Jacqueline Arninkhof (inf.administratie@ou.nl)

Chapter 3

EXTREME:
EXecuTable Requirements Engineering, Management, and Evolution

Ella Roubtsova
Open University of The Netherlands, The Netherlands

ABSTRACT

Requirements engineering is a process of constantly changing worlds of intentions, goals, and system models. Conventional semantics for goal specifications is synchronous. Semantics of conventional system modeling techniques is asynchronous. This semantic mismatch complicates requirements engineering. In this chapter, we propose a new method EXTREME that exploits similarities in semantics of goal specification and executable protocol models. In contrast with other executable modelling techniques, the semantics of protocol modelling is based on a data extended form of synchronous CSP-parallel composition. This synchronous composition provides advantages for relating goals and system models, reasoning on models, requirements management, and evolution.

INTRODUCTION

It is ''reasonably well known that requirements will never be totally complete, finished, and finalized as long as a system is in service and must evolve to meet the changing needs of its customers and users" (Firesmith, 2005). However, there is a temporary notion of adequate completeness at some moment in time when the stakeholders are agreed on requirements. Adequate completeness of requirements is needed to estimate the development costs and to avoid incorrect assumptions for implementation decisions.

One of the powerful instruments to get adequately complete requirements is executable system modeling. Psychology studies show that people's thinking is context related (Tversky & Simonson, 1999). For requirements engineering this means that stakeholders can identify missing or tacit requirements at the moment they see the behaviour of the system model. Hence, the executable models offer to stakeholders the contextual basis for identification of incompleteness. The semantics of executable modelling should be consistent with the semantics of goals.

In practice there is a semantic mismatch. The semantics of goals is synchronous. The conventional executable system modeling techniques are asynchronous. Asynchronous execution of the

DOI: 10.4018/978-1-4666-4217-1.ch003

Copyright © 2013, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Jacqueline Arninkhof (inf.administratie@ou.nl)

models gives birth to states that are not expressed by the goals. In such states, stakeholders do not understand the execution of the models and cannot properly evaluate the models and reason on them.

In this chapter we propose a new method EXTREME that exploits similarities in semantics of goal specification and executable protocol models in order to simplify executable requirements engineering, management and evolution. Protocol models use a data extended form of synchronous CSP-parallel composition. The combination of protocol models and goal-oriented approaches semantically coherent, all states can be goal interpreted and this eases reasoning on models in terms of goals, goal refinement and identification of missing requirements.

Before showing the EXTREME method, we first remind elements of goal modelling. Then we remind elements of protocol modeling and show how to create protocol models corresponding to goals. The process is illustrated with a case from the insurance domain. We discuss the semantic elements of Protocol modelling that make it suitable for combination with goal-oriented modeling.

GOAL MODELLING

Goal-Oriented Requirements Engineering (GORE) is a well-established group of approaches (Kavakli, 2002; van Lamsweerde, 2004; Darimont & Lemoine, 2006; Regev & Wegmann, 2012). The aim of a goal-oriented approach is to justify requirements by linking them to higher-level goals.

The notion of a goal is used as a partial description of a *system state* being a result of an execution of the system. The authors of the GORE methods emphasize the similarity between goals, requirements, and concerns and propose to combine them in one tree structure. Goals are refined by requirements and concerns. The goal models are used to keep the business motivation in mind of requirement engineers and to elaborate the strategic goals with requirements and concerns.

An example of a goal tree is shown in Figure 1. The top nodes of Figure 1 present business goals of a simplified system supporting insurance business. The goals are:

- A product is composed.
- A policy is bought by a registered customer.
- A claim of a client with a bought policy is handled.

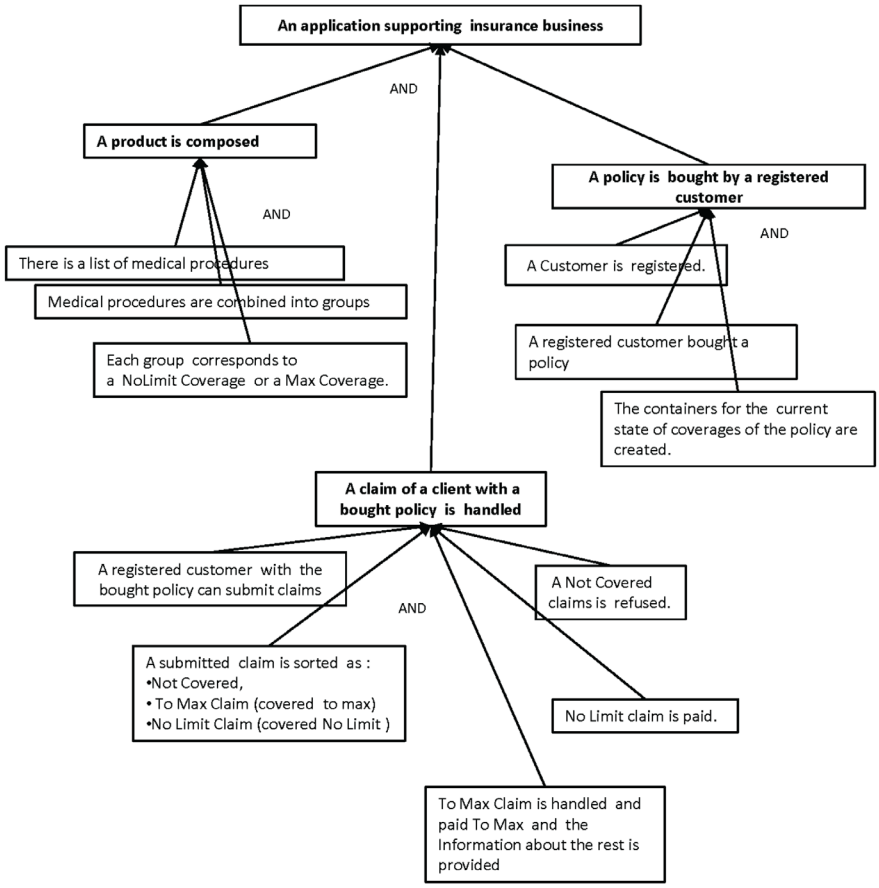
Each parent goal (the one pointed to by the arrow) is refined with a list of sub-goals and requirements. The leaves of the tree present system requirements. Business and strategic goals are expressed using concepts of the stakeholders' vocabulary. Lower-level goals are typically expressed using words from the stakeholders' vocabulary as well as specific technical terms introduced in the model on purpose and where necessary (Respect-IT, 2007).

Identifying goals is not proceeding exclusively from either a top-down or a bottom-up approach. In most cases the two approaches are used at the same time. Refining goals in a goal model often follows a so-called "milestone approach" (although there are many other decomposition approaches). Milestone goals represent goals as intermediate states in a process aimed to achieve the top goals. For example, the goal *"A product is composed"* (Figure 1) is refined by goals "*There is a list of medical procedures", "Medical procedures are combined into groups", "Each group corresponds to a NoLimit(Coverage) or MaxCoverage"*.

GORE trees are also used to relate goals and structural elements of the: Entities, Agents, and Operations. Entities represent passive objects in contrast with Agents that represent active objects. Agents are either human beings or automated components that are responsible for achieving requirements. The goals of this level assigned to the humans are called expectations. Software agents are responsible for requirements. Agents, Entities and their Relations are captured in an

Jacqueline Arninkhof (inf.administratie@ou.nl)

Figure 1. Goal tree of an insurance business



object model. Often goals are assigned to several agents rather than a single one.

In order to achieve goals software agents perform operations. The operation model in GORE sums up all the behaviors that agents need to have to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. Those operations work on objects (entities and agents) described in the object model: they can create objects, provoke object state transitions or trigger other operations through the send and receive events.

GORE operation diagrams are either data flow or control flow diagrams. Data flow and control flow diagrams have asynchronous semantics. This is the place where the operation model introduces states that cannot be related to goals. Even if the operational model is executable, not all of its states can be related to goals. A simple example is asynchronous arriving of two data items to reach a state expressed with a goal. In the asynchronous model the items arrive one after another and produce intermediate state when one item has arrived and the other has not arrived yet. This state cannot be interpreted from the goal perspective and may be seen by stakeholders as an evidence of wrong model behaviour. Letier et al. (2008) note that in order to be semantically equivalent to the synchronous goal models, the operation models need to refer explicitly to timing events. It seems that the object and operation models present the abstraction level that is lower than the

Jacqueline Arninkhof (inf.administratie@ou.nl)

level needed for the requirement analysis by the stakeholders. GORE methods need synchronous compositional executable behavioural models corresponding to goals.

PROTOCOL MODELLING

Protocol models have elements of synchronization needed to present the system behavior at the higher level of abstraction than the operation diagrams. We propose to relate goals to protocol machines instead of class and operation diagrams. Synchronously composed protocol machines form together the protocol model corresponding to goals. In Figure 2 a dashed arrow is drawn from a box presenting a protocol machine to the box presenting the corresponding requirement. In this section, we discuss the elements of protocol models and the advantages of using them in goal–oriented approaches.

Protocol Modelling approach was developed by McNeile and Simons (2006). This approach can be viewed as a combination of object life-cycle modelling and the data-extended synchronous CSP-parallel composition. The initial ideas of this composition technique were borrowed from the process algebra of Communicating Sequential Processes (Hoare, 1985) and then extended in order to enable composition of models with data. In this part we present main elements of Protocol Modelling. We also show that this approach produces the system models, all states of which can be related to goals.

A protocol machine is a state-transition structure with data storage that defines ability of a system to interact with the environment by accepting events from environment or refusing events. A protocol machine can be seen like an object that exists even without its creation in its initial state. An object goes into its active state with a creating event.

For example, the protocol machine *Medical Procedure* defines *the attributes* and *stored states* of the life cycle and transitions of every object of type *Medical Procedure*. Transitions define the interactions with the environment recognized by the object.

```
OBJECT Medical Procedure
  NAME Name
  ATTRIBUTES Name: String, MPGroup:MPGroup
  STATES created, added
  TRANSITIONS
      new*Create Medical Procedure=created
      created*AddMPintoGroup=added.
      added*Submit Claim=added.
```

The named interactions are specified as event types. Event types are presented as data structures. The attributes of event types are used as data containers for the data exchange with the environment.

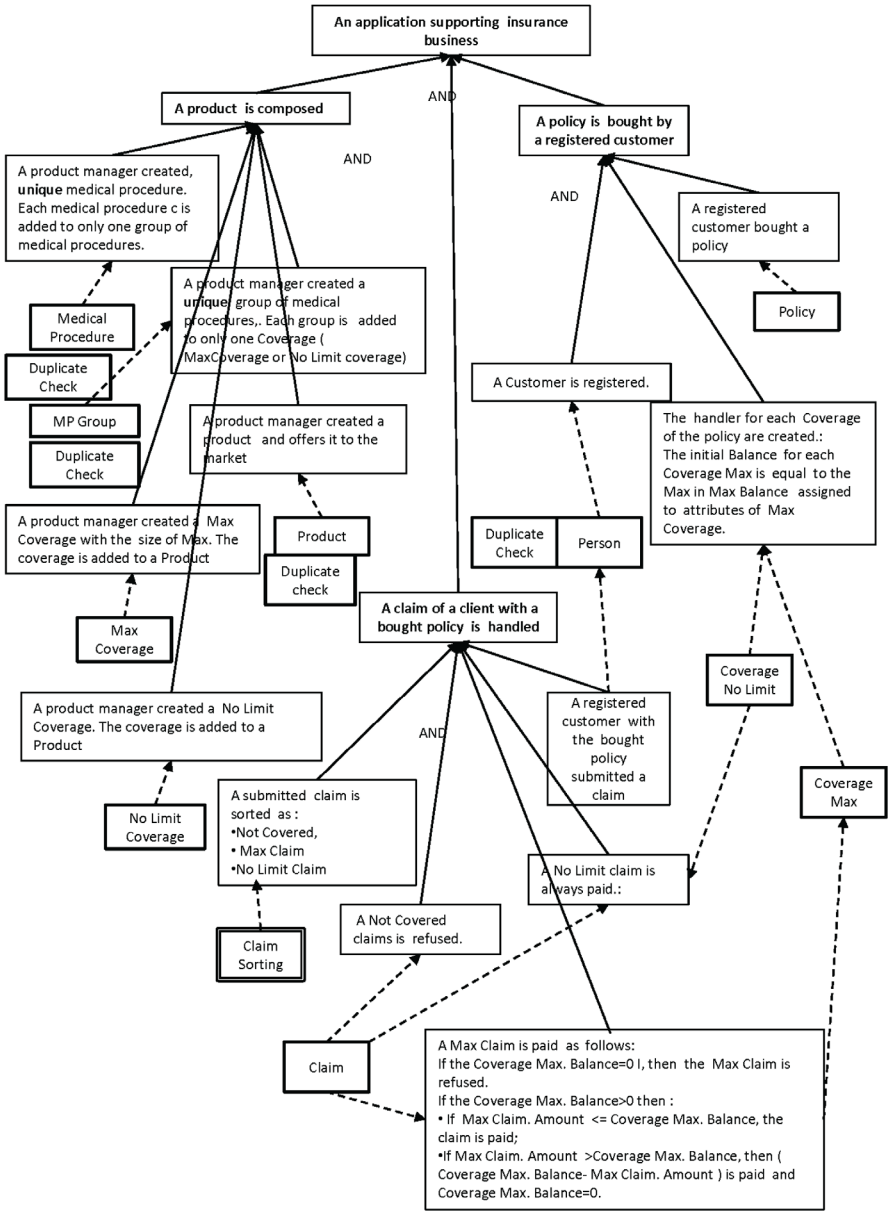
For example, the definition of event *Create Medical Procedure* shown below tells that this event is used by the protocol machine *Medical Procedure* and it takes a string from environment to name this medical procedure.

```
EVENT Create Medical Procedure
ATTRIBUTES
    Medical Procedure: Medical Procedure,
Name: String
```

Being in a state specified by a transition, the protocol machine accepts the event of this transition. If the protocol machine is not in the state where a given event causes a transition, this event is refused even if the event is recognized by the protocol machine. If an event has been accepted, it is processed until the quiescent state of the protocol machine. During this processing the other events are refused. This behavior of protocol machines is different from behavior of state machines. The UML state machines accumulate all recognized events in a queue so that they may cause a transition in the future (UML2.OMG, 2007). Accumulating events in the queues causes extra states of the

Jacqueline Arninkhof (inf.administratie@ou.nl)

Figure 2. Goals and corresponding protocol machines



model and non-determinism in behavior of state machines. Protocol machines are deterministic.

A protocol model is a synchronous CSP parallel composition of all protocol machines in the model (McNeile & Simons, 2006). The composition is used to compose different views on the system expressed as protocol machines. Protocol machines work synchronously resulting in observable behavior. That is why an event is only accepted by the model if all protocol machines recognizing this event accept it. Otherwise the event is refused. This is the core of the CSP parallel composition.

Jacqueline Arninkhof (inf.administratie@ou.nl)

For example, event *AddMPintoGroup* synchronizes two protocol machines *Medical Procedure* and MPGroup:

```
EVENT AddMPintoGroup
ATTRIBUTES
  Medical Procedure: Medical Procedure,
MPGroup:MPGroup
```

We say that there is a CSP parallel composition: *Medical Procedure || MPGroup.*

Event *Submit Claim* synchronizes protocol machines *Claim, Polis* and *Medical Procedure* and takes from the environment the *Claim Number* and its *Amount*.

```
EVENT Submit Claim
ATTRIBUTES
  Claim:Claim, Polis:Polis, Medical
Procdure: Medical Procedure,
  Claim Number:String, Amount:
Currency
```

The result of synchronization is the CSP parallel composition: *Claim || Polis || Medical Procedure.*

The complete protocol model in our insurance case is the CSP parallel composition the instances of 12 protocol machines. The number of instances is not restricted and depends on the interactive process of model execution. The meta-code of the model is given in the appendix.

The data extension of the initial CSP parallel composition semantics concerns with the ability of protocol machines to read but not alter the state of other protocol machines, so that the state causing accepting or refusing events can be formulated using states and local storages of all protocol machines in the model and the data from events.

Another consequence of the data extension of the CSP composition is the ability of protocol machines to derive own states from the states of other protocol machines. Derived states are calculated from the values of the stores states specified for protocol machines. A derived state extends the state space of the system model and used to generalize the state of different protocol machines for a specific system view. For example, if all medical procedures have been included into a group, the state "the group has been completed" can be derived. Having derived states we can separate stored state space and derived state space for reasoning and analyses.

The updating of the stored space of a protocol model is restricted by accepting one event at a time and handing it until the new quiescent state of the model. Only quiescent states visible from the environment are included into protocol models. As only quiescent states are specified in the goals, the semantics of protocol model corresponds to the semantics of goal specification.

A protocol machine called *Behaviour* may be included into another protocol machine. This means that an instance of the *Behaviour* is automatically created with the instance of the including protocol machine.

Behaviours are equally CSP parallel composed with other protocol machines.

Deriving state and updating state of several protocol machines demand some search of instances of protocol machines and their attributes. These search commands are specified in small java files using a set of search functions built into the Modelsope tool. There are three types of search functions:

1. Function *selectByRef("Behaviour_Name","Attribute_Name")* returns an array of instances, all of which include the specified behaviour (or object) and have the specified attribute referencing this.
2. Function *selectInContext ("Behaviour_Name","Event_Name")* returns an array of instances, all of which include the specified behaviour and have the specified event with the specified subscript in context.

Jacqueline Arninkhof (inf.administratie@ou.nl)

3. Function *selectInState ("Behaviour_Name","State")* returns an array if instances, all of which include the specified behaviour.

The data extension of the CSP parallel composition for protocol machines makes protocol models flexible in presentation of any modeling abstraction, such as objects and crosscutting concerns (see details in McNeile & Roubtsova, 2008, 2010) and adopting any model change as a separate protocol machines. The experimental studies demonstrate scalability and change adoptability on the applications of industrial size (Verheul & Roubtsova, 2011).

Events are identified in the goal-oriented approaches, but they are not used for object communication and do not contain data. Events in the goal-oriented approaches trigger operations. Protocol models work at the higher level than the level of operations. Protocol machines communicate with environment accepting or refusing events and by generating events to the environment. Dealing with events with data in protocol machines allows abstracting from the send-receive-operations and avoiding the non-determinism caused by them. The use of operations is an implementation decision which the protocol models avoid as ''requirements have to describe what the system does, not how its does it (Zave & Jackson, 1997).

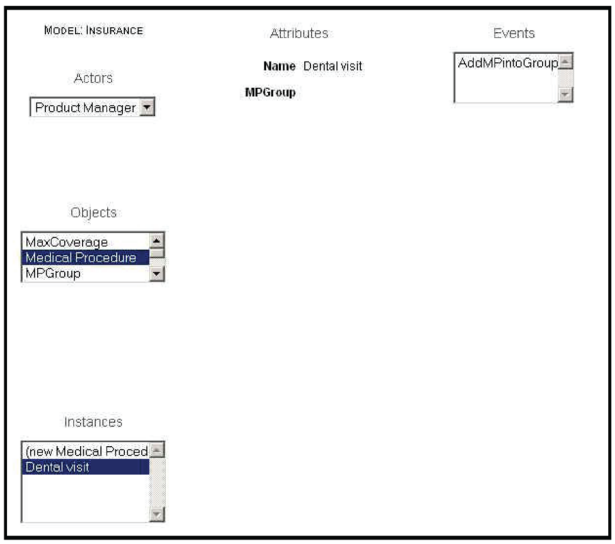
Protocol models are directly executable in the Modelscope tool (McNeile & Simons, 2011). The tool provides a generic interface for execution of any protocol model allowing submitting events and observation of results, protocol machine, and their attributes. The interface generated by the Modelscope tool for this model is shown in Figure 3.

Local Reasoning on Protocol Machines

Protocol Models are unique in the sense that they possess the property of local reasoning on each protocol machine about the behavior of the whole system. The local reasoning on protocol machines was proven in McNeile and Roubtsova (2008), and it was discussed in detail in McNeile and Roubtsova (2010) and Roubtsova (2011).

Local reasoning in Protocol Modelling is based on a property of CSP composition.

Figure 3. Execution of the protocol model



Jacqueline Arninkhof (inf.administratie@ou.nl)

- Let us take a sequence, *S*, of events that is accepted by the CSP parallel composition of protocol machines (*M1* || *M2*) of the two machines *M1* and *M2*.
- Then let us take the subsequence, *S0*, of *S* obtained by removing all events in *S* that are not recognized by the protocol machine *M1*.
- S0 will be accepted by the machine *M1* by itself.

In other words, composing *M1* with another machine with cannot "break its trace behavior of *M1*. We can use this property to support local reasoning on each protocol machine about the behaviour of the whole model. If by removing all events in *S* that are not recognized by *M1*, we have got a sequence *S0* that were not acceptable to *M1* or *M2*, then the original sequence *S* could not have been acceptable to *(M1* || *M2)*.

Each protocol machine has usually 1-5 states (Figure 4). The set of its sequences is observable. The loops can produce infinite traces, but testing of the finite set of traces for each simple protocol machine is sufficient to test one protocol machine. This means that verification of any requirement may be reduced to testing of a finite set of traces of relevant protocol machines. The testing of goals means that for each goal there is a reachable state and there are no states that do not correspond to one or another intermediate or final goal leading to the final state.

PROTOCOL MODEL CORRESPONDING TO THE GOAL MODEL

The state-transition part of protocol models can be presented graphically. Graphical presentation of CSP composition is possible but not necessary. The CSP parallel composition is comparable with an interpreter that executes the model interacting with its environment.

The graphical presentation of our case is shown in Figure 4. States of any protocol machine are ellipses, events are labels on arcs and transitions are triples of two ellipses and a labeled arc between them. The graphical presentation does not allow adequate specification of data.

We discuss this correspondence of goals and protocol machines goal by goal.

Goal "A Product is Composed"

Defining the goal *"A product is composed"* via sub-goals and requirements we identify *concepts Medical procedure*, *MPGroup*, *NoLimit Coverage*, *Max Coverage* and *Product*. Each of the concepts is specified as a protocol machine.

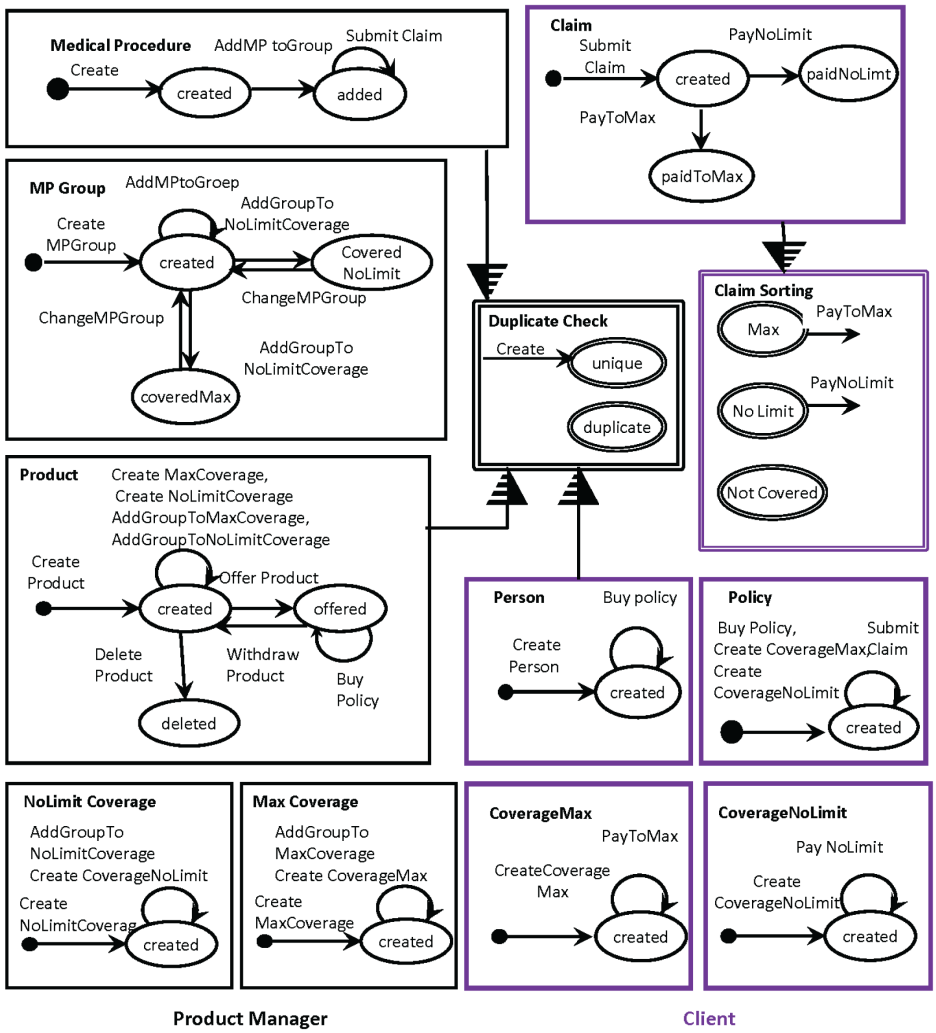
In order to create an instance of a *Product* the concept *Medical procedure* is populated with instances. Chosen instances of *Medical Procedure* are combined in one *MPGroup* and this group is assigned to a *NoLimit Coverage* instance. Another group is assigned to *Max Coverage*.

Several instances of *Max Coverage* can be defined with different attribute *Max Balance*. A completely composed product is offered to the market. Acceptance of the event *Offer Product* transits the instance of a *Product* to state *offered.* If a product is in the state *offered* (Figure 4), it is available for clients willing to buy a policy. At this moment the actor *Client* may submit events to the model.

Almost all events recognized by described protocol machines are submitted to the system by the actor called *Product Manager* and this is described in the meta-code in the Appendix. The specification of an actor selects protocol machines visible to a particular interacting actor from the actor specification and metacode of corresponding protocol machines the *Modelscope* tool generates the *Product Manager* interface to test the model.

Jacqueline Arninkhof (inf.administratie@ou.nl)

Figure 4. Protocol model



Unfolding Hidden Requirements

Very often during the execution of the life cycle events, a stakeholder may recognize a tacit requirement.

For example, a stakeholder executes the model and creates two medical procedures with the same name. The stakeholder decides that this is not what he wants and any *Medical procedure*; any *MPGroup* and any *Product* should be unique in the system. To achieve this crosscutting requirement a protocol machine that controls the duplication can be added and CSP composed with the model. The meta-code of the protocol machine *Duplicate Check* is presented below and in Figure 4. This protocol machine allows proceeding of event *Create* only if the created instance does not exist.

```
BEHAVIOUR !Duplicate Check
    STATES unique, duplicate
    TRANSITIONS @any*Create =unique
```

Graphical presentation of protocol machines with derived states demands particular attention.

Jacqueline Arninkhof (inf.administratie@ou.nl)

We depict derived states as double line ellipses. The function of state derivation is specified in a java file and it is not presented in the Figure 4.

Derived states should not form pairs to a specify transition. A derived state constraints the acceptance of an event. If a derived state presents a pre-state of an event then an outgoing arc is labeled with this event. If a derived state presents a post-state of an event then an ingoing arc is labeled with this event.

If the event recognized by the machine with derived states is accepted by the model then the resultant state is defined by protocol machines with stored states accepting this event.

The behaviour *Duplicate Check* is a cross-cutting concern, as it is included into protocol machines *Medical Procedure*, *MP Group* and *Product*. This means that an instance of *Duplicate Check* is created with any instance of *Medical Procedure, MP Group* and *Product* and CSP composed with the model.

The exclamation symbol in the metacode *BEHAVIOUR !Duplicate Check* means that there is a java code corresponding to the protocol machine. The java code is shown below. It finds all instances of the hosting protocol machine and checks if there is an instance with the same name and the same identification. The *Duplicate Check* protocol machine with the corresponding code is used for modelling the uniqueness constraint. The user does not see the java code but she executes the model and is able to create and use only unique instances.

```
package Insurance;
import com.metamaxim.modelscope.callbacks.*;

public class DuplicateCheck extends
Behaviour {
public String getState() {
         String myName=getString("Name");
          Instance[] existingIns = this
selectInState(this.getObjec
Type(), "@any");
        for (int i = 0; i < existingIns.
length; i++)
        if(existingIns[i].getString("Name").
equals(myName)&& !existingIns[i].
equals(this))
    return "duplicate";
    return "unique";
  }
  }
```

Goal "A Policy is Bought by a Registered Customer"

For this goal concepts *Person* and *Policy* have been recognized and the corresponding protocol machines have been specified. A person and a policy should be unique, so the *Duplicate Check* is included into the protocol machines *Person* and *Policy*.

Goal "A Claim of a Client with a Bought Policy is Handled"

Unfolding Hidden Requirements

1. Concept Claim is identified from this goal. A claim can be paid without limit or paid to maximum. This specification hides the need of classification of claims.
2. Buying a policy means some obligations for the insurance business to create containers for coverages for handling the policy limits. When the payment takes place the corresponding "container" is updated. Creating containers are expressed in requirements. The rules of these updates are not presented in the requirements.
3. We improve the situation with tacit requirements about the claim classification by adding a protocol machine *Claim Sorting* included into protocol machine *Claim*. The behaviour *Claim Sorting* checks if the medical procedure of the submitted claim belongs to the group assigned to the *NoLimit Coverage* or one of the Max *Coverages* in the policy of the client. If the medical procedure is not assigned to a group, it is not covered. If

Jacqueline Arninkhof (inf.administratie@ou.nl)

it is assigned, then correspondingly state *Max* or *NoLimit* is derived for protocol machines *Claim Sorting* (Figure 4). In state *Max* event *PayToMax* is allowed. In state *NoLimit* event *Pay NoLimit* is allowed.

4. We improve the situation with the specification of claim handling. We have identified that we need concepts of "containers" that will be updated with claim handling. To support modeling of claim handling from the point of view of the insurance business we create protocol machines CoverageNoLimit and *CoverageMax*. We put java file *BuyPolicy* into correspondence to event *Buy Policy*, so that this event generates events *createCoverageMax* and *createCoverageNoLimit* and submits them to the environment. Protocol machines *CoverageMax* and *CoverageNoLimit* accept these events and create the instances. All coverages of the product are found in the *Product* protocol machine and collected in an array *Instance[] myMaxCoverages.* For each coverage a creating event is generated (for example Event *createCoverageMax* and all attributes are filled in with the data. The presented code does not specify any implementation; it translates the event *BuyPolicy* to the concepts of protocol machines *CoverageMax* and *CoverageNoLimit.* The Modelscope tool finds the java file with the same name *Buy Policy* and executes it, so that the stakeholder executes the model and tests her requirements.

```
package Insurance;
import com.metamaxim.modelscope.callbacks.*;
public class BuyPolicy extends Event {
 public void handleEvent() {
        this.submitToModel();
        //Add the associated CoveragesProce-
dures to the Policy
        Instance myProduct= this=
getInstance("Product");
        String myProductName=myProduct.
getString("Product Name");
        Instance[] myMaxCoverages =
        this.getInstance("Product")
selectByRef("MaxCoverage", "Product");
        for (int i = 0; i < myMaxCoverages.
length; i++) {
        String myName=myMaxCoverages[i].
getString("MaxCoverage Name");
        int myMax=myMaxCoverages[i].
getCurrency("MaxBalance");
             Event createCoverageMax = this
createEvent("Create CoverageMax");
             createCoverageMax.
setNewInstance("CoverageMax", "Coverage-
Max");
              createCoverage-
Max.setInstance("MaxCoverage",
myMaxCoverages[i]);
              createCoverageMax.
setString("CoverageMax Name", myName);
createCoverageMax.setCurrency("Balance",
myMax);
               createCoverageMax.submitTo-
Model();
}
        Instance[] myNoLimitCoverages =
        this.getInstance("Product").
selectByRef("NoLimitCoverage", "Product");
        for (int i = 0; i < myNoLimitCover-
ages.length; i++) {
        String
myNoLimit=myNoLimitCoverages[i].
getString("NoLimitCoverage Name");

       Event createCoverageNoLimit = this.
createEvent("Create CoverageNoLimit");
      createCoverageNoLimit.setNewInstance("
CoverageNoLimit", "CoverageNoLimit");
      createCoverageNoLimit.
setInstance("NoLimitCoverage",
myNoLimitCoverages[i]);
       createCoverageNoLimit.
setString("CoverageNoLimit Name", myNoLim-
```

Jacqueline Arninkhof (inf.administratie@ou.nl)


```
it);
       createCoverageNoLimit.submitToMod-
el();
             }
         }
    }
```

5. Handling of claims means updating containers. The given requirements of these updates are not precise enough for executable modelling. For example, the goal *"A Max-claim is paid to Max" is* ambiguous. The executable model demands refinement of this goal formulation to a simple algorithm on how to calculate the payment and the rest of the coverage. This algorithm is presented in the state-function of the protocol machines *CoverageMax*:

```
package Insurance;
import com.metamaxim.modelscope.callbacks.*;

public class CoverageMax extends Behaviour {
public void
processPayToMax(Event event, String sub-
script) {

int newBalance=this.getCurrency("Balance");
int newAmount=
event.getInstance("Claim").
getCurrency("Amount");

int newPayment=0;
    if (newBalance >= newAmount) {
        newBalance=newBalance-newAmount;
        newPayment=newAmount;
     }     else {
        newPayment=newBalance;
        newBalance=0;
            }
    this.setCurrency("Balance", newBalance);
    this.setCurrency("PaymentToMax", newPay-
ment);
}
}
```

We can see that as a result of executable modeling we have refined initial requirements to the point that they may be executed. Even our simple case shows that the initial goal specification is refined because the executable protocol model demands more precision to be executed.

DISCUSSION

EXTREME is a New Method for EXecuTable Requirements Engineering, Management, and Evolution

If the ideas of a collection of methods used together produces more knowledge then each of combined methods, this collection presents a new method.

Our collection of methods combines the ideas of goal-oriented methods and protocol modeling.

Goal-oriented modeling methods along suggest strategic structuring of wishes of stakeholders as goals and refinement of the goals to requirements up to structural elements of system implementation: classes, their attributes and operations. In the absence of system implementation, verification of the produced specification demands methods of model checking. The properties are specified in a formal logic. Goal–oriented methods do not give instructions for transformation of requirements into formal properties. The properties are specified by an analyst, which is an error prone process. Then the state space specified by classes and attributes is built and the sequences of operation calls are generated to verify the truth of specified properties or the false of negations of properties.

The results of model checking need to be interpreted to stakeholders who specified goals. The interpretation may be misunderstood and the feedback of stakeholders may be lost at this stage. In this case the knowledge about correspondence between the specifications to the wishes of

Jacqueline Arninkhof (inf.administratie@ou.nl)

stakeholders will be fully identified only at the implementation stage.

Protocol modeling along allows producing executable models of requirements with any possible decomposition; however, this method does not answer the question how the protocol models should be built from system requirements. Tracing requirements and management of requirements is not addressed in protocol modeling.

Business practice needs methods that consider requirements engineering, evolution and management as parts of one process. Using model-checking techniques in this process (with specification of system properties and interpretation of results to requirements engineers) is not effective. This is like an extra step of translating to model checking after every change in requirements and translating the results of model checking back to requirements. Requirements engineers should be able to fulfill the self-validation of their requirements. By executing requirements on the protocol model in EXTREME, they are able to do this.

EXTREME replaces refinement of goals to the specification of classes, attributes, and operations with refinement of goals to protocol models. Being compositional in nature, the protocol models contain protocol machines directly corresponding to requirements. Protocol models possess the property of local reasoning described in section 3. Local reasoning means that properties of each protocol machine are preserved in the behaviour of the whole protocol model. That is why building and analyzing of large state space of the system by model checking may be replaced with execution of a limited number of protocol machines collaboratively responsible for the tested requirement.

The tested requirements should not be transformed into formal property. Each requirement usually describes the sequential steps of a protocol machine or update of local storage of a protocol machine. This means that a requirement is directly visible in one of protocol machines. The execution of the protocol model in de Modelscope tool shows the protocol machines responsible for the requirement. The requirements implemented as a protocol machine is preserved in the CSP parallel composition of protocol machines. The presence or absence of a requirement is an additional indication of the correctness of model evolution.

The user and requirements engineer provide their feedback as many times as needed by looking at model execution. In such a way EXTREME produces the knowledge about the correspondence of requirement specification to the wishes of stakeholders before the implementation of the system and more knowledge is produced than in original goal-oriented methods.

The improvement of the requirement engineering process means

- Producing adequately complete requirements.
- Enabling easy management of requirements including easy changing.
- Enabling local reasoning on parts of the model about the behavior of the whole.

Producing Adequately Complete Requirements

Protocol Modelling of requirements results in a refined goal tree shown in Figure 2. Contextual playing with executable requirements caused recognizing tacit requirements.

- We have extended initial requirements with uniqueness of instances. For example, *any Medical Procedure, Medical Procedure Group, Product* and *Person* should be unique. This concern is specified as a separate protocol machine *Duplicate Check*. This protocol machine accompanied with the corresponding call-back function is included into each of the named protocol machines and CSP composed with them.

  Duplicate check is a good example of a crosscutting concern. Protocol models provide the necessary flexibility for

Jacqueline Arninkhof (inf.administratie@ou.nl)

presentation of crosscutting concerns and other refinements of requirements (McNeile,A., Roubtsova E. (2010)).

- We have completed the requirements needed for claim handling by adding containers of coverages for each policy and by specification of updating procedure of those containers.

As we can see the ambiguities of the first definitions of claim handling and product definition are resolved by adding new protocol machines and CSP composing them with the protocol machines representing life cycle of identified objects. In fact each protocol machine is a presentation of a requirement that can be tested. If a requirement cannot be presented as a protocol machine it has to be refined. An executable protocol model if requirements indicate their adequate completeness.

Enabling Easy Management of Requirements Including Easy Changing

Goals have a clear tree structure and combining of them with protocol models puts protocol machines as leaves of the goal tree. The tree structure is good for search of goals and corresponding protocol machines. It is easy to delete an existing goal (protocol machine) and add a new goal (protocol machine).

The only problem with the trees is the accommodation of crosscutting concerns. In a tree structure, the crosscutting requirement and the corresponding protocol machine will inevitable appear several times.

In order to solve this inconvenience, the name of the protocol machine in the tree may contain a link to the place of the metadata specifying this protocol machine in the textual document. Modification of a requirement (goal) will result in search of the corresponding protocol machine in order to correct it.

In other methods combining of goal and operation models (Respect-IT (2007), Van, H.T. et.al. (2004)), refinement of requirements usually results in remodeling. The crosscutting concerns cause a lot of error prone remodeling activities. This is explained by the composition techniques used in operation models, namely, the sequential composition and the hieratical composition. If a concern is added, the sequences have to destroyed and built again. The change of the hierarchy usually causes complete remodeling. Example of these problems in conventional operation models are shown in McNeile,A., Roubtsova.E. (2007).

ENABLING LOCAL REASONING ON PARTS OF THE MODEL ABOUT THE BEHAVIOR OF THE WHOLE

EXTREME inherits the property of local reasoning from Protocol Modelling enabling also direct relations with the requirements for local reasoning.

Let us take a requirement "A submitted claim is sorted as: *Not Covered, Max Claim No Limit Claim*."

This requirement corresponds to the protocol machine *Claim Sorting*. We see that this machine has the specified states: *Not Covered, Max, No Limit*. *Claim Sorting* is included into protocol machine *Claim*.

The state of the protocol machine *Claim Sorting* is derived when the protocol machine *Claim* accepts event *Submit Claim*. Event *Submit Claim* contains information about *Medical Procedure*. If this *Medical Procedure* has been included into a group for Max *Coverage*, then the claim should be sorted as *Max Claim*. The behaviour of *Claim and Claim Sorting* is tested locally without analysis of the complete state space of the model. The proposed tests are the Submit *Claim* eventwith a medical procedure from the group *Max Coverage* and the check if it is sorted as *Max Claim*.

Jacqueline Arninkhof (inf.administratie@ou.nl)

FUTURE RESEARCH DIRECTIONS

Modeling of More Abstract Business Concepts

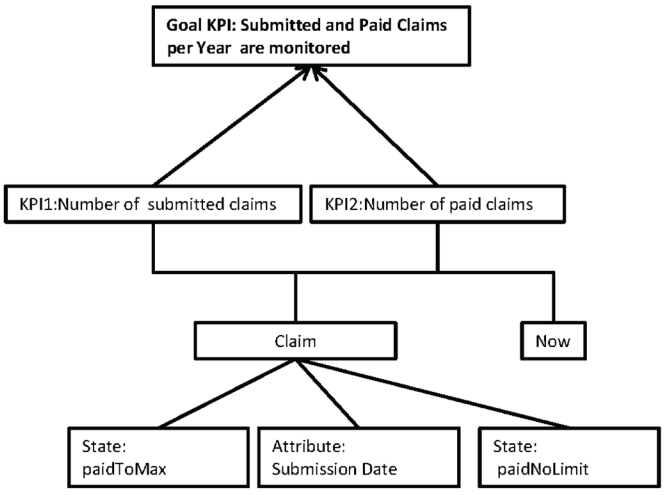
Controllable development of business is impossible without monitoring of its state, capacities, calculating key performance indicators and making decisions for correct and timely investment. Such monitoring happens both at the level of enterprises, enterprise departments and educational institutions, but also at the level of ministries and government. The major problem with monitoring and decision support is different interpretation of state, capacities, and key performance indicators. This problem has an objective reason as even businesses of the same branch have different variations. Ambiguity and different understanding of state, capacities and key performance indicators cause problems for management in making right decisions and for employees in directing their efforts. Cloud technologies and mobile technologies introduce new indicators that need to be unambiguously understood.

Our preliminary studies show that conceptual models of business capacities and KPIs lead to unambiguous definitions. EXTREME allows for building conceptual models of abstract business concepts and key performance indicators using ideas of goal-oriented approaches and protocol modelling. Analysis of KPIs models may result in useful standardisation of operational KPIs. The classified KPIs and business capacities will speed up solution of many business intelligence tasks, lead to easy building of business analytics into information systems and eventually will result in better decision support for business and better decisions.

At the moment performance indicators are not included into system models, but maintaining of high performance is always a goal of a system. The compositional nature of Protocol Models used in our method to make requirements operational gives the opportunity to raise the level of abstraction in models and relate many complex business concepts to behavioral models. Such concepts as business capabilities, Key Performance Indicators (KPIs) and motivation models may extend behavior models.

For example, let us see the monitoring of KPIs as a new goal "*The numbers of submitted and paid claims per year are calculated.*"

Figure 5. KPI: Submitted and paid claims per year



Jacqueline Arninkhof (inf.administratie@ou.nl)

These KPIs are conceptually related to the concept *Claim*, its attribute *Submission Date*, the time concept *Now* and two values of the concept *State, namely* the state value *paidToMax and* the state value *paidNoLimit.* The conceptual model is shown in Figure 5. The claims that have these values of their *State* concept are paid.

In order to calculate *KPI1: Number of submitted claims,* the date *oneYearAgo* from the value of Now is calculated. Then the claim instances with the value of the attribute *Submission Date* after *oneYearAgo* are found and the *KPI1: NumberOfClaims* is calculated (see function *getNumberOfClaims()).*

In order to calculate *KPI2:Number of paid claims,* the claims situated in the state *paidToMax* or *paidNoLimitthat* submitted after the *Submission Date* are found and their number is calculated (function *getNumberOfPaidClaims()).*

The ability to derive states makes it possible to calculate KPIs in Protocol Models. The metacode of the Protocol Model and the corresponding java code calculating the number of claim instances is shown below.

OBJECT ***CounterNumberOfClaims***

NAME Name

ATTRIBUTES Name: String, ***!NumberOfClaims:Integer, !NumberOfPaidClaims:Integer***

STATES created

*TRANSITIONS @new*GetNumberOfClaims=created*

EVENT GetNumberOfClaims

ATTRIBUTES CounterNumberOfClaims:CounterNumberOfClaims, Name:String

#----------

```
package Insurance;
import com.metamaxim.modelscope.callbacks.*;
import java.util.*;

public class CounterNumberOfClaims extends
Behaviour {
    public int getNumberOfClaims() {
             int NumberOfClaims=0;
             Date d = new Date();
             Calendar cal = Calendar.getIn-
stance();
     cal.add(Calendar.YEAR, -1);
     Date oneYearAgo = cal.getTime();
     Instance[] existingIns =
selectInState("Claim", "@any");
             for (int i = 0; i < existingIns.
length; i++) {
    Date SD=existingIns[i].
getDate("SubmissionDate");
             if (SD.compareTo(oneYearAgo)>0)
             NumberOfClaims=NumberOfClaims+1;
    }
             return NumberOfClaims;
    }
    public int getNumberOfPaidClaims() {
        int NumberOfPaidClaims=0;
        Date d = new Date();
        Calendar cal = Calendar.getIn-
stance();
     cal.add(Calendar.YEAR, -1);
     Date oneYearAgo = cal.getTime();
      Instance[] PaidToMaxIns =
selectInState("Claim", "paidToMax");
           for (int i = 0; i < PaidToMaxIns.
length; i++) {
      Date SD=PaidToMaxIns[i].
getDate("SubmissionDate");
           if (SD.compareTo(oneYearAgo)>0)
           NumberOfPaidClaims=NumberOfPaidC
laims+1;
    }
    Instance[] PaidNoLimitIns =
selectInState("Claim", "paidNoLimit");
            for (int i = 0; i < PaidNoLimi-
tIns.length; i++) {
    Date SD=PaidNoLimitIns[i].
getDate("SubmissionDate");
         if (SD.compareTo(oneYearAgo)>0)
         NumberOfPaidClaims=NumberOfPaidCla
ims+1;
```

Jacqueline Arninkhof (inf.administratie@ou.nl)


```
}
     return NumberOfPaidClaims;
}
}
```

It is possible that during the modelling and execution the stakeholder will decide that the date of KPI monitoring is fixed and this concept *Fixed Monitoring Date* will replace concept *Now* in the conceptual model.

The discovery of patterns of protocol models of different KPIs and other abstract business concepts is one of directions for future work.

Tool Support for Traceability of Requirements in Models

The goals and requirements are naturally kept in a tree structure. From this tree structure is possible to generate a textual document for metadata of protocol machines. The goals and requirements may be presented in this document as comments following the hierarchy of requirements in the goals tree. The meta-code of each protocol machine may be written then under own requirements as we have presented in the appendix. In case of crosscutting concerns, the meta-code of a protocol machine may have a link to the requirement in the tree structure and the description of the relevant requirements from the branch of the goal tree may be generated in its comments when necessary. This means that the problem of traceability of requirements in protocol models will be solved. The traceability of requirements in protocol models needs to be supported with a tool in near future.

CONCLUSION

The correspondence between the strategic and the operational levels of a business is a success factor of the business. In this chapter, we have presented a new method EXTREME that combines the ideas of thegoal models at the strategic level with the executable protocol models at the operational level. The uniqueness of the proposed combination is in synchronous semantics both at the strategic and the executable levels that easies both the goal refinement and the executable modeling.

The name of the method EXTREME can be associated with Extreme Programming (Cockburn, 2001b). The associating is not wrong as the ideas of Extreme Programming to improve a software project on the basis of five essential principles; communication, simplicity, feedback, respect, and courage are present in the method EXTREME. The difference is that the EXTREME can be also used for modeling of businesses, not necessarily for programming software, and the five principles are used already at the stage of requirements engineering before the implementation phase.

The extended CSP parallel composition used in Protocol Modelling gives extra decomposition flexibility and ability to reason locally avoiding the state space explosion at the stage of analysis.

The goal-orientation brings the order and simplicity into decomposition, testing and evolution. New goals are refined as requirements and corresponding new protocol machines and CSP composed with the rest of the model. The existing model parts are not changed and preserve their behaviour in the growing model. Synchronous nature of protocol models does not add any states that cannot be explained by goals and their refinement. Therefore the execution of models can be easily controlled from the goal perspective by the stakeholders.

The method has been tested for insurance products in Oracle Nederland (Verheul & Roubtsova, 2011). Currently there is a running experiment in two companies that want to improve requirements management of product releases by application of the EXTRIME method.

Jacqueline Arninkhof (inf.administratie@ou.nl)

REFERENCES

Alsumait., et al. (2003). Use case maps: A visual notation for scenario-based requirements. In *Proceedings of the 10th International Conference on Human-Computer Interaction.* IEEE.

Cockburn, A. (2001a). *Writing effective use cases*. Reading, MA: Addison-Wesley.

Cockburn, A. (2001b). *Agile software development*. Reading, MA: Addison-Wesley Professional.

Dardenne, A., van Lamsweerde, A., & Fickas, S. (1991). Goal-directed requirements acquisition. *Science of Computer Programming*, *20*(1-2), 3–50. doi:10.1016/0167-6423(93)90021-G.

Darimont, R., & Lemoine, M. (2006). Goal-oriented analysis of regulations. In *Proceedings of the International Workshop on Regulations Modelling and their Validation and Verification, REMO2V'06,* (pp. 838-844). REMO2V.

Firesmith, D. G. (2005). Are your requirements complete? *Journal of Object Technology*, *4*(1), 27–43. doi:10.5381/jot.2005.4.1.c3.

Harel, D., & Kugler, H. (2002). Synthesizing state-based object systems from LSC specifications. *Foundation of Computer Science*, *13*(1), 5–51. doi:10.1142/S0129054102000935.

Hoare, C. (1985). *Communicating sequential processes*. New York: Prentice-Hall International.

ITU. (2008). *Formal description techniques (FDT) – User requirements notation recommendation Z.151 (11/08)*. Retrieved September 3, 2012, from http://www.itu.int/rec/T-REC-Z.151-200811-I/en

Jensen, K. (1997). *Coloured Petri nets: Basic concepts, analysis methods, and practical use*. Berlin: Springer Verlag. doi:10.1007/978-3-642-60794-3.

Kavakli, E. (2002). Goal-oriented requirements engineering: A unified framework. *Requirements Engineering*, *6*(4), 237–251. doi:10.1007/PL00010362.

Letier, E. et al. (2008). Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, *15*(2), 175–206. doi:10.1007/s10515-008-0027-7.

McNeile, A., & Roubtsova, E. (2007). Protocol modelling semantics for embedded systems. In *Proceedings of the Special Session on Behavioural Models for Embedded Systems at the IEEE Second International Symposium on Industrial Embedded Systems, SIES'2007*. Lisbon, Portugal: IEEE.

McNeile, A., & Roubtsova, E. (2008). CSP parallel composition of aspect models. In *Proceedings of the International Workshop on Aspect-Oriented Modelling, AOM'08.* ACM Press.

McNeile, A., & Roubtsova, E. (2010). Aspect-oriented development using protocol modeling. *Transactions on Aspect-Oriented Software Development*, *7*, 115–150.

McNeile, A., & Simons, N. (2006). Protocol modelling: A modelling approach that supports reusable behavioural abstractions. *Software & Systems Modeling*, *5*(1), 91–107. doi:10.1007/s10270-005-0100-7.

McNeile, A., & Simons, N. (2011). *Modelscope*. Retrieved September 3, 2012, from http://www.metamaxim.com

Regev, G., & Wegmann, A. (2011). Revisiting goal-oriented requirements engineering with a regulation view. *Lecture Notes in Business Information Processing, 109.*

Respect-IT. (2007). *A KAOS-tutorial*. Retrieved September 3, 2012, from http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf

Jacqueline Arninkhof (inf.administratie@ou.nl)

Roubtsova, E. E. (2011). Reasoning on models combining objects and aspects. *Lecture Notes in Business Information Processing*, *109*, 1–18. doi:10.1007/978-3-642-29788-5_1.

Tversky, A., & Simonson, I. (1993). Article. *Management Science*, *39*(10), 1179–1189. doi:10.1287/mnsc.39.10.1179.

UML2.OMG. (2007). *Unified modeling language: Superstructure version 2.1.1*. Formal/2007-02-03.

Van, H. T., et al. (2004). Goal-oriented requirements animation. In *Proceedings of RE'04: 12th IEEE International Requirements Engineering Conference,* (pp. 218-228). IEEE.

van Lamsweerde, A. (2004). Goal-oriented requirements engineering: A roundtrip from research to practice. In *Proceedings of the 12th IEEE International Requirements Engineering Conference*. Kyoto, Japan: IEEE.

Verheul, J., & Roubtsova, E. (2011). An executable and changeable reference model for the health insurance industry. In *Proceedings of the 3rd International Workshop on Behavioural Modelling - Foundations and Applications.* ACM.

Yu, E. (1995). *Modelling strategic relationships for process reengineering*. (Ph.D. Thesis). Dept. of Computer Science, University of Toronto, Toronot, Canada.

Zave, P., & Jackson, M. (1997). Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, *6*(1), 1–30. doi:10.1145/237432.237434.

KEY TERMS AND DEFINITIONS

Adequate Completeness of Requirements: Requirements are adequately complete if all specified requirements are met by an executable system model and the model can be used to reason about the system.

CSP Parallel Composition: Is an algorithm of constructing sequences of events accepted by the abstracting from data. Processes P and Q must both be able to perform event before that event can occur. Processes communicate via synchronous message passing. $(a \rightarrow P) \,||\, [\{a\}] \,||\, (a \rightarrow Q)$.

CSP Parallel Composition Extended for Machines with Data: Is an algorithm of constructing sequences of events accepted by the modeled system including data storages and state spaces.

Event: A recognized happening in an environment that can be expressed as a data structure. One element of this data structure is an event type.

Goal: A general name of a piece of system functionality; a description of a system state being a result of an execution of the piece of system functionality.

Goal Tree: A tree the root of which is a system, the next nodes is the system goals; the goals are refined to requirements.

Protocol Machine: Is a state-transition construction with data storage that defines ability of a system to accept events from environment. If a pre-event or a post-event constraint on the data storage is not met then the machine refuses the event.

Requirement: A description of system reactions on related events.

Jacqueline Arninkhof (inf.administratie@ou.nl)

APPENDIX

Meta Code of a Protocol Model of an Insurance Business

MODEL Insurance
#---
#Product is composed.
#A product manager created, unique medical procedure. Each medical procedure c is added to only one #group of medical procedures.

```
OBJECT Medical Procedure
     NAME Name
     INCLUDES Duplicate Check
          ATTRIBUTES Name: String, MPGroup:MPGroup
          STATES created, added
          TRANSITIONS          @new*Create Medical Procedure=created,
                               created*AddMPintoGroup=added,
                               added*Submit Claim=added,

BEHAVIOUR !Duplicate Check
     STATES unique, duplicate
          TRANSITIONS @any*Create =unique
```

#A product manager created a unique group of medical procedures, Each group is added to only #one #Coverage (MaxCoverage or No Limit coverage.

```
OBJECT MPGroup
     NAME Name
     INCLUDES Duplicate Check
     ATTRIBUTES
          Name: String,
          !CurrentState:String,
          MaxCoverage:MaxCoverage,
          NoLimitCoverage:NoLimitCoverage
          STATES created, coveredMax,
          coveredNoLimit
     TRANSITIONS @new*Create MPGroup=created,
                 created*AddMPintoGroup=created,
                 reated*!AddGroupToNoLimitCoverage=coveredNoLimit,
                 created*!AddGroupToMaxCoverage= coveredMax,
                 coveredMax*ChangeMPGroup=created,
                 coveredNoLimit*ChangeMPGroup=created,
```

Jacqueline Arninkhof (inf.administratie@ou.nl)

#A product manager created a Max Coverage with the size of Max. The coverage is added to a #Product.

```
OBJECT MaxCoverage
     NAME Name
     ATTRIBUTES
          Name: String,
          MaxBalance:Currency,
          Product:Product,
          Product Name:String
          STATES created,
TRANSITIONS @new*Create MaxCoverage =created,
                  created*AddGroupToMaxCoverage= created,
                  created*Create CoverageMax= created,
```

#A product manager created a No Limit Coverage. The coverage is added to a Product.

```
OBJECT NoLimitCoverage
NAME Name
     ATTRIBUTES
     Name: String,
     Product:Product,
     Product Name:String
     STATES created,
     TRANSITIONS @new*Create NoLimitCoverage =created,
                 created*AddGroupToNoLimitCoverage= created,
                 created*Create CoverageNoLimit= created,
```

#A product manager created a product and offers it to the market.

```
OBJECT Product
     NAME Name
     INCLUDES Duplicate Check
     ATTRIBUTES Name:String
     STATES created, offered
     TRANSITIONS @new*Create Product=created,
                 created* Create MaxCoverage=created,
                 created*Create NoLimitCoverage=created,
                 created*AddGroupToMaxCoverage=created,
                 created*AddGroupToNoLimitCoverage=created,
                 created*Offer Product=offered,
                 offered*Buy Policy=offered

EVENT Create MaxCoverage
ATTRIBUTES   MaxCoverage:MaxCoverage,
                  Name: String,
                  MaxBalance:Currency,
                  Product:Product
EVENT Create NoLimitCoverage
ATTRIBUTES   NoLimitCoverage:NoLimitCoverage,
               Name:String,
               Product:Product
```

Jacqueline Arninkhof (inf.administratie@ou.nl)


```
EVENT AddMPintoGroup
ATTRIBUTES   Medical Procedure: Medical Procedure,
               MPGroup:MPGroup
EVENT Create Medical Procedure
ATTRIBUTES   Medical Procedure: Medical Procedure,
               Name:String
EVENT Create MPGroup
ATTRIBUTES     Name: String, MPGroup:MPGroup
EVENT AddGroupToNoLimitCoverage
ATTRIBUTES   MPGroup:MPGroup,
             NoLimitCoverage:NoLimitCoverage,
             Product:Product
EVENT AddGroupToMaxCoverage
ATTRIBUTES   MPGroup:MPGroup,
               MaxCoverage:MaxCoverage,
               Product:Product
EVENT ChangeMPGroup
ATTRIBUTES    MPGroup:MPGroup

EVENT Create Product
ATTRIBUTES    Product:Product,
                Name:String

EVENT Offer Product
ATTRIBUTES    Product:Product

GENERIC Create
MATCHES Create Medical Procedure,Create Product,Create MPGroup
```

#A policy is bought by a registered customer.
#A Customer is registered.

```
OBJECT Person
NAME Name
INCLUDES Duplicate Check
       ATTRIBUTES
       Name: String,Policy: Policy
       STATES created
       TRANSITIONS @new*Create Person=created,
                   created*Buy Policy=created,
```

#A registered customer bought a policy.

```
OBJECT Policy
     NAME Name
     INCLUDES Duplicate Check
     ATTRIBUTES Name: String, Product:Product, Person:Person
          STATES created, deleted, offered
     TRANSITIONS @new*Buy Policy=created,
                 created*Submit Claim=created,
                   created*Create CoverageMax=created,
                      created*Create CoverageNoLimit=created
```

Jacqueline Arninkhof (inf.administratie@ou.nl)

#The handler for each Coverage of the policy are created.
#The initial Balance for each Coverage Max is equal to the Max in Max Balance assigned to #attributes #of Max Coverage.

```
OBJECT CoverageMax
NAME CoverageMax Name
    ATTRIBUTES CoverageMax Name: String,MaxCoverage:MaxCoverage,
       Balance:Currency PaymentToMax:Currency, Policy:Policy
              STATES created
              TRANSITIONS @new*Create CoverageMax =created,
                          created*!PayToMax=created,

OBJECT CoverageNoLimit
NAME CoverageNoLimit Name
     ATTRIBUTES CoverageNoLimit Name:String,PaymentNoLimit:Currency,
Policy:Policy
              STATES created
        TRANSITIONS @new*Create CoverageNoLimit=created,
                    created*!PayNoLimit=created,

EVENT !Buy Policy
ATTRIBUTES Policy:Policy, Policy Number:String, Person:Person, Product:Product
#-------------------------------------------------------------------------------
```

#A Claim of a client with a bought policy is handled.
#A registered customer with the bought policy submitted a claim.

```
OBJECT Claim
     NAME Name
     INCLUDES Duplicate Check,Sorting Claim,
     ATTRIBUTES Name: String, Policy:Policy, Medical Procedure: Medical
Procedure,
                Amount:Currency, SubmissionDate:Date,
                CoverageMax:CoverageMax, CoverageNoLimit:CoverageNoLimit,
     STATES created, paidToMax, paidNoLimit
     TRANSITIONS @new*Submit Claim=created,
                 created*PayToMax=paidToMax,
                 created*PayNoLimit=paidNoLimit
```

Jacqueline Arninkhof (inf.administratie@ou.nl)

#A submitted claim is sorted as:
#Not Covered,
#Max Claim
#No Limit Claim
#A Max Claim is paid as follows:
#If the Coverage Max. Balance=0 I, then the Max Claim is refused.
#If the Coverage Max. Balance>0 then:
#If Max Claim. Amount <= Coverage Max. Balance, the claim is paid;
#If Max Claim. Amount >Coverage Max. Balance, then (
#Coverage Max. Balance- Max Claim. Amount) is paid and
#Coverage Max. Balance=0.

#A Not Covered claims is refused.
#A No Limit claim is always paid.

```
BEHAVIOUR !Sorting Claim
           ATTRIBUTES
     STATES Max, NoLimit, NotCovered
     TRANSITIONS Max*PayToMax=@any,
                 NoLimit*PayNoLimit=@any

EVENT PayToMax
ATTRIBUTES CoverageMax:CoverageMax, Claim: Claim

EVENT PayNoLimit
ATTRIBUTES CoverageNoLimit:CoverageNoLimit, Claim:Claim


EVENT Create CoverageMax
ATTRIBUTES CoverageMax:CoverageMax, Policy:Policy, CoverageMax Name:String,
Balance:Currency, MaxCoverage:MaxCoverage

EVENT Create CoverageNoLimit
ATTRIBUTES CoverageNoLimit:CoverageNoLimit, Policy: Policy, CoverageNoLimit
Name: String, NoLimitCoverage:NoLimitCoverage

EVENT Submit Claim
ATTRIBUTES Claim:Claim, Policy:Policy, Claim Number:String, Medical Procedure:
Medical Procedure, Amount:Currency, SubmissionDate:Date

EVENT Create Person
ATTRIBUTES Person:Person, Name: String
#-------------------------------------------------------------------------
```