Computing Support for Problem Solving in Virtual Communities of Practice

María Clara Casalini, Elsa Estevez
Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur
Bahía Blanca, Argentina
{mcca,ece}@cs.uns.edu.ar
and
Tomasz Janowski
United Nations University International Institute for Software Technology
Macau
tj@iist.unu.edu

ABSTRACT

The paper presents a formal model for a knowledge repository shared by members of a Virtual Community of Practice (VCPs), describes how the repository can be used to underpin collaborative problem solving, and how to build computer support for such processes. The repository comprises the resources used and developed by VCPs particularly through problem solving. As a case study, the paper illustrates how the problem solving process and the underlying repository can be applied in disaster prevention and handling. The repository and the process are formally described using the RAISE Specification Language.

Keywords: Virtual Community of Practice; Knowledge Repository; Collaborative Problem Solving; Semantic Web; Formal Specifications

1. INTRODUCTION

Communities of Practice (CoPs) are groups of individuals with a common interest in a particular area of knowledge, interacting regularly by sharing experiences and taking part in joint activities, with the aim of learning from each other and developing the area [19]. Virtual Communities of Practice are CoPs supported by technology [16].

CoPs are characterised by [19]: *domain* – the subject of interest that brings people together; *community* – defined by members interacting through various activities, all shaping the community and the relationships between members; *practice* – the repertoire of resources, such as experiences, tools, and ways of addressing recurring problems. CoPs have a positive impact on organizations and individuals since they contribute to: (1) providing the context for people to interact, discuss and learn from one another, (2) making existing knowledge explicit and generating new knowledge, (3) introducing collaborative processes and (4) promoting professional development.

Members of VCPs get involved in direct or indirect collaborations. The former takes place when two or more members communicate directly, the latter when a member applies the knowledge other members made available in a shared repository [6]. For both types of collaborations, a number of tools exist to offer various kinds of computing support. However there is little support for the process of collaborative problem solving and limited understanding of what the process, if one can be systematized, involves.

This paper presents a formal model for a shared knowledge repository, and explains how the repository can underpin the problem solving process defined in [5]. We also illustrate how the process can be applied to solve problems in the domain of disaster prevention and handling. We also elaborate on available tools and technologies that make VCPs, and the implementation of the model presented here, possible.

The rest of the paper is organised as follows. Section 2 introduces related work and provides examples of computer support for collaborations in VCPs. Sections 3 and 4 present the model of the repository and how it can be used as part of the process for collaborative problem solving. Section 5 presents a case study in the use of the process and the repository in disaster prevention and handling. Finally, Section 6 presents some conclusions.

2. RELATED WORK

The activities carried out in VCPs depend on the nature of the community. A wide range of computer support exists for various functions and activity types, including member registration, uploading and browsing of resources, chat rooms, emails, text editors and spreadsheets.

Examples of tools include: *ACE* – a collaborative editor [1] and *Google Docs & Spreadsheets* – a text editor and spreadsheet from Google [10]. Forum engines like *phpBB* [15], while still evolving, are mature applications that provide support for discussions and information sharing, allowing member registration, roles and profiles management, and content search, among others. Wikis are an easy way to collaboratively build a shared knowledge base [13] - a freely expandable collection of interlinked Web pages, a hypertext system for storing and modifying information and a database where each page can be edited by users with a forms-capable Web browser. Wikis are becoming the first choice for collaboration portals. Probably the most illustrative example of the use of Wikis is Wikipedia - the collaborative encyclopedia [20]. This type of software is known as *groupware*, since it supports Computer-Supported Cooperative Work (CSCW) [3][11].

A weakness of most tools is inability to interpret the semantics of information, therefore lack of support for generating new knowledge through exploring relationships among data. To overcome this, we take advantage of the concepts and technologies used by Semantic Web – an extension of the current Web, in which information is given well-defined meaning, enabling computers and people to work in cooperation [2]. Semantic Web is implemented through a set of technologies and standards, such as the Resource Description Framework (RDF) [17].

RDF is a language for representing and exchanging metadata about web resources, describing them in terms of properties and their values. Resource descriptions are called statements and consist of a *subject*, a *predicate* and an *object*. The *subject* identifies the resource, the *predicate* determines a property of the resource, and the *object* defines the value of the property. Resources and properties are identified using Unique Resource Identifiers (URIs). Figure 1 shows a concrete RDF graph with three sentences relating the `GrandForks` resource with the `United States` resource and with two other data elements.

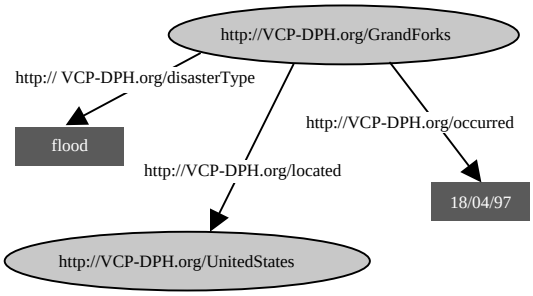

Figure 1: Resources, Properties and Statements

3. MODEL - REPOSITORY

The process for collaborative problem solving proposed in [5] relies on a repository of resources built and developed by the community. This section presents in detail the model of the repository using the notation of RSL [9].

The repository is modelled using three components: *resources* – community assets categorized into types, *properties* – data about resources or relations between them, and *statements* – expressions about resources and their properties. Following RDF definitions, resources are uniquely identified and described through *properties* - binary relations between pairs of resources or between resources and simple values, and *statements* - triples of a subject (resource), property and object (resource or data). Statements are written as `[sub,prop,obj]`. Resource types (categories) are organized into a tree-like hierarchy.

First, we introduce the abstract type `Element` representing all the elements in the repository: data, resources, properties and statements.

```
scheme ELEMENT = class type Element end
```

Second, the module `CATEGORY` introduces the abstract type `Cat`, and concrete types `Cats'` and `Cats`. `Cats'` maps a category to the set of all descendant categories in the hierarchy. `Cats` models categories that are well formed using the function `iswf` to assess if a category is well formed – it has a root element, the root is not a subcategory of itself, all categories are subcategories of the root, and every category has a single parent. The values `root` and `init` model the root element and the initial value of a category hierarchy respectively. In addition, a hierarchy can be modified by adding categories and subcategories and by deleting categories.

```
scheme CATEGORY = class
type
  Cat,
  Cats' = Cat -m-> Cat-set,
  Cats = {| cs: Cats' :- iswf(cs) |}
value
  root: Cat,
  init: Cats = [root +> {}]
value
  iswf: Cat' -> Bool
  iswf(cs) is
    isCat(root,cs) /\
    ~isSubCat(root, root,cs) /\
    (all c1,c2: Cat :-
      isCat(c1,cs) =>
      isSubCat(c1,root,cs) /\
      (isSubCat(c2,c1,cs) => isCat(c2,cs)) /\
      (isCat(c2,cs) => cs(c1) inter cs(c2) = {})
    )
  value
    addCat: Cat >< Cats -~-> Cats,
    addSubCat: Cat >< Cat >< Cats -~-> Cats,
    deleteCat: Cat >< Cats -~-> Cats
end
```

In addition to resources, data types representing values of resource attributes are also organized into a hierarchy. `DATA` and `RESOURCE` modules are introduced to model the corresponding hierarchies. Both include the abstract types `Data` and `Res`, the corresponding data/resource types, and the functions to add and delete data/resource types and to determine if a given type belongs to a category. The functions are defined in terms of functions in the `CATEGORY` module. Here is the `DATA` module:

```
scheme DATA = class
  object C: CATEGORY
  type
    Data,
    DType = C.Cat,
    DTypes = C.Cats
  value
    addDType: DType >< DTypes -~-> DTypes
    addDType(dt, dts) is
      C.addCat(dt, dts)
      pre canAddSubDType(dt,root,dts),
    isDType: DType >< DTypes -> Bool
    isDType(dt,dts) is C.isCat(dt,dts)
    ...
end
```

And here is the `RESOURCE` module:

```
scheme RESOURCE = class
  object
    C: CATEGORY
  type
    Res,
    RType = C.Cat,
    RTypes = C.Cats
  value
    addRType: RType >< RTypes -~-> RTypes
    addRType(rt,rts) is
      C.addCat(rt,rts)
      pre canAddSubRType(rt,root,rts),
    isRType: ResT >< ResTs -> Bool
    isRType(rt,rts) is C.isCat(rt,rts)
    ...
end
```

Information about resources is expressed through their properties. A property maps a resource type to a simple data type – associating a data value to a resource, or to another resource type - establishing a relationship among resources. Consequently, an object in a property can be a data type or a resource type. Before we introduce the `PROPERTY` module, we define the `VALUE` module for representing possible values of an object. The `Value` type contains two kinds of values: `plain` - instances of `Data` and `complex` - instances of `Res`. Likewise, `VType` are defined as plain or complex data/resource types and `VTypes` represents the hierarchies of `DTypes` and `RTypes`.

```
scheme VALUE(D: DATA, R: RESOURCE) =
class
  type
    Value == plain(D.Data) | complex(R.Res),
    VType == plain(D.DType) | complex(R.RType),
    VTypes:: plain: D.DTypes complex: R.RTypes
  ...
end
```

Properties relate subjects (resources) to objects (values) according to the resource and value types existing in the repository. The type `Prop` is defined as a record composed of an `RType`, a `VType` and the name of the property. A property is correct (`isProp`) if its `subType` is a valid `RType` and its `objType` is a valid `VType`.

```
scheme PROPERTY (D:DATA,R:RESOURCE,V:VALUE(D,R)) =
class
  type
    Name,
    Prop::
      subType: R.RType
      objType: V.VType
      name: Name
  value
    isProp: Prop >< D.DTypes >< R.RTypes -> Bool
    isProp(p, dts, rts) is
      R.isRType(subType(p), rts) /\
      V.isVType(objType(p), V.makeVTypes(dts,rts))
  ...
end
```

Statements are triples of: subject - identifies the resource described by the statement, property - identifies a property describing the resource, and object - identifies the value of the property, which can be a data instance or another resource depending on the property. In a correct statement, the property must be present in the repository and the types of the subject and object parts must conform to the resource/values types required by the property.

```
scheme STATEMENT(D: DATA, R: RESOURCE,
  V: VALUE(D,R), P: PROPERTY(D, R, V)) = class
  type
    Stat:: sub: R.Res prop: P.Prop obj: V.Value
  value
    isStat: Stat >< D.DTypes >< R.RTypes -> Bool
    isStat(s, dts, rts) is
      P.isProp(prop(s), dts, rts) /\
      R.hasRType(sub(s),P.subType(prop(s)),rts) /\
      V.hasVType(obj(s),
        P.objType(prop(s)),
        V.makeVTypes(dts, rts))
  ...
end
```

In order to manage elements of the repository, resources, properties and statements are grouped into collections. The module `COLLECTION` is introduced to define an abstract type `Collection`, a special value `empty` to represent the empty collection, and some functions. Among them, `hasElem` determines whether an element belongs to a collection and `addElem` adds an element to the collection.

```
scheme COLLECTION(E: ELEMENT) = class
  type
    Col
  value
    empty: Col,
    hasElem: E.Element >< Col -> Bool
  axiom
    (all e:Element:- ~hasElem(e, empty))
  value
    addElem: E.Element >< Col -> Col
    addElem(e, c) as c'
      hasElem(e, c') /\
      (all e': E.Element :-
        hasElem(e',c) => hasElem(e',c')
      )
  ...
end
```

Finally, the module `REPOSITORY` defines the repository along with the operations on it. It creates the instances of the modules defined earlier, and defines types to represent collections of resources, properties and statements.

```
scheme REPOSITORY = class
  object
    D: DATA,
    R: RESOURCE,
    V: VALUE(D, R),
    P: PROPERTY(D, R, V),
    S: STATEMENT(D, R, V, P),
    ER : class type Element = R.Res end,
    EP : class type Element = P.Prop end,
    ES : class type Element = S.Stat end,
    CR : COLLECTION(ER),
    CP : COLLECTION(EP),
    CS : COLLECTION(ES)
  type
    Ress = CR.Collection,
    Props = CP.Collection,
    Stats = CS.Collection
  ...
end
```

A set of functions is also defined in `REPOSITORY` to determine if a given resource, property or statement exists on the repository, based on the `hasElem` function of the `COLLECTION` module. In addition, `existsValue` determines if a value exists in the collection of resources.

```
value
  existsRes: R.Res >< Ress -> Bool,
  existsProp: P.Prop >< Props -> Bool,
  existsStat: S.Stat >< Stats -> Bool,
  existsValue: V.Value >< Ress -> Bool
```

The type `Repository'` is defined as a record containing collections of resources, properties and statements, along with the hierarchies of data and resource types. The type `Repository` models well-formed repositories, in which resources, properties and statements are all well-formed.

```
type
  Repository'::
    dts:  D.DTypes <-> re_dts
    rts:  R.RTypes <-> re_rts
    res:  Ress     <-> re_res
    prop: Props    <-> re_prop
    stat: Stats    <-> re_stat,
  Repository = {| s:Repository' :- iswf(s)|}
value
  iswf: Repository' -> Bool
  iswf(r) is
    iswfRes(res(r), rts(r)) /\
    iswfProp(prop(r),dts(r),rts(r)) /\
    iswfStat(stat(r),res(r),prop(r),dts(r),rts(r))
```

Statements are well-formed if all statements are correct and all their elements exist in their collections.

```
value
  iswfStat: Stats >< Ress >< Props ><
    D.DTypes >< R.RTypes -> Bool
  iswfStat(ss, rs, ps, dts, rts) is
    (all s: S.Statement :-
      existsStat(s, ss) =>
      S.isStat(s, dts, rts) /\
      existsRes(S.sub(s), rs) /\
      existsProp(S.prop(s), ps) /\
      existsValue(S.obj(s), rs)
    )
```

A repository is build beginning from empty hierarchies and collections and later adding the relevant types, resources, properties and statements. Functions are defined to perform all these operations. For example, the `addStat` function is defined as follows:

```
value
  addStat: S.Stat >< Repository -~-> Repository
  addStat(s, rep) is
    let ss = stat(rep), ss' = CS.addElem(s, ss)
    in re_stat(ss', rep) end
    pre canAddStat(s, rep),
  canAddStat: S.Stat >< Repository -> Bool
  canAddStat(s, rep) is
    ~existsStat(s, stat(rep)) /\
    existsRes(S.sub(s), res(rep)) /\
    existsProp(S.prop(s), prop(rep)) /\
    case S.obj(s) of
      V.plain(_) -> true,
      V.complex(r) -> existsRes(r, res(rep))
    end /\
    S.isStat(s, dts(rep), rts(rep))
```

4. MODEL - PROCESS

The problem solving process defined in [5] relies on the repository to carry out problem-solving, and generates new resources, properties and statements as one outcome of the process. The process is carried out collaboratively by community members in a series of six steps:

1) *Problem Registration* – A new problem description is registered as a problem resource by a member.
2) *Problem Exploration* – The problem is analyzed and the repository is explored to find related resources. New statements are added in the process
3) *Problem Matching* - The problem is matched against similar problems existing in the repository to find similarities, dependencies and partial solutions.
4) *Solution Design* - The problem is decomposed into a set of sub-problems and these sub-problems are analyzed for possible dependencies.
5) *Solution Refinement* - The solution is refined by gradually replacing sub-problems by their solutions.
6) *Solution Integration* – Integrating all partial solutions into complete solution to the problem.

Figure 2 depicts the process in its entirety. It is easy to see how the number of statements about the problem increases with every step, and how the number of unsolved problems in the repository increases during Solution Design and decreases during Solution Refinement.

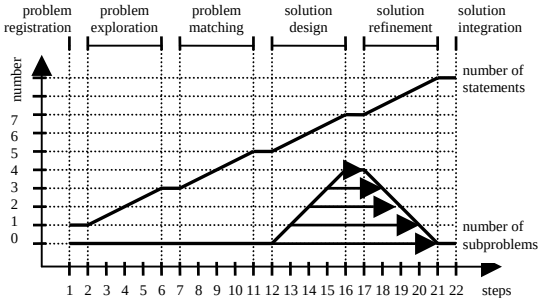

Figure 2: Process for Collaborative Problem Solving

The process starts with Problem Registration when a member posts a new problem and publishes it as the new Problem resource in the repository. During Problem Exploration, the problem is defined in more detail by exploring the repository for related resources and adding the corresponding statements, with the problem resource figuring as a subject or an object of such statements.

Problem Matching enables comparing the problem with existing problems. The comparison between a pair of resources is done by looking at the statements describing each of them and defining a degree of similarity. For instance, a pair of problems may exhibit the following similarities: (1) the same sets of properties and the same values, (2) the same sets of properties but different values, (3) a subset/superset of properties with the same values, (4) a subset/superset of properties with different values. If the problems exhibit condition (1), this may be an indication that they are closely related to one another.

Solution Design consists in decomposing the problem into a set of sub-problems. Sub-problems are registered in the repository as `Problem` type resources, thus recursively initiating the problem solving process. During Solution Refinement, all knowledge generated about the solution is added as statements. In addition, statements specifying how to combine solutions to sub-problems into solution to the original problem are also added.

Finally, once all sub-problems were solved, and the constructed solution was completely documented, Solution Integration enables linking the obtained solution to the original problem, indicating that the problem has been solved and the process is concluded.

5. A CASE STUDY

We present a case study illustrating the problem-solving process and the use of the repository. The case study refers to a VCP focusing on Disaster Prevention and Handling [4]. First, the three components of the VCP are presented: the domain, the community and the practice. The structure of the repository is depicted and illustrated. Finally, a problem posted by a member is described and the problem solving process for this problem is developed.

The case study considers a Virtual Community of Practice for Disaster Prevention and Handling (VCP-DPH) - a community performing research and developing recommendations in the area of prevention and handling of disaster events. Members of this community comprise representatives of non-governmental organizations, civil defense, police, medical doctors, lifeguards, researchers and detectives, and generally all people involved with organization and coordination of actions to prevent and handle natural or man-made disasters. The community carries out various activities: develops and publishes best practices; organizes workshops, conferences and seminars; compiles data on documented cases; carries out online discussions; and elaborates on the statistics obtained.

VCP-DPH relies in a shared repository of resources. These resources have been gathered by members during the lifetime of the community, and produced through their interactions and activities in which they participate. The repository is organized according to a hierarchy of resource and simple data types defined by the community administrator. The root of the hierarchy is the type `All` and immediate descendants are: `Document`, `Person`, `Organization`, `Instance`, `Country`, `Problem` and `Solution`. `Problem` and `Solution` are two pre-defined types of resources used by the problem-solving process. The hierarchy may have several levels. For instance, `Member`, `Staff` and `Volunteer` are descendents of `Person`. The repository includes definition of data types. A partial hierarchy of resource types is shown in Figure 3.

All resources added to the repository must belong to an existing type. Properties are defined with a name and a description comprising two elements. The first element is the type of resource the property describes, and the second element is a resource type or a data type. For instance, the property `located` in Figure 1 is defined with its name and the pair `<Instance,Country>`, since it maps an instance of a disaster and the country where it happened.

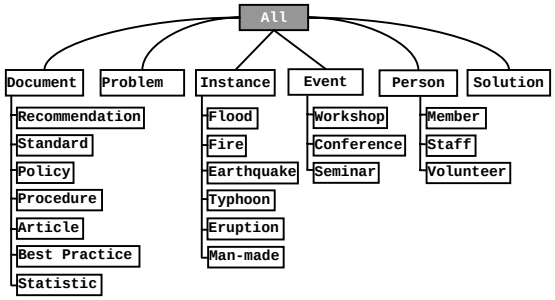

Figure 3: Repository Structure – Resource Types

Members use the repository while searching for solutions to problems. The solutions found and the knowledge acquired are all recorded in the repository, allowing the community to grow and expand.

Figures 4 and 5 show some resources, properties and statements existing in the repository. References to real cases of disasters, recommendations made and standards adopted were obtained from [7], [8] and [14].

Type	Instances
Flood	Flood1: 1997-GrandForks, North Dakota
Earthquake	EarthQuake1: 2003-Bam, Kerman
Country	Country1: Japan
	Country2: Iran
Recommendation	Recommendation1: What to do before a flood, FEMA
	Recommendation2: What to do during a flood, FEMA
Standard	Standard1: 2006-Standard Flood Hazard Determination Form
Problem	ProblemX: list the equipment required in rescue operations
	ProblemY: define a general chain of commands for a rescue operation
Solution	SolutionX: list EQ of equipment
	SolutionY: chain CC of commands

Figure 4: Sample Resources

Properties: <subject, name, object>
<Problem, about, Topic>
<Problem, coversDisaster, Disaster>
<Problem, coversArea, Area>
<Solution, isSolving, Problem>
<Solution, solved, Problem>
<Solution, contains, Problem>
Statements: [resource, property, value]
[ProblemX, about, equipment]
[ProblemX, coversDisaster, generalDisasters]
[ProblemX, coversArea, anyArea]
[ProblemY, about, humanResources]
[ProblemY, coversDisaster, generalDisasters]
[SolutionX, solved, ProblemX]
[SolutionY, solved, ProblemY]

Figure 5: Sample Properties and Statements

Problem Description

When a member of the community presents a new problem, the process starts. All resources and all members can be involved in searching a solution to the problem. Suppose the following problem is posted by a member:

```
ProblemP: Specify a plan for disaster handling
for the case of a flooding taking place in a
city located in a sea area.
```

The presented problem is only an example and is not meant to be a real guide on how to specify a plan for disaster handling. Parts of the example were written following the guidelines taken from [12].

Problem Solving

The process for solving `ProblemP` includes the six steps as described in the general case earlier:

1) Step 1: Problem Registration
   The problem is added as a new resource of the type `Problem` – the repository has one more resource and the members start working on the problem.

2) Step 2: Problem Exploration
   During problem exploration, the members analyse the problem by determining the topics to which it is related, categorising it, and adding statements to the repository with the information collected. In particular for the posted problem, members determine that the problem is about three specific topics - equipment, human resources and rescue techniques, that it focuses on one type of disaster - flooding, and that it may take place in a specific area - seaside. The statements added are:

   ```
   [ProblemP,about,equipment]
   [ProblemP,about,humanResources]
   [ProblemP,about,rescueTechniques]
   [ProblemP,coversDisaster,flood]
   [ProblemP,coversArea,sea]
   ```

3) Step 3: Problem Matching
   The result of comparing `ProblemP` against all the problem resources existing in the repository shows that it has the same properties as `ProblemX` and a superset of `ProblemY`. Two statements are added to the repository reflecting this fact:

   ```
   [ProblemP,sameProps,ProblemX]
   [ProblemP,supProps,ProblemY]
   ```

4) Step 4: Solution Design
   The design of solution starts by adding a new resource to the repository - `SolutionS`. The statement `[SolutionS,isSolving,ProblemP]` is added as well, linking the solution to the problem. Next, `ProblemP` is decomposed into:

   ```
   a) SubP1: Define the chain of command and
      the tasks for each position in the
      chain for a rescue operation in the
      case of flooding
   b) SubP2: Design communications between
      team members
   c) SubP3: List the equipment required
   d) SubP4: List rescue operations
   ```

   The four sub-problems are registered, initiating the problem solving process for each. Four statements are added relating `SolutionS` to the sub-problems:

   ```
   [SolutionS,contains,SubP1]
   [SolutionS,contains,SubP2]
   [SolutionS,contains,SubP3]
   [SolutionS,contains,SubP4]
   ```

   Since communication between members of the team can be designed only after the chain of commands is defined, there is a dependency between `SubP1` and `SubP2` - the problem solving process for `SubP2` waits for the results of the `SubP1` process. This fact is reflected by adding the statement `[SubP2,dependsOn,SubP1]`.

5) Step 5: Solution Refinement
   Once all the problems identified in Step 4 have been solved, their solutions can be combined to conclude the construction of `SolutionS`. More information is added to the repository in the form of statements `[descriptionS,documents,SolutionS]` where `descriptionS` is a resource of the type `Document` containing a detailed explanation of how the solution

is constructed. The document will explain: that the chain of commands defined in the solution to SubP1 must be set in place; that communications between members of the chain must follow the guidelines of the document produced as the solution to SubP2; that the equipment provided to the working teams follows the process applied to the problem SubP3. Finally, the document will describe how the rescue operations explained in the solution to SubP4 should be applied to the case of flooding.

6) Step 6: Solution Integration
The last step in the process involves removing the statement [SolutionS,isSolving,ProblemP] added during Step 4, and adding instead the statement [SolutionS, solved,ProblemP].

Sub-Problem Solving

While solving ProblemP, four sub-problems were identified. The processes for solving these sub-problems are explained in the following:

1) **Sub-Problem 1:** SubP1 - "Define the chain of commands and tasks for each position in the chain for a rescue operation in the case of flooding" was added as a problem resource. Here is how this problem is solved through a six-step process.

In Problem Registration, SubP1 is registered in the repository. The exploration of SubP1 gives members the insight that the problem is about human resources for the operations in the case of flooding. As a result, the statements [SubP1,about,humanResources] and [SubP1, coversDisaster,flood] are added to the repository. During problem matching, SubP1 is matched against other problems in the repository, showing that it shares the same properties with the problem ProblemPY. As a result, the statement [SubP1,sameProps,ProblemPY] is added. During Solution Design, a new resource of the type Solution - SolutionS1, and the statement [SolutionS1,isSolving,SubP1] linking it with SubP1 are added. The similarities between this problem and ProblemPY, and further analysis of the problem lead to the conclusion that this problem is simple enough to be solved directly using the solution to ProblemPY. Therefore it is not decomposed further. For refining the solution, a new document descriptionS1 describing the solution to this problem is written and added as a resource. The solution is added through the statement [descriptionS1, documents, SolutionS1]. SolutionS1 reuses the resource SolutionY – solution to ProblemPY. The chain of commands defined for a general disaster event is taken as a base design; necessary modifications are introduced to create the appropriate chain of commands for this particular case and define the responsibilities and tasks assigned to each position in the chain – explained in detail by descriptionS1. Finally, the statement [SolutionS1,isSolving,SubP1] is removed from the repository and the new statement [SolutionS1,solved,SubP1] is added during solution integration.

2) **Sub-Problem 2:** SubP2 defined as "Design communications between team members" is added to the repository. A dependency between this problem and ProblemP1 was found during Step 4 of the process for solving ProblemP. The dependency indicates that SubP2 could only be solved after some results from the problem solving process for SubP1 are obtained. In this case, since SubP2 aims to design communication between members of the team, the solution to SubP1 must be ready to start this process.

The first step in a solution includes adding SubP2 as a new resource in the repository. Problem exploration provides the knowledge that the problem is about designing communications between members, so the statement [SubP2, about, communication] is added. The comparison of SubP2 against existing problems in the repository does not find useful matching, so no statements are added in this case. As this problem is not further decomposed into smaller sub-problems during Step 4, only the new resource SolutionS2 is added along with the statement [SolutionS2, isSolving, SubP2]. During solution refinement, the members in charge of solving this problem search for documentation about communication designs for teamwork. In the process, members may acquire books, articles and recordings, and all these resources will be added as new instances to the repository. Experts who are also resources of the community will be consulted, and based on their advice and the research done the communication for the work team will be defined. The result will be included as solution description, along with statement [descriptionS2, documents, SolutionS2]. Finally, the statement added in Step 4 is replaced by [SolutionS2, solved, SubP2].

3) **Sub-Problem 3:** The process for SubP3 is similar to the one for SubP1: similarities between SubP3 and ProblemPX are found; the problem is not further decomposed into sub-problems; and the solution for SubP3 is constructed by reusing the solution to ProblemPY. Based on the list of equipment provided by SolutionSY and advice from the members with expertise in flooding events particularly in the coastal areas, a new list is written. The list is added as a new resource to the repository and a document is written explaining this in detail. This generates the new resource descriptionS3.

4) **Sub-Problem 4:** The approach for solving SubP4 is similar to the one for SubP2, without reusing other existing solutions. For solving this problem, members search for documentation about techniques to apply in rescue operations, in particular for operations that need to take place in the indicated area and under a flooding event. Members may acquire books, articles, recordings, and all these resources will be added as new instances to the repository. A new list is written with the rescue operations required and included in the description of the solution.

During the solution design phase of the posted problem, four sub-problems were identified and the process was recursively applied for solving these sub-problems.

6. CONCLUSIONS

The paper presented a formal model for a repository of resources to underpin the process of collaborative problem solving in Virtual Communities of Practice. The application of the process and the utilization of the repository were illustrated through a case study.

Following analysis of the existing tools providing computer support to VCPs and their limitations for defining and interpreting semantic information, RDF concepts were applied to define the structure of the repository and to drive the problem solving process. The process utilizes the knowledge owned by the community and enriches the community by enabling the systematic growth of the underlying repository. A formal model for the repository was described using the notation of RSL. The problem solving process was explained, and a case study was developed that showed how the repository and the process can be applied. While the process enables automating some steps of the process, human intervention cannot be eliminated since members are responsible for interpreting and selecting the results obtained.

Future work includes the implementation of the problem solving process, further automation of some of its steps, and the integration with other similar tools supporting Virtual Communities of Practice.

ACKNOWLEDGEMENTS

We would like to thank Adegboyega Ojo, Irshad Khan, and Gabriel Oteniya for useful discussions and comments.

REFERENCES

[1] ACE A Collaborative Editor, http://ace.iserver.ch
[2] T.Berneers-Lee, J. HEndler, O. Lassila. The Semantic Web. Scientific American, 2001.
[3] B. Bruegge, A. Houghton. Computer-Supported Cooperative Work. Discussion Summary. Software Engineering Education and Practice, 1996.
[4] M.C.Casalini, E.Estevez, T.Janowski. Collaborative Problem Solving in Virtual Communities of Practice – A Case Study in Disaster Prevention and Handling. XII CACIC, 2006.
[5] M.C.Casalini, T.Janowski, E.Estevez. A Process Model for Collaborative Problem Solving in Virtual Communities of Practice. 7th Working Conference on Virtual Enterprises PRO-VE'06, Helsinki, Finland. Springer Verlag, 2006.
[6] W.Conen, G. Neumann. Prerequisites for Collaborative Problem Solving. Proceedings of WETICE 96, IEEE 5th Intl. Workshops on Enabling Technologies, Standford, CA, 1996.
[7] Federal Emergency Management Agency (FEMA), http://www.fema.gov
[8] First American Flood Data Services, http://fafds.floodcert.com/news/index.asp?ID=103
[9] C.George et al. The RAISE Specification Language. Prentice Hall, 1992.
[10] Google Docs & Spreadsheets, http://docs.google.com
[11] J. Grudin. Computer-Supported Cooperative Work: History and Focus. Computer, 1994
[12] Interpol Disaster Handling Procedures, http://www.interpol.int/Public/DisasterVictim/guide/chapitre2.asp
[13] B.Leuf, W.Cunningham. The Wiki Way: Quick Collaboration on the Web. Addison-Wesley Professional.
[14] Natural Disaster Handling Procedures, http://library.thinkquest.org/03oct/00758/en/home.html
[15] phpBB - Creating Communities, http://www.phpbb.com
[16] C. E. Porter. A Typology of Virtual Communities: A Multi-Disciplinary Foundation for Future Research. Journal of Computer-Mediated Communication. 10 (1) Article 3, 2004.
[17] Resource Description Framework (RDF) - Semantic Web Activity - World Wide Web Consortium, http://www.w3.org/RDF
[18] Structured Analysis Wiki www.yourdon.com/strucanalysis wiki
[19] E. Wenger. Communities of Practice, Introduction, 2004.
[20] Wikipedia, http://www.wikipedia.org