Implementation of Decision Modules

Serguei Roubtsov
Technical University Eindhoven, the Netherlands
s.roubtsov@tue.nl

Ella Roubtsova
Open University of the Netherlands
ella.roubtsova@ou.nl

ABSTRACT

Separation of concerns can have different forms. The well accepted concern is an object behavior usually specified as a life cycle module. In this paper, we define another type of concern, a decision module, and identify it in requirements and models. Separation of decision modules in programs may improve traceability of requirements and simplify code analysis. We present the results of our experiments with implementation of decision modules. We implement the modules using object composition, reflection, the publisher-subscriber design pattern and aspects. We present the possibilities of different implementation forms and illustrate our observations of pros and cons with an example of a document submission system.

Categories and Subject Descriptors

D.2 [**SOFTWARE ENGINEERING**]; D.2.2 [**Design Tools and Techniques**]; D.2.3 [**Coding Tools and Techniques**]; D.3.3 [**Language Constructs and Features**]

General Terms

Modules and Interfaces, State Diagrams, Object-oriented programming, Language Constructs and Features

Keywords

Decision Module, Requirement, Model, Program, Protocol Contracts

1. INTRODUCTION

Modules are useful instruments for handling complexity of software systems. Modules are created for various purposes depending on which different approaches to modularisation exist. Among the goals of modularization are traceability of requirements and ease of code modification, reuse, and testing.

The most commonly used modularization of object life cycles hides the process control points, i.e. the decisions on which path to follow, inside the objects. To be able to modify a program or generate a test, the control points inside life cycle modules have to be analysed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
BM-FA '14, July 22 2014, York, United Kingdom
Copyright 2014 ACM 978-1-4503-2791-6/14/07 ...$15.00.
http://dx.doi.org/10.1145/2630768.2630774.

In this paper, we define the decision modules and separate them from the life cycle behaviour of objects. Section 2 defines a decision module. Section 3 presents the approaches that separate the modules similar to the decision modules.

The next sections of the paper contribute to the definition of a decision module from the perspective of a phase of system development.

Section 4 presents the 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 [14], explains the composition of decision modules and life-cycle modules and recognizes the attractive properties of modules in protocol modelling. These properties facilitate model changing and testing. The same properties are decried in the implemented systems. Therefore, we formulate our research question: *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?*

Section 7 applies different software development techniques (object composition, reflection, the publisher-subscriber design pattern and aspects) for implementation of decision modules and estimates whether the implementation preserves the desired properties of protocol models.

Section 8 concludes the paper and draws perspectives for the future work.

2. DEFINITION OF DECISION MODULE

- *We define a decision module as an abstract description of the system actions and the states before and after these actions allowing or forbidding these actions action or allowing a choice between several actions.*
- *We separate a decision module as a module because it can be associated with different objects as a separate entity.*
- *We name this module a decision module because it forms the condition for the acceptance or refusion of an action. The condition is derived from the pre- and post-states of the action in the life cycle modules.*

3. RELATED WORK

Modules similar to our decision modules have been also recognized in Business Rules. The modules are called *enablers* [2].

"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 [2] and often called an integrity constraint, a condition or a test.

The enablers represent only a subset of our decision modules because the decision modules 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 [12]. Protocol modeling makes use of Communicating Sequential Processes (CSP) [6] parallel composition of modules which possess internal data. The CSP parallel composition produces observationally consistent models. This means that the protocol model allows one to add and delete modules as the behaviour of modules is preserved in the the whole behaviour [11]. The modules separated in protocol models possess the following properties:

- able to read (but not change) attributes, an event prestate of other modules, and predict the event poststate for the given event [14],
- able to be composed with different life cycle modules (objects) in such a way that the life cycle modules do not know about the decision modules (remain oblivious) and, thus, should not be changed as the latter are added or changed [11; 12].

The decision modules are easily separated in protocol models as protocol machines with derived states. However, the protocol machines with derived states cover not only the decision modules. They possess the expressive pover for separation of a wider class of modules.

4. CASE STUDY: PREPARATION OF A DOCUMENT BY SEVERAL PARTICIPANTS

We illustrate the proposed modularization with a case study. It demonstrates how the declarations of the decision modules can be transformed into modules of executable synchronous protocol models. The goal of the case is to show the advantages of decision modules for model changes.

Let us consider a system that controls a joint 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. A 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 request 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.

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 (Figure 1). A snapshot is a visible abstract state captured after or before an event.

For example, the declaration *"A document can be submitted before the deadline"* can be presented as a decision module *DeadlineControl*. It shows that the event *Submit Document* can only take place if the deadline is not expired.

Figure 1 presents the decision module *DeadlineControl* and other modules taken from the case description. We depict an abstract state as a double line oval. An oval may have an ingoing or outgoing arc ladled with an action that can happen. If an action can happen only if it results in described state, then the arc is ingoing. The example is the *DeadlineControl*. If an action can happen only in the described state, then the arc is outgoing. The example is the constraint *Document Submittable: If all parts are ready, the document can be submitted.*

The state descriptions in decision modules are abstracted from the life cycle of entities and agents 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, the progress constraint `Act` presents a possibility to progress by creating a new participant (event *Create Participant*) who can write the part (event *Assign Part*).

The elements of the life cycle of entities and agents (objects) in the model are also presented in requirements as declarations of decision modules. For example, we can read in requirements what an instance of the *Coordinator* can do when it is created (state "*Created*"). It can *Create Document*, *Submit Document*, *Create Participant*, *Create Part*, *Assign Part* and *Cancel Document.*

The declarative specifications are not executable. However, there is a way to preserve decision modules as modules of executable models. We show this way of modularization in the next sections.

6. DECISION MODULES IN PROTOCOL MODELS

6.1 Protocol Model

The building blocks of a Protocol Model [14] are protocol machines and events. They are instances of, correspondingly, *protocol machine types* and *event types.*

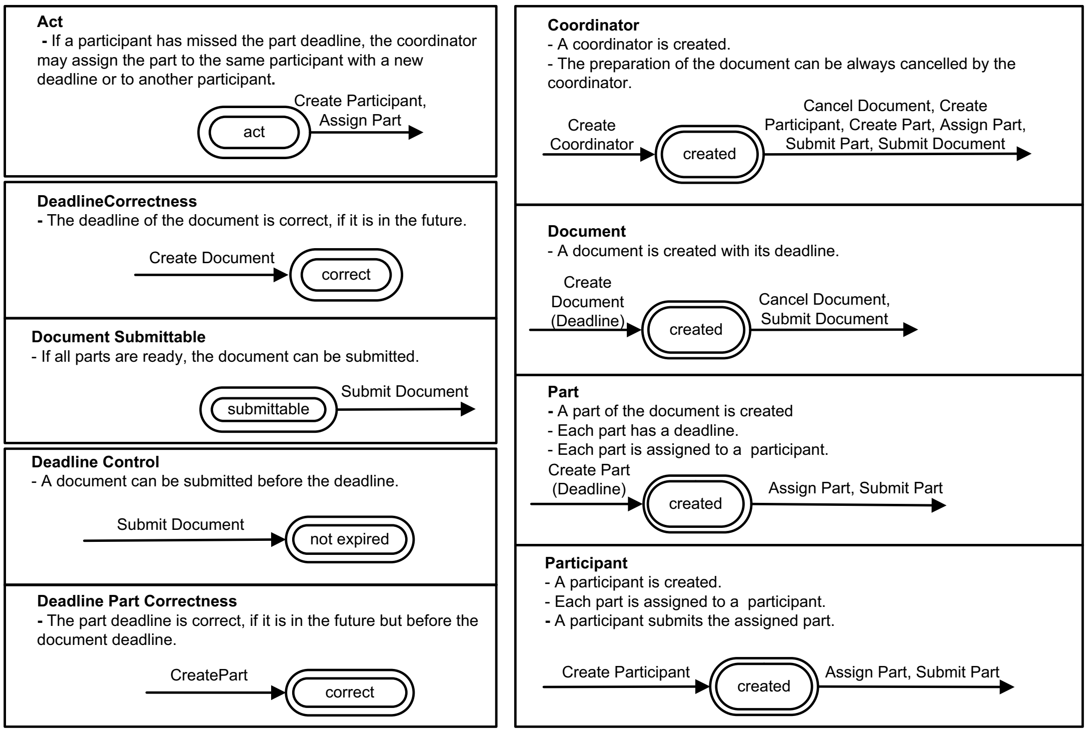

Act
- If a participant has missed the part deadline, the coordinator may assign the part to the same participant with a new deadline or to another participant.
act
Create Participant, Assign Part
DeadlineCorrectness
- The deadline of the document is correct, if it is in the future.
Create Document
correct
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.
CreatePart
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

Figure 1: Declarative specification

A protocol machine type is an LTS (Labelled Transition System) extended to enable modelling with data:

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

- s_i^0 is the initial state;
- S_i is a non-empty finite set of states.
- E_i is a finite set of recognized event types e_i, coming from the environment.
- A_i is a finite set of attributes of different types. The set 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 for updating the values of the attributes, states and events of the protocol machines of the protocol model. $PM_1, ..., PM_n$ are the protocol machines of the protocol model. $E_1, ..., E_m$ are events of the protocol model.
- $T_i \subseteq S_i \times E_i \times S_i$ 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 may be updated without callback functions. The values of the attributes, states and events may be updated using the callback function only as a result of a transition, i.e., as a result of event acceptance.

In order to facilitate reuse, protocol machines come in two variants: 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 [1; 12].

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

- A^e is a finite 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 event calculates attributes of generates other events from the state of the model.

Within the Protocol Modelling, callback functions are the instrument for data handling. In the ModelScope tool [13] supporting execution of protocol models, the callbacks are coded as small 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.

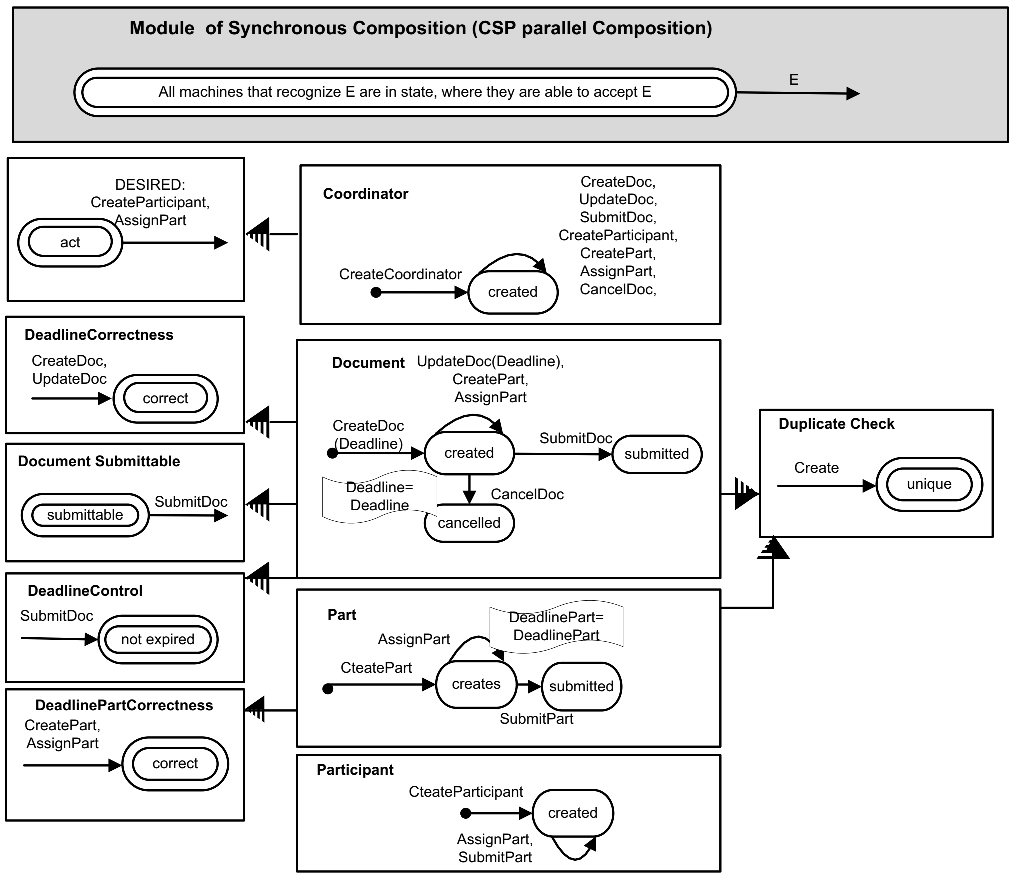
CSP parallel composition. In any state, a system model PM is a CSP parallel composition of finite set of *instances of protocol machines.*

$$PM = \underset{i=1}{\overset{n}{\|}} PM_i = \quad (s_0, S, E, A, CB, T), \quad n \in N.$$

A Protocol Model PM is also a protocol machine, the set of states of which is the Cartesian product of states of all composed protocol machines [14]:

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

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



EVENT CreateCoordinator
ATTRIBUTES
NameCoordinator:String,
Coordinator:Coordinator

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

EVENT CreateDoc
ATTRIBUTES
Document:Document,
Name:String, Deadline:Date,
Coordinator:Coordinator
EVENT UpdateDoc
ATTRIBUTES
Document:Document,
Name:String,
Deadline:Date,
Coordinator:Coordinato
EVENT SubmitDoc
ATTRIBUTES
Document: Document,
Coordinator:Coordinator
EVENT CancelDoc
ATTRIBUTES
Document:Document,
Coordinator:Coordinator

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

GENERIC Create
MATCHES
CreateDoc,
CreatePart[Document

GENERIC E
MATCHES
all Events

Figure 2: Executable Protocol Model

$E = \bigcup_{i=1}^{n} E_i$ 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.

Dependent Protocol Machines. Derived states. Usually transitions T_i of a protocol machine PM_i enable updates of its own states of the state set S_i. On the other hand, protocol machines can read the states of other protocol machines, although cannot change them. This property makes possible dependency of protocol machines. The dependency means that one protocol machine needs to read the state of another machine to calculate its own state and/or the attributes. Such calculated states are called *derived states*, which distinguishes them from the protocol machine states denoted in the model, which are called *stored states* [14]. Callback functions CB_i are used to update attributes and calculate derived states.

The ability of protocol machines to read the state of other protocol machines is an asset for separation of decision modules. Decision modules read the information of other modules and use it to make a decision according to the rules coded in the module.

Two possibilities are used in dependent machines:
(1) The pre-state of a transition can be calculated. The pre-state is similar to guards calculated in CPN [8] and the UML state machines [15].
(2) The post-state of a transition can be calculated and used to allow or refuse the event. This semantics does not exist in the UML and the CPN.

6.2 Protocol Model with Decision Modules in the case study

The protocol model of our case study is shown in Figure 2. The protocol model is executed in the ModelScope tool [13].

A protocol model defines a set of classes of objects. For example, *Document*, *Coordinator* etc. A finite set of *EVENTS* is defined for a protocol model. An event is described as a set of attributes of different data types including the classes of the model. For example, the event type *CreateDoc* is a tuple of variables of types *Document, Coordinator, String* and *Date*. An instance of the event will contain values of those types (Listing 1).

Listing 1: EVENT CreateDoc
```
EVENT CreateDoc
ATTRIBUTES
    Document:Document,
    Name:String,
    Deadline:Date,
    Coordinator:Coordinator
```

The life cycle of any object in the protocol model are described as a labeled state transition system.

A life cycle module presents an object and is described with the finite sets *STATES, ATTRIBUTES* including *NAME* and *TRANSITIONS*.

A transition is triple $(state_1, event, state_2)$, where $state_1, state_2 \in STATES$ and the $event \in EVENTS$.

If an object is created then the $state_i = @new$.

Listing 2 shows the labelled transition system of the object *Document.*

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
```

Protocol Modelling provides an advanced semantics to separate decision modules as it permits not only to derive a state from the pre-states of a transition in life-cycle modules, but also to predict the post-state of the transition and use it to derive the state of the decision module.

In order to establish the functional relations between the states of objects and the abstract states of decision modules, the decision module is described as a labelled transition system with callbacks.

For example, the decision module *DeadlineControl* contains the transition $@any * SubmitDoc = notexpired$. State $@any$ literally means any possible combination of the states of the life cycle modules in the model. State *not_expired* presents the post state of the transition caused by the event *SubmitDoc*. It extends the state space of the model. It should be calculated using the *Deadline* attribute of the *Document* in question and the current date.

The relation between the *DeadlineControl* and *Document* is specified with the *INCLUDE* sentence in the *Document*. The functional dependency of the attribute *Deadline* of the document is defined in the java class *DeadlineControl* shown below 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* in a 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.

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";
  }
}
```

6.3 Properties of Decision Modules in Protocol Models

Analysing the decision modules rendered in Protocol Modelling we can name the following *properties of decision modules in Protocol Modelling*:

- *Modularity:* a decision module modularizes the decision making rules (separates with the reuse purpose);

  For example, the module *DeadlineControl* can restrict the behaviour of objects *Document* and *Part* in the same way.

- *Unidirect 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). [1]

  For example, *DeadlineControl* reads the value of attribute *Deadline* of the object *Document* and predicts state *expired*, *not expired* after event *SubmitDoc.* The object *Document* remains oblivious.

- *Mechanism to achieve the properties* is the *event-driven design with CSP parallel composition.*

  Decision modules are incorporated into the whole 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 easy adding and deleting decision modules. The CSP composition rules are:

  - If a event is not recognised by the protocol model, it is ignored.

  - If an event is recognised by the protocol model and all protocol machines are able to accept this event, 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.

Executable protocol models enable separation of decision modules defined in requirements. This makes requirements 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 assigned, one part has been submitted and another part has not been submitted;
  (2) a document has been created, the parts have 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 implementation promises to provide 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 in Java.

7. DECISION MODULES IN JAVA

The implementation of decision modules using main-stream programming languages is the question that needs investigation. For the best of our knowledge, there are no systematic implementation approachers for separation of enablers. We expect that the aspect-oriented languages [10; 17] and mixin-based languages [1] may contain means for implementation of enables and decision modules. However, first, we would like to investigate the implementation in main stream programming environments.

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?

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 using specialised libraries and frameworks usually makes the implementation less generic with respect to, for

[1] *Obliviousness:* the protocol machine into which the decision module is included does not aware of the behaviour of the decision module. [5]

example, underlying architecture. It can also make the solution platform- and vendor-specific violating a well known Java principle ”write once, run everywhere”.

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 5 both OBJECT *Document* and BEHAVIOUR *DeadlineControl* are shown as Java classes, the former includes the latter as an instance variable.

Listing 5: 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 expDate = this.getDate("deadline");
       Date currentDate = new Date();
       return
         currentDate.compareTo(expDate) > 0
                   ? "expired" : "not expired";
    }
}
```

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

- 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 *deadlineControl.getState()* method.
- The decision module implementation is also dependen, because it has to know the exact name, the type, and the value of a constrained attribute (e.g., *Date deadline*) Consequently, changing and adding new functionality within decision modules would require refactoring and subsequent regression testing of all affected life cycle modules.
- The communication and composition is not event-driven.

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.

Listing 6 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 implemented in the parent class *Behaviour.*

Listing 6: Implementation using Publisher-Subscriber design pattern and Java Reflection

```
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");
    }
/* Implemetetion of listener method
from SubmitDocEventListener interface */
    @Override
    public void submitDocEventReceived(){
       if (deadlineControl.getState(this) ==
            State.NOT_EXPIRED) {
          this.setState(State.SUBMITTED);
       }
    }

}
public class DeadlineControl extends Behaviour{
   private String deadlineAttribute;

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

```
        public State getState(Behaviour inst){
        Date expDate =
           inst.getDate(this.deadlineAttribute);
        Date currentDate = new Date();
        return
           currentDate.compareTo(expDate) > 0
            ? State.EXPIRED : State.NOT_EXPIRED;
    }
}

 class Behaviour {
 /*...*/
    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);
    }
}
```

In the enhanced code above we also make use of the static class *State* containing the enumeration of all possible states of the objects in the model.

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 event-based communication of modules is implemented. The dependency of modules is bi-directional.

7.3 Using Aspects within Enterprise Java Beans Framework

One way to invoke the decision module almost completely independently from the life cycle object is to use the module as an aspect. The standard Java currently has only one aspect mechanism implemented in the Java Enterprize Edition (Java EE [4]), which supports Enterprise Java Beans 3 (EJB3) specification. EJB3 supports special objects called interceptors, which have 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 subjected to certain additional functionally provided by the container. In our case, it's the interceptor functionality. The container is instructed by means of an EJB3 annotation to call an interceptor before the invocation of a business method of a bean.

In the Listing 7 , class *Document* is a "stateless" bean [4], which the corresponding annotation *@Stateless* declares. The only thing the code developer has to do with the life cycle module is to annotate the business method *SubmitDoc()* with the *@Interceptors* annotation. This annotation informs the application server that before submitting the document the corresponding deadline control has to be invoked.

Listing 7: 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 = State.SUBMITTED;
   }
}
```

Interceptor *DeadlineControlInterceptor* (Listing 8) 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 instance. The Java reflection mechanism provides access to the *deadLine* attribute of the *Document* object.

Listing 8: 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 is not called
       } else {
          return ic.proceed();
       }
    }
}
```

The implementation above is generic enough as it allows using the same decision module among multiple classes of different types. The unidirect dependency and the event-based communication and composition can be implemented. The only restriction remains that the name of the constrained attribute "deadLine" has to be the same among all of them.

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 needed to be extended by decision modules may have the same external behaviour, e.g., *Document* and *Part* in our running example. 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 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 10 the deadline control functionality is implemented as a decorator class *DocumentDeadlineControlDecorator*. It has the *Document* or *Part* class

Technique	Modularity	Unidirect 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

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

injected via their common interface *DocumentRemote*. The *@Inject* annotation uses the dependency injection mechanism [4] 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.

Listing 9: 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();
         }
   }
}
```

7.5 Properties of decision modules in 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.

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, unidirect dependency with other modules and event-based communication and composition of modules.

7.6 Future work: AspectJ and mixins

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 justified enough. 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 [16], 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 [17]. It has an additional benefit, as it supports the AspectJ [3] specification, which implements the aspect paradigm much more thoroughly than EJB3 does. This gives the developer more flexibility in the choice of different types of aspect semantics: before, after or around invoke. We didn't experiment with Spring yet, but a code snippet like the one below can be already envisioned.

Listing 10: Implementation of DeadlineControl as an aspect using Spring framework specification

```
@Aspect
public class DeadlineControlAspect {
/* ... */
   @Before(
   "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. It is the subject of further investigation, whether or not AspectJ as a special library for plain Java can be used to implement decision modules.

Another promising approach would be to program with mixins [18]: "When a class includes a mixin, the class implements the interface and includes, rather than inherits, all the mixin's attributes (fields, properties) and methods. They become part of the class during compilation". 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 obscure 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 [7] does not support them either.

A partial solution could be to use newly introduces in Java 8 SE [7] 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 [9] and special libraries [3] can be used to program decision modules with mixins.

8. CONCLUSION

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

Decision modules separate elements of the control flow allowing their reuse by different life cycle modules. This facilitates requirements traceability, test generation and modification of models.

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 mainstream object-oriented language techniques so that such implementations would have the same properties as the decision modules in executable protocol models. However, we have found that the ways of implementation of decision modules in modern Java are not ideal. Some of them are not complete enough, some lead to not necessarily justifiable overhead. As the functionality of mixins appears to be very close to the one of BEHAVIOUR in protocol models, we consider them as promising candidates to implement decision modules. However, proving this and finding a generic approach to doing it is the subject of further investigation.

References

[1] 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.

[2] Business Rules Group. Defining Business Rules. What Are They Really? *http://www.businessrulesgroup.org/first-paper/BRG-whatisBR_3ed.pdf*, 2000.

[3] Eclipse. ASpectJ project. http://projects.eclipse.org-/projects/tools.aspectj.

[4] *EJB 3.2 Expert Group. JSR-318 Enterprise JavaBeans, Version 3.2*, 2013.

[5] R. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development.* Addison-Wesley, 2004.

[6] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[7] *JSR-000337 Java SE 8 Release* , 2014.

[8] K. Jensen. *Coloured Petri Nets.* Springer, 1997.

[9] Kerflyn's Blog. Java 8: Now You Have Mixins? http://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/.

[10] 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.

[11] A. McNeile and E. Roubtsova. CSP parallel composition of aspect models. *AOM'08*, pages 13–18, 2008.

[12] A. McNeile and E. Roubtsova. Aspect-Oriented Development Using Protocol Modeling. *LNCS 6210*, pages 115–150, 2010.

[13] A. McNeile and N. Simons. http://www.metamaxim.com/.

[14] A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *Software and System Modeling*, 5(1):91–107, 2006.

[15] OMG. *Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03.* 2003.

[16] Oracle. JavaEE Compatibility. http://www.oracle.com/-technetwork/java/javaee/overview/compatibility-jsp-136984.html/.

[17] Spring. Spring Framework. http://projects.spring.io/spring-framework/.

[18] Wikipedia. Mixin. http://en.wikipedia.org/wiki/Mixin.