A State-based Join Point Model for AOP

Noorazean Mohd Ali, Awais Rashid

Computing Department, Infolab 21, South Drive, Lancaster University,
LA1 4WA Lancaster, UK
{mohdali, marash}@comp.lancs.ac.uk

Abstract. The implementation of crosscutting concerns in a safety-critical system needs constant system state monitoring and responding to particular (desirable or undesirable) system states. Such states and their transition cannot easily be captured by low-level fixed code-based behavioural join points exposed by current AOP join point models. In order to find ways to implement crosscutting concerns in such a system using current available join point models, a programmer is forced to write code that does not clearly represent the programmer's intention or map to the system designs it is supposed to implement, e.g., state-transition models pertaining to fault tolerance. In this paper we present the fundamental basis for a high level join point model which is based on state transitions. By defining system states pertaining to a crosscutting concern as aspects and using the transitions of these states as join points, the proposed State-based Join Point Model offers a high-level means of defining and capturing crosscutting concerns of a safety-critical system.

1 Introduction

Aspect oriented programming (AOP) [10] aims at providing a better separation of concerns [8] at development level by modularising concerns that would otherwise be tangled and scattered across other concerns. To support the implementation of crosscutting concerns AOP techniques rely on the notion of a join point model to specify where, when and how aspects observe, augment or alter the program execution. Different join point models offer different means of implementing crosscutting concerns in an application. Hence, the ability of an AOP language to support modularisation of crosscutting concerns is very much dependent on its join point model.

Research in AO software development areas has extended from language design to earlier stages of software development such as requirements engineering [3][12][13] and software design [4][6]. Work in these early aspect areas generally aims at bridging together software artifacts from different development phases, aiming for better traceability, maintainability, reusability and extensibility throughout the software development life cycle. Motivated by what we perceive as the deficiencies of existing low-level fixed code-based behavioural join point models, e.g., [1][2][12], in supporting the implementation of crosscutting concerns in systems that need constant state monitoring, our work focuses on improving the contributions of AOP in both language design and traceability of specific design and fault tolerance models to the system implementation.

There are different views of the same system, with each view emphasises or hides different aspect of the system. We specifically focus on state transition models which are often employed in safety-critical systems to depict and analyse the various system states as well as potential faults.

In this paper we present the fundamental basis for a high level join point model which supports the implementation of more *intentional* crosscutting concerns and enables code writing that represents closer mapping to the state-based system designs. Note that though we focus on safety-critical systems as our motivating problem domain, state-based modelling is not limited to this domain. The UML, for instance, has the notion of state charts which are widely used. Therefore, the aim of this paper is not to present a domain specific join point model but a general state-based join point model with safety-critical systems as a motivating example.

The aims of our proposed state-based join point model are threefold:

- To expose high-level join points in the code based on the states and state transitions of the system, by providing a state-based AOP language platform that allows such join points to be exposed;
- Allow high-level means of capturing the exposed state-based join points for the implementation of systems which need constant state monitoring, with focus on safety-critical systems;
- To allow a clear representation of the programmer's intention in implementing the relevant state-based designs when writing the code for such systems.

The remainder of this paper is organised as follows: Section 2 presents an example of a simple automatic driverless train system to illustrate the need for a high-level state-based join point model. Section 3 explains the notion of aspect modules, join points and aspect implementation in the proposed State-based Join Point Model. Finally section 4 discusses related work and section 5 concludes the paper.

2 Motivation

Capturing system states is an important part of monitoring and controlling a safety-critical system. Whether the system is manually controlled by an operator or semi/fully controlled by a computer system, the information on current system states is needed to decide the next action to be taken during system operation. For a safety-critical system, awareness of the system states is crucial. Failing to take preventive or corrective actions when the system is in dangerous states can result in fatal accidents [9]. On the other hand, desirable system states not only indicate that it is safe to continue maintaining current states, but also provide opportunities to take further actions for additional gains. For instance, in the following train system example, an awareness that the last scheduled train has left and is within a safe distance from the last station could provide the opportunity to put an extra train on the track during peak business hours.

To illustrate the issues that motivate the needs for a State-based Join Point Model, we use an example of a simple safety-critical automatic driverless train system. The reason we choose this kind of system as our motivating example is mainly because the implementation of this system needs constant system state monitoring and therefore, it should be implemented using a state-based approach. Implemented as a state-based system, the implementation of this system offers a suitable ground for testing the efficiencies of existing join point models in capturing system states and state transitions.

In this simple example, we assume that although the train system has a central controlling unit which is responsible for monitoring the overall operation of the system, the operation of running a train is controlled by the train's control unit itself. In general the train running operation comprises of a train entering the first station for boarding, leaving the station when the boarding time is up, and moving on the track heading for the next station. When the train arrives at its destination the operation cycle starts all over again.

2.1 Crosscutting Safety-Critical Concern

As a safety-critical system, one of the most important requirements is to prevent train collision during train operation. Therefore, as dictated by the safety requirements, at all times there must be a safe distance between every train running on the same track. To prevent collision between trains, throughout their movement on the track, i.e. entering/leaving a station or moving on a track:

- a train decelerates, and eventually stops if needed, if its distance from the train in front is less than 200m;
- a train maintains its speed if its distance from the train in front is within 200-400m;
- a train accelerates only if its distance from the train in front is more than 400m.

In this example, the implementation of the train operation and the safety element involves constant system state monitoring. The following Figure 1(a) illustrates the state transitions of the train operation, with the thick grey lines in *entering_stn*, *leaving_stn* and *moving* states represent the safety element that must be applied in these states. Figure 1(b) details out how the safety state influents the train operation, and Figure 1(c) illustrates the state transitions of the safety element, with conditions that determine each state of the safety element – whether it is *safe to accelerate* or *unsafe to accelerate but safe to maintain* – listed next to the states.

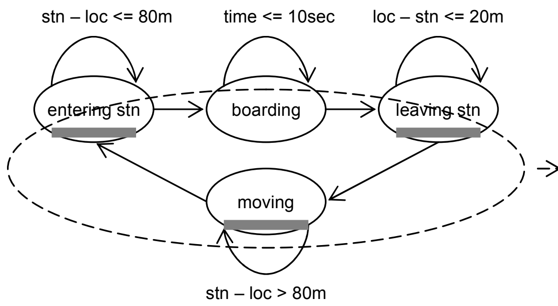

Fig. 1(a). State transitions of the train operation.

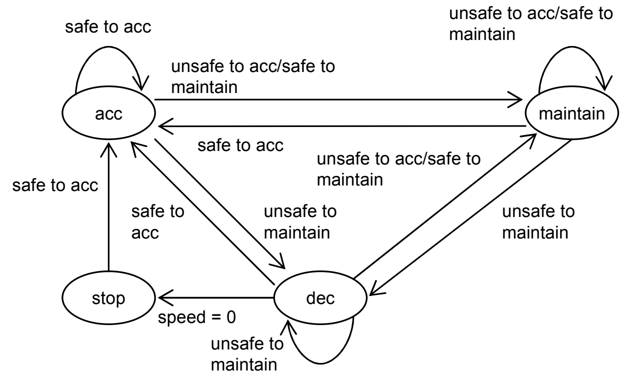

Fig. 1(b). How the safety state influents the train's operation.

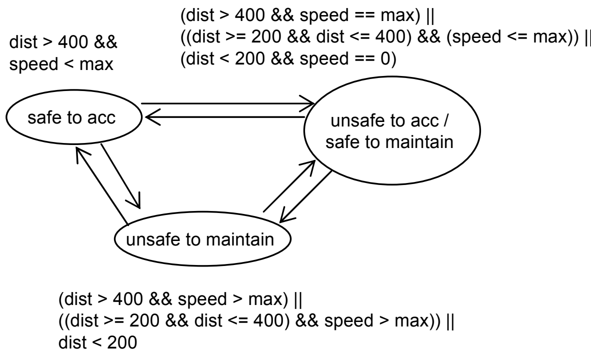

Fig. 1(c). State transitions of the safety element.

As illustrated in Figure 1(b), the safety state in Figure 1(c) plays an important role throughout a train's operation. For instance, while Train A is accelerating if Train B, which is the train in front, decelerates causing the distance between the two trains to decrease to between 200-400m and changing the safety state to *unsafe to accelerate/safe to maintain*, Train A will stop accelerating and maintain its current speed. If Train B continues to decelerate or suddenly makes an emergency stop causing the distance to further decrease to less than 200m and the state to change to *unsafe to maintain*, Train A will start decelerating. If Train B does not start accelerating and increase the distance back to more than 200m, Train A will continue to decelerate and eventually come to a complete stop. In other words, to ensure the enforcement of the safety element, throughout its operation each train's control unit must constantly monitor the train's safety state and carry out action in response to it.

As indirectly represented by Figure 1(a), the safety element implementation is a crosscutting safety-critical concern. It crosscuts a range of train operation states: entering a station, leaving a station and moving on track. The following Figure 2 illustrates the crosscutting nature of the safety element.

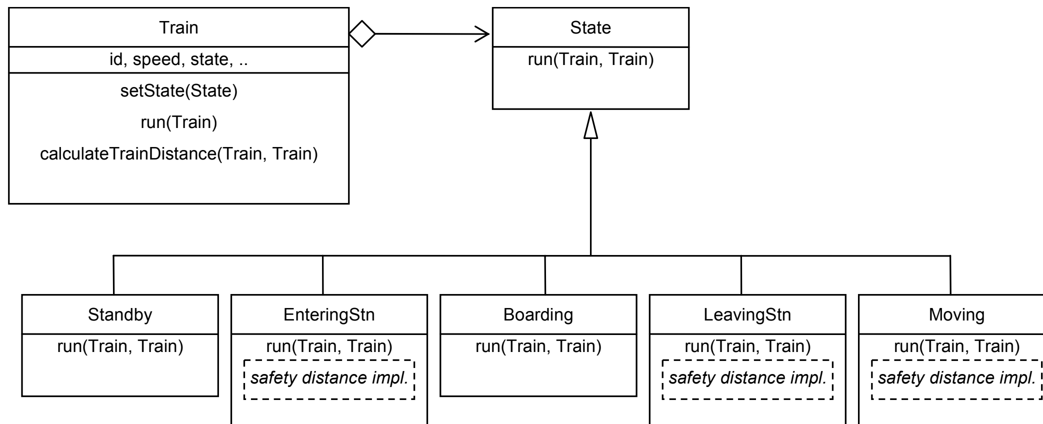

Fig. 2 UML class diagram for the states of the train operation implemented using state pattern.

As illustrated in the UML diagram for the state pattern in Figure 2, the code for the safety element – which is represented by the dashed boxes in EnteringStn, LeavingStn and Moving states – must be applied in multiple states of the train operation. Thus, it should be abstracted in an aspect.

2.2 Problem Statement

The following AspectJ code in Listing 1 shows the implementation of *EnteringStn* and *Moving* classes that implement two of the train operation states (i.e. entering station and moving on track), and the *SafetyDistance* aspect that implements the crosscutting safety element.

Listing 1: An example implementation of the safety concern in AspectJ

```
public class EnteringStn extends State {

   public void run(Train t, Train t_ifront) {

      if (t.isAtPlatform()) {
         t.stop();
         t.setState(t.BOARDING);
      }

      else { //go to platform
         //maintain speed=5
         if (t.speed < 5)
            t.adj_mode = "acc";
         else if (t.speed == 5)
            t.adj_mode = "maintain";
         else if (t.speed > 5)
            t.adj_mode = "dec";
      }

      t.adjustSpeed();
      t.updateLocation();
   }
}

public class Moving extends State {

   public void run(Train t, Train t_ifront) {

      t.adj_mode = "acc";
      t.adjustSpeed();
      t.updateLocation();
```

```
        if ((t.station - t.front_end) <= 80)
           t.setState(t.ENTERING_STN);
      }
  }

  public aspect SafetyDistance {

    void around(Train train, String adj_mode) :
      cflowbelow(call(void Train.run(..)))
      && target(train)
      && set(public String adj_mode)
      && args(adj_mode) {

      int dist = calculateTrainDistance(train, train.t_ifront);

      if (train.state == train.ENTERING_STN ||
         train.state == train.LEAVING_STN)  {

         if (dist >= 1 && dist < 200)
            adj_mode = "dec";
      }

      else { //state==MOVING
         if (dist >= 1 && dist < 200)
            adj_mode = "dec";
         else if (dist >= 200 && dist <= 400)
            adj_mode = "maintain";
      }


      proceed(train, adj_mode);
    }

    public int calculateTrainDistance(Train t2, Train t1) {...}
  }
```

In the code example, the aspect ensures that the safety requirement is met before a train is allowed to move and change its location on its track – which happens when the train enters/leaves a station or moves forward on its track. In this example, capturing the join point where the aspect should observe and intercept the program is not very straightforward. Since current low-level fixed code-based behavioural join point models do not support conditional checking at join points – which allows a condition to be fulfilled before a join point is captured, the aspect has to observe and intercept program execution each time a train moves. This is to determine the current state of the safety element and whether it should superimpose its advice at the join point. In the example, this is achieved by capturing the join point where the train is setting the value of its speed adjustment mode (i.e. `adj_mode`).

Like most of system states that are normally determined by a combination of separate system parts, determining the safety state involves capturing the speed and the location of the train and the speed and the location of the train in front, and calculating the distance between the two trains and the minimum time needed for the train in question to stop without colliding with the train in front. Although, in this simple example, gathering all system contexts needed is made possible by pointcut `target(train)`, in a real complex system not all system contexts needed can be gathered at such a low-level single join point. If that is the case, then implementing a state-based system using one of these join point models is going to be a tricky challenge to the programmer.

In this example, the implementation of the safety requirement is made possible only by having the aspect monitoring the train movement, instead of the safety state. Other than the fact that the train movement is hard to monitor at such fine granularity in a real system, this also creates a misalignment between the code and the designer's (or analyst's) intention as well as the state-based representation of the system design in Figure 1(a) and 1(b).

In conclusion, from this example, we believe that there is a need for a high-level state-based join point model that allows aspects to observe system states and join points to be captured based on the state

transitions. It is important to state here that there are real-time issues that must be considered in a real train system implementation. However, in order to narrow down the problem scope, we will not consider the real-time issues at this early stage of our join point model development and will focus on achieving a better implementation and representation of a state-based system in AOP programs.

3 State-based Join Point Model

This section presents our notion of a State-based Join Point Model (SJPM), which in general refers to defining a crosscutting system state as an abstract state machine, and using the transitions of this abstract state machine that are controlled by state guards, to identify the join points for aspect superimposition. In this section we present the SJPM by discussing the notion of aspect modules (i.e. aspect definition), join points and aspect implementation within the model.

3.1 Aspect Module

To implement the crosscutting concerns of the train system in a state-based approach, SJPM must allow *aspectual states* – the states that crosscut multiple classes – to be explicitly treated as first class program elements (i.e. aspects). In the train system example, the safety state is an aspectual state and therefore, is defined as an aspect. As first class program elements the aspect can be instantiated and attached to the train class on a per instance basis. This per instance attachment is essential in our example because each train object may be in a different safety state.

The attributes characterising an aspect in SPJM are as follows:

1. *Name* – the aspect has a name to be identified with, e.g., *SafetyDistance*.
2. *Local variables*.
3. *Local methods*.
4. *States* – each state has a *state guard,* that guards the state's value to true. If any of the conditions that specified by the state guard is not met, the state's value becomes false. This means the aspect's state is then set to another state, because only one state is allowed to be active or has true value at one time.

Although we are still working on defining the syntax and semantics of the language, a rough example of the *SafetyDistance* aspect implemented using SJPM looks like the following code in Listing 2. Note that the keyword `thisBoundObject` in the aspect refers to the Train object that is bound to the aspect.

Listing 2: An example of an aspect in State-based Join Point Model

```
aspect SafetyDistance {
    //local variable definitions
    int dist = calculateDistance(thisBoundObject,
                   thisBoundObject.t_infront);
    int speed = thisObjectBound.speed;
    int max = thisObject.MAX_SPEED;

    state safe_to_acc :
       dist > 400 && speed < max;

    state unsafe_to_acc_safe_to_maintain :
       (dist > 400 && speed == max) ||
       ((dist >= 200 && dist <= 400) && (speed <= max)) ||
       (dist < 200 && speed == 0);

    state unsafe_to_maintain :
       (dist > 400 && speed > max) ||
       ((dist >= 200 && dist <= 400) && speed > max)) ||
       dist < 200;

    //local method
    public int calculateDistance(Train t1, Train t2) {...}
}
```

3.2 State-based Join Points

As stated earlier on in this section, SJPM uses the transitions of the aspect's state to identify the join points during system execution. Using state guards as controlling mechanism, the aspect's state changes to another state whenever the condition of its state guard is violated, exposing join point in the program execution when and where aspect's advice should be superimposed on the main program. These join points are categorised as *state-based join points* because they are captured as state transitions that the aspect is interested in. The ultimate goal that we hope to achieve through these state-based join points is the ability to specify that an action should be taken (i.e. aspect superimposition) whenever the system change to a particular state, and regardless of the event that is happening in the system. For a start, we have identified three types of state-based join points that SJPM should expose.

1. Dependency-based join points

Dependency-based join points are the join points exposed when a state transition that the system is interested in occurs. For instance, a dependency-based join point allows an action to be taken when a transition occurs from state A to state B. Figure 3 below, which is one of the transitions of safety state in Figure 1(c), illustrates how the dependency-based join point allows the train system execution to be intercepted when a state transition occurs from *safe_to_acc* to *unsafe_to_acc/safe_to_maintain*, so that an action can be carried out to prevent further acceleration.

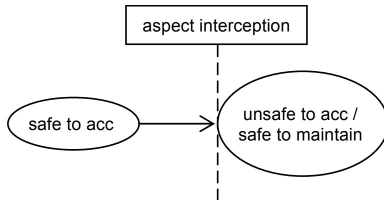

Fig. 3. A simple dependency-based join point.

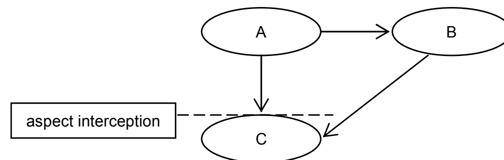

Fig. 4. A more complex dependency-based join point.

Other than a simple state dependency example in Figure 3, the dependency-based join points can also be used to expose more complex join points based on special dependency transitions. For instance, a dependency-based join point can be used to specify that there must be a transition to state B after state A, before a transition to state C is allowed (as illustrated in Figure 4), or otherwise an action should be taken.

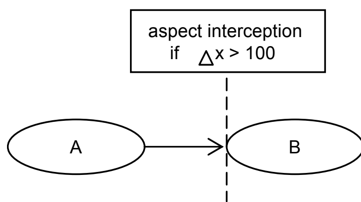

Fig. 5. A scope-based join point.

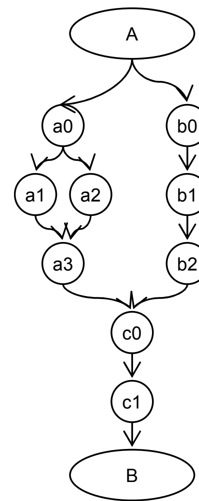

Fig. 6. A transition flow join point.

2. Scope-based join points

Scope-based join points are the join points exposed when a state transition that the system is interested in occurs, and the change in data state caused by the transition is of interest to the system. A scope-based join point is similar to a dependency-based join point because it exposes join points as a specified transition occurs. However, a scope-based join point differs from a dependency-based join point because it considers the data state change caused by the transition before deciding whether to superimpose the aspect at the join points. For example, in Figure 5 below, when a transition occurs from state A to state B, an action is taken only when the change of value of x is more than 100.

3. Transition flow join points

Transition flow join points are similar to AspectJ cflow join points. These join points expose the program execution points within the transition flow from one state to another. For example as depicted in Figure 6, a transition flow join point can be used to capture the execution points (i.e. a0, a1, a2, a3, b0, b1, b2, c0 and c1) within the transition flow from state A to state B.

3.3 Aspect Implementation Module

For a better modularity reason, the aspect definition (i.e. the aspect's states) and the aspect implementation are separated in different program files. While aspect definition defines the aspect's states and attributes, aspect implementation deals with specifying how transitions of the aspect's states can be used to capture join points and what action must be taken once the join points are captured. An initial sketch of an aspect implementation module example (which currently has an AspectJ-like advice) that captures the transitions of the states specified in the *SafetyDistance* aspect in Listing 2, looks like the following code in Listing 3:

Listing 3: An example of an aspect implementation in State-based Join Point Model

```
class SafetyRules {

    bind(SafetyDistance, Train);

    void around() : trans(SafetyDistance.safe_to_acc →
       SafetyDistance.unsafe_to_acc_safe_to_maintain) {

       //implementation of an action to ensure that
       //the train does not accelerate until
       //the state change back to safe_to_acc
    }

    void around() : trans(SafetyDistance.safe_to_acc →
       SafetyDistance.unsafe_to_maintain) {

       //implementation of an action to ensure that
       //the train immediately decelerates and
       //continues to decelerate until
       //the state change either to safe_to_acc
       //or unsafe_to_acc_safe_to_maintain
    }
}
```

In Listing 3 above the `pointcut trans(SafetyDistance.safe_to_acc → SafetyDistance.unsafe_to_acc_safe_to_maintain)` captures the dependency join point that is being exposed when transition from *safe_to_acc* to *unsafe_to_acc_safe_to_maintain* occurs. In reference to Figure 1(c), a transition from *safe_to_acc* can also happen to *unsafe_to_maintain* state. In both cases, actions must be taken to ensure the train must not accelerate when it is no longer in *safe_to_acc* state. Please note here that at this point, we are still working on identifying how the implementation of such an action can be implemented using our join point model.

4 Related Work

Our views on the needs for a state-based join points is also supported by Boucke and Holvoet [5]. In [5] they point out the need for high-level join points as a means to solve the problems in capturing abstract system states encountered when developing an application for controlling automatic guided vehicles in a warehouse management system. The main difference between our ideas of state-based join points lies in the way our join points are captured. In the initial sketch of their state-based join points they consider the use of an event-based approach to trigger the quantification for the specified state-based join points. In contrast, our state-based join point model is based on state transitions to identify and capture join points during system execution. However, both works demonstrate that aspectual states which crosscut other concerns must be defined as explicit program elements in order to support a better implementation of crosscutting concerns of a system that requires state monitoring.

Some other approaches have also taken a state-based perspective on AOSD. In [14] Whittle and Araújo propose an aspect-oriented scenario modelling approach for modelling crosscutting requirements during a requirement identification process. Their modelling approach focuses on using system scenarios to provide a step-by-step process to refine crosscutting and non-crosscutting (incomplete) system requirements; separate and model them independently; and merge them as a complete requirement set, before validating the complete requirements. A state-based approach is used to merge aspectual and non-aspectual scenarios – which are originally derived from the incomplete requirements – where each scenario is translated into a set of state machines and composed together to form a complete validated state machine description of the system requirements. Their work demonstrates how a state-based approach in requirements engineering naturally eases the process of identifying and validating aspect-oriented system requirements, since thinking in terms of scenarios and state transitions allows stake holders, system designers and system developers to work together on the requirements at the same level of understanding. For future target, it is interesting to see how our join point model supports a seamless means of implementing system requirements identified by any similar state-based approach.

5 Conclusion

This paper presents the fundamental basis for a state-based join point model to support the needs for a high-level means of capturing system states. Using a simple automatic train system as a motivating example, we demonstrate how the implementation of a state-based system could be more difficult using low-level fixed code-based behavioural join points. This is because states and their transition cannot easily be captured using these join points. As a result, this could lead to misalignment between the code, the designer's intention and the state-based representation of the system designs. As a solution we propose the notion of defining aspectual system states as aspects and using the transitions of these aspectual states as a means of capturing the join points in the program execution. So far we have identified three types of state-based join points for the state-based join point model (i.e. dependency-based, scope-based and transition flow join points), and identified the foundation for state-based aspect implementation. Our future work will focus on refining our join points, and aspect implementation (e.g., pointcuts and aspect advice), constructing the implementation language for the join point model, developing the weaver to support the proposed join point model and addressing the open issues of conflicts and aspect interactions.

Acknowledgement

This work is funded by the Public Service Department of Malaysia and National University of Malaysia.

References

1. AspectJ Home Page: http://eclipse.org/aspectj
2. AspectWerkz Home Page: http://aspctwerkz.codehaus.org

3. Baniassad, E., Clarke, S.: Theme: An Approach for Aspect-Oriented Analysis and Design. In: Proceedings of the 26th International Conference on Software Engineering (ICSE). Edinburgh, Scotland (2004) 158-167
4. Baniassad, E., Clarke, S.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)
5. Boucke, N., Holvoet, T.: State-based join points: Motivation and requirements. In: Filman, R. E., Haupt, M., Hirschfeld, R. (eds): Proceedings of the Second Dynamic Aspects Workshop (2005) 1-4
6. Clarke, S., Walker, R. J.: Generic Aspect-Oriented Design with Theme/UML. In: Filman, R. E., Elrad, T., Clarke, S., Aksit, M. (eds): Aspect-Oriented Software Development. Addison-0Wesley (2004)
7. Cigas, J.F.: The art of the states. SIGCSE (1992) 153-156
8. Dijkstra, E.: A discipline of programming, Prentice Hall (1976)
9. Jacky, J.: Safety-critical computing: Hazards, practices, standards and regulation. In: Kling, R (ed): Computerization and Controversy: Value conflicts and social choices. 2nd edn. Academic Press (1996) 767-792
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J-M., Irwin, J.: Aspect-oriented programming. Lecture Notes in Computer Science, Vol. 1241. Springer-Verlag, Berlin Heidelberg New York (1997) 220-242
11. JAC Home Page: http://jac.objectweb.org/
12. Rashid, A., Moreira, A., Araújo, J.: Modularisation and composition of aspectual requirements. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Design (AOSD). ACM Press (2003) 11-20
13. Rashid, A., Sawyer, P., A., Moreira, A., Araújo, J.: Early aspects: A model for aspect-oriented requirements engineering. In RE 2002 (2002) 199-202
14. Whittle, J., Araújo, J.: Scenario Modelling with Aspects. In: Rashid, A, Moreira, A, Teknierdogan. B (eds): Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. IEE Proceedings (2004)