A Conceptual Schema for a Conference Management Application

(Final Report, April 2005)

Ruth Raventós

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
raventos@lsi.upc.edu

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

Conference management applications have been widely used as examples for the study of conceptual schemas because of their manageable size and diversity of cases. However, as far as we are concerned, the conceptual schemas available only refer to the structural or static part of these systems. Therefore we have considered useful and necessary for our work to develop a whole specification, including the behaviour of a conference management application. The conceptual schema proposed in this paper is based on the study of the code of a real existing product: MYREVIEW. However this paper does not intend to be an accurate description of such system. Moreover some changes have been made to ease the understanding of the system or to include some features of two other existing products: OPENCONF and COMMENCE.

Additionally, this paper has also permitted to test or consolidate the *real* application of some proposals. The first group consists on alternate mechanisms to define integrity constraints and derived elements proposed in [1, 2] by Olivé. The root of these proposals is the definition of constraint and derived elements by means of operations. Secondly, it aims at exploiting the benefits of modelling events as entity types proposed by Olivé [3]. Finally, it also includes the distinction between domain events, action request events and query events in which Olivé and Raventós are currently working. Note that the specification of this case has been defined in UML 2.0 [4] and OCL 2.0 [5].

This document is structured as follows: first, a description of the application; second, a complete specification including some previous explanations in each section; finally, conclusions.

2. A Conference Management Application

The specification proposed in this paper is mainly based on the study of the MYREVIEW system with some additional features of OPENCONF and COMMENCE systems.

MYREVIEW is a web-based conference management software which was implemented in may-july 2003 with PHP and MySQL by Philippe Rigaux[1]. It was used initially for managing the ACM conference on Geographic Information Systems (See *http://www.esri.com/events/acm/index.htm*).

OPENCONF is an open source conference management system, under active development by Zakon Group LLC. Since its release in 2003, OpenConf has been used by over 90 conference and journals worldwide (See *http://www.openconf.org/technology/openconf-conferences.shtml*).

COMMENCE System is a web application developed by a team from The University of Queensland and Singapor Polytechnic. The latest version is hosted on Sourceforge (See *iaprcommence.sourceforge.net*).

The three systems are open-source web-applications written in PHP linked to a MySQL database, used for managing Technical Conferences.

Note that most of the explanations described in this paper may also be found in the document of the description of MYREVIEW (in *http://myreview.lri.fr/index.php?action=doc*)

Our conference management application, in the following CMA, includes the traditional functionalities of such systems which are summarized as follows:

1. **Conference preparation support**.

The system allows to create, to change, to remove and to query the following information: basic data of the conference, the list of research topics, the list of reviewers, and the list of evaluation criteria which are required to evaluate papers.

2. **Paper submission support.**

Authors can submit an abstract, along with the author's list, the contact author and the main topics of the paper. They receive an id and a password which must be used later on to submit the full paper (a file which is stored by the system).

3. **Assignment of papers to reviewers support**.

The organizer can use a manual or automatic assignment of papers to the reviewers. The assignment of papers is the most time-consuming task when managing the submission phase of a conference. CMA proposes an automatic assignment which relies on a variant of weighted matching algorithms for bipartite graphs.

[1] The system distribution is based on the GPL license. Basically it can be freely used and modified by anybody – but nobody may claim that he/she created it and any modifications should be public.

Reviewers are required to rate the submitted papers, based on the title, abstract and authors information, and these ratings are used by the algorithm to obtain the best possible assignment.

4. **Review and evaluation support**.

Reviewers can connect to the system by using their email as login, and a password. They can download the papers they have been assigned to, submit their evaluation, and modify it at any moment. Based on the reviews, the chairman can "accept" or "reject" papers. An acceptance or rejection notification is sent to the contact author of each paper, together with the (anonymous) reviews.

3. The Conceptual Schema

The conceptual schema of CMA includes all relevant general static and dynamic aspects. The part of the conceptual schema that describes the static part is called the structural schema and the part that deals with dynamic aspects is called the behavioural schema. Both are described next.

3.1 The Structural Schema

The structural schema describes the static part of a conceptual schema, i.e. entity types, relationship types, integrity constraints and the state derivation rules that may derive information from the entity and relationship types in the domain.

The structural schema is described by a class diagram where concepts relevant to the system and their relationships are represented. The relevant integrity constraints are included by assigning an operation with the stereotype «IC» to each constraint, as suggested in [1]. Analogously, derived elements are described as suggested by Olivé in [2].

Figure 1 shows the structural schema of CMA. The explanation of the elements shown in the Figure 1 and the specification of the data types, initial values, operations and integrity constraints, in OCL, will be described in the upcoming.

Conference

CMA comes with a set of parameters that must be configured when organizing the conference:

- Name: used in some mails and HTML pages.
- Acronym: used as a short identification of the conference.
- URL: used as base URL in several mails and HTML links.
- Email: used as the 'From' and 'Reply-to' value for mails.
- Chair mail: used to send copies of mails.
- Submission open: when 'True' authors are allowed to submit papers. Switching from 'True' to 'False' closes the submission phase.
- Review open: when 'True' reviewers are allowed to submit reviews. Switching from 'True' to 'False' closes the review phase.
- Camera-ready open: when 'True' authors are allowed to submit the camera-ready version of their accepted papers. Switching from 'True' to 'False' closes the camera-ready phase.
- Reviewers per paper: number of reviewers which must be assigned to each paper.
- LastPaperIdentifier: the last number assigned to a paper used to generate, in an abstract submission, a new identifier of paper
- ConflictGap : Two reviews are deemed conflicting if there is a gap larger than (or equal to) the number of *conflictGap* between their overall rates.
- Submission, review and camera-ready deadlines: are the deadlines of the corresponding phases.

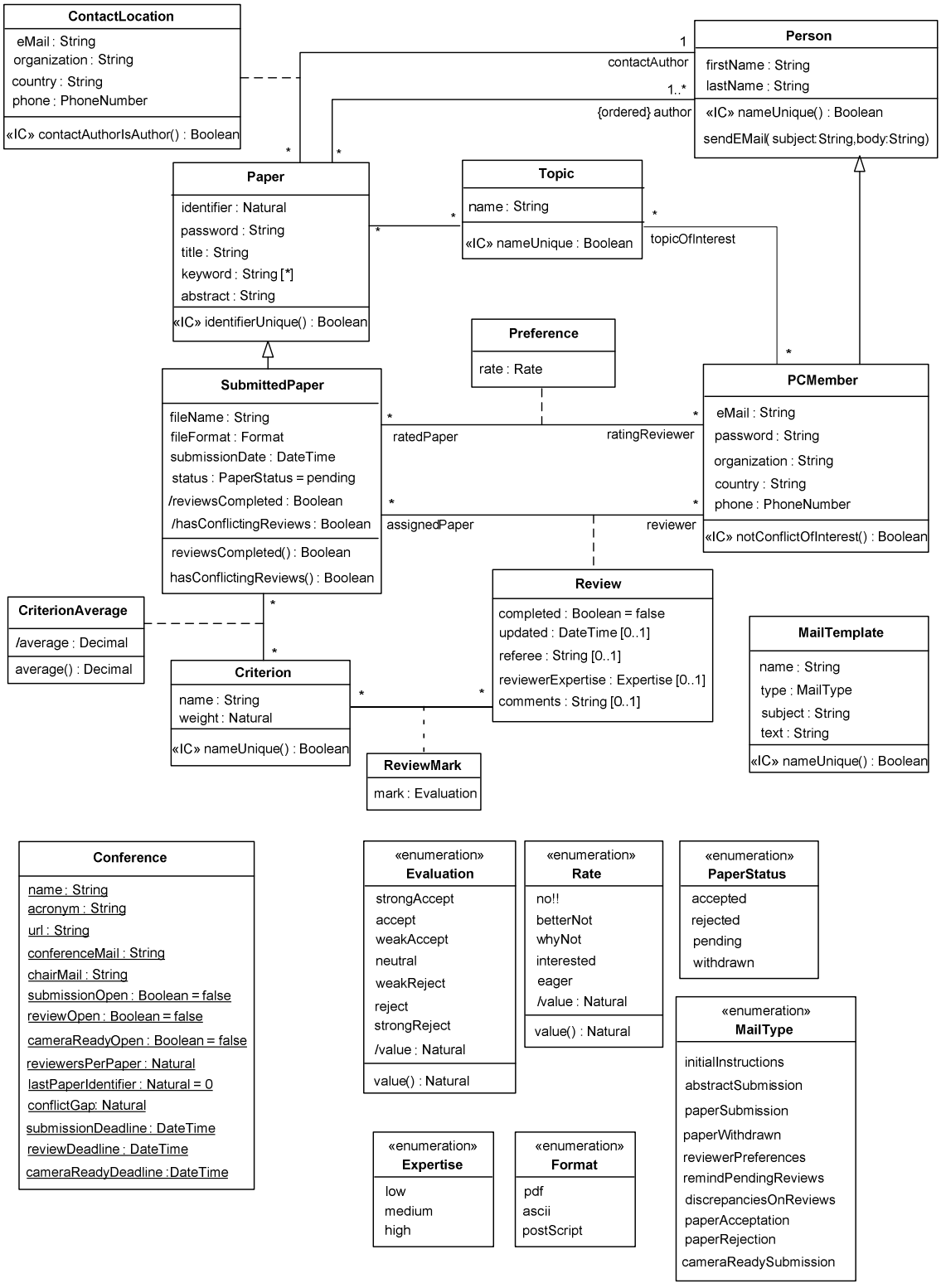

Fig. 1. Structural Schema of CMA

Person

Persons are identified by name and last name. They may be authors and/or Program Committee (PC) members. Authors are created when submitting a paper and the contact author must provide the following information: email, organization, country and phone number. The intended effect of the operation sendEMail is that an email message, with some information given in the operation parameter, is sent to the corresponding person (see Message below).

Program Committee Member

Additional data is kept for the members of the program committee: their email, password, organization, country and phone number.

Topic

CMA allows registering a list of research topics. In such a case, PC members are asked to select their preferred topic of interest, and ask authors to assign topics to the paper(s) they submit. Preferences can help during the manual assignment of papers to reviewers.

Paper

When the submission phase of the conference is "open", authors may submit papers. Initially authors must submit an abstract, along with some basic information concerning their paper, then the file of the paper must be uploaded. The contact author receives an acknowledgment of their submission, along with a paper id and a password. This id/password can be used during all the submission phase to modify the submission information and the paper itself. Additionally to the identifier and password the following information is kept in relation to the papers: a title, the name of the file, keywords, the abstract, the format in which is submitted and the last submission date. The status indicates whether a paper is "pending" (default value), i.e., it has been submitted but not reviewed, "accepted" or "rejected" after reviews, or "withdrawn", i.e., it has been removed from the conference.
It is unavoidable that some reviewers diverge on their evaluations. Two reviews are deemed conflicting if there is a gap larger than (or equal to) the number of *gapConflict* between their overall rates (defined in hasConflictingReviews attribute).

Preference

Each PC member can rate a paper to express his/her willingness/expertise to review that paper. The rate is raging from "I do not want this paper" to "I am eager to review this paper".

Review

The information needed for each review on any paper is: whether it has been completed, the date of its last update, the name of the referee if someone different from the PC member has reviewed the paper, the expertise of the reviewer in the paper, comments of the reviewer and for each criterion (see Criterion below) the punctuation given.

Criterion and Criterion Average

The list of criteria (e.g., originality, quality, relevance, presentation, recommendation) used to evaluate a paper is customizable. Moreover, a weight (natural) is associated to each criterion and the overall mark of a review is the weighted average of the marks to the criterion. If (C_1, w_1), (C_2, w_2),…,(C_n, w_n) is the list of criteria, along with their weights, and $m_1,m_2,\ldots,m_n$ are the marks of a review, then the overall evaluation is given by:

$$\frac{\sum_{i=1}^{n} w_i \times m_i}{\sum_{i=1}^{n} w_i}$$

The recommended configuration is to give a weight of 1 to "recommendation" criterion (this is the only compulsory criterion) and 0 to the others.

Message and Mail Template

Mails in CMA are sent automatically as consequences of the following actions: abstract submission, paper download and review submission. A message consists of the following parts:

- The sender: it is always set to the conferenceMail attribute.
- The subject: it is always set to the acronym attribute plus some information specified in the subject attribute.
- The "To" field, set to one email address.
- The "cc" field, set to chair email of the conference.
- The message body that often consists of text of a template depending on its MailType but may be personalized.

Data Types

The specification, in OCL, of the value attribute of the data types defined in the structural schema is the following:

```
context Evaluation::value():Natural
body:  if self = Evaluation::strongAccept then 7
       else if self = Evaluation::accept then 6
       else if self = Evaluation::weakAccept then 5
       else if self = Evaluation::neutral then 4
       else if self = Evaluation::weakReject then 3
       else if self = Evaluation::reject then 2
       else if self = Evaluation::strongReject then 1
       endif

context Rate::value():Natural
body:  if self = Rate::no!! then 0
       else if self = Rate::BetterNot then 1
       else if self = Rate::WhyNot then 2
       else if self = Rate::Interested then 3
       else if self = Rate::Eager then 4
       endif
```

Initial Values

Some initial or by default values are expressed in the structural schema of Figure 1 as:

```
SubmittedPaper.status : PaperStatus = pending
Review.Completed : Boolean = false
Conference.submissionOpen : Boolean = false
Conference.reviewOpen: Boolean = false
Conference.cameraReayOpen: Boolean = false
Conference.lastPaperIdentifier : Natural = 0
```

Integrity Constraints

The specification, in OCL, of the integrity constraints defined in the structural schema is the following:

```
«IC»
context PCMember::notConflictOfInterest:Boolean
body: assignedPaper.contactLocation.organization->excludes(organization)
```

```
«IC»
context Person::nameUnique():Boolean
body:  Person.allInstances() -> isUnique(firstName.concat(lastName))


«IC»
context ContactLocation::contactAuthorIsAuthor():Boolean
body:  self.paper.author -> includes(contactAuthor)

«IC»
context Topic::nameUnique():Boolean
body:  Topic.allInstances() -> isUnique(name)

«IC»
context Criterion::nameUnique():Boolean
body:  Criterion.allInstances() -> isUnique(name)

«IC»
context Paper::identifierUnique():Boolean
body: Paper.allInstances() -> isUnique(identifier)

«IC»
context MailTemplate::nameUnique():Boolean
body:  MailTemplate.allInstances() -> isUnique(name)
```

Derivation Rules

The specification, in OCL, of the derivations rules defined in the structural schema is the following:

```
context CriterionAverage::average():Decimal
body:  let total=self.submitedPaper.review.reviewMark->
             select(criterion=self.criterion).mark.value->sum()
       let numberOfReviews= self.submitedPaper.review->size()
       in
       if submittedPaper.reviewsCompleted then total/numberOfReviews
       else 0 endif

context SubmittedPaper::reviewsCompleted():Boolean
body:  review->forAll(completed)

context SubmittedPaper::hasConflictingReviews():Boolean
-- A paper has a conflict if there are at least two recommendation marks with
a difference of the maximum gap considered.
body:  self.review –> exists (r1, r2 | r1<>r2 and r1.completed and r2.completed
       and r1.reviewMark->any(criterion.name='recommendation').mark.value -
        r2.reviewMark->any(criterion.name='recommendation').mark.value) >=
        Conference.conflictGap )
```

3.2 The Behavioural Schema

The behavioural schema is the part of the conceptual schema that deals with dynamic aspects. This paper adopts the representation of events as entities as suggested by Olivé in [3]. Events may be classified in *domain events*, *action request events* and *query events* as shown in Figure 2.

A **domain event** is a state change that consists of a set of structural events that are perceived or considered as a single change in the domain.
An **action request** is a request to the IS to perform an action. The net effect of an action may be a change to the IB and/or the communication of some information or command to one or more recipients. Changes to the IB are performed only by means of domain events. An action changes the IB by generating one or more domain events.
A **query** is an external action that provides some information to the initiator of the action request.

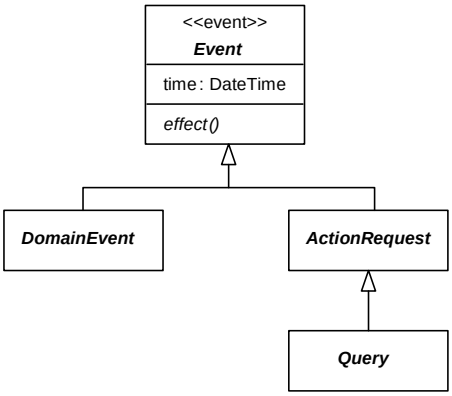

Fig. 2. Top-level taxonomy of event types

The modelling of the events, corresponding to functionalities summarized in Section 2, may be grouped in packages in order to ease understanding as shown in Figure 3:

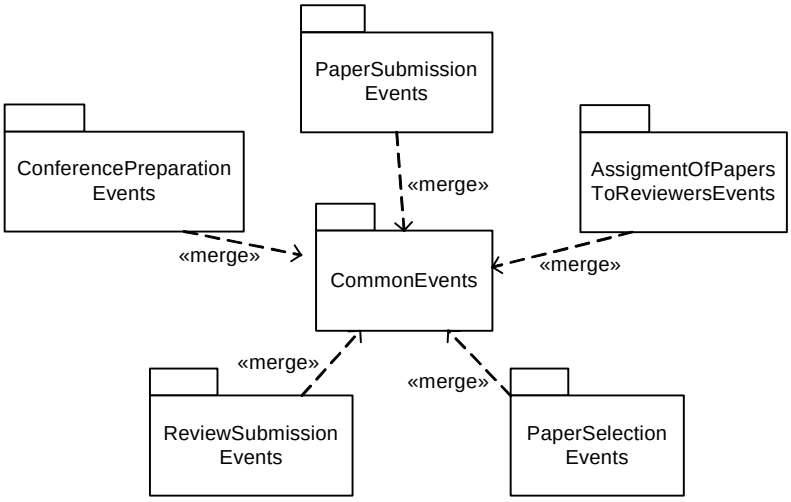

Fig. 3. Packages of the behavioural structure

3.2.1. *CommonEvents* Package

One of the advantages of event taxonomies is that event types with common characteristics, constraints, derivation rules and effect can be defined in a generalized event type, instead of repeating then in each event type.

The *CommonEvents* package defines the event types that generalize some of the events types defined on the other packages. In UML, a short-hand way of explicitly defining those generalizations is through package merge [5]. A package merge is a relationship between two packages, where the contents of the target package (in this case, the *CommonEvents* package) is merged with the contents of the source package through specialization and redefinition, where applicable. The *CommonEvents* package includes the top-level taxonomy described in Figure 2. The rest of events included in this package is defined below, shown in Figure 4, 5 and 6. The specification, in OCL, of the derivation rules and integrity constrains is shown below each figure. Additional details of these events are introduced where they arise.

Generalized Action Request Events

Figure 4 shows the following generalized events of action request events:

- *ExistingPCMemberRequest* is a generalized event type that applies to the instances of the following action request events: *ModifyPCMemberRequest* and *RemovePCMemberRequest* (see Figure 8), *SelectTopicsRequest* and *NewPreferencesRequest* (see Figure 17),

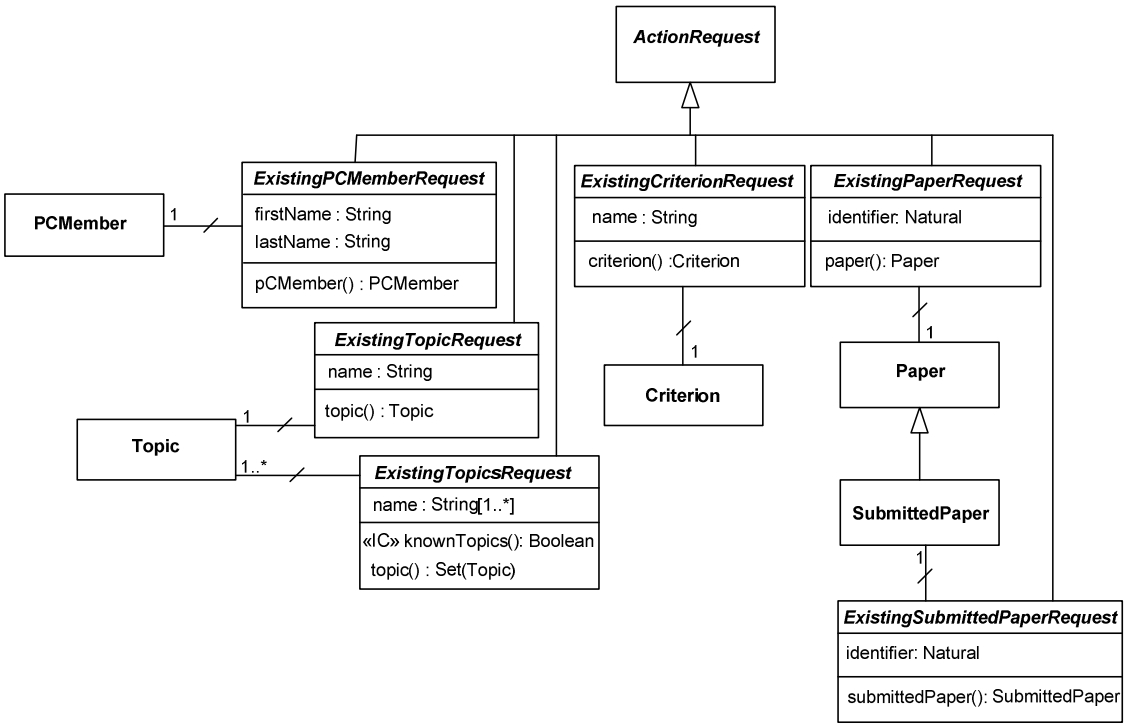

Fig. 4. Definition of the events that are generalizations of action request events types

- *SetAssignmentRequest* and *UnsetAssigmentRequest* (see Figure 19), *ReviewSubmissionRequest* (see Figure 21). There is an association between *ExistingPCMemberRequest* and an instance of *PCMember*, which may be derived from the *firstName* and *lastName* attributes. Additionally, the constraint that the PC member must exist has been expressed as a cardinality constraint in *ExistingPCMemberRequest.*
- *ExistingTopicRequest* is a generalized event type that applies to the instances of the following action request events: *ModifyTopicRequest* and *RemoveTopicRequest* (see Figure 9). There is an association between *ExistingTopicRequest* and an instance of *Topic*, which may be derived from the *name* attribute. Additionally, the constraint that the topic must exist has been expressed as a cardinality constraint in *ExistingTopicRequest.*
- *ExistingTopicsRequest* is a generalized event type that applies to the instances of the following action request events: *AbstractSubmissionRequest* (see Figure 13), *PaperSubmissionRequest* (see Figure 14) and *SelectTopicsRequest* (see Figure 17). There is an association between *ExistingTopicsRequest* and a set of instances of *Topic*, which may be derived from the *name* attribute. Additionally, the constraint that the topics must exist has been expressed in the integrity constraint *knownTopics.*
- *ExistingCriterionRequest* is a generalized event type that applies to the instances of the following action request events: *ModifyCriterionRequest* and *RemoveCriterionRequest* (see Figure 10). There is an association between *ExistingCriterionRequest* and *Criterion*, which may be derived from the *name* attribute. Additionally, the constraint that the criterion must exist has been expressed as a cardinality constraint in *ExistingCriterionRequest.*
- *ExistingPaperRequest* is a generalized event type that applies to the instances of the *PaperSubmissionRequest* action request (see Figure 14). There is an association between *ExistingPaperRequest* and *Paper*, which may be derived from the *identifier* attribute. Additionally, the constraint that the paper must exist has been expressed as a cardinality constraint in *ExistingPaperRequest.*
- *ExistingSubmittedPaperRequest* is a generalized event type that applies to the instances of the following action request events: *WithdrawalRequest* (see Figure 15), *SetAssigmentRequest* and *UnsetAssigmentRequest* (see Figure 19), *ReviewSubmissionRequest* (see Figure 21), *AcceptanceRequest* and *RejectionRequest* (see Figure 24) and *CameraReadySubmissionRequest* (see Figure 26). There is an association between *ExistingSubmittedPaperRequest* and *SubmittedPaper*, which may be derived from the *identifier* attribute. Additionally, the constraint that the criterion must exist has been expressed as a cardinality constraint in *ExistingSubmittedPaperRequest.*

The derivation rules and integrity constraints shown in Figure 4, in OCL, are the following:

```
context ExistingPCMemberRequest::pCMember():PCMember
body: PCMember.allInstances() ->any(self.firstName = firstName and
                                    self.lastName = lastName)

context ExistingTopicRequest::topic():Topic
body:  Topic.allInstances()->any(self.name=name)

«IC»
context ExistingTopicsRequest::knownTopics():Boolean
body:  Topic.allInstances().name -> includesAll(name)

context ExistingCriterionRequest::criterion():Criterion
body: Criterion.allInstances()->any(self.name = name)

context ExistingPaperRequest::paper():Paper
body: Paper.allInstances()->any(self.identifier = identifier)

context ExistingSubmittedPaperRequest::submittedPaper():SubmittedPaper
body: SubmittedPaper.allInstances()->any(self.identifier = identifier)
```

Generalized Query

CMA provides several functionalities to examine the abstracts and papers which have been submitted. Since the number of papers may be quite large, some criteria can be used to restrict the papers which are examined. These restrictions may be applied to different queries, therefore it includes several criteria (related to reviews or reviewers) that are not useful in the early phase of paper submission. The criteria have been defined as the following attributes in *SubmittedPaperQuery*: *titleContains* and *authorContains* to full text search on title and author names, *status* that may be 'accepted', 'rejected', 'pending' or 'review', *nameReviewerContains* which allows to find all the papers assigned to a given reviewer, *topicName* which allows to find all the papers associated to a topic and *withConflict* which allow to find all the papers with conflicting reviews. *SubmittedPaperQuery* applies to the instances of the following events: *ListSubmittedPapers* (see Figure 18) and *ListReviews* (see Figure 22)

Figure 5 shows the generalized query event.

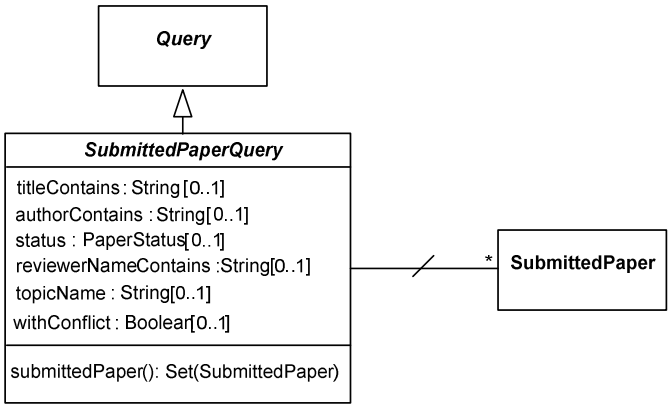

Fig. 5. Definition of SubmittedPaperQuery

The specification of the *submittedPaper* operation shown in Figure 5 and the specification of the operation *includesSubString()* defined in *String*, are the following:

```
context String:: includesSubString(sub:String): Boolean
body: self->exists (i:Integer, j:Integer| i<j and substring(i,j) = sub)
```

```
context SubmittedPaperQuery::submittedPaper():SubmittedPaper
body: let byTitle: Set(SubmittedPaper) =
        SubmittedPaper.allInstances()->
             select(sp| sp.title-> includesSubString(titleContains))
      let byAuthor: Set(SubmittedPaper) =
        SubmittedPaper.allInstances()->
             select(sp| sp.author.allInstances -> firstName->
             includesSubString(authorContains) or
             lastName->includesSubString(authorContains))
      let byStatus: Set(SubmittedPaper):
        SubmittedPaper.allInstances()-> select(sp| sp.status=status)
      let byReviewer: Set(SubmittedPaper) =
        SubmittedPaper.allInstances()->
             select(sp| sp.reviewer.allInstances -> firstName->
             includesSubString(nameReviewerContains) or
             lastName->includesSubString(nameReviewerContains))
      let byTopic: Set(SubmittedPaper) =
        SubmittedPaper.allInstances()->select(sp| sp.topic -> name= topicName)
      let byConflict: Set(SubmittedPaper) =
        SubmittedPaper.allInstances()->
             select(sp| sp.hasConflictingReviews)
      in
      SubmittedPaper.allInstances()->asSet()->
      (if titleContains->notEmpty() then intersection(byTitle) endif)->
      (if authorContains->notEmpty() then insersection(byAuthor) endif)->
      (if status->notEmpty() then intersection(byStatus) endif)->
      (if nameReviewerContains->notEmpty() then intersection(byReviewer)
      endif) -> (if topicName->notEmpty() then intersection(byTopic) endif) ->
      (if withConflict->notEmpty() then intersection(byConflict) endif)
```

Generalized Domain Events

Figure 6 shows the following generalized events of domain event types:

- *SubmissionPhase* is a generalized event type that applies to the instances of *AbstractSubmission* (see Figure 13) and *PaperSubmission* (see Figure 14) domain events. *SubmissionPhase* includes the integrity constraint *submissionOpen* that ensures that any abstract or paper may only be submitted if the submission phase is open.
- *ReviewPhase* is a generalized event type that applies to the instances of *SetAssigment* (see Figure 19), *ReviewSubmission* (see Figure 21), and *Acceptance* and *Rejection* (see Figure 24) domain events. *ReviewPhase* includes the integrity constraint *reviewOpen* that ensures that any assignment, review, acceptance or rejection of a paper may only be submitted if the review phase is open.
- *CameraReadyPhase* is a generalized event type that applies to the instances of *CameraReadySubmission* (see Figure 26) domain event. Again, *CameraReadyPhase* includes the integrity constraint *cameraReadyOpen* that ensures that any camera-ready paper may only be submitted if the camera-ready phase is open.
- *PhaseOpening* is a generalized event type that applies to the instances of *OpenSubmission* (see Figure 12), OpenReview (see Figure 16) and OpenCameraReady (see Figure 25) domain events. *PhaseOpening* includes the integrity constraint *deadlineAfterNow* that ensures that the deadline of the opening phase, i.e., submission, review or camera-ready phase, is at any time after its creation.

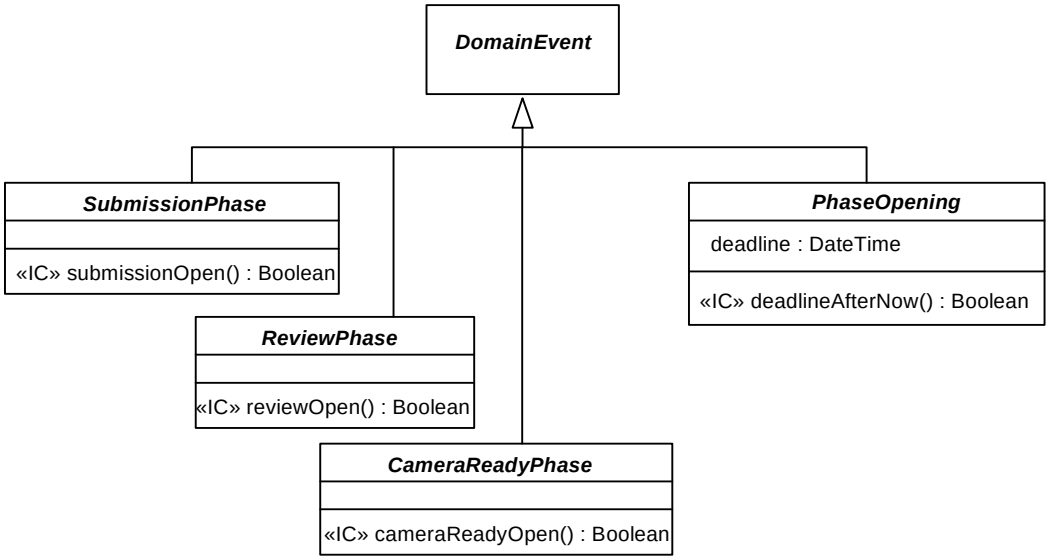

Fig. 6. Definition of the events that are generalizations of domain events types

The specifications of the event types shown in Figure 6 are the following:

```
«IC»
context SubmissionPhase::submissionOpen:Boolean
body: Conference.submissionOpen

«IC»
context ReviewPhase::reviewOpen:Boolean
body: Conference.reviewOpen

«IC»
context CameraReadyPhase::cameraReadyOpen:Boolean
body: Conference.cameraReadyOpen

«IC»
context PhaseOpening::deadlineAfterNow: Boolean
body:  deadline > time + 1
```

3.2.2. *ConferencePreparationEvents* Package

The *ConferencePreparationEvent* package includes all that happen during the preparation of the conference: setting the parameters of the new conference; creating, modifying and removing new PC members, topics and criteria, and telling PC members the initial instructions. From Figure 7 to Figure 11 the definition of these events is shown. The specification of the effects, in OCL, of each event is described below each figure.

Conference Configuration

Figure 7 shows the definition of *NewConference* domain event type, whose purpose is to configure the new conference with all the information related to the conference. This includes: the name of the conference, its acronym, the url of the submission site, etc (see *Conferece* of the structural schema for more details). Note that the *submissionOpen, reviewOpen, cameraReadyOpen* and *lastPaperIdentifier* attributes of *Conference* don't need to be settled since they are set to a value by default in the definition of *Conference,* as shown in Figure 1.

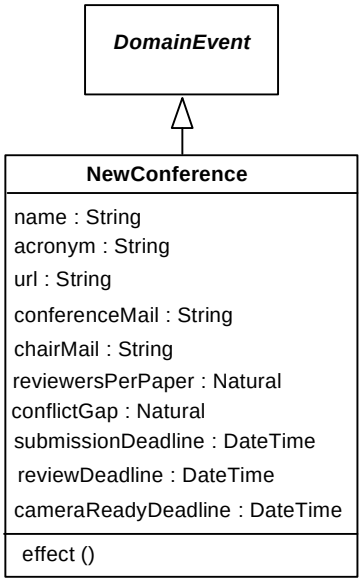

Fig. 7. Definition of *NewConference* domain event

The specification of the *NewConference* event shown in Figure 7 is the following:

```
context NewConference::effect()
post:  Conference.name = name and
       Conference.acronym = acronym and
       Conference.url = url and
       Conference.conferenceMail = conferenceMail and
       Conference.chairMail = chairMail and
       Conference.reviewersPerPaper = reviewersPerPaper and
       Conference.conflictGap = conflictGap and
       Conference.submissionDeadline = submissionDeadline and
       Conference.reviewDeadline =reviewDeadline and
       Conference.cameraReadyDeadline = cameraReadyDeadline
```

Program Committee

The program committe is a list of *members*. Figure 8 shows the definition of the events related to the creation, modification, removal and listing of PC members:

- *NewPCMember* domain event type, whose purpose is the creation of new instances of *PCMember,*
- *ModifyPCMember* domain event type in order to modify some data of an instance of *PCMember*,
- *RemovePCMember* domain event, whose effect is the removal of an instance of *PCMember*,
- *ModifyPCMemberRequest* and *RemovePCMemberRequest* action request, whose effect is to generate the *ModifyPCMember* and *RemovePCMember* domain events respectively and
- *ListPCMembers* query which gives some information of instances of *PCMember* ordered by last name.

Note that *RemovePCMember* includes the integrity constraint *withoutPapers* to avoid the removal of a PC member if there are papers assigned to her.

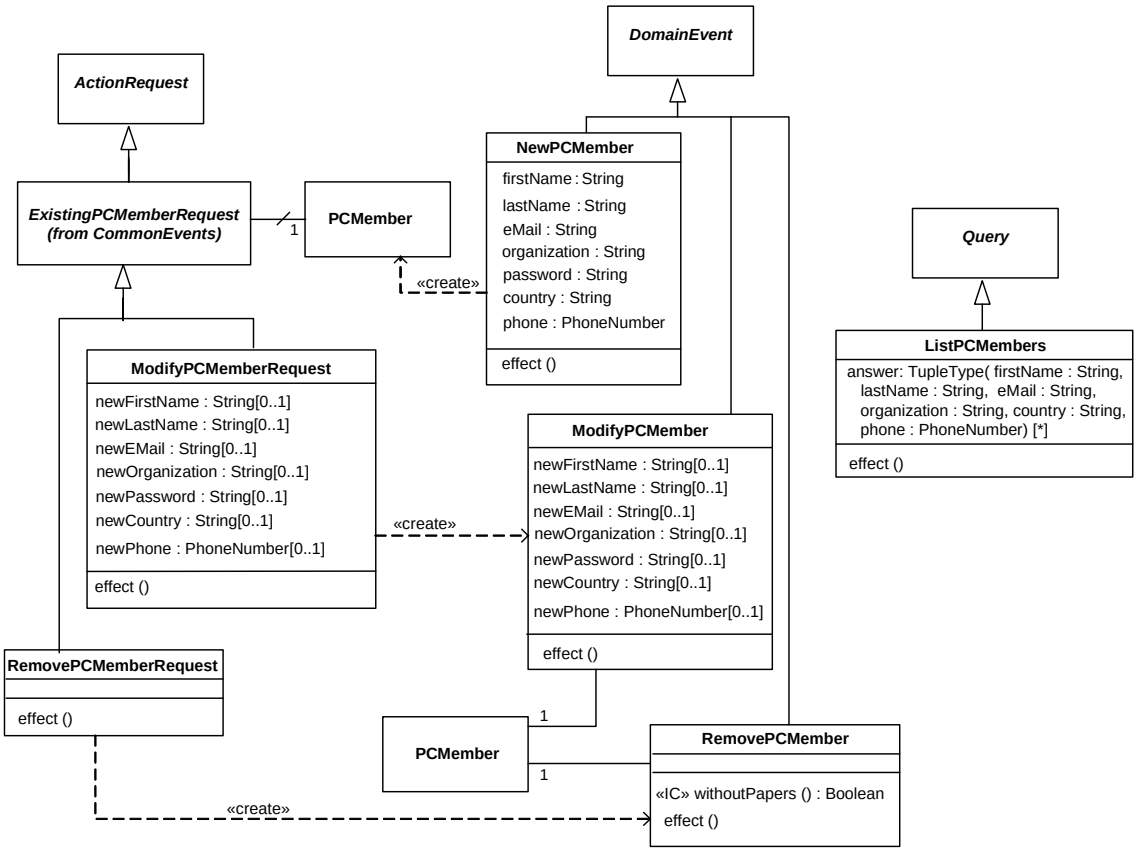

Fig. 8. Definition of the events related to the creation, modification, removal and query of PC members

The specifications of the event types shown in Figure 8 are the following:

```
context ModifyPCMemberRequest::effect()
post Generate a domain event ModifyPCMember:
      m.oclIsNew() and m.oclIsTypeOf(ModifyPCMember) and
      m.pCMember = pCMember and m.newFirstName = newFirstName and
      m.newLastName = newLastName and m.newEMail = newEMail and
      m.newOrganization = newOrganization and m.newPassword = newPassword and
      m.newCountry = newCountry and m.newPhone = newPhone

context RemovePCMemberRequest::effect()
post Generate a domain event RemovePCMember:
      r.oclIsNew() and r.oclIsTypeOf(RemovePCMember) and
      r.PCMember = pCMember

context NewPCMember::effect()
post:  np.oclIsNew() and np.oclIsTypeOf(PCMember) and
      np.firstName = firstName and np.lastName = lastName and
      np.email = email and np.organization = organization and
      np.password = password and np.country = country and np.phone = phone

context ModifyPCMember::effect()
post:  if newFirstName<>null then pCMember.firstName = newFirstName endif and
      if newLastName<>null then pCMember.lastName = newLastName endif and
      if newOrganization<>null then pCMember.organization = newOrganization
      end if and
      if newPassword<>null then pCMember.password = newPassword endif and
      if newCountry<>null then pCMember.country = newCountry endif and
      if newPhone<>null then pCMember.phone = newPhone endif

context RemovePCMember::effect()
post:  PCMember.allInstances() -> excludes(pCMember)
```

```
«IC»
context RemovePCMember::withoutPapers():Boolean
post: pCMember.assignedPaper -> isEmpty()

context ListPCMembers::effect()
post:  answer = PCMember.allInstances-> sortedBy(lastName)->
        collect (pc| Tuple{firstName=pc.firstName, lastName=pc.lastName,
                           eMail=pc.eMail, organization=pc.organization,
                           country=pc.country, phone=pc.phone})
```

Topics

CMA allows registering, modifying, removing and list a set of research topics. Figure 9 shows the definition of the following events:

- *NewTopic* domain event type, whose purpose is to create a new instance of *Topic*,
- *ModifyTopic* domain event type in order to change the name of an existing topic,
- *RemoveTopic* domain event type in order to remove an existing topic,
- *ModifyTopicRequest* and *RemoveTopicRequest* action request, whose effect is to generate a *ModifyTopic* and *RemoveTopic* domain event respectively and
- *ListTopics* query that gives the names of the instances of *Topic*.

Note that *ModifyTopic* and *RemoveTopic* domain events include the integrity constraint u*nreferenced* to ensure that a topic that has already been referenced by a paper or selected as a topic of interest by a PC member is not modified or removed.

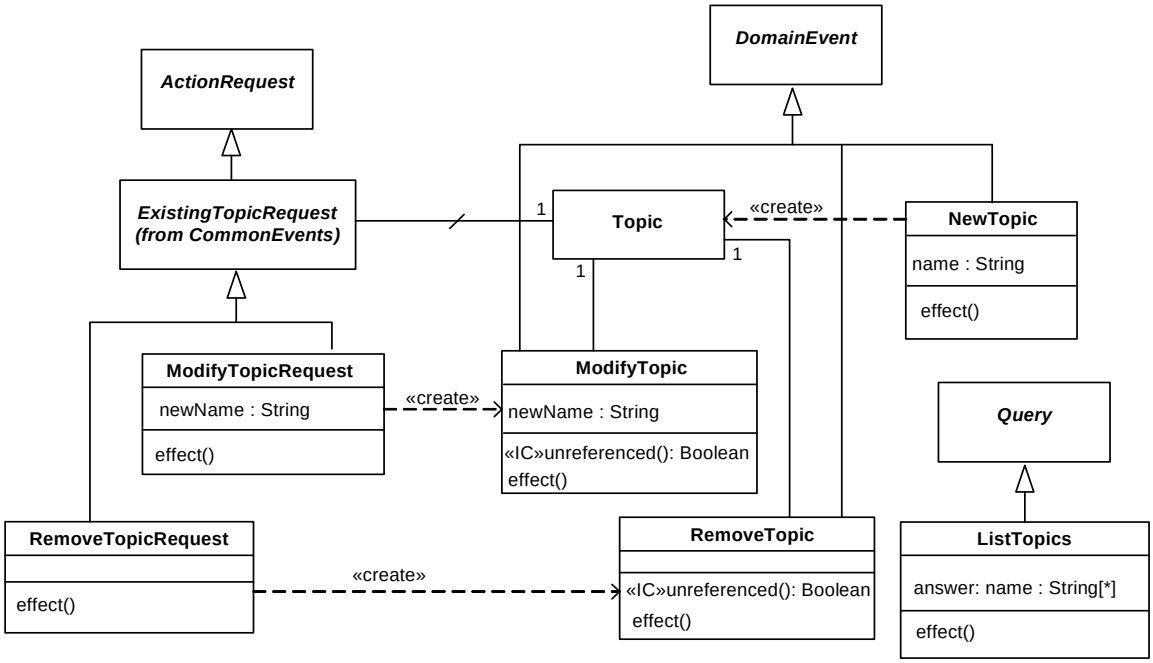

Fig. 9. Definition of the events related to the creation, modification, removal and query of topics

The specifications of the event types shown in Figure 9 are the following:

```
context ModifyTopicRequest::effect()
post Generate a domain event ModifyTopic:
      mt.oclIsNew() and mt.oclIsTypeOf(ModifyTopic) and mt.topic = topic and
      mt.newName = newName

context RemoveTopic::effect()
post Generate a domain event RemoveTopic:
      rt.oclIsNew() and rt.oclIsTypeOf(RemoveTopic) and rt.topic = topic

context NewTopic::effect()
post:  t.oclIsNew() and t.oclIsTypeOf(Topic) and t.name = name
```

```
context ModifyTopic::effect()
post:  topic.name = newName

«IC»
context ModifyTopic::unreferenced():Boolean
body: topic.paper->isEmpty() and topic.pCMember->isEmpty()

context RemoveTopic::effect()
post:  Topic.allInstances() -> excludes(topic)

«IC»
context RemoveTopic::unreferenced():Boolean
body: topic.paper->isEmpty() and topic.pCMember->isEmpty()

context ListTopics::effect()
post:  Topic.allInstances -> sortedBy(name)-> collect(name)
```

Criteria

CMA allows registering, modifying, removing and list a set of criteria used to evaluate a submitted paper. Figure 10 shows the definition of the following events:

- *NewCriterion* domain event type, whose purpose is the creation of a new instance of *Criterion*,
- *ModifyCriterion* domain event type to change the name or/and weight of an existing criterion,
- *RemoveCriterion* domain event type in order to remove an existing criterion,
- *ModifyCriterionRequest* and *RemoveCriterionRequest* action request, whose effect is to generate a *ModifyCriterion* and *RemoveCriterion* domain events respectively and
- *ListCriterion* query which gives the names and weights of the instances of *Criterion* created ordered by name.

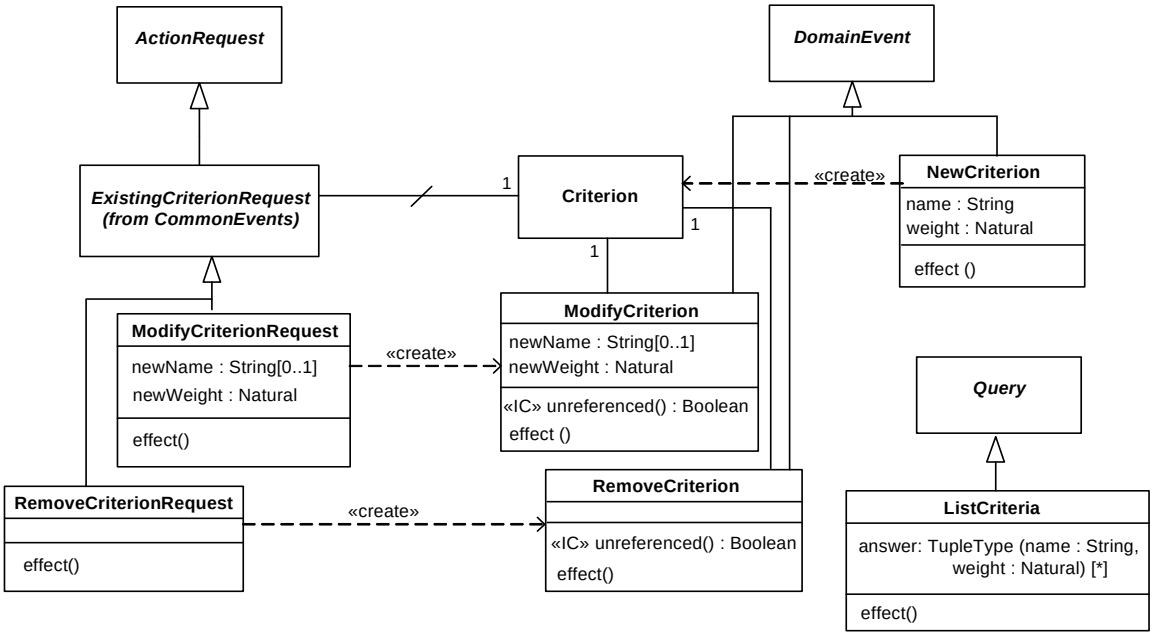

Fig. 10. Definition of the events related to the creation, modification, removal and query of criteria

Note that *ModifyCriterion* and *RemoveCriterion* domain events include the integrity constraint *unreferenced* to ensure that a criterion that has already been used in a review is not modified or removed. There is an association between *ExistingCriterionRequest* (from *CommonEvents*) and *Criterion*, which may be derived from the *name* attribute and it applies to *ModifyCriterionRequest* and *RemoveCriterionRequest*. Additionally, the constraint that the criteria must exists has been expressed as a cardinality constraint in *ExistingCriterionRequest*.

The specifications of the event types shown in Figure 10 are the following:

```
context ModifyCriterionRequest::effect()
post Generate a domain event ModifyCriterion:
       mc.oclIsNew() and mc.oclIsTypeOf(ModifyCriterion) and
       mc.criterion = criterion and
       if newName<>null then mc.newName = newName endif and
       c.newWeight = newWeight

context RemoveCriterion::effect()
post Generate a domain event RemoveCriterion:
       rc.oclIsNew() and rc.oclIsTypeOf(RemoveCriterion)
       and rc.criterion = criterion

context NewCriterion::effect()
post:  c.oclIsNew() and c.oclIsTypeOf(Criterion) and c.name = name and
       c.weight = weight

context ModifyCriterion::effect()
post:  if newName->notEmpty() then criterion.name = newName endif and
       criterion.weight = newWeight

«IC»
context ModifyCriterion::unreferenced():Boolean
body: criterion.review->isEmpty()

context RemoveCriterion::effect()
post:  Criterion.allInstances() -> excludes(criterion)

«IC»
context RemoveCriterion::unreferenced():Boolean
body: criterion.review->isEmpty()

context ListCriteria::effect()
post:  Criterion.allInstances -> sortedBy(name) -> collect(c|
                          Tuple{name=c.name, weight=c.weight})
```

Mailing Instructions

Figure 11 shows the definition of:

- *InstructionsNotification* action request to send the initial instructions to all reviewers. The effect of the action request *InstructionsNotification* is the invocation of the operation *sendEmail()* of *Person*.

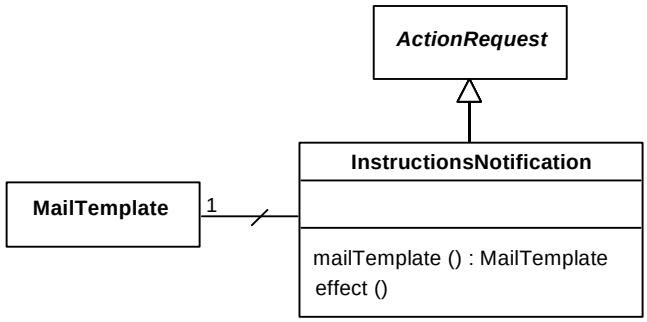

Fig. 11. Definition of the *InstructionsNotification* action request

The specifications of the operations of the event types shown in Figure 11 are the following:

```
context InstructionsNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          PCMembers.allInstances() -> forAll (p |p^sendEmail(subject, body))
```

```
context InstructionsNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::initialInstructions)
```

3.2.3. *PaperSubmissionEvents* Package

The *PaperSubmissionEvents* Package includes all that happens during the papers submission phase.

Once the phase submission is opened, authors are allowed to submit abstracts and full papers. The paper submission is organized in two steps: first authors must submit an abstract, along with some basic information concerning their papers, and then, the file of the paper must be uploaded. Authors may also ask for the withdrawal of a submitted paper if it has not been already completely reviewed. At any time, the chairman of the conference may lists all the submitted papers.

Submission Phase

Figure 12 shows the definition of:

- *OpenSubmission* domain event, whose effect is to open the submission phase, that is, to set to *true* the attribute *openSubmission* of *Conference* and to set a time point as the deadline of paper submission.

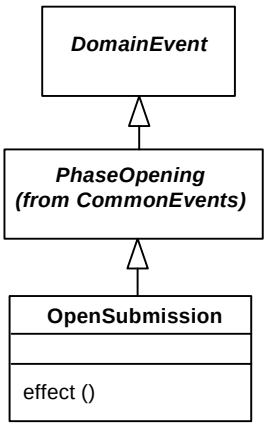

Fig. 12. Definition of the *OpenSubmission* domain event

The specification of the *effect* operation shown in Figure 12 is the following:

```
context OpenSubmission::effect()
post:  Conference.submissionOpen = True and
       Conference.submissionDeadline = deadline
```

Abstract Submission

In order to submit an abstract, the contact author must fill the required information for the submission (title, keyword, abstract, topics, and for each author, his or her first name, last name, and organization; for the contact author the email, organization, country and phone number must be included). Submitting an abstract implies the creation of a new paper and additionally, the contact author receives an acknowledgment of their submission, along with the new paper identification and a password. This identification and password can be used during all the submission phase to modify some of the information submission (abstract, keywords, topic and comments) and the paper file itself.

Figure 13 shows the definition of:

- *AbstractSubmission* domain event type, whose effect is the creation of a new instance of *Paper,*
- *AbstractSubmissionRequest* action request, whose effect is to generate an *AbstractSubmission* domain event and an *AbstractSubmissionNotification* action request and
- *AbstractSubmissionNotification* action request, whose effect is to send an acknowledgment of their submission to the contact author, along with the paper identifier and the password through the invocation of the operation *sendEmail()* of *Person*.

Note that *AbstractSubmission* includes the integrity constraint *onlyOneContactAuthor* to ensure that only one of the authors has been marked as the contact author for mailing purposes.

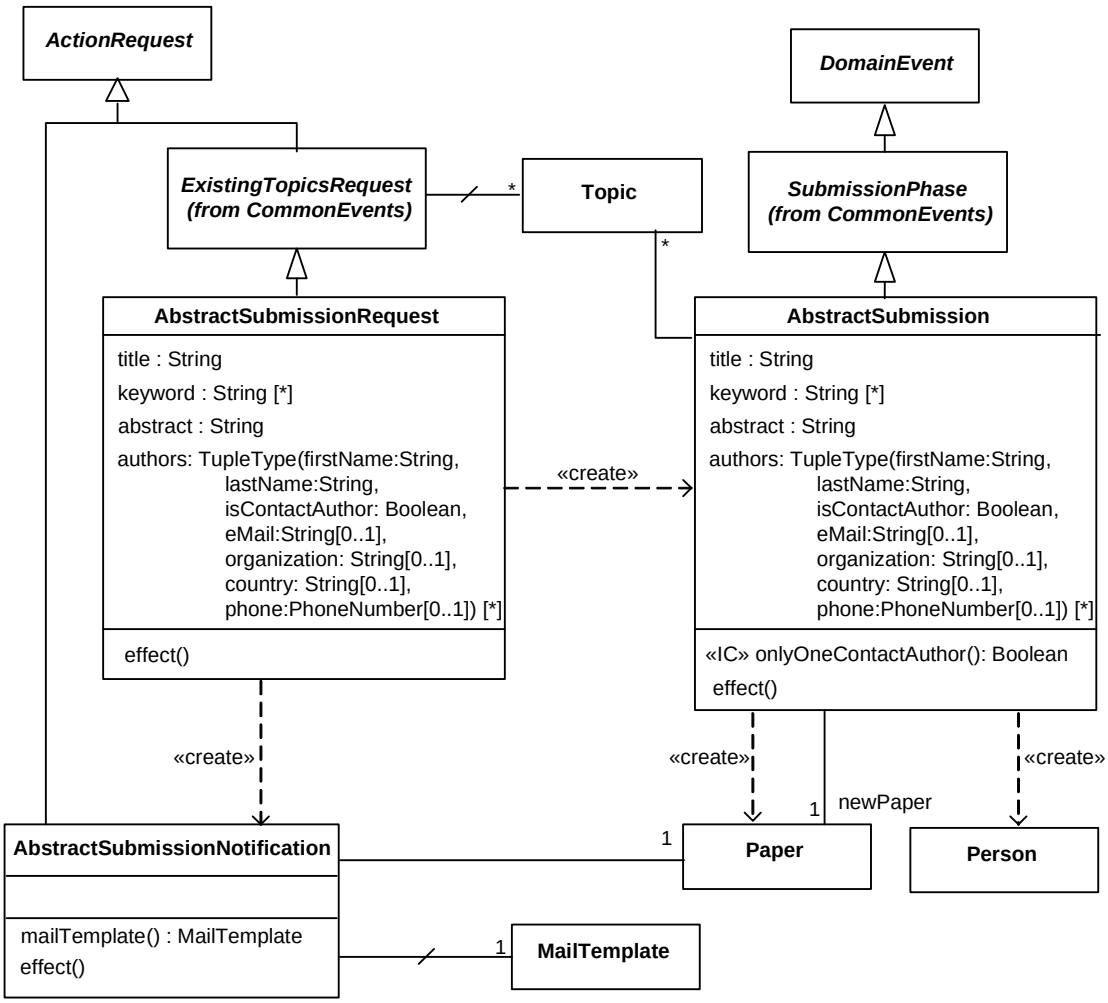

Fig. 13. Definition of the events related to an abstract submission

The specifications of the operations of the event types shown in Figure 13 are the following:

```
context AbstractSubmissionRequest::effect()
post Generate a domain event AbstractSubmission:
       ab.oclIsNew() and ab.oclIsTypeOf(AbstractSubmission) and
       ab.title  = title and
       ab.keyword = keyword and
       ab.abstract = abstract and
       ab.topic = topic and
       and ab.authors = authors and
       -- Generate an action request AbstractSubmissionNotification
       s.oclIsNew() and s.oclIsType(AbstractSubmissionNotification) and
       s.paper = ab.newPaper

context AbstractSubmission::effect()
post:  p.oclIsNew() and p.oclIsTypeOf(Paper) and
       Conference.lastPaperIdentifier = Conference.lastPaperIdentifier@pre+1
       and p.identifier = Conference.lastPaperIdentifier and
       p.password = **** and p.title = title and p.abstract = abstract and
       p.keyword = keyword and
       p.topic=topic and
       self.authors –> forAll (a|
       if Person.allInstances()->exists(aut:Person|
              aut.firstName=a.firstName and aut.lastName=a.lastName)
       then -- the person already exists
```

```
            per = Person.allInstances()-> any (aut:Person|
             aut.firstName=a.firstName and aut.lastName=a.lastName)
      else
           -- create a new instance of Person
            per.oclIsNew() and per.oclIsTypeOf(Person) and per.firstName =
            a.firstName and per.lastName = a.lastName
      endif
      p.author->includes(per)
      -- contact author
      if a.isContactAuthor then
         if ContactLocation.allInstances()->exists(ca:ContactLocation|
            ca.eMail =a.e.Mail)
         then
            ca.paper ->includes(p)
         else
            co.oclIsNew() and co.oclIsTypeOf(ContactLocation) and
            co.eMail = a.eMail and co.organization = a.organization and
            co.country = a.country and co.phone = a.phone and
            co.paper->includes(p) and co.contactAuthor->includes(per)
         endif
      endif)

«IC»
context AbstractSubmission::onlyOneContactAuthor():Boolean
body: self.authors –> select(a.isContactAuthor)->size=1

context AbstractSubmissionNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject)
       let body:String = mailTemplate().text in
          paper.contactAuthor^sendEmail(subject, body)

context AbstractSubmissionNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::abstractSubmission)
```

Paper Submission

A paper may be submitted several times and a submission replaces the previous version, if it exits; this is, if the title, password, abstract, or keywords are given, they replace the previous ones. However, if topics or authors are specified, they are just added (not replaced) to the previous ones.

Figure 14 shows the definition of the following events:

- *PaperSubmission* domain event type, whose effect is that the paper that was previously created becomes an instance of *SubmittedPaper* with the corresponding attributes,
- *PaperSubmissionRequest* action request, whose effect is to generate an instance of *PaperSubmission* domain event and an instance of *PaperSubmissionNotification* request and
- *PaperSubmissionNotification* request which, similarly to *AbstractSubmissionNotification,* sends the acknowledgement of the submission of the paper to the contact author.

Note that *PaperSubmissionRequest* includes the constraint *knownPassword* that checks that the password introduced is the same that was sent to the contact author.

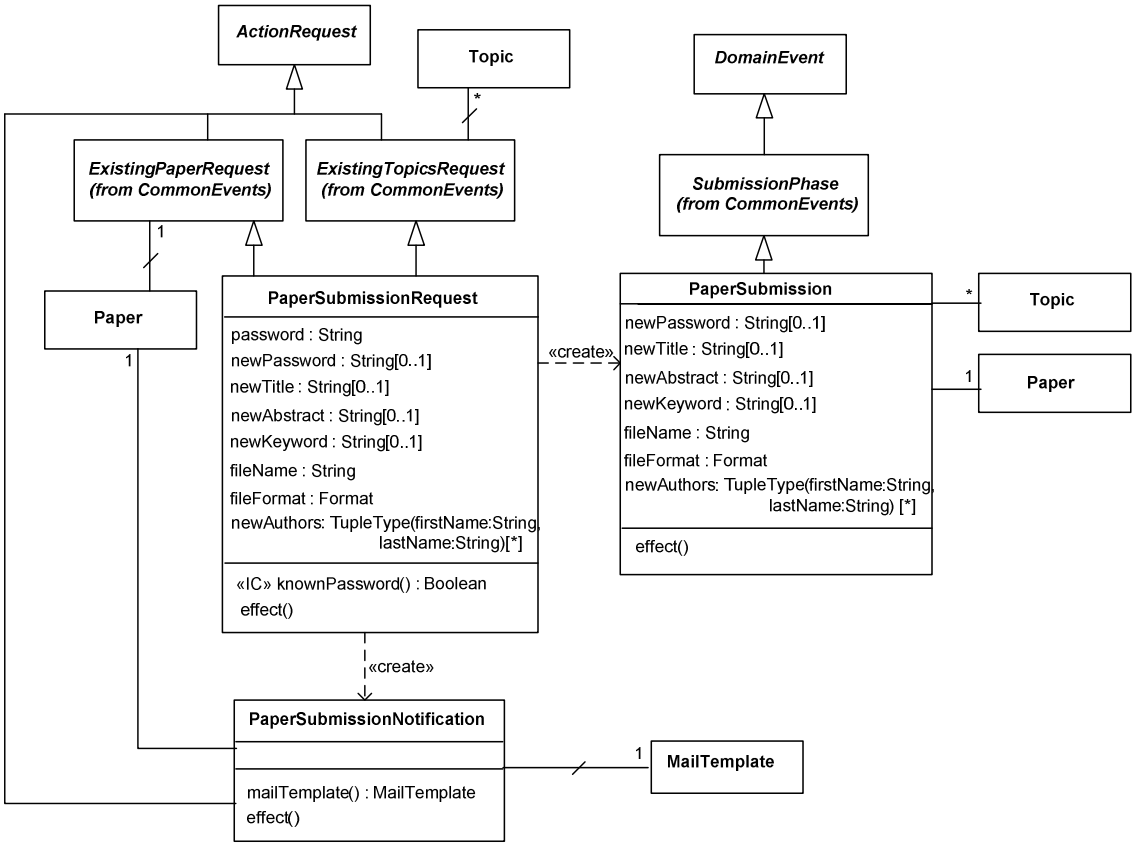

Fig. 14. Definition of the events related to the paper submission

The specifications of the operations of the event types shown in Figure 14 are the following:

```
context PaperSubmissionRequest::effect()
post Generate a domain event PaperSubmission:
       ps.oclIsNew() and ps.oclIsType(PaperSubmission) and ps.topic = topic and
       ps.paper = paper and ps.newPassword = newPassword and
       ps.newTitle = newTitle and ps.newKeyword = newKeyword and
       ps.newAbstract = newAbstract and ps.fileName=fileName and
       ps.fileFormat=fileFormat and ps.newAuthors = newAuthors
post Generate an action request PaperSubmissionNotification:
       psn.oclIsNew() and psn.oclIsType(PaperSubmissionNotification) and
       psn.paper = paper

«IC»
context PaperSubmissionRequest::knownPassword:Boolean
body   self.password = paper.password

context PaperSubmission::effect()
post:  newPassword->notEmpty() implies paper.password = newPassword and
       newTitle->notEmpty() implies paper.title = newTitle and
       newAbstract->notEmpty() implies paper.abstract = newAbstract and
       newKeyword->notEmpty() implies paper.keyword = newKeyword and
       paper.topic->includesAll(self.topic) and
       paper.oclIsTypeOf(SubmittedPaper) and
       paper.oclAsType(submittedPaper).fileName = fileName and
       paper.oclAsType(submittedPaper).fileFormat = fileFormat and
       paper.oclAsType(submittedPaper).submissionDate = time and
       if newAuthors->notEmpty() then forAll (a|
           if exists(aut:Person|a.firstName = aut.firstName and
              a.lastName =aut.lastName) then
              paper.author= Person.allInstances()
              ->any(aut:Person|a.firstName = aut.firstName and
```

```
            a.lastName =aut.lastName)
          else
            per.oclIsNew() and per.oclIsTypeOf(Person) and per.firstName =
            a.firstName and per.lastName = a.lastName and
            paper.author ->includes(per)
       endif

context PaperSubmissionNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          paper.contactAuthor^sendEmail(subject, body)

context PaperSubmissionNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::paperSubmission)
```

Paper Withdrawal

Users may request to withdraw a submitted paper. When this request is received, the status of the submitted paper has to change to "withdrawn", the paper has to be removed from its reviewers (if any) and the reviewers have to be informed that they do not have to review the paper.

Figure 15 shows the definition of the following events:

- *PaperWithdrawal* domain event type, whose effect is to withdraw a paper (to change its status and remove from assigned reviewers) that was previously submitted and its status is still pending,
- *WithdrawalRequest* action request, whose effect is to to generate an instance of *PaperWithdrawal* domain event and an instance of *WithdrawalNotification* request and
- *WithdrawalNotification* request that informs the assigned reviewers to such a paper that they do not have to review the paper any more.

Note that *WithdrawalRequest* includes the constraint *knownPassword* that checks that the password introduced is the same that was sent to the contact author.

Note also, that the events related to the withdrawal of a paper are not exclusively of the submission phase of the conference they may also occur in the review phase of the conference.

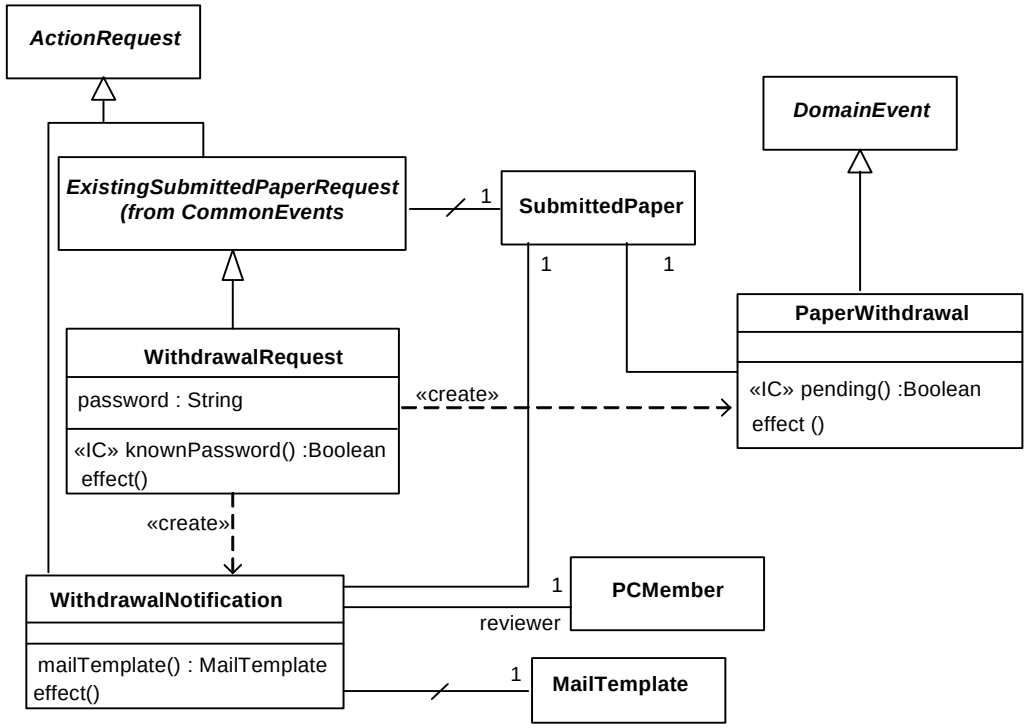

Fig. 15. Definition of the events related to the withdrawal of a paper

The specifications of the event types shown in Figure 15 are the following:

```
context WithdrawalRequest::effect()
  post   Generate a domain event PaperWithdrawal:
       pw.oclIsNew() and pw.oclIsTypeOf(PaperWithdrawal) and
       pw.submittedPaper = submittedPaper
  post   Inform the pending reviewers:
       paper.review@pre -> forAll(r|
          not r.completed implies
              wn.oclIsNew() and wn.oclIsTypeOf(WithdrawalNotification) and
              wn.reviewer = r and
              wn.submittedPaper = submittedPaper)

«IC»
context WithdrawalRequest::knownPassword:Boolean
body   self.password = paper.password

context PaperWithdrawal::effect()
    post: submittedPaper.status = PaperStatus::withdrawn and
          submittedPaper.reviewer-> isEmpty()

«IC»
context PaperWithdrawal::pending():Boolean
  body: submittedPaper.status = PaperStatus::pending

context WithdrawalNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
       reviewer^sendEmail(subject, body)

context WithdrawalNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::paperWithdrawal)
```

3.2.4. *AssignmentOfPapersToReviewersEvents* Package

The *AssignmentOfPapersToReviewersEvents* package includes all that happens in the assignment of papers to reviewers. This is, when the deadline of the submission phase is met the submission phase is closed and the review phase may open. On the review phase all submitted papers must be assigned to the reviewers. This is difficult and time-consuming task but CMA provides the following functionalities:

- Manual assignment. The organizer may set or unset assignments directly after consulting the preferences of reviewers in order to decide for any assignment.
- Automatic assignment. The assignment may be done automatically by using an automatic algorithm which computes the best possible assignment, given the rating of reviewers to papers. This requires that reviewers provide preference *rates* for all set of submitted papers.

Review Phase

Figure 16 defines the the following events:

- *CloseSubmision* domain event, whose effect is to close the submission phase, that is, to set to *false* the attribute *openSubmission* and consequently no more submissions of papers are allowed and
- *OpenReview* domain events, whose effect is to open the review phase, that is, to set to *true* the attribute *openReview* of *Conference* and to set a time point as the deadline to submit reviews.

OpenReview domain event includes the integrity constraint *recommentationCriteriaDefined* that ensures that the 'recommendation' criterion has been defined when opening the review phase.

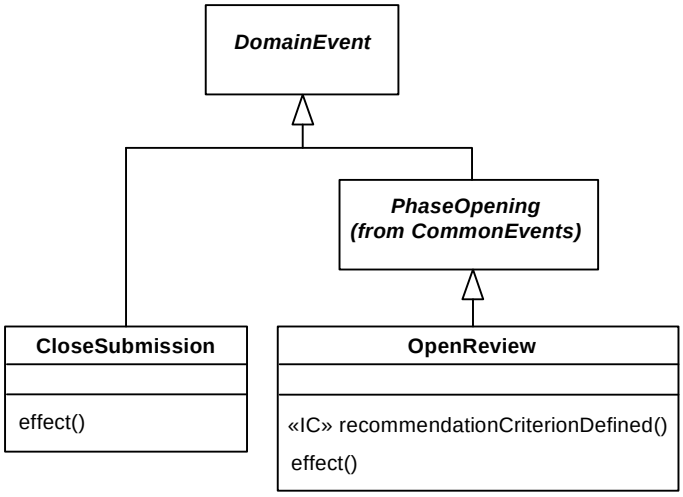

Fig. 16. Definition of the *CloseSubmission* and *OpenReview* domain event types

The specifications of the operations of the event types shown in Figure 16 are the following:

```
context CloseSubmission::effect()
post:  Conference.submissionOpen = False

context OpenReview::effect()
post:  Conference.reviewOpen = True and
       Conference.reviewDeadline = deadline

context OpenReview:recommentationCriterionDefined::Boolean
body:   Criterion.name->includes('recommendation')
```

Reviewers' Preferences

Manual and automatic assignment of papers to reviewers should be based on the reviewers' preferences. Figure 17 shows the definition of the events related to the collecting of topics of interest and preferences for reviewing any paper:

- *SelectTopics* domain event type, whose effect is to create associations between an instance of *PCMember* a set of instances of *topicOfInterest*,
- *NewPreference* domain event type, whose effect is the creation of an instance of *Preference*, that is the *rate* that a PC member gives for a given paper,
- *SelectTopicsRequest* and *NewPreferencesRequest* action request, whose effect is to generate instances of *SelectTopics* and *NewPreference* domain events respectively,
- *AskingPreferencesRequest* action request, whose effect is to send an email to each PC member, asking her or him rate preference for each paper,
- *ListTopicsOfInterest* query which gives the list, for each PC member, of his or her topics of interest and
- *ListReviewerPreferences* query which gives the list, for each PC member, of his or her reviewing preferences about all papers.

Note that *NewPreferences* generates a set of instances of *NewPreference* domain event, i.e., one for each tuple of *preferences* attribute.

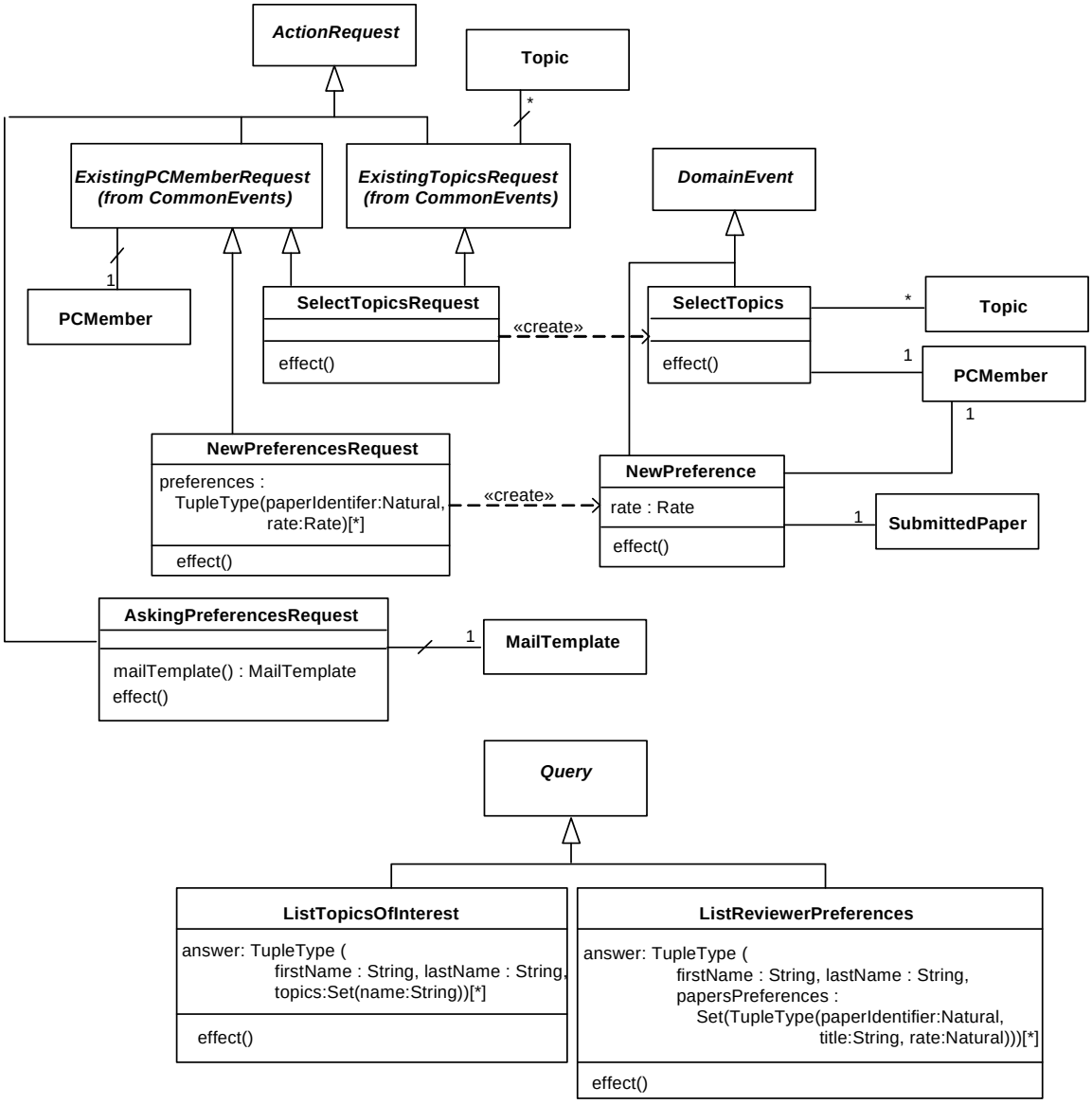

Fig. 17. Definitions of the events previous to the assignment

The specifications of the event types shown in Figure 17 are the following:

```
context AskingPreferencesRequest::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          PCMember.allInstances() -> forAll (pc |
              pc^sendEmail(subject, body))

context NewPreferencesRequest::effect()
post   Generate a set of domain events NewPreference:
       preferences -> forAll(p |
       pref.oclIsNew() and pref.oclIsTypeOf(NewPreference) and
       pref.submittedPaper= SubmitedPaper.allInstances() ->
             any(identifier=p.paperIdentifier) and pref.pCMember = pCMember)

context SelectTopicsRequest::effect()
post Generate a domain event SelectTopics:
       top.oclIsNew() and top.oclIsTypeOf(SelectTopics) and top.topic=topic and
       top.pCMember = pCMember


context SelectTopics::effect()
```

```
post: pCMember.topicsOfInterest -> includesAll(topic)

context NewPreference::effect()
post: pr.oclIsNew() and pr.oclIsTypeOf(Preference) and
       pr.ratingReviewer->include(pCMember) and
       pr.ratedPaper->include(submittedPaper) and pr.rate = rate
```

Submitted Papers

Figure 18 shows the definition of:

- *ListSubmittedPapers* query which gives some information of all the submitted papers. This information may be useful to make the manual assignment..

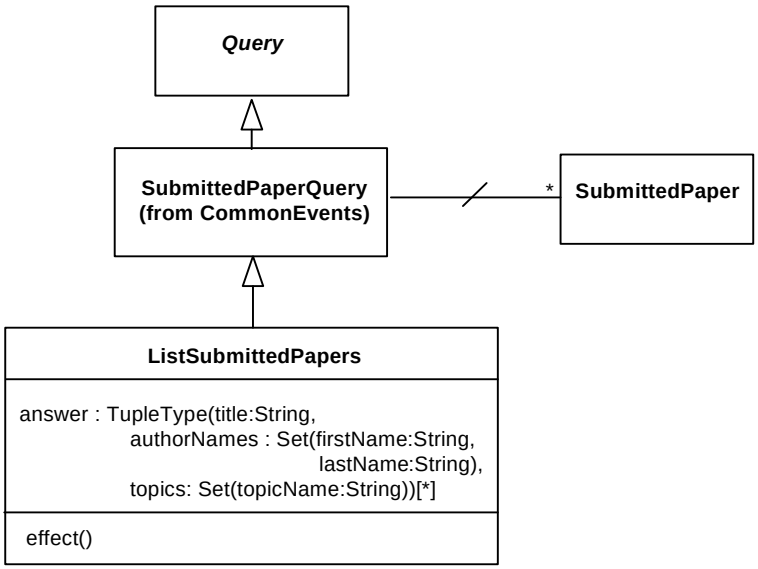

Fig. 18. Definition of *ListSubmittedPapers* query

The specifications of the effect operation shown in Figure 18 is the following:

```
context ListSubmittedPapers::effect()
post: answer = submittedPaper-> collect(sp |
       Tuple( title = sp.title,
       authorNames = sp.authors-> collect (a| Tuple(firstName=a.firstName,
                                        lastName=a.lastName),
       topics= sp.topics-> collect(t|topicName=t.name)))
```

Assigment of Papers to Reviewers

As we stated before, the assignment of papers may be manual or automatic. In the manual assignment, the organizer may set or unset assignments directly after consulting the submitted papers and the preferences of reviewers in order to decide for any assignment. The automatic assignment consists on an algorithm that computes the best possible assignment based on the rating of reviewers to papers.

Figure 19 shows the definition of the following events:

- *SetAssigment* domain event type, whose effect is to create an instance of *Review* relating an instance of *PCMember* (the *reviewer*) and an instance of *SubmittedPaper* (the *assignedPaper*),
- *UnsetAssignment* domain event type, whose effect is the removal of an instance of *Review* and its associations,
- *SetAssignemtRequest* and *UnsetAssigmentRequest* action request, whose effect is to generate instances of *SetAssignment* and *UnSetAssigment* domain events respectively,
- *AutomaticAssigment* action request, whose effect is to generate automatically a set of instances of *SetAssigment* and *UnsetAssigment* domain events. *SetAssigment* domain event includes the integrity constraint *paperNotAssignedToReviewer* that ensures that the same assignment must only

be done once. *UnsetAssigment* domain event includes the integrity constraint *paperAssignedToReviewer* to ensure the existence of the assignment previous to its removal.

The details of the *AutomaticAssigment* will be described in the upcoming.

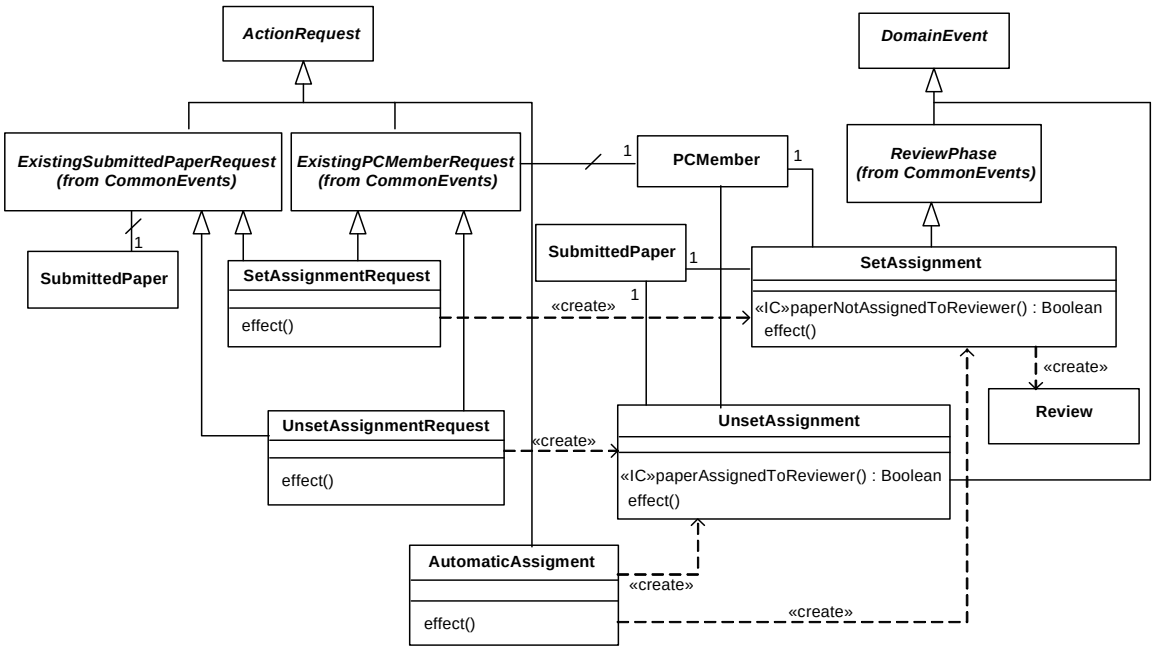

Fig. 19. Definition of the events related to the assignment of papers to reviewers

The specifications of the event types shown in Figure 19 are the following:

```
context SetAssignmentRequest::effect()
post   Generate a domain event SetAssignment:
       sa.oclIsNew() and sa.oclIsTypeOf(SetAssignment) and
       sa.pCMember = pCMember and sa.submittedPaper = submittedPaper

context UnsetAssignmentRequest::effect()
post   Generate a domain event UnsetAssignment:
       unsa.oclIsNew() and unsa.oclIsTypeOf(UnsetAssignment) and
       unsa.pCMember = pCMember and unsa.submittedPaper = submittedPaper

context SetAssignment::effect()
post: re.oclIsNew() and re.oclIsTypeOf(Review) and
       re.reviewer=pCMember and re.assignedPaper=submittedPaper

«IC»
context SetAssignment::paperNotAssignedToReviewer: Boolean
body:  submittedPaper.reviewer -> excludes (pCMember)

context UnsetAssignment::effect()
post:  submittedPaper.reviewer -> excludes (pCMember)

«IC»
context UnsetAssignment::paperAssignedToReviewer: Boolean
body:  submittedPaper.reviewer -> includes (pCMember)
```

The automatic assignment is based on an optimal weighted matching algorithm (WMA) for bipartite graphs that delivers the best possible assignment. More precisely, the WMA applies to a bipartite graph $G = (V,E)$, with $V = U \cup W$. Every edge in G is of the form $\{u_i, w_j\}$ where $u_i \in U$ and $w_j \in W$. Further, it is assumed that G is *complete* (an edges exists for each pair (u_i, w_j)), and *weighted*, i.e., we are given a number $wt_{ij} \geq 0$ for each edge (u_i, w_j). A *matching* M of G is a subset of the edges with the property that no two edges of M share the same node. The *weight* of M, denoted $wt(M)$, is $\sum_{e \in M} wt(e)$. A (weitghted) matching M_o of G is optimal if every other matching M of G is such that $wt(M_o) \geq wt(M)$.

This is illustrated in Figure 20 which shows the graph G together with the rating/weight on each edge. The matching M is represented by thick lines: it can can be verified that its weight is $wt(M) = 1+4+3 = 8$, and that any other matching yields a lower value. Note that, if no ones like a paper (for instance Paper 2), it will get reviewers with low rating but this is unavoidable.

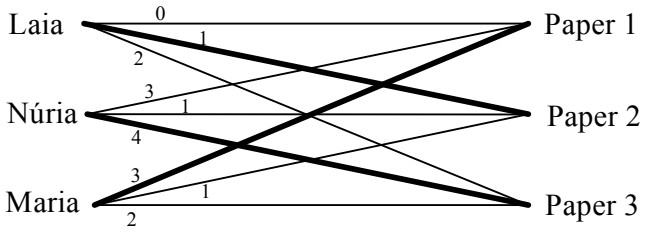

Fig. 20. Weighted matching algorithm

```
context AutomaticAssigment
def:   maxReviewersPerPaper(posAssig:Set(TupleType(subPa:SubmittedPaper,
       rate:Natural,rev:PCMember))):Boolean =
              posAssig.subPa-> size()=  Conference.reviewersPerPaper

def:   maxPapersPerReviewer(posAssig:Set(TupleType(subPa:SubmittedPaper,
       rate:Natural,rev:PCMember))):Boolean =
              posAssig.rev -> size() <= (
              totalNumberPapers * (Conference.reviewersPerPaper /
              totalNumberReviewer)->abs())

def:   correctAssigment(posAssig:Set(TupleType(subPa:SubmittedPaper,
       rate:Natural,rev:PCMember))):Boolean =
       maxReviewersPerPaper(posAssig) and maxPapersPerReviewer(posAssig) and
       posAssig->forAll(a1, a2|a1.subPa = a2.subPa implies a1=a2) and
       posAssig->forAll(a1, a2|a1.rev = a2.rev implies a1=a2)

def:   preferencesSet:Set(TupleType(subPa:SubmittedPaper, rate:Natural,
       pcMem:PCMember) =Preference -> collect(pr|
       Tuple{subPa=pr.submittedPaper, rate=pr.rate, pcMem=pc.pCMember})

def:   cardAssig=preferencesSet->size()


context AutomaticAssigment::allSolutions():
              Set(TupleType(aSolution:Set(TupleType(subPa:SubmittedPaper,
       rate:Natural,rev:PCMember)),totalRate:Natural))
pre:   preferencesSet->notEmpty()
       result-> size()=2^cardAssig and
       result-> forAll(ss| preferencesSet->includes(ss.aSolution) and
                     correctAssigment(ss.aSolution) and
                     ss.totalRate=ss.aSolution.rate->sum())

context AutomaticAssigment::effect()
let    totalNumberReviewers:Natural = PCMember.allInstances->size()
let    totalNumberPapers:Natural = SubmittedPaper.allInstances->size()

in
post:  allSolutions()-> sortedBy(totalRate)->last().aSolution->
       forAll(bs| sa.oclIsNew() and sa.oclIsTypeOf(SetAssignment) and
       sa.pCMember = bs.rev and sa.submittedPaper = bs.subPa)
```

3.2.5. *ReviewAndEvaluationEvents* Package

Once the assignment is done, reviewers can download their assigned papers and submit their reviews. The chairman may send reminders to reviewers with pending reviews, or ask to reviewers to minimize discrepancies when there is a paper with conflicting reviews. When all reviews are completed, the PC accepts or rejects each paper and a notification is sent to the contact author, together with the (anonymous) reviews. Finally, authors of accepted papers must submit their final camera-ready paper and the camera-ready phase finishes.

Review

A review may be submitted several times and a submission replaces the previous version, if it exits. Figure 21 shows the definition of the following events:

- *ReviewSubmission* domain event, whose effect is to complete a review, this is, to update the corresponding attributes of *Review* and for each criteria evaluated to create an instance of *ReviewMark* with its corresponding evaluation and
- *ReviewSubmisisonRequest* action request, whose effect is to generate an instance of *ReviewSubmission* domain event.

Note that *ReviewSubmission* includes the integrity constraint *paperAssignedToReviewer* to ensure that a review is only submitted if the paper was previously assigned to that reviewer.

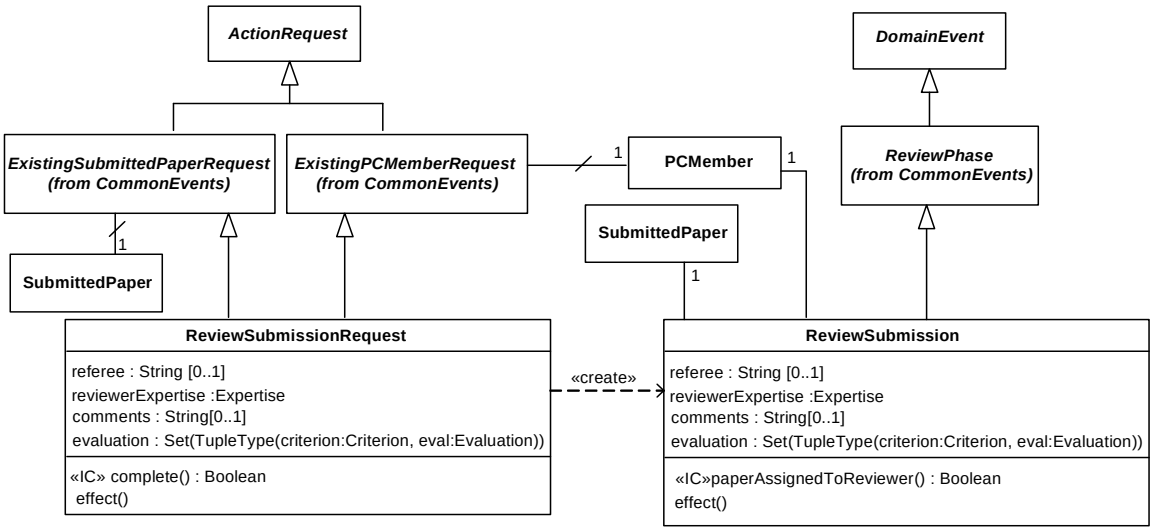

Fig. 21. Definition of the *ReviewSubmissionRequest* and *ReviewSubmission* events

The specifications of the event types shown in Figure 21 are the following:

```
context ReviewSubmissionRequest::effect()
post   Generate a domain event ReviewSubmission:
       rs.oclIsNew() and rs.oclIsTypeOf(ReviewSubmission) and
       rs.pCMember = pCMember and rs.submittedPaper = submittedPaper and
       rs.referee = referee and rs.reviewerExpertise = reviewerExpertise and
       rs.comments = rs.comments and rs.evaluation = evaluation

«IC»
context ReviewSubmissionRequest::complete: Boolean
body:  Criterion.allInstances() = evaluation.criterion
```

```
context ReviewSubmission::effect()
post:  let theReview:Review= pCMember.review-> any
            (assignedPaper = submittedPaper) in
        theReview.completed = true and theReview.updated = time and
        theReview.referee=referee and
        theReview.reviewerExpertise=reviewerExpertise and
        rs.comments = rs.comments and
        evaluation -> forAll(e |
              if theReview.reviewMark->exists(r|
                    r.criterion=e.criterion) then
                    theReview.reviewMark ->
                    any(r|r.criterion=e.criterion).mark=e.eval
              else
                    rm.oclIsNew() and rm.oclIsTypeOf(ReviewMark) and
                    rm.mark =e.eval and rm.review =theReview and
                    rm.criterion= e.criterion
              endif)


«IC»
context ReviewSubmission::paperAssignedToReviewer: Boolean
body:  submittedPaper.reviewer -> includes (pCMember)
```

List Reviews

Figure 22 shows the definition of:

- *ListReviews* query, whose effect is to obtain all reviews.

The chairman, in order to obtain the reviews, can use some criteria to restrict the papers to examine which have been defined in *SubmittedPaperQuery* (from *CommonEvents*).

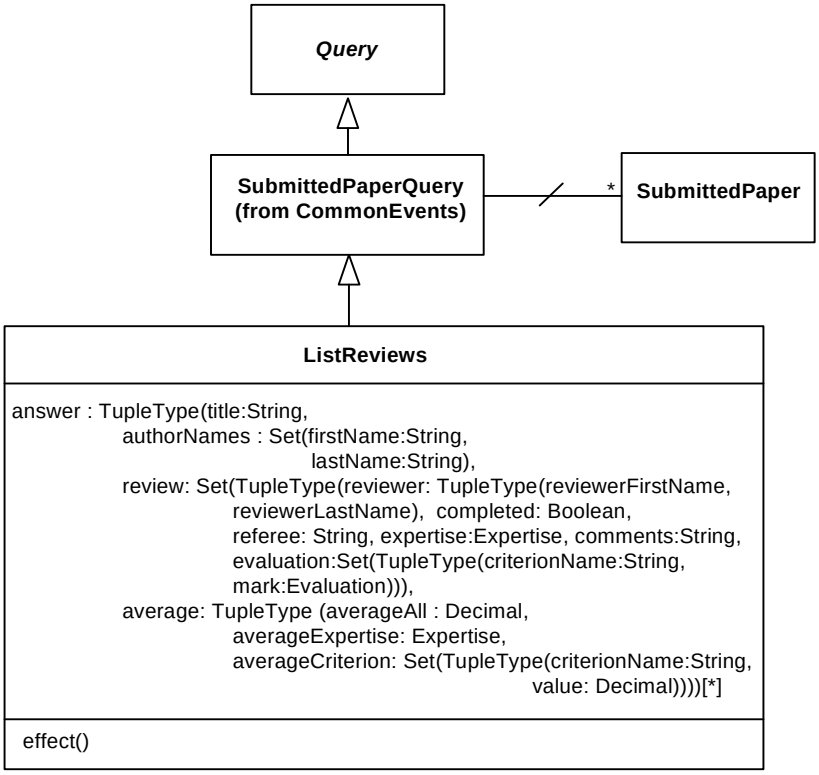

Fig. 22. Definiton of *ListReviews* query

The specification of the effect of the query shown in Figure 22 is the following:

```
context ListReviews::effect()
post: answer = submittedPaper-> collect(sp |
      Tuple( title = sp.title,
      authornames = sp.authors-> collect (a| Tuple(firstName=a.firstName,
            lastName=a.lastName)),
      review= sp.review->
            collect(r|reviewer=Tuple(reviewerFirstName=r.reviewer.firstName,
            reviewerLastName=r.reviewer.lastName), completed=r.completed,
            referee = r.referee, expertise=r.expertise, comments=r.comments,
            evaluation= r.reviewMark->collect (m|
            Tuple(criteriaName=m.criteria.name, mark=m.mark)),
      average = Tuple( averageAll = sp.review.allInstances()->
            forAll(rev|rev.reviewMark -> collect(value * criteria.weight)->
            sum()/Criteria.allInstances-> size())->asSet()->sum()/
            sp.review-> size(),
            averageCriteria = sp.criteriaAverage->collect
            (ca|Tuple(criteriaName=ca.criteria.name, value=ca.value))
```

Reviewing Reviews

Before making the final evaluation of a submitted paper, the chairman can send mails with reminders or check whether there exist conflicts on some papers. In the latter case, an email is sent to the reviewers to ask them to minimize discrepancies.

Figure 23 shows de definition of the following events:

- *RemindPendingReviews* action request, whose purpose is to send a reminder to all reviewers with pending reviews and
- *AskToMinimizeDiscrepancies* action request, whose purpose is, to send a mail to reviewers in order to minimize discrepancies among the reviewers. The effect of both action requests is the invocation of the operation *sendEMail()* of *Person*.

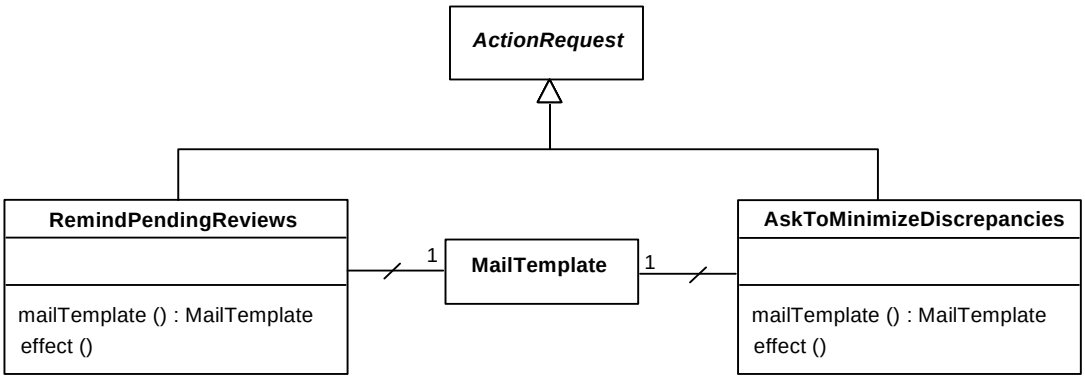

Fig. 23. Definition of *RemindPendingReviews* and *AskToMinimizeDiscrepancies* action requests

The specifications of the operations of the event types shown in Figure 23 are the following:

```
context RemindPendingReviews::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          Review.allInstances()-> forAll(r| not(r.completed) implies
              r.reviewer^sendEmail(subject, body))

context RemindPendingReviews::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::remindPendingReviews)
```

```
context AskToMinimizeDiscrepancies::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          SubmittedPaper.allInstances()-> forAll(p|p.hasConflictingReviews and
              p.reviewer-> forAll(r|r^sendEmail(subject, body)))

context AskToMinimizeDiscrepancies::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::discrepanciesOnReviews)
```

Evaluation

Once all reviews have finished the PC may accept or reject the papers and send the corresponding notification to their contact authors.

Figure 24 shows the definition of the following events:

- *Acceptance* and *Rejection* domain event types, whose effect is to change the *status* of a submitted paper to *accepted* or to *rejected* respectively,
- *AcceptanceRequest* and *RejectionRequest* action requests, whose effect is to generate an instance of *Acceptance or Rejection* domain event respectively and the corresponding instances of *AcceptanceNotification and RejectionNotification* action request and
- *AcceptanceNotification* and *RejectionNotification* request that informs the contact author whether its paper has been accepted or rejected for the conference.

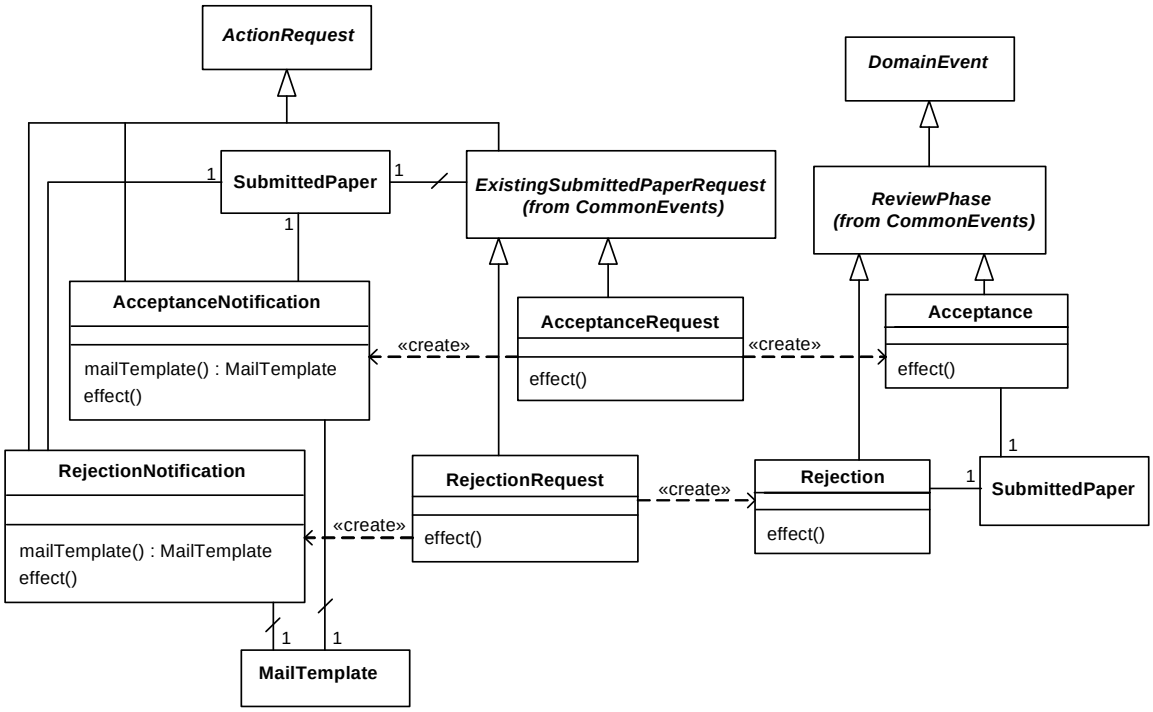

Fig. 24. Defition of the events related to the acceptance and rejection of submitted papers

The specifications of the events types shown in Figure 24 are the following:

```
context AcceptanceRequest::effect()
post   Generate a domain event Acceptance:
       ac.oclIsNew() and ac.oclIsTypeOf(Acceptance) and
       ac.submittedPaper = submittedPaper
post    Inform the contact autor:
       an.oclIsNew() and an.oclIsTypeOf(AcceptanceNotification) and
       an.submittedPaper = submittedPaper
```

```
context RejectionRequest::effect()
post   Generate a domain event Rejection:
       re.oclIsNew() and re.oclIsTypeOf(Rejection) and
       re.submittedPaper = submittedPaper
post    Inform the contact autor:
       an.oclIsNew() and an.oclIsTypeOf(RejectionNotification) and
       an.submittedPaper = submittedPaper

context Acceptance::effect()
post:  submittedPaper.status=Status.accept

context Rejection::effect()
post:  submittedPaper.status=Status.accept

context AcceptanceNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
       Submittedpaper.contactAuthor^sendEmail(subject, body))

context AcceptanceNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::paperAcceptation)

context RejectionNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
       Submittedpaper.contactAuthor^sendEmail(subject, body))

context RejectionNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::paperRejection)
```

Camera-ready Phase

Figure 25 defines the following events:

- *CloseReview* domain event, whose effect is to close the review phase, that is, to set to *false* the attribute *openReview* and consequently no more reviews of papers are allowed and
- *OpenCameraReady* domain events, whose effect is to open the camera-ready phase, that is, to set to *true* the attribute *openCameraReady* of *Conference* and to set a time point as the deadline to submit the camera-ready papers.

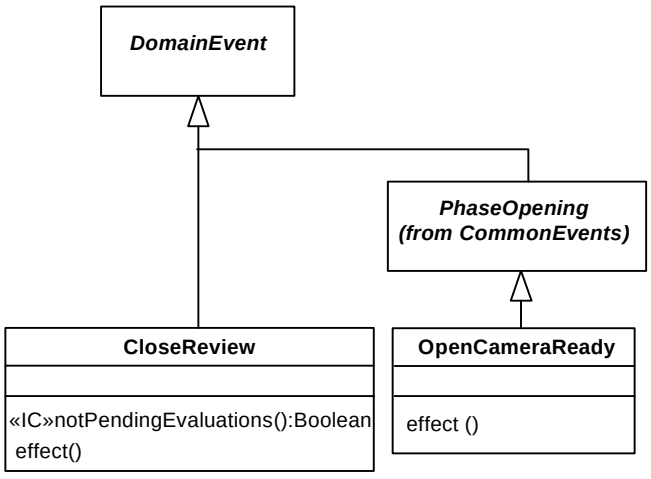

Fig. 25. Definition of *CloseReview* and *OpenCameraReady* domain events

The specifications of the event types shown in Figure 25 are the following:

```
context CloseReview::effect()
post:  Conference.reviewOpen = False
```

```
«IC»
context CloseReview::notPendingEvaluations: Boolean
body:  submittedPaper.allInstances -> forAll(status<>Status:pending)

context OpenCameraReady::effect()
post:  Conference.cameraReadyOpen = True and
       Conference.cameraReadyDeadline = deadline
```

Camera-ready Submission

All contact authors of the accepted papers must submit the final version of their papers.
Figure 26 shows the definition of the following events:

- *CameraReadySubmission* domain event type, whose effect is that the corresponding attributes of an accepted paper are updated,
- *CameraReadySubmissionRequest* action request, whose effect is to generate an instance of *CameraReadySubmission* domain event and an instance of *CameraReadySubmissionNotification* request and
- *CameraReadySubmissionNotification* request, whose effect is to send the acknowledgement of the submission of the camera-ready paper to the contact author.

CameraReadySubmission includes the constraint *acceptedPaper* that checks the paper to be submitted has been already accepted. Additionaly, *CameraReadySubmissionRequest* includes the constraint *knownPassword* that checks that the password introduced is the same that was sent to the contact author.

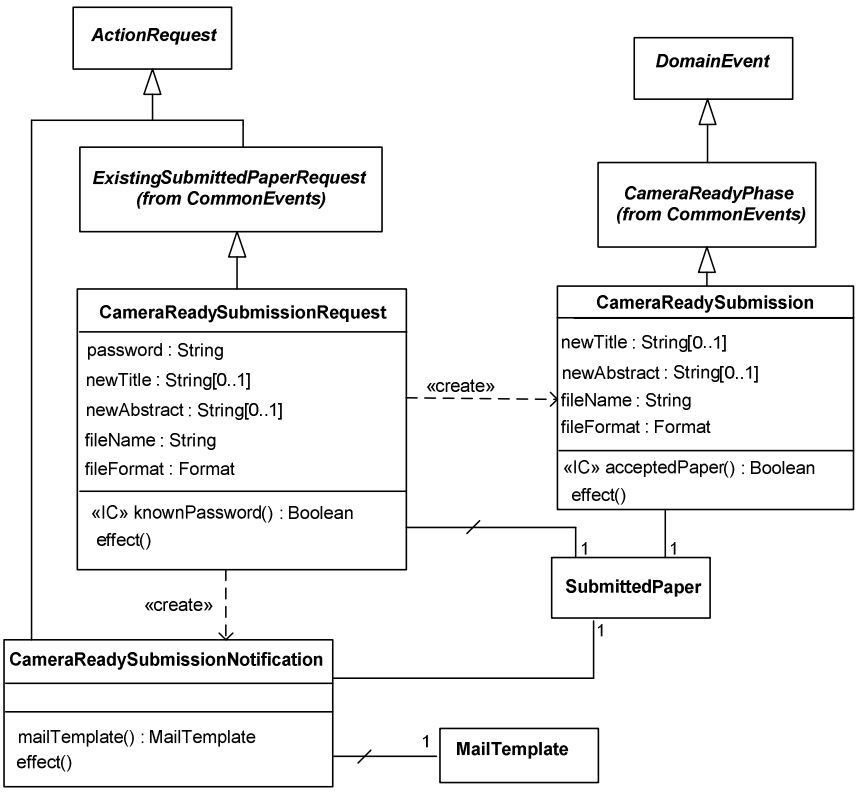

Fig. 26. Definition of the events related to the camera-ready submission of papers

The specifications of the event types shown in Figure 26 are the following:

```
context CameraReadySubmissionRequest::effect()
post Generate a domain event CameraReadySubmission:
       cr.oclIsNew() and cr.oclIsType(CameraReady) and
       cr.submittedPaper = submittedPaper and
       if newTitle<>null then cr.newTitle = newTitle endif and
       if newAbstract<>null then cr.newAbstract = newAbstract endif and
       cr.fileName=fileName and cr.fileFormat=fileFormat and
```

```
post Generate an action request CameraReadySubmissionNotification:
       crn.oclIsNew() and crn.oclIsType(CameraReadySubmissionNotification) and
       crn.submittedPaper = submittedPaper

«IC»
context CameraReadySubmissionRequest::knownPassword:Boolean
body   self.password = submittedPaper.password

context CameraReadySubmission::effect()
post:  if newTitle->notEmpty() then submittedPaper.title = newTitle endif and
       if newAbstract->notEmpty() then submittedPaper.abstract = newAbstract
       endif and submittedPaper.fileName =fileName and
       submittedPaper.fileFormat =fileFormat

«IC»
context CameraReadySubmission::acceptedPaper:Boolean
body   submittedPaper.status = Status:accepted

context CameraReadyNotification::effect()
post:  let subject:String = Conference.acronym.concat(mailTemplate().subject))
       let body:String = mailTemplate().text in
          submittedPaper.contactAuthor^sendEmail(subject, body)

context CameraReadySubmissionNotification::mailTemplate():MailTemplate
post: MailTemplate.allInstances()->any(type=MailType::cameraReadySubmission)
```

Closing Camera-Ready Phase

Figure 27 defines:

- *CloseCameraReady* domain events. The effect is to close the camera-ready phase, that is, to set to *false* the attribute *openCameraReady* and consequently no more camera-ready paper submissions are allowed.

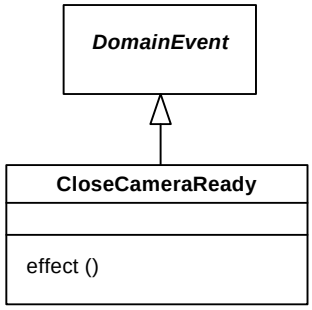

Fig. 27. Definition of *CloseCameraReady* domain event

The specifications of the effect operation shown in Figure 27 is the following:

```
context CloseCameraReady::effect()
post:  Conference.cameraReadyOpen = False
```

4. Conclusions

The main objective of this work is to develop a complete conceptual schema of a system including behaviour. It has also permitted to consolidate the application of modelling events as entities.

It has been proved that modelling events as entities (albeit of a special kind) provides substantial benefits to behavioural modelling. The reason is that the uniform treatment of event and entity types implies the majority of (if not all) language constructs available for entity types can be used also for event types. In particular, event types with common characteristics, constraints, derivation rules and effects have been generalized so that common parts have been defined in a single place, in the *CommonEvents* package, instead of repeating them in each even type. In practice, many event types have used the generalized events defined in the *CommonEvents* package.

Acknowledgements

I would like to thank people of the GMC group and particularly to Antoni Olivé for their many useful comments to previous drafts of this paper. This work has been partly supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIC2002-00744.

REFERENCES

1. Olivé, A. *Integrity Constraints Definition in Object-Oriented Conceptual Modeling Languages*. in *Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings*. 2003: Springer.
2. Olivé, A., *Derivation Rules in Object-Oriented Conceptual Modeling Languages*, in *Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June 16-18, 2003, Proceedings*, J. Eder and M. Missikoff, Editors. 2003, Springer.
3. Olivé, A. *Definition of Events and their Effects in Object-Oriented Conceptual Modeling Languages*. in *23rd International Conference on Conceptual Modeling (ER 2004)*. 2004. Shangai, China.
4. OMG, *UML 2.0 Superstructure Adpted Specification*, in *OMG Adopted Specification (ptc/03-08-02)*, O.M. Group, Editor. 2002.
5. OMG, *UML 2.0 OCL Specification*, in *Doc. ptc/03-10-14*, O.M. Group, Editor. 2003.