution of two, or more, threads. The join concept is the symmetrical construction. It allows the synchronisation of concurrent activities. Forks and joins are also super-transition and their translations resembles the one for decisions and merges but with the significant difference that all input, or output, must be considered together. Figure 4 presents a translation example.




```
Boolean st1()
  if (cas[1].isActive())
    as[1].action();
    as[1].setInactive();
    as[2].setActive();
    as[3].setActive();
    return TRUE;
  else
    return FALSE;
```

```
Boolean st2()
  if (cas[2].isActive()
      AND
      cas[3].isActive())
    as[2].action();
    as[3].action();
    as[2].setInactive();
    as[3].setInactive();
    as[4].setActive();
    return TRUE;
  else
    return FALSE;
```

Fig. 4. – Fork and Join translation

3.4 Translation of Subactivities States

Each subactivity state is translated to an aggregation between the class containing the subactivity state and the class containing the activity diagram inside the subactivity state. If an upper bound on resources is important, the aggregation should be replaced by composition. This will typically be the case for some embedded systems which can not allow or do not even support, the unpredictability of dynamic memory allocation.

4 Execution

The execution of the resulting code is made step by step. The event acquisition is made outside the step execution. After a step execution, the event acquisition is restarted.

The step execution starts by creating a copy of the event objects which will be used to determine which transitions (that depend on events) are active. It also creates a copy of the action states that will be used during the super-transition method execution. These copies will be used at the end of the step to update the event objects and the action state objects. In this way, and in the absence of conflicts in the model,

a deterministic execution is guaranteed. After the creation of the two copies, the events vector is processed sequentially. For each active event the associated transition methods are executed. The transition methods only change the state of the system if the significant input action states are active. This execution can, in turn, update the events state. In short:

```
start event handling step
let ce be a copy of the event vector
let cas be a copy of the action state object
for each event ev in ce
  if ev is active
    for each transition t associated to the ev
       execute t associated method
update event vector with ce
update action state object with cas
stop event handling step
```

As already presented in the previous sections, the transitions associated methods succeed if the input action states are active, and the guards evaluate to true, in the copy of the action states objects. The actions state vector is updated each time a transition method succeeds.

The stepped execution and the buffering of the events state, also enable the handling of deferred and non deferred events [1] in a unified way.

5 The Role of Activity Diagrams in User Interface Modelling

Statecharts and Petri Nets have already been used to model the controller part of the Model-View-Controller pattern [3][4] in graphical user interfaces design [5][6][7]. Activity diagrams, as a particular case of state diagrams (which are derived from Statecharts) and Petri Nets, can, naturally also be used. They present the significant advantage of a simpler formalism.

Only the Controller part will be specified by the Activity Diagram model. The controller inputs and outputs are modelled by events. Transitions firing depend not only of the input events but also on the controller internal state, which is modelled by the action states. Transition guards depend, typically, on the Model and can even modify it. This is because some system evolutions should only occur if some external action was successful, for example a connection to a remote machine. All events, super-transitions and action states (representing the system state) are part of an activity diagram corresponding to the high-level view of the controller model. This system state is modelled by the state of the action states which, as already mentioned, can act like preconditions for event acceptance.

A translation similar to the one here proposed, but based on a simple class of Petri nets, was already applied to the modelling of an Operation Terminal for a remote telecontrol and telemeasurement station for industrial process control [7]. The model and respective translation were applied not only to the model of the Graphical User Interface, but also to the handling of asynchronous events coming from the remote station. The model using the simple class of Petri nets was extremely useful in the design and implementation phases. It allowed the building of a rigorous and easily

understandable design which was of great help for the developers, all with very different backgrounds as regard to modelling tools.

The used Petri net class had very similar capabilities to the ones presently offered by activity diagrams. The only significant difference was in the Petri net capability of multiple and simultaneous activation of one activity, which is achieved by the existence of multiple token in one place. The code resulting from the net specification of the controller was smoothly integrated with a C++ framework as well as the development environment automatically generated code.

6 Conclusion

The paper presented a systematic, but simple and intuitive, way to translate activity diagrams to class diagrams with executable code. This translation to object-oriented code is shown to be simple and direct enough to allow its use even without automating tools, although these have obvious advantages. The interface of the generated code enables an easy integration with other code, even if automatically generated by other tools. This was already verified in the making of a telecontrol application centred on the Model-View-Controller pattern.

In the very near future we will formalise the mapping between activity diagram and class diagrams, through the use of UML meta models, so as to avoid any possible ambiguity and, soon after, we will start the development of a translation tool between both diagrams. We believe that the tool development will bring vital contributions to the formalisation effort.

A second aspect to be considered in the translation tool development will be the availability of reverse engineering enabling the analysis of code and automatic production of class diagrams and activity diagrams meta models.

Acknowledgement: The authors thank the anonymous reviewer for constructive remarks.

References

1.“OMG Unified Modeling Language Specification”, online, available at http://www.omg.org/cgi-bin/doc?ad/99-06-08.ps, August 16, 2000.
2. Bruce Powel Douglass, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns, Addison-Wesley, 1999.
3. Dave Collins, *Designing Object-Oriented User Interfaces*, Benjamin/Cummings Publishing Company, Inc., 1995.
4. John Hunt, “Constructing Modular User Interfaces in Java”, Java Report, Vol. 2, No. 8, Sept.1997
5. Ina Horroks, “Constructing the User Interface with Statecharts”, Addison-Wesley, 1999.
6. Rémi Bastide, Philippe Palanque, “A Petri Net Based Environment for the Design of Event-Driven Interfaces”, in Application and Theory of Petri Nets’95; Lecture Notes in Computer Science; vol. 935; Giorgio De Michelis, Michel Diaz(eds.); Springer, Berlin; pp 66-83
7. Luis Gomes, João Paulo Barros and Anikó Costa, "Detailed design of the INNOVA Station Operation Terminal"; Esprit 21017 INNOVA project internal report, EP21017/023/UNI, February 1998.