Decision Modules in Models and Implementations

Serguei Roubtsov[1] and Ella Roubtsova[2]

[1] Technical University Eindhoven,The Netherlands
[2] Open University of the Netherlands

Abstract. We define a type of concern called a decision module. Decision modules can be seen as a specific subset of often changeable business rules, identified in requirements. We present decision modules as protocol machines in protocol models. The proven property of such protocol machines is their unidirectional dependency from other protocol machines. The composition technique used in protocol models allows for such local changes in a protocol machine that the behaviour of unchanged machines in the whole system is preserved.
We analyse different Java implementation techniques in order to find the possibility of building decision modules having the same properties as in protocol models. We implement decision modules using object composition, reflection, the publisher-subscriber design pattern, interceptors and aspects. The results of our experiments are illustrated with an example of a document submission system. We discuss the functionality of a generic library that we build for adopting the new style of locally changeable implementations with separated decision modules.

1 Introduction

Modules are useful instruments for handling complexity of software systems. Modules are created for various purposes giving birth to different approaches to modularisation. Among the goals of modularisation are traceability of requirements and ease of code modification, reuse, testing and support of system evolution.

The major goal of the modularisation technique presented in this paper is the support of system evolution. The system evolution starts with new requirements caused by new business ideas, changes in laws, regulations and business rules. "International Data Corporation (IDC) asked in a survey: How often do you want to customize the business rules in your software? 90% of respondents reported they try to change it annually or more frequently. 34% said monthly... A conventionally programmed software package can seldom be reprogrammed this often"[9].

New policies, laws and business rules result in instructions on what to do in a given situation. An instruction combines a description of the situation, the expected actions or events and the directives permitting or forbidding events in the described situation. A model or implementation of such an instruction usually combine states of system objects, as well as events and control flow constructions corresponding to the description. In this paper, we define such a combination as a decision module. If a business rule could be presented as a decision module in models and implementations and if this module could be locally changed without the necessity to change the rest of the model or implementation, then such a modularisation would ease system evolution. However, the separation of such modules is not a common practice in conventional modelling and implementation approaches.

The most commonly used modularization is the separation of object life cycles. This means that a domain concept is modelled as a class of objects. Each object is created, exists in some states, makes decisions on the basis of its states and recognised events. Eventually, the object is destroyed or deleted. Modularisation of object life cycles hides the states of objects and process control points inside objects. To implement a new policy, modify a program or generate a test, the control points inside life cycle modules have to be analysed. As a result, each modification of policies and business rules requires re-modelling and re-implementing all the life cycle modules contributing to the decisions made by these policies and business rules.

The research on business rules [4, 10] investigates the ways of modularisation of business rules and the methods of their integration. However, the business rules engines are usually based on the workflow concept and the Business Process Modeling Language (BPML). They integrate the entire control flow and, thus, do not provide the possibility for local changes of the business rules.

There is a modelling technique called Protocol Modelling that supports the separation of the modules called "behaviours" that derive own state from the states of different objects and combine event descriptions with control flow elements. Protocol Modelling is based on the CSP parallel event-based composition technique [8] extended for models with data [22]. This composition technique, implemented in a composition engine, supports separation and composition of both life cycle modules and "behaviours". The property of observational consistency proven in Protocol Modelling for modules and the system they form perfectly supports local changes of modules without the need to validate the rest of the model. Thus, the properties of "behaviours" facilitate model changing and testing. The same properties are desirable for the decision modules both in models and implementations. In this paper we intend to investigate *if it is possible to protocol model decision modules and implement them using the available object-oriented techniques in such a way that the implementations would have the same properties of "behaviours" as in executable protocol models.*

The rest of the paper is structured as follows.
Section 2 defines a decision module.
Section 3 presents related work.
Section 4 introduces a case study used for illustration of our modularization vision.
Section 5 shows how the decision modules are identified in requirements.
Section 6 identifies the decision modules in executable protocol models. It ex-

plains the composition of decision modules and life-cycle modules and recognizes the attractive properties of modules in Protocol Modelling.
Section 7 applies different software development techniques (object composition, reflection, the publisher-subscriber design pattern, interceptors and aspects) for implementation of decision modules and evaluates how the implementations preserve the desired properties of protocol models.
Section 8 discusses the advantages of the proposed approach for the modelling and implementation as well as tackling the obstacles in its adoption.
Section 9 concludes the paper and draws perspectives for the future work.

2 Definition of Decision Module

In this section, we give a notation independent definition of a decision module. Later in section 6, a decision module will be defined as a protocol machine.

A model of a system is described as a set of interacting objects. A behaviour of an object is its description in terms of states and transitions. The behaviour of a system of objects is defined on the basis of the states and events of objects. However, a system has its own goals and the behaviour of the system is often different from the disjoint union of the individual behaviours of objects. Decision modules direct behaviours of objects in particular combinations of states towards system goals.

Definition 1. *A decision module represents a specification of an instruction on how to make the decision about allowing or forbidding certain events.*

- *The instruction is defined on a set of objects selected in a model. The type of each selected object contains specifications of its possible states and the events allowed in the specified states. All possible combinations of the states of selected objects and a subset of the union of all events of the set of selected objects are used for specification of a decision module.*
- *The instruction can be seen as an "if (case)"- construction*

$$if\ (State\ Function() = value)\ then\ allow\ Event;$$

The domain of the State Function() is defined by the partitioning of the state space of the set of selected objects into disjoint subsets. The partitioning can be done before (or after) the Event in hand.

The range of the state function is a set of names of these disjoint subsets presented with a nominal variable called "state of the decision module". The nominal values are the abstractions (often - business names) of the conditions for a decision.

For example, let's consider a decision module which can belong to an object *Aircraft*. It's state function:

$$If(State\ Function() = \text{``}All\ Passengers\ Are\ On\ Board''),\ then\ allow\ Start.$$

The *Aircraft*'s behaviour is coordinated with the behaviors of its *Passengers*. The selected event is *Start*. The state space has been partitioned before the *Start* into two subsets *'All Passengers Are On Board"* and *'Not All Passengers Are On Board"*.

We separate a decision module as a *module* because it can be associated (in terms of programming - reused) with different objects as a separate entity. In our example, it can be associated with an *Aircraft* or, say, a *Bus*.

We name this module a *decision* module because it forms the condition for the acceptance or refusal of an action processing an event. It is not a pre-condition because it can be derived both from the pre- and post-states of the actions in the life cycles of the set of selected objects.

A decision module cannot be classified as an object. An object owns (may change independently) its state, whereas the state of a decision module *is derived* from the states of other modules. A decision module only permits or forbids the actions specified in objects from the set of selected objects.

A decision module does not fit exactly into the definition of a crosscutting concern or an aspect [7]. It can be seen as a high-level management concern or a control concern. It can be crosscutting or not.

Finally, what we can observe about a decision module is that:

- it is recognized and can be separated at the early stages of system development such as requirements engineering;
- it is often not separated as an entity in conventional models and implementations;
- it tends to undergo frequent changes at the later stages such as maintenance and evolution.

3 Related Work

Modules similar to our decision modules have been also recognized in Business Rules community. These modules are called *enablers* [1, 3].

"An enabler is a type of action assertion which, if true, permits or leads to the existence of the correspondent object." An enabler has varying interpretations depending on the nature of the correspondent object: it may permit (i.e. enable) the creation of a new instance; permit another action assertion; permit an action execution [3] and is often called an integrity constraint, a condition or a test.

The enablers represent only a subset of our decision modules because the decision modules

- describe assertions of a set of actions using the state space of a set of selected objects as an input;
- can both enable and disable (refuse) an action execution, the creation of a new instance and another action assertion.

In rather advanced form, such an approach to modularization can be seen in protocol models [20]. The modules called "behaviours", separated in protocol models, possess the following properties:

- use state functions that are able to read (but not change) attributes, an event pre-state of other modules [22];
- use state functions that are able to predict the post-state of other modules for the given event [22];
- often use the control flow constructions to permit events at specified values of the state function;
- are composed with different life cycle modules (objects) in such a way that the life cycle modules do not know (remain oblivious) about the "behaviours" and do not need to be changed as the "behaviours" are added or changed [19, 20].

Protocol modeling makes use of an extended form of the Communicating Sequential Processes (CSP) [8] parallel composition of modules which possess internal data. The CSP parallel composition produces observationally consistent models. This means that a protocol model allows one to modify modules locally, add and delete modules so that the behaviour of unchanged modules is preserved in the behaviour of the whole system [19].

In section 6, we will show that the decision modules are easily separated in protocol models as "behaviours" or, in other words, as protocol machines with derived states.

4 Case study: Preparation of a document by several participants

We illustrate the proposed modularization with a case study. It will be used to show how the decision rules declared in requirements can be transformed into modules of executable synchronous protocol models. It will be also used to illustrate our investigation of the applicability of Java-based techniques for implementation of decision modules.

Let us consider a system that controls a preparation of a document, e.g. a proposal, a paper or a report, by several participants. One of the participants usually plays the role of the coordinator responsible for submitting the document. There is a deadline for the document submission.

The coordinator creates the parts of the document and chooses participants. Each part is assigned to a participant. A part has its own deadline before the deadline of the document and should be submitted by the participant so that the coordinator has time to combine parts and submit the document.

If a participant misses the part deadline, the coordinator sends a reminder to the delaying participant. The coordinator can change the deadline or assign the document to another participant. Only the coordinator can cancel the preparation of the document.

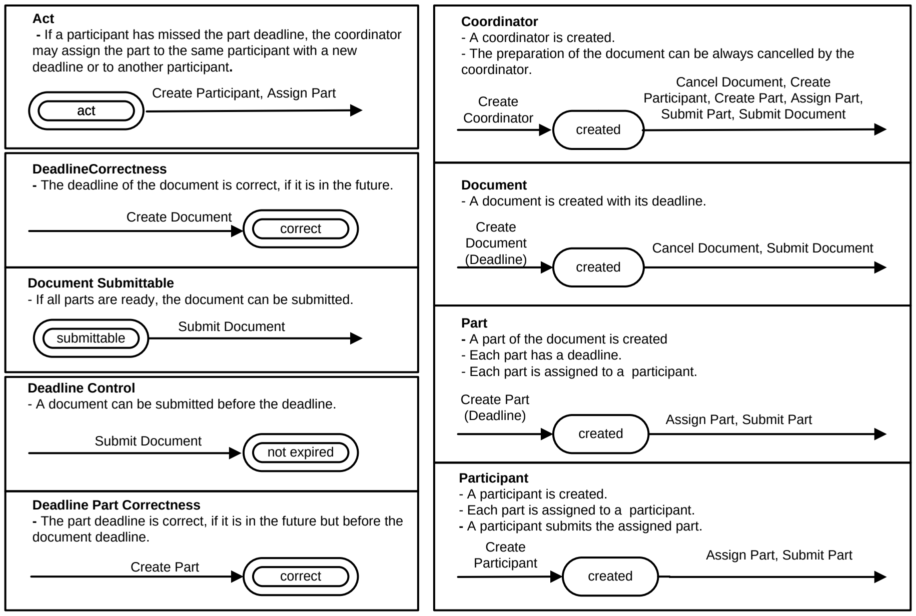

Act
- If a participant has missed the part deadline, the coordinator may assign the part to the same participant with a new deadline or to another participant.
act
Create Participant, Assign Part
DeadlineCorrectness
- The deadline of the document is correct, if it is in the future.
Create Document
correct
Document Submittable
- If all parts are ready, the document can be submitted.
submittable
Submit Document
Deadline Control
- A document can be submitted before the deadline.
Submit Document
not expired
Deadline Part Correctness
- The part deadline is correct, if it is in the future but before the document deadline.
Create Part
correct
Coordinator
- A coordinator is created.
- The preparation of the document can be always cancelled by the coordinator.
Create Coordinator
created
Cancel Document, Create Participant, Create Part, Assign Part, Submit Part, Submit Document
Document
- A document is created with its deadline.
Create Document (Deadline)
created
Cancel Document, Submit Document
Part
- A part of the document is created
- Each part has a deadline.
- Each part is assigned to a participant.
Create Part (Deadline)
created
Assign Part, Submit Part
Participant
- A participant is created.
- Each part is assigned to a participant.
- A participant submits the assigned part.
Create Participant
created
Assign Part, Submit Part

Fig. 1. Declarative specification

5 Decision Modules in Requirements

In our experience of requirements engineering we have found that requirements often describe the decision modules informally.

We start with an observation that almost every sentence of requirements presents a snapshot of the desired system behaviour. A snapshot is a visible abstract state captured after or before an event. Figure 1 presents all the snapshots corresponding to the case study in section 4 as a declarative specification. We depict an abstract system state as a double line oval. An oval may have an ingoing or outgoing arc labeled with an action that can happen.

For example, the declaration *"A document can be submitted before the deadline"* can be presented as a decision rule or a decision module *DeadlineControl*. It shows that the event *Submit Document* can only be accepted if the deadline is "not expired". In this case, the arc is ingoing.

If an event can only be accepted in the described state, then the arc is outgoing. The example is the decision module *Document Submittable: After all parts are ready, the document can be submitted.*

The state descriptions in decision modules are abstracted from the life cycle of entities of the system. An abstract state may present the state of a set of system concepts, a subset of states of the system, etc. For example, state *submittable* of the decision module *Document submittable* depends on the states of all parts of the document.

Often the decision modules give instructions or polices on what to do in a situation described as an abstract state. For example, *Act* presents a possibility to progress by creating a new participant (event *Create Participant*) who can write a part of the document (event *Assign Part*).

The elements of the life cycles of entities in the model are also present in requirements and shown in Figure 1 as declarations. However, the states in such description are not abstract, they are the states of objects. We depict a state of an object as a single line oval. For example, we can read in requirements what an instance of the *Coordinator* can do when it is in state *"created"*. It can *Create Document*, *Submit Document*, *Create Participant*, *Create Part*, *Assign Part* and *Cancel Document*.

6 Decision Modules in Protocol Models

The declarative specifications are not executable. However, there is a way to present decision modules as modules of executable protocol models. We show this way of modularization after a short introduction of Protocol Modelling developed by A.McNeile [22].

6.1 Protocol Modeling

Protocol Modeling splits the Universe into a system and its environment. A protocol model represents the modelled system. The environment submits events to the system. The system may change its state reacting to events.

The building blocks of a protocol model [22] are protocol machines and events. They are instances of, correspondingly, *protocol machine types* and *event types*. Each protocol machine "recognises" a finite set of event types, i.e. uses the names of these event types in the specification of a protocol machine type.

In order to facilitate reuse, there are two types of protocol machines: Objects and Behaviours. Behaviours cannot be instantiated on their own but may extend functionality of Objects. In a sense, Behaviours are similar to mixins or aspects in programming languages [2, 20].

A protocol machine type is an LTS (Labelled Transition System) extended with attributes and call-backs to enable modelling with data:

$$PM_i = (s_i^0, S_i, E_i, T_i, A_i, CB_i,),\ where$$

- s_i^0 is the initial state;
- S_i is a non-empty finite set of states;
- E_i is a finite set of transition labels being the "recognized" event types e_i, coming from the environment. The set can be empty.
- $T_i \subseteq S_i \times E_i \times S_i$ a finite set of transitions: $t = (s_x, e, s_y),\ \ s_x, s_y \in S_i,\ \ e \in E_i$. The set of transitions can be empty. The states are updated by transitions.

- A_i is a finite set of attributes of the specified types. The standard data types such as *String, Integer, Currency, Date*, etc. plus the types of protocol machines can be used for specification of attributes. The attributes are the data containers of a protocol machine. A protocol machine Object contains at least one attribute, the Name of the Object. The set of attributes of a Behaviour protocol machine can be empty.
- $CB_i(PM_1, ..., PM_n, E_1, ..., E_m) = (PM_1, ..., PM_n, E_1, ..., E_m)$ is a callback function. $PM_1, ..., PM_n$ are the protocol machines of the protocol model. $E_1, ..., E_m$ are the events of the protocol model. We list all protocol machines of the protocol model and the events "recognized" by PM_i as the arguments of the callback function CB_i because the elements of all protocol machines and all "recognized" events can be used as inputs for updating the values of the attributes, states and events of the protocol machines. These values can be updated using the callback function only as a result of a transition, i.e. as a result of event acceptance.

  If no calculation is needed for updating attributes of the protocol machine PM_i, the set of callback functions is empty.

"Recognized" events are modelled from the system perspective. Each event belongs to a specified type telling the system what kind of attributes can be found in this event.

An event type is a tuple $e = (A^e, CB^e)$, where

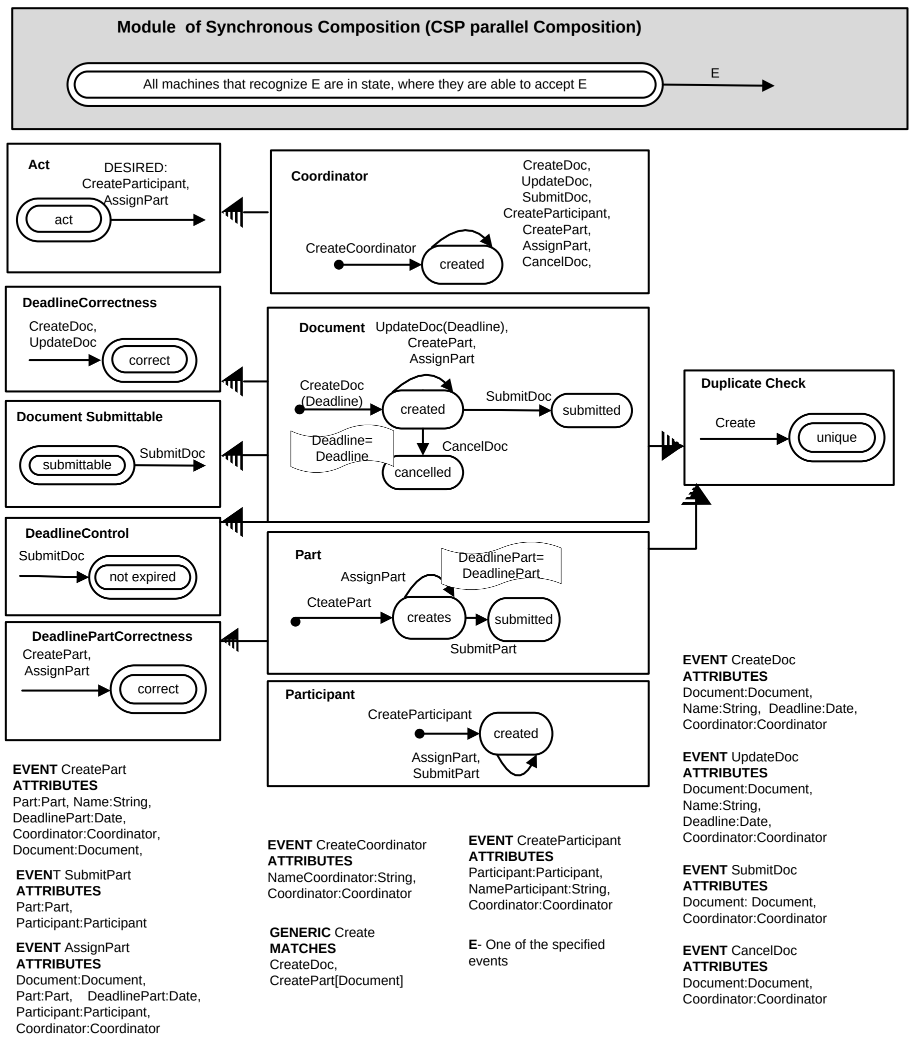
- A^e is a finite not empty set of attributes of the event.
- $CB^e(PM_1, ..., PM_n, E_1, ..., E_m) = (PM_1, ..., PM_n, E_1, ..., E_m)$ is a callback function corresponding to this event. The callback function for an event is used if the protocol model generates other events using the attributes of this event.

Within Protocol Modelling, callback functions are the instruments for data handling. In the ModelScope tool [21] supporting the execution of protocol models, the callbacks are coded as Java classes with methods changing and/or returning the values of attributes and states of instances of protocol machines. They may also change attributes of events and generate event instances.

CSP parallel composition.

A protocol model (PM) is a CSP parallel composition of a finite number of *instances of protocol machines*. A PM is also a protocol machine, the set of states of which is the Cartesian product of states of all composed protocol machines [22]:

$$PM = \mathop{\|}_{i=1}^{n \in N} PM_i = (s_i^0, S_i, E_i, T_i, A_i, CB_i) = (s_0, S, E, T, A, CB,).$$



EVENT CreatePart
ATTRIBUTES
Part:Part, Name:String,
DeadlinePart:Date,
Coordinator:Coordinator,
Document:Document,

EVENT SubmitPart
ATTRIBUTES
Part:Part,
Participant:Participant

EVENT AssignPart
ATTRIBUTES
Document:Document,
Part:Part, DeadlinePart:Date,
Participant:Participant,
Coordinator:Coordinator

EVENT CreateCoordinator
ATTRIBUTES
NameCoordinator:String,
Coordinator:Coordinator

GENERIC Create
MATCHES
CreateDoc,
CreatePart[Document]

EVENT CreateParticipant
ATTRIBUTES
Participant:Participant,
NameParticipant:String,
Coordinator:Coordinator

E- One of the specified events

EVENT CreateDoc
ATTRIBUTES
Document:Document,
Name:String, Deadline:Date,
Coordinator:Coordinator

EVENT UpdateDoc
ATTRIBUTES
Document:Document,
Name:String,
Deadline:Date,
Coordinator:Coordinator

EVENT SubmitDoc
ATTRIBUTES
Document: Document,
Coordinator:Coordinator

EVENT CancelDoc
ATTRIBUTES
Document:Document,
Coordinator:Coordinator

Fig. 2. Executable Protocol Model

$s_0 = \bigcup_{i=1}^{n} s_i^0$ is the initial state;

$S = \prod_{i=1}^{n} S_i$ is the set of states;

$E = \bigcup_{i=1}^{n} E_i$ is the set of events;

$A = \bigcup_{i=1}^{n} A_i$ is the set attributes of all machines;

$CB = \bigcup_{i=1}^{n} CB^i$ is the set of callbacks of all machines.

The set of transitions T of the protocol model is defined by the rules of the CSP parallel composition [8]. The rules synchronise transitions of protocol machines.

The CSP composition rules in Protocol Modelling are:

- If an event is not recognised by the protocol model, it is *ignored*.
- If an event is recognised by the protocol model and all protocol machines, recognising this event, are able to accept it, the event is *enabled*.
- If an event is recognised by the protocol model, but at least one protocol machine, recognising this event, is not able to accept it, the event is *refused*.

As the result, the composition may contain the union of transitions of composed protocol machines if the sets of the "recognised" events of protocol machines are disjoint. If the sets of the "recognised" events are not disjoint, the set of allowed transitions is defined using the Cartesian product $\prod S_i$ of states of machines, the set of events and the rules of CSP parallel composition. An algorithm of calculation of the set of transitions can be found in [26].

Dependent protocol machines. Derived States.

Transitions T_i of a protocol machine PM_i enable updates only its own states; namely, those in S_i. On the other hand, protocol machines can read the states of other protocol machines, although cannot change them. Callback functions CB_i are used to read states of specified protocol machines and update attributes and calculate derived states of protocol machines of type Behaviour.

Callback functions create dependencies between protocol machines. The dependency means that one protocol machine (usually the included protocol machine of type Behaviour) needs to read the state of other protocol machines to calculate its own state. Such calculated states are called *derived states*, which distinguishes them from the *stored states* denoted in the model [22]. A protocol machine with derived (calculated) states is called dependent.

A transition of a dependent protocol machine contains a derived state and an event, permitted in this state. The derived state can be either an input or output state of this event.

As all protocol machines are composed with he same CSP parallel composition rules, a dependent protocol machine specifies an extra "restrictions" on the acceptance of an event by other protocol machines of the protocol model. Note that these "restricted" protocol machines are not necessarily the same protocol machines, states of which have been read to derive the state of the dependent protocol machine.

The ability of protocol machines to read the state of other protocol machines is an asset for separation of decision modules. Decision modules need this to read the information of other modules and use it to specify a decision about event acceptance.

Further, there are two types of derived states possible in dependent machines, which can be used in decision modules:
(1) The pre-state of a transition which can be calculated. The pre-state is similar to guards calculated in Coloured Petri Nets (CPN) [12] and the UML state machines [23].
(2) The post-state of a transition which can also be calculated. We mentioned already that the callback functions can update the values of the attributes, states and events only as a result of a transition, i.e. as a result of event acceptance. If a post-state refuses the event caused its calculation, the event is rolled back, i.e. the system sends a messages about the post-state value and it is returned into the state that preceded to the event acceptance. This semantics does not exist either in the UML, CPN or BPMN. An example of a decision module using a post-state will be shown in the next sub-section.

6.2 Protocol Model with Decision Modules in the Case Study

The protocol model of our case study is shown in Figure 2. This protocol model can be executed in the ModelScope tool [21].

A finite set of *EVENTS* is defined for this protocol model. For example, the event type *CreateDoc* is a tuple of variables of types *Document, Coordinator, String* and *Date*. The elements *Document* and *Coordinator* are the protocol machines of types *Document* and *Coordinator* correspondingly (Listing 1).

Listing 1. EVENT CreateDoc

```
EVENT CreateDoc
ATTRIBUTES
    Document:Document,
    Name:String,
    Deadline:Date,
    Coordinator:Coordinator
```

As we see, the event is described as a set of attributes of standard data types and the types of protocol machines *Coordinator* and *Document*.

The life cycles entities (objects) *Coordinator, Participant, Part* and *Document* are specified as protocol machines. Figure 2 shows the protocol machines graphically.

Listing 2 shows the textual specification of a protocol machine of type *Document.* Before its creation, any object is in the state @*new*. Accepting events, any object transits to states of its life cycle. For example, being in state @*new* and accepting event *CreateDoc* an object of type *Document* transits from state @*new* to state *created.*
The corresponding transition is depicted as follows: *@new*CreateDoc= created.*

Listing 2. OBJECT Document

```
OBJECT Document
NAME Name
INCLUDES      DeadlineControl,
              DocumentSubmittable,
              DeadlineCorrectness,
              DuplicateCheck
ATTRIBUTES
          Name:String,Deadline:Date,
          Coordinator:Coordinator
STATES created, submitted, cancelled
TRANSITIONS @new*CreateDoc= created,
              created*UpdateDoc=created,
              created*CreatePart=created,
              created*AssignPart=created,
              created*SubmitDoc=submitted,
              created*CancelDoc=cancelled
```

Using the protocol machines presenting the life cycles of system entities, the decision modules can be specified. In order to establish the functional relations between the states of life cycle modules (objects) and the derived states of decision modules, the decision module is described as a labelled transition system with callback functions.

For example, the *decision module DeadlineControl* in our protocol model consists of a description of the labeled transition system (Listing 3) and the corresponding java class (of the same name, Listing 4) describing functional relation between the states of the decision module and the life cycle modules.

The relation between the *DeadlineControl* and *Document* is specified with the *INCLUDE* sentence in the *Document* (Listing 2). To facilitate reuse, the *DeadlineControl* decision module can be included in any other object that have an attribute *Deadline.*

Listing 3. BEHAVIOUR DeadlineControl

```
BEHAVIOUR !DeadlineControl
# Allows SubmitDoc only if
# the deadline is not expired
        STATES expired , not expired
        TRANSITIONS @any*SubmitDoc= not expired
```

Listing 4. Java Callback for BEHAVIOUR DeadlineControl

```
import java.util.Date;
public class DeadlineControl extends Behaviour{
  public String getState(){
    Date expDate = this.getDate("Deadline");
    Date currentDate = new Date();
    return currentDate.compareTo(expDate)>0
    ?"expired":"not␣expired";
  }
}
```

The decision module *DeadlineControl* contains transition @*any*∗*SubmitDoc* = *not expired*. State @*any* literally means any possible combination of the states of the life cycle modules in the model.

The functional dependency between *DeadlineControl* and the attribute *Deadline* of the *Document* is defined in the java class *DeadlineControl* shown in Listing 4 as a callback. Behaviour *DeadlineControl* relates an instance of the *Document* with the system clock which is invisibly present in the model. The system clock gives the current date. The current date is compared with the *Deadline* of the *Document*. The derived state *"expired"* or *"not expired"* is returned to the protocol machine *DeadlineControl*.

The decision module *DeadlineControl* is an example of the Behaviour that calculates the post-state after proceeding of the event *SubmitDoc*. If the derived state of *DeadlineControl* after processing an event of type *SubmitDoc* has the value "expired", then the event is rolled back and the system returns into the state before processing of this event.

Figure 2 also shows the examples of the Behaviours that use the pre-state of other protocol machines for making the decision. For example, Behaviour *Document Submittable* is included into the Object *Document*. *Document Submittable* derives its state *submittable* only if all *Parts* of the *Document* are in state *submitted* (Listings 5 and 6).

Listing 5. BEHAVIOUR DocumentSubmittable

```
BEHAVIOUR !DocumentSubmittable
# Ensures that a document  cannot be
#submitted if it has unfinished Parts
ATTRIBUTES !Document Status: String
        STATES submittable , not submittable
        TRANSITIONS submittable*SubmitDoc=@any
```

Listing 6. Java Callback for BEHAVIOUR DocumentSubmittable

```
public class DocumentSubmittable extends Behaviour {
 public String getState() {
  boolean allSubmitted =true;
    Instance[] myParts =
        this.selectByRef("Part","Document");
    if (myParts.length==0)
                {allSubmitted = false;}
    for (int i = 0; i < myParts.length; i++) {
     if ((myParts[i].getState("Part").equals("created")))
            allSubmitted = false;
                }
   return allSubmitted?"submittable":"not␣submittable";
 }
 public String getDocumentStatus() {
   return this.getState("DocumentSubmittable");
        }
}
```

Figure 2 presents six decision modules: *DeadlineCorrectness, DeadlinePart-Correctness, DeadlineControl, Document Submittable, Act, Duplicate check.* The arc with the half-dashed triangle end shows the INCLUDE relation.

Several decision modules may be included into the same object. For example, four decision modules are included into object *Document.* All the decision modules processing the submitted event need to be executed to make a decision about its proceeding. Any event proceeds only if all the protocol machines recognising this event permit its proceeding.

6.3 Properties of Decision Modules in Protocol Models

We can name the following *properties of decision modules in Protocol Models*:

1. *Modularity:* a decision module localises the decision making rules (separates them for the purpose of reuse);
   For example, the module *DeadlineControl* can restrict the behaviour of objects *Document* and *Part* in the same way.
2. *Unidirectional dependency*: the decision modules can read the state of other modules, but other modules do not know how the decision is made (other modules are oblivious [7]).
   For example, *DeadlineControl* reads the value of attribute *Deadline* of the object *Document* and predicts state *expired*, *not expired* after event *Submit-Doc*. The object *Document* remains oblivious.
3. *Mechanism to achieve the properties* is *event-driven with CSP parallel composition.*

Decision modules are incorporated into the protocol model on the basis of their ability to react to predefined events following the rules of CSP parallel composition. The CSP parallel composition of all modules in protocol model allows for local modification, adding and deleting of modules without affecting the ordering in the specified behaviour sequences of existing modules.

Executable protocol models enable separation of decision modules defined in requirements. Requirements become traceable in executable models. There are obvious advantages of modularisation of decision modules for traceability of requirements and testing and modification of models.

Traceability. Traceability of requirements in models is prescribed in standards and considered as a prerequisite of a proper system evolution, modifiability and long life. The developers should convince themselves and their customers that the system does what it was required to do. Modularisation of decision modules directly transforms the declarations or items of requirements into modules of the model. For example, the item "If all parts are ready, the document can be submitted" is traced in the decision module *Document Submittable.*

Testing. Modularisation of decision modules defines the testing strategy. Each of the decision modules specifies a finite set of tests. The set of tests is finite because the decision module partitions the data into groups. Each group results in a decision. Testing only one representative from each group is sufficient to test the decisions and the variants of behaviour resulting from this decision. For example, in order to test the decision module *Document Submittable*: "If all parts are ready, the document can be submitted" two tests should be designed:
(1) a document has been created; at least two parts have been assigned; one part has been submitted and another part has not been submitted;
(2) a document has been created; the parts have been assigned and all parts have been submitted.

Modification. In our model we have not separated the decision module *Cancel Document.* However, we can easily modularize cancellation of a document and compose it with the model. A new decision module will define that *"If a document is in a state created, it can be canceled or submitted"*.

Systematic separation of decision modules from requirements to models and implementation promises advantages for traceability of requirements, testing and modification of the implementation. In the next section we investigate if the decision modules with the same properties as in protocol models can be implemented using such a mainstream programming language as Java.

7 Decision Modules in Java

The implementation of decision modules using mainstream programming languages is the question that needs investigation. To the best of our knowledge, there are no systematic implementation approaches for separation of enablers.

The research question of this paper is the following:
Is it possible to implement the decision modules using mainstream object-oriented

language techniques in such a way that the implementation of decision modules would have the same properties as the decision modules in executable protocol models, namely:
- modularity;
- unidirectional dependency;
- event-based composition?

For our experiments with the implementation of decision modules we have chosen Java as one of the mainstream object-oriented programming languages.

First, we investigated if decision modules can be implemented within common Java paradigm, that is, without using any frameworks and special libraries. We consider this rather important because relying upon specialised libraries and frameworks usually makes the implementation less generic with respect to, for example, underlying architecture. It can also make the solution platform- and vendor-specific violating a well known Java principle "write once, run everywhere".

Further, we also investigate the expressivity of Enterprise Java Beans (EJB3) and aspect-oriented Java (AspectJ) [15, 27] for implementation of enables and decision modules.

7.1 Using Object Composition

It seems that a simple way to implement decision modules is to use object composition where they are included in life cycle modules as object fields. In the Listing 7 both OBJECT *Document* and BEHAVIOUR *DeadlineControl* are shown as Java classes, the former includes the latter as an instance variable. As we said in 6.1, the difference between life cycle modules(objects) and behaviours in Protocol Modelling is that behaviours cannot be instantiated on their own but rather extend functionality of objects. In the implementations in this paper, we do not implement this restriction. That's why there is no need in a separate class Object and both classes extend the same parent class Behaviour.

Listing 7. Implementation using Object Composition

```
class Document extends Behaviour{
    private String name;
    private Date deadLine;
/* 'INCLUDES' in the model is implemented
as object composition */
    private DeadlineControl deadlineControl;
    public Document(String name, Date deadLine){
        this.name = name;
        this.deadLine = deadLine;
        this.state = "created";
/*if deadline changes DeadlineControl
has to be somehow notified */
```

```
        this.deadlineControl =
         new DeadlineControl(deadLine);
    }
/* This method has to check itself the state
of DeadlineControl */
   public void submitDoc(){
      if(deadlineControl.getState().
        compareTo("not expired") == 0) {
           this.setState("submitted");
        } else {
          this.setState("cancelled");
        }
   }
}

public class DeadlineControl extends Behaviour {

/* Date needs to be passed to DeadlineControl */
    private Date deadline;
    DeadlineControl(Date deadLine) {
        this.deadline = deadLine;
    }
    @Override
    public String getState() {
       Date currentDate = new Date();
       return
        currentDate.compareTo(deadline) > 0
                 ? "expired" : "not expired";
    }
}
```

Such an implementation is quite traditional and completely within the scope of plain Java. However, its limitations are obvious:

- The communication of objects is not event-driven.
- The dependency of modules is bi-directional. The life cycle module *Document* is not oblivious about the functionality of the decision module *DeadlineControl* because it has to
  - specify *DeadlineControl* as its object field and
  - explicitly invoke the *deadlineControl.getState()* method.
  - even the state of the decision module *DeadlineControl* “not expired” is used in the code of the life cycle module *Document.*
- The implementation of the decision module is also dependent, because it has to know the exact name, the type, and the value of a constrained attribute (e.g. *Date deadline.*)

Consequently, changing (for example, changing the module name or the state name "not expired") or adding new functionality within decision modules would require refactoring and subsequent regression testing of all affected life cycle modules. The limitations above make such decision modules not generic enough to be used to implement shared behaviours among different life cycle objects.

7.2 Using Publisher-Subscriber Design Pattern and Java Reflection

Further generalization can be done using Java reflection and the publisher-subscriber design pattern. Java reflection makes it possible to retrieve the name of a field of a known type to the decision module. Using publisher-subscriber design pattern, we can implement event-driven mechanism, which is in the core of the Protocol Modeling approach.

Here we implemented the generic functionality of the behaviour protocol machines (section 6) in the parent class *Behaviour*. In particular, *Behaviour* implements the reflection on all allowed generic data types. The *Behaviour* class can be put in a separate Java package among other application independent elements of Protocol Modeling such as *Object*, *State*, *Attribute*, or *Event*. This package - we will further refer to it as a *"behaviour engine"* - should also include generic Protocol Modelling mechanisms such as object instantiation and the CSP composition mechanism. We will introduce the latter in the following subsections.

Listing 8 shows the *Document* class, which now implements interface *SubmitDocEventListener* within the publisher-subscriber design pattern.

DeadlineControl has now a new attribute *deadlineAttribute*, which is used to invoke the name of the checked attribute *deadline* of the class *Document* via Java reflection inside the *getDate()* method. This method is defined in the parent class *Behaviour*.

Listing 8. Implementation using publisher-subscriber design pattern and Java Reflection

```
class Behaviour {
 /*
 ...
 The rest of BEHAVIOUR functionality
...
 */
    public Date getDate(String dateFieldName){
    //Reflection to get access to the value
    //of dateFieldName of type Date
         Field field;
         field =
            this.getClass().
             getDeclaredField(dateFieldName);
         field.setAccessible(true);
```

```
        return (Date) field.get(this);
    }
}


public class Document extends Behaviour
    implements SubmitDocEventListener {

    private String name;
    private Date deadLine;
/*    'INCLUDES' in the model is implemented
as object composition */
    private DeadlineControl deadlineControl;

    public Document(String name, Date deadLine){
        this.name = name;
        this.deadLine = deadLine;
        setState(State.NEW);
        //passes the deadline attribute
        this.deadlineControl =
          new DeadlineControl("deadLine");
    }
/* Implementation of listener method
from SubmitDocEventListener interface */
    @Override
    public void submitDocEventReceived(){
       if (deadlineControl.getState(this) ==
           DocManState.NOT_EXPIRED) {
          this.setState(DocManState.SUBMITTED);
       }
    }

}
public class DeadlineControl extends Behaviour{
   private String deadlineAttribute;

    DeadlineControl(String deadline) {
        this.deadlineAttribute = deadline;
    }

       public DocManState getState(Behaviour inst){
       Date expDate =
         inst.getDate(this.deadlineAttribute);
       Date currentDate = new Date();
       return
```

```
            currentDate.compareTo(expDate) > 0
             ? DocManState.EXPIRED : DocManState.NOT_EXPIRED;
    }
}
```

In the enhanced code above we also make use of the class *DocManState*, which specialises the application dependent functionality of the behaviour engine's generic class *State* and contains the enumeration of all possible states of the objects in the *Document Manager* model.

The publisher-subscriber design pattern implements the Protocol Modelling event-based communication of modules. Java reflection allows reducing the dependency of the decision module on a particular life cycle module.

Still, OBJECT *Document* has to be aware of the functionality of the BEHAVIOUR *DeadlineControl* as it has to invoke it inside the event handler *submitDocEventReceived()*. The dependency of modules is bi-directional.

7.3 Using Interceptors within Enterprise Java Beans Framework

A decision module can be seen as a managerial concern or a control concern. In mainstream languages, concerns are often implemented using the aspect mechanism.

We intend to investigate if the aspect mechanisms in Java can support implementation of decision modules that have only unidirectional dependency with life cycle modules, that is, the decision modules can read the state of the life cycle modules and permit or forbid proceeding of events while the life cycles are oblivious to decision modules.

The standard Java currently has only one aspect mechanism implemented in the Java Enterprise Edition (Java EE [6]), which supports Enterprise Java Beans 3 (EJB3) specification. EJB3 supports special objects called interceptors, which have the "around invoke" aspect semantics. Interceptors are invoked by the Java EE container run by an application server. Each EJB may have a set of "business methods" which can be surrounded by additional functionally provided by a decision module via container. The decision module is implemented as an interceptor. The container is instructed by an EJB3 annotation *@Interceptors* to call an interceptor before the invocation of a business method of a bean.

In the Listing 9, the life cycle module is implemented as class *Document*. It is a "stateless" bean [6], which the corresponding annotation *@Stateless* declares. The only thing the code developer has to do with the life cycle module is to choose the business method (or methods) that should be intercepted and annotate this business method with the *@Interceptors(DeadlineControlInterceptor.class)* annotation. In our case, this is the *SubmitDoc()* method. This annotation informs the application server that before submitting the document the corresponding deadline control interceptor has to be invoked.

Listing 9. Implementation of OBJECT type Document as a stateless bean using EJB3 specification

```
@Stateless
public class Document implements DocumentRemote {

   private String name;
   private static Date deadLine;
   private String state;

   public Document() {
      this.state = "@new";
        /*..*/
   }

   @Interceptors(DeadlineControlInterceptor.class)
   @Override
   public void submitDoc() {
      this.state = DocManState.SUBMITTED;
   }
}
```

Interceptor *DeadlineControlInterceptor* (Listing 10) is a Java class. It has one special method annotated as *@AroundInvoke*. Via its only parameter *InvocationContext*, it has access to the life cycle module's instance. The Java reflection mechanism provides access to the *deadLine* attribute of the *Document* object.

Listing 10. Implementation of DeadlineControl as an interceptor using EJB3 specification

```
class DeadlineControlInterceptor {

    @AroundInvoke
    public Object getState(InvocationContext ic)
    throws Exception {
         Date currentDate = new Date();
/* Using InvocationContext to get the object
   and reflection to get the value of its
  "deadLine" attribute */
        Field fld = ic.getMethod().
         getDeclaringClass().
            getDeclaredField("deadLine");
        fld.setAccessible(true);
        Date dt = new Date();
        Date expDate = (Date) fld.get(dt);
        if (currentDate.compareTo(expDate) > 0) {
           return null; //Method submitDoc() is not called
```

```
      } else {
         return ic.proceed();
      }
   }
}
```

Using *InvacationContext*, the decision module *DeadlineControlInterceptor* obtains the name of the attribute *"deadLine"* to read its value from the life cycle module. The value of the *"deadLine"* is assigned to the *expDate (expiration Date)* and compared with the current date.

The implementation above is generic enough as it allows using the same decision module among multiple life cycle modules. The only restriction remains that the name of the constrained attribute "deadLine" has to be the same among all of them. The unidirectional dependency is achieved using the interceptor mechanism supported by the application server. The event-based communication and composition is also used (although not shown in the listings above).

7.4 Using Enterprise Java Beans Framework and Decorator Design Pattern

One may argue that using reflection is not safe and should be avoided whenever it's possible. In some cases, life cycle modules to be extended by decision modules may have the same external behaviour, e.g. *Document* and *Part* in our running case study. In such a case, decision modules may be implemented as wrappers to life cycle modules using the Decorator design pattern. In the EJB3 specification this pattern is supported as well. Decorators implement a mechanism that is close to interceptors. They add functionality to the decorated classes. However, instead of implementing cross-cutting concerns useful for different class types, they extend the behaviour of a class implementing a certain interface.

In the following Listing 11 the deadline control functionality is implemented as a decorator class *DocumentDeadlineControlDecorator*.

Listing 11. Implementation of DeadlineControl using Delegation within EJB3 specification

```
@Decorator
public abstract class
   DocumentDeadlineControlDecorator
      implements DocumentRemote {

   @Inject
   @Delegate
   DocumentRemote doc;

   @Override
   public void submitDoc() {
        Date currentDate = new Date();
            if(currentDate.
                compareTo(doc.getDeadLine()) >0){
                System.err.println("Expired");
            } else {
                doc.submitDoc();
            }
    }
}
```

The code shows the *Document* or *Part* class injected via their common interface *DocumentRemote*. The *@Inject* annotation uses the dependency injection mechanism [6] to give the decorator access to the decorated class. The *@Delegate* annotation gives the container access to all exposed methods of all the classes implementing the *DocumentRemote* interface. In our example, the call of the *submitDoc()* method of *Document* happens only if the deadline is not expired. As one can see, the techniques based on the dependency injection mechanism provide the implementation means to produce the decision modules with all the desired properties: modularity, unidirectional dependency with other modules, event-based communication and composition of modules.

The disadvantage of the decision modules' implementation approach using EJB3 is obvious: it's too heavy. The overhead of running the application server just for the sake of support of decision modules is not sufficiently justified. However, if the system is already implemented as an enterprise application, this may be a viable solution. EJB3 is supported by a large variety of certified application servers [25], both open source and proprietary. In order to completely avoid a vendor lock the EJB3 platform may be substituted by a platform independent solution, for example, the Spring framework [27]. It has an additional benefit, as it supports the AspectJ [5] specification, which implements the aspect paradigm much more thoroughly than EJB3 does. We didn't experiment with Spring, but a code snippet like the one below can be already envisioned (Listing 12).

Listing 12. Implementation of DeadlineControl as an aspect using Spring framework specification

```
@Aspect
public class DeadlineControlAspect {
/* ... */
   @Around("execution(* documentmanager.submitDoc(..))")
   public void DeadlineControl(JoinPoint joinPoint){

       /* add decision module functionality here */

   }
}
```

Still, Spring is an additional layer on top of an application server. In the following section we show how AspectJ as a special library for plain Java can be used to implement decision modules.

7.5 Using AspectJ

The idea to implement business rules as aspects is not new. The authors of [17, 16, 13] point out that aspects allow the developers to separate the business logic from the application's core functionality encapsulating in aspects both crosscutting features as well as their "connectors" [17] to business objects. This makes it possible to "completely remove the source code pertaining to the business rules" [13] from the business objects. In our terminology this is called unidirectional dependency or obliviousness. In this subsection we show how the entire set of Protocol Modelling properties may be implemented using aspect technology for the plain Java.

We implemented our running example using an AspectJ plugin for Eclipse [5]. In the implementation, we continued to separate the generic Protocol Modelling behaviour from the application (*Document Manager*) specific functionality.

Apart from the *Behaviour* class, already implemented earlier, we added to the generic implementation ("behaviour engine") two public interfaces *LifecycleModule* and *DecisionModule*.

The interface *LifecycleModule* contains the list of its decision modules and the declarations of methods linking a life cycle module to its decision modules. This is the implementation of the protocol model INCLUDES declaration (Listing 2).

The interface *DecisionModule* declares the *decide()* method, which, being implemented, have to return *true*, if the decision module is in the right state allowing proceeding the event specified for this decision module or *false* otherwise.

Further, we implemented the protocol model generic behaviour as an abstract aspect *BehaviourProtocol* (Listing 13).

Listing 13. Implementation of the Protocol Modeling behaviour as an abstract aspect using AspectJ

```
public abstract aspect BehaviourProtocol {
  public abstract pointcut stateChanges(LifecycleModule lc);

  void around(LifecycleModule lc): stateChanges(lc) {
    for (int i = 0; i<lc.getDecisionModules().size(); i++){
          if (!((DecisionModule) lc.getDecisionModules()
                .elementAt(i)).decide(lc)){
                System.out.println(" NO GO!" );
                return;
          }
        }
        proceed(lc);
    }
/* ... */
}
```

The most important part of our "engine" is the AspectJ advice *stateChanges()* with "around invoke" semantics. It iterates through all the life cycle module's (LC) decision modules (DM) using their *decide()* methods. If the state of each decision module permits to proceed, the AspectJ's *proceed()* method is invoked, which allows the corresponding LC module's method to run. As one can see in Listing 13, if several DM's are used for a single LC, the AND semantics is implemented with respect to the DM invocation: each of them has to be in the right state that permits proceeding. If necessary, any other semantics of logical composition of aspects (e.q., OR, XOR) can be realized generically at the implementation level.

Listing 14 shows the application specific part of the AspectJ implementation. There is an aspect implementing the *BehaviourProtocol* abstract aspect. It declares the *Document* class as a LC and the *DeadlineControl* class as its DM and also links them to the aspect. Next, the *DeadlineControl*'s *decide()* method is implemented by the aspect. Finally, the pointcut is created, which links the advice *stateChanges* to the *Document*'s method *submitDocEventAJ()*.

Listing 14. Implementation of the application specific behaviour using AspectJ

```
public aspect BehaviourProtocolDocManImpl
                                extends BehaviourProtocol {
/* Document as a life cycle module */
  declare parents: Document
                                implements LifecycleModule;
  public Object Document.getData() { return this; }
/* Its decision module */
  declare parents: DeadlineControl
                                implements DecisionModule;
/* GO-No GO method for the decision module */
  public boolean DeadlineControl.decide(LifecycleModule lc)
  {
        System.out.println("DeadlineControl␣Works!");
/* Here the DM gets the information about types of the LC
  fields to check and their correct states for GO */
     if (getState((Behaviour) lc.getData()) ==
                                    DocManState.NOT_EXPIRED ){
        return true;
     } else{
        return false;
     }
  }
/* stateChanges is invoked for each target being a life
   cycle module with the signature of the method aspect
   is weaved to */
  public pointcut stateChanges(LifecycleModule lc):
     target(lc) &&  call(public void submitDocEventAJ())
}
```

Listing 15 shows the AspectJ implementation of the *Document* class. As one can see, this is a "purely oblivious" implementation without any knowledge about the decision module. The decision module can be implemented the same way as shown in Listing 8. To obtain its current state, it still needs Java reflection and the knowledge about the LC's field type. However, as we pointed out before (subsection 7.2), it is possible to implement any type-related functionality, which uses reflection, as a member method of the generic class *Behaviour*.

Listing 15. Document class in the AspectJ implementation

```
public class Document extends Behaviuor
    private String name;
    private Date deadLine;
    public Document(String name, Date deadLine) {
        this.name = name;
        this.deadLine = deadLine;
        setState(State.NEW);
    }
    public void submitDocEventAJ() {
            this.setState(DocManState.SUBMITTED);
    }
/* ... */
    }
}
```

To support modularity, the AspectJ implementation clearly separates the core and business rule logics between LC and DM implementation classes. The former are completely oblivious to the latter. The even-driven mechanism can be easily implemented using the publisher-subscriber pattern as we have shown before in subsection 7.2. Moreover, the generic Protocol Modelling behaviour can be further separated from the application specific one. This way, the usability of the approach can be facilitated.

7.6 Using Mixins

Another promising approach would be to program with mixins [18]. When a class implements a mixin, it implements an interface extended with this mixin and, this way, implements mixin's attributes and methods. This description is very close to the functionality of *BEHAVIOUR* in protocol models. Unfortunately, direct realisation of mixins in mainstream languages is largely absent, at least without making use of special or not well known libraries. Despite some anecdotal claims, the support of mixins in Java is hardly expected in foreseeable future as well. The newly released Java 8 SE specification [11] does not support them either.

A partial solution could be to use newly introduces in Java 8 SE [11] so called "virtual extension methods", which simply allow one to add default method implementations to the interface not changing the implementation classes. Whether or not such a feature could be sufficient enough for decision module implementations needs further experiments. In our future study we intend to investigate as to how some known ad-hoc approaches [14] can be used to program decision modules with mixins.

7.7 Properties of Decision Modules in Java Implementations

Technique	Modularity	Unidirectional dependency	Mechanism: Event-Driven (CSP||)
Object Composition	yes	no	no
Publ.-Subscr.& Java Reflection	yes	partially, state reading:yes obliviousness:no	yes
EJB 3 with Interceptors	yes	yes	yes
EJB 3 with Delegation	yes	yes, for a given interface	yes
Aspects with AspectJ	yes	yes	yes

Table 1. Protocol Modelling decision modules properties in different Java implementations

Table 1 summarises the implementation examples above with respect to their adherence to the properties of decision modules in Protocol Modelling as they described in subsection 6.3. The table shows that the aspect-oriented implementation techniques provide full support for the modularization with decision modules.

8 Discussion

8.1 Decision Abstractions in Behaviour Modelling Practice

Decision modules are the abstractions of the business system modelling domain.

The practice shows that the most changeable parts of functionality of information and case management systems are the parts that present the rules of handling the cases and information. The rules of handling insurance claims change every year. The rules of proposal selection for funding and crediting are constantly modified following the changes in the economic situation. The rules of dealing with private health information in patient files, the rules of using the information on the web, all undergo changes. Localization of such rules in separate modules and the ability to modify those modules without changes in other modules are the desired requirements supporting changeability of systems during their evolution. Complex automated control systems may experience less changes but they still need to overcome modernization. The localization of decision points in such a way that their changes does not change the rest of the system can simplify regression testing of system modifications.

The need of localising decisions seems being realised in the draft version of the new Case Management Model and Notation (CMMN) [24] standard developed by OMG. The standard includes a decision module called *Sentry*. "Sentry watches out for important situations to occur (or events), which influence the further proceedings of a Case .... A Sentry is a combination of an event and/or condition. When the event occurs, a condition might be applied to evaluate whether the event has effect or not" [24](p. 23)

Thus, on the one hand, the need of decision points as separate entities has been recognised by the OMG community. On the other hand, the Protocol Modelling semantics forms a solid basis for building executable models with decision modules. There is also the ModelScope tool [21] that supports the execution of protocol models. This is a good starting point. In this paper we have shown the possibilities of implementation of decision models with the same properties as in protocol models.

However, there are barriers to widespread adoption of the new protocol modelling style.

- *Unawareness.* The first barrier is unawareness of the modelling community about this style of modelling. The CMMN standard is young and still under development. The description of the standard does not exactly follow the protocol modelling semantics. The mutual adaptation of the Protocol Modelling and the CMMN standard would contribute to the awareness about decision modules.
- *A small number of success stories.* For the moment, success stories have been collected at the official web page of the company called Metamaxim [21]. A systematic application was fulfilled for the use case of basic insurance for Oracle Nederland [28]. In order to convince businesses to use the new modelling style, more success stories have to be collected and distributed to the modelling community and businesses.

– *Legacy models.* There are many legacy models in the key business areas like banking and government services. These models are built using the traditional process modelling style. To adopt a new modelling style, companies need a very good motivation for new investments in those models. Modelling does not give direct return on investments. Better understanding and reasoning are not tangible enough. The advantages of the new modelling style need further valorisation.

8.2 Decision Modules in Implementations. The Way to Go?

The bottleneck in system development is often the implementation of changes. For example, the maintenance teams of insurance applications experience stress every year as the new rules of claim handling are accepted by the governments, say, in November-December. The changes often need to be implemented and work perfectly from the first day of January. Most of the time, the modifications concern decision rules. If the implementation follows the new modelling style and supports local modifications which guarantee that the not-modified parts preserve their behaviour, the time for implementation and testing activities will be shortened. This evidence will be the best argument in favour of the new modelling and implementation style. We see the best way to obtain such an evidence in refactoring an existing application in some traditional case management domain. To do so, we need a proper tool support for different stages of the software development process.

– *Refactoring of conventional process models.* The first group of tools should be able to refactor or transform the conventional process models into protocol models with separated decision models. These tools should help to overcome the barrier of legacy systems and also increase the awareness in the new style of modelling and the properties of models with separated decision modules.
– *An open source execution tool for CMMN models with Protocol Modelling semantics.* The most convincing way of promoting the new approach is to let the user to play with it. Even more importantly, executable Protocol Modelling would allow the developers to get insights into the model at early stages of the development and avoid more costly mistakes at the later stages.
– *Implementation libraries and plugins.* The Protocol Modelling implementation style has to be accompanied by open source libraries which should implement its generic functionality. In our experiments with Java and AspectJ we have found that this generic part can be factored out from the application specific behaviour into a separate package. Further, using the plugin mechanism, a "behaviuor engine" package and, thus, the decision module-based approach can be made a part of the development environments such as Eclipse or NetBeans.

In our future work we intend to follow the directions indicated in the above list.

9 Conclusion

In this paper we have defined decision modules and investigated the possibilities of Java implementation of decision modules identified in requirements and modularized in protocol models.

Decision modules separate a specification of the state of a non-empty set of system objects in the form of a calculated state allowing or forbidding a set of system actions. The calculated states serve as conditions for making decisions about the possible system actions. Separation of such modules facilitates requirements traceability, test generation and modification of models and implementations.

We have shown a possible way to separate decision models in declarative models, executable protocol models and Java programs.

To answer our research question, we conclude that it is indeed possible to implement functionality of decision modules using such mainstream object-oriented language techniques as EJB3 and AspectJ so that such implementations would have the same properties as the decision modules in executable protocol models. These techniques support all the decision modules' properties and provide the means to make generic implementations. The usability of such a solution depends on the development of necessary libraries and plugins. In this paper we have collected the experience for creating a "behaviour composition engine" for the implementation of applications with life-cycle and decision modules.

References

1. B. Halle von. *Business Rules Applied.* Wiley, 2001.
2. G. Bracha and W. Cook. Mixin-based inheritance. *OOPSLA/ECOOP '90 Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, 1990.
3. Business Rules Group. Defining Business Rules. What Are They Really? *http://www.businessrulesgroup.org/firstpaper/BRG-whatisBR_3ed.pdf*, 2000.
4. C.J. Date. *What not How: The Business Rules Approach to Application Development.* Addison-Wesley, 2000.
5. Eclipse. AspectJ project. http://projects.eclipse.org/projects/tools.aspectj.
6. *EJB 3.2 Expert Group. JSR-318 Enterprise JavaBeans, Version 3.2*, 2013.
7. R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development.* Addison-Wesley, 2004.
8. C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
9. IDC. IDC survey, http://ceiton.com/CMS/EN/workflow/introduction.html#Customization, 2007.
10. J. Taylor and N. Raden. *Smart (Enough) Systems* . Prentice Hall, 2007.
11. *JSR-000337 Java SE 8 Release* , 2014.
12. K. Jensen. *Coloured Petri Nets.* Springer, 1997.
13. A. Kellens, K. D. Schutter, T. D'Hondt, V. Jonckers, and H. Doggen. Experiences in modularizing business rules into aspects. In *ICSM'08*, pages 448–451, 2008.
14. Kerflyn's Blog. Java 8: Now You Have Mixins? http://kerflyn.wordpress.com/-2012/07/09/java-8-now-you-have-mixins/.

15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1241:220–242, 1997.
16. M. A. Cibrán, M. D'Hondt. Composable and reusable business rules using AspectJ. In *In Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on AOSD. Boston, USA* , 2003.
17. M. A. Cibrán, M. D'Hondt and V. Jonckers. Aspect-Oriented Programming for Connecting Business Rules. In *6th Proceedings of International Conf. on Business Inforamtioon Systems,Colorado Springs, USA* , 2003.
18. M. Flatt, S.Krishnamurthi, M. Felleisen. A Programmers Reduction Semantics for Classes and Mixins. In *In Formal Syntax and Semantics of Java. Lecture Notes in Computer Science. Volume 1523*, pages 241–269. Springer, 1999.
19. A. McNeile and E. Roubtsova. CSP parallel composition of aspect models. *AOM'08*, pages 13–18, 2008.
20. A. McNeile and E. Roubtsova. Aspect-Oriented Development Using Protocol Modeling. *LNCS 6210*, pages 115–150, 2010.
21. A. McNeile and N. Simons. http://www.metamaxim.com/.
22. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.
23. OMG. *Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03.* 2003.
24. OMG. *Case Management Model and Notation. Version 1.0, formal/2014-05-05.* 2014.
25. Oracle. JavaEE Compatibility. http://www.oracle.com/technetwork/java/-javaee/overview/compatibility-jsp-136984.html/.
26. E. Roubtsova and S. Roubtsov. A Test Generator for Model-Based Testing. *Proceedings of the Fourth International Symposium on Business Modeling and Software Design, BMSD 2014, 24-26 June, 2014, Luxembourg.*, 2014.
27. Spring. Spring Framework. http://projects.spring.io/spring-framework/.
28. J. Verheul and E. Roubtsova. An Executable and Changeable Reference Model for the Health Insurance Industry. *The 3rd International Workshop on Behavioural Modelling - Foundations and Application. BM-FA 2011, Birmingham, UK. ACM DL*, pages 33–40, 2011.