Augmenting Complex Problem Solving with Hybrid Compute Units*

Hong-Linh Truong[1], Hoa Khanh Dam[2], Aditya Ghose[2], Schahram Dustdar[1]

[1] Distributed Systems Group, Vienna University of Technology, Austria
{truong,dustdar}@dsg.tuwien.ac.at
[2] University of Wollongong, Australia
{hoa,aditya}@uow.edu.au

Abstract. Combining software-based and human-based services is crucial for several complex problems that cannot be solved using software-based services alone. In this paper, we present novel methods for modeling and developing hybrid compute units of software-based and human-based services. We discuss high-level programming elements for different types of software- and human-based service units and their relationships. In particular, we focus on novel programming elements reflecting hybridity, collectiveness and adaptiveness properties, such as elasticity and social connection dependencies, and on-demand and pay-per-use economic properties, such as cost, quality and benefits, for complex problem solving. Based on these programming elements, we present programming constructs and patterns for building complex applications using hybrid services.

1 Introduction

Recently, several novel concepts have been introduced to exploit human computing capabilities together with machine computing capabilities. This combination has introduced a new form of "computing model" that includes both machine-based and human-based "computers". In this emerging computing model, machine-based and human-based computing elements are interconnected in different ways, thus it is possible to support different programming models built on top of them.

Indeed, there are different ways to develop applications atop such a new computing model. In the current research approaches, human-based capabilities are usually provisioned via "crowdsourcing" platforms [1] or specific human-task plug-ins [2,3]. These approaches achieve human and software integration mainly using (specific) platform integration. The main programming model is mostly the workflow which is however not flexible enough for programming different types of interactions among multiple types of services. In these approaches, essential programming elements representing software-based services (SBS) and human-based services (HBS) cannot be programmed directly into applications. Furthermore, these approaches do not provide a uniform view of SBS and HBS, and let the developer perform the complex tasks of establishing relationships between SBS and HBS. In addition, although SBS and HBS can be provisioned using

* The work mentioned in this paper is partially supported by the EU FP7 FET SmartSociety.

cloud provisioning models (thus they can be requested and initiated on-demand under different quality, cost and benefit models), there is a lack of mechanisms to program explicitly quality, cost, and benefit constraints for complex elastic applications.

In this paper, we view the "new computing model" as a collection of diverse and heterogeneous SBS and HBS that can be provisioned (e.g., by cloud computing models) on-demand under different cost, benefits and quality models. This view is very different from human-based workflows of which tasks and flows are (statically) mapped to humans. More specifically, our model considers humans as a service unit, like software service units, and takes into account diverse types of relationships among human-based and software-based service units, quality, cost and benefit properties. Our approach provides concepts for developing such applications where *hybrid service units*, their *relationships*, and *cost, quality and benefits* are first-class programming elements. Hence, our approach provides a higher level of abstraction and a flexible way for combining hybridity, collectiveness and adaptiveness of sohuman-based and software-based services.

The rest of this paper is organized as follows: Section 2 discusses background, related work and our approach. Section 3 serves to describe programming elements covering units, relationships and non-functional parameters. In Section 4 we describe high-level programming constructs. Section 5 illustrates an example of how our approach works in practice. We conclude the paper and outline our future work in Section 6.

2 Background and Related Work

Software-based service units constructs: There exist several frameworks for engineering and executing cloud applications using different IaaS, PaaS and SaaS, such as Aneka [4], BOOM [5]. They abstract cloud resources and support different prorgamming models, such as MapReduce and dataflows. But they do not consider hybrid services consisting of SBS and HBS and do not provide high level programming constructs for modelling the relationships among HBS and SBS. Most of them rely on traditional relationships among SBS, such as control and data dependencies, modelled in specific application structure descriptions, workflows and declarative programming languages.

Human computation programming frameworks: There have been an increasing number of programming frameworks for human computation introduced in recent years. Most of existing work (e.g., Crowdforge [1], TurKit [2]) consider human workers as being homogeneous and interchangeable, which is useful in developing crowdsourcing solutions where scalability and availability are the main issues. Such frameworks, however, provide limited notion of identity, human skills, and social relationships which are important in developing an ecosystem of connected, heterogeneous people and software. The recent Jabberwocky framework [6] has addressed this issue to some extent by providing a programming environment for both human and machine computation. However, Jabberwocky does not allow to explicitly model the relationships between people and machines. General-purpose programming languages for human computation, such as CrowdLang [7], do not rely on service models and do not consider quality, cost, benefits and elasticity as first-class entities in programming and constructing hybrid compute units.

High-level constructs for hybrid compute units: Using several low level APIs for accessing SBS, like JClouds, Boto, and OpenStack, the developer can define SBS objects and establish data and control flows. Our previous work (e.g., [8]) has focused on providing well-defined APIs for provisioning HBS. However, there is a lack of support for programming different types of relationships among SBS and HBS. The developer has to do this on his/her own. As a result, they would find it difficult to code such relationships due to the lack of well-defined programming elements, in particular those related to cost, benefit, quality constraints and to mixed compositions of SBS and HBS. The use of generic "building blocks" abstracting patterns and providing them via APIs to simplify the developer task is well-known in SBS in clouds[9]. However, no high-level program constructs and code generation have been proposed for HBS and SBS in cloud environments.

Compared with existing work we are focusing on combining HBS and SBS for hybrid compute units using service computing and cloud computing models. Our approach supports unified framework for human and software, and provide high-level programming constructs for different types of services, relationships, and cost, quality, and benefits models.

3 Fundamental Elements for Hybrid Compute Units

3.1 Service-based Compute Units

In our model, at the core of SBS and HBS there are "processing units", realized via either machine CPUs/cores or human brains. To program an application, the developer can exploit an SBS or HBS via an *abstract service unit*. Therefore, an application developed in our framework is abstractly viewed as consisting of a number of service-based compute units (see Figure 1) and their interactions. A *Unit* can perform a number functions (e.g., detecting a pattern in or enriching the quality of an image) with input and output data. A unit also has a number of cost, benefit, and quality properties (see section 3.3 for more details). A unit can be either a *SBS* (Software-Based Service) or *HBS* (Human-Based Service). We further divide HBS into ICU (Individual Compute Unit – representing a service offered by an individual) and SCU (Social Compute Unit – representing a service offered by a team). Both HBS and SBS units can potentially support elasticity in terms of capability (resource), cost and quality [10]. For example, a SBS for data analytics can increase its cost when being asked to provide higher analysis accuracy or a SCU can reduce its size and the cost when being asked to reduce the quality of the result. To support solving complex problems with elastic service units, we model elasticity capability (*ElasticityCapability*) and associate it with *Unit*.

A SBS unit can be in number of known software forms offered in cloud computing models, such as IaaS (e.g., Amazon EC), DaaS (e.g., Microsoft Azure Data Marketplace), PaaS (e.g., Google App Engine) or SaaS (e.g., Salesforce.com). Although many ongoing work is still being developed for SBS, SBS are already extensively explored in terms of service management, capabilities, and function modeling. Therefore, we rely on existing common models for representing SBS. For HBS, their computing capability is specified in terms of human skills and skill levels. Therefore, in our model a HBS unit has a set of `Skills`, each of which is associated with a skill level. Those skills and

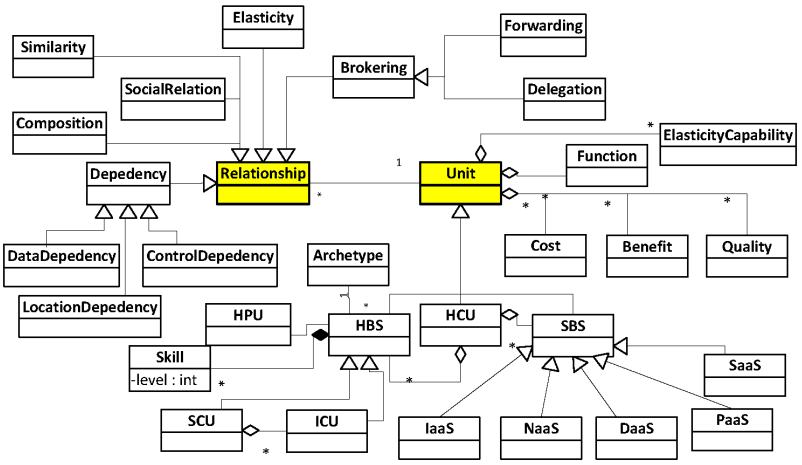

Fig. 1. A conceptual model for elements in programming hybrid compute units

skill levels can be defined consistently within a particular service provisioning platform (using evaluation techniques, benchmarking, or mapping skills from different sources into a common view for the whole platform). Therefore, we associate each HBS with a Human Power Unit (HPU) [8], a value defined by the HBS provisioning platform to describe the computing power of the HBS based on its skills and skill levels, which are always associated with specific `Archetypes` indicating the domain in which the skills are established.

By combining a set of HBS and SBS, we introduce *hybrid compute units* (HCUs). A `HCU` is a collective, hybrid service-based units among which there exist different types of relationships, covering human-specific, software-specific, as well as human-software specific ones. A *HCU*, as a collective unit, can be elastic: it can be expanded and reduced based on specific conditions.

3.2 Relationships between Service Units

Using cloud computing provisioning models in which SBS and HBS are abstractly represented under the same service unit model with pay-per-use and on-demand service usage, a range of programming elements reflecting relationships among different types of service units are important and useful in building complex applications. Table 1 describes different types of relationships between service units that we consider as important programming elements, each of which applies to HBS, SBS or HCU.

Similarity: Given that certain tasks can be conducted by software or human, developers will need to compare HBS and SBS in order to select suitable ones for the tasks. We extend traditional similarity among SBS for HBS (e.g., simulation result analysis can be provided by two different research teams which are similar in terms of archetype and/or cost) or between HBS and SBS (e.g., specific image patterns can be detected by scientists or image processing software). From the programming perspective, similarity

Relationship Type	HBS	SBS	HCU	Description
Similarity	Yes	Yes	Yes	This traditional type of relationship indicates how similar a service is to another. In principle, similarity can be measured in terms of *functions*, *non-functional parameters* and *social contexts*.
Composition	Yes	Yes	Yes	This well-known type of service relationships indicates that a service is composed of several other services.
Data dependency	Yes	Yes	Yes	A service depends on another service if the former requires the latter for providing a certain data for one of its functions.
Control dependency	Yes	Yes	Yes	A service depends on another service if the outcome of latter determines whether former should be executed or not.
Location dependency	Yes	Yes	Yes	The locations of two service units are dependent, e.g., co-located in the same data center or country
Forwarding	Yes	Yes	Yes	This is a form of brokering/outsourcing in which a task is forwarded form one service to another.
Delegation	Yes	Yes	Yes	This is a form of brokering/outsourcing in which a service delegates a task to another service.
Social relation	Yes	No	Yes	This relationship describes different types of social relations (e.g. family or Linkedin connection) between two services.
Elasticity	Yes	Yes	Yes	This relationship describes how service units can be scaled in/out, replaced or (de)composed to offer similar functions but different cost, benefit and quality at runtime.

Table 1. Different types of relationships between services

can be specified in applications in terms of cost and quality (for all unit types), archetype (between HBS units), capability (between SBS units) and function (between HBS and SBS units).

Composition: Composing HBS and SBS units for complex tasks are possible. Therefore, we extend traditional composition relationships to cover also composites of hybrid services, such as describing how ICU can be composed with SBS to establish human-based filter. Composition can be in different forms such as data or control decomposition, and can be structured in different ways (e.g., star vs. ring structure).

Dependency: We support the classical view of dependency between services in terms of data (a service requires data provided by another service) and control (a service requires an successful completion of another service). Data and control dependencies can be programmed for any types of SBS and HBS. In particular, data exchange between two units can be conducted via other service units (e.g., two HBS can exchange data via Dropbox – a SBS). Furthermore, we consider *location dependency* which is crucial in clouds due to not only performance but also compliance requirements. Developers can use the location dependency to control the co-location of services.

Brokering: We consider brokering relationships for work distribution among service units. Two types of brokering relationships are considered: *delegation* (a service manipulates a request/response and delegates the request/response to/from another services) and *forwarding* (a service just forwards request/result to/from another service). With

hybrid services, such relationships can also be established between a SBS and a HBS, e.g., a SBS can decide where a SBS or an HBS will be used for evaluating the quality of data based on the type of the data.
Social relation: When using HBS for certain tasks in complex applications, we may require specific social relations among HBS solving the tasks, for example, two scientists who have conducted a joint research before. To support this, social relations are considered as programming elements.
Elasticity: This emerging relationship is due to the elasticity capability of services at runtime [10]. To the consumer, elasticity means that the expected service function is unchanged but the cost, benefits and/or quality can be scaled up/down at runtime. To the service provider, to enable the elasticity of costs, benefits, and/or quality, at runtime service units can be replaced by different variants or similar units or (re)composed by adding/removing appropriate units, or new compositions are introduced.

3.3 Quality, Cost, and Benefits

SBS and HBS have common and distinguishable quality, cost and benefit properties. In order to allow programmers to specify these properties, we support the following programming elements:

- Quality: represents common quality metrics and models for processing units and data. Quality can be further classified into `Performance` for processing capabilities of service units and `QoD` (quality of data) for input/output of service units. `Performance` and `QoD` can have several other sub entities, such as Response-Time, Availability, Accuracy, and Completeness.
- Cost: represents monetary pricing models, such as charging or rewarding models.
- Benefits: represents non-monetary benefits. It is classified into different entities, such as Return-on-Opportunity or Promotion.

We consider these properties as first-class programming elements since service units are constrained by various types of cost, benefit and quality models and the service provider wants to program her SBS/HBS/HCU to be able to scale in/out with expected quality under desirable cost and benefit at runtime. For example, in a situation with several real-time events signaling an emergency situation, an HCU might be programmed to reduce the accuracy of analytics in order to meet the response time to quickly react to the situation. On the other hand, in non-critical situations it could be programmed to utilize more (cheap) HBS to minimize the cost, maximize the accuracy, but accept an increasing response time as a trade-off. Therefore, treating these properties as first-class programming elements will allow the developer to explicitly specify, control, and enforce elastic constraints.

4 High-level Constructs for Hybrid Compute Units

From our proposed fundamental elements, in order to assist the development of complex applications, we develop a number of high-level constructs for service units and the relationships between them that help establish interactions among units in a hybrid

compute unit. Those constructs correspond to the conceptual model elements presented in Figure 1. Constructs for service units have a set of APIs that can be called upon the units. Constructs for a relationship have a set of (usage) patterns that can be used to establish the relationship. Constructs for cost, quality and benefits also have a set of APIs for specifying expected costs, quality and benefits. Using high-level programming constructs the developer can focus on the logic of the hybrid compute unit, instead of dealing with implementation-specific details of service units and complex algorithms for establishing relationships among units.

Construct	**Description**
$similarity(U, V, criteria)$	true if U is similar to V w.r.t. $criteria$
$datadependency(U, D, [M,]V)$	U producing data D which is needed by V. The optional $medium$ is the location associated with a DaaS (e.g., a Dropbox URL) where the data will be placed and shared.
$controldependency(U, V)$	declares that V should execute only after U finishes.
locationdependency(U, V, ctx, path)	declares that U and V should be linked in a given location context (e.g., country or data center) with a path in that context (e.g., city or server rack)
$composition(structure, type, U_1, U_2, \cdots, U_n)$	construct a composition of $U_1, U_2, ..., U_n$ for a given structure model and type
$forward(U, t, V)$	U forwards task t to V.
$delegate(U, t, V)$	U delegates task t to V.
$socialrelation(U, V, ctx, path)$	returns a distance relation between U and V in a given social context.
$?elasticity(U, [Func,]NFPs, x)$	x is a new form of U or x provides function $Func$ to satisfy given cost/quality/benefit models specified in $NFPs$.

Table 2. High-level constructs for relationships in hybrid compute units

Table 2 presents main programming constructs for relationships, each of which is abstractly represented as a function which takes a number of arguments. There are two types of functions: one that takes grounded variables (denoted as capital letters) as arguments, and one that takes free variable (denoted as lower case letters) as arguments. The latter is denoted with the symbol "?" in the function name. In the following, we explain some possible algorithmic patterns for high-level constructs for relationships:

Similarity: The construct $similarity(U, V, criteria)$ represents a similarity relationship between units U and V with regard to a given $criteria$ (namely "Cost", "Quality", "Archetype", or "Function"). A variant of this construct is $?similarity(U, x, criteria)$ which returns a set of units similar to U with regard to a given $criteria$ and store them in a free variable x. Pseudo algorithmic for this construct usages is shown below.

```
if (criteria == "Cost") return simCost(U, V);
else if (criteria == "Quality") return simQuality(U, V);
else if (criteria == "Archetype" && U.type == HBS && V.type ==
    HBS)   return (U.Archetype == V.Archetype);
else if (criteria == "Function") return U.Function==V.Function;
return false;
```

Data dependency: The construct $datadependency(U, D, M, V)$ states that V depends on U for data D and medium M where the data is stored. Variants of this construct include $?datadependency(x, D, M, V)$ (find unit x which provides data D needed by unit V), $?datadependency(U, D, M, x)$ (find unit x which needs data D), and $?datadependency(U, D, x[c], V)$ (find a medium x that can be used to share D between U and V satisfying a given constraint c). Pseudo algorithmic code for data dependency constructs are shown in the following:

```
datadependency(U, D, M, V) {
  Unit storageUnit = M;
  if (M==null) storageUnit = getDedaultMedium()
  request U stores D into storageUnit
  //get the URI indicating the location of the data
  URI uri = storageUnit.getURI(D)
  request V access D from uri
}
```

Location dependency: The construct $locationdependency(U, V, ctx, path)$ establishes a location dependency between U and V based on a specific context ctx and a specific $path$ in cxt. Here ctx can represent human-specific location context, such as the cloud platform providing HBS (e.g., based on Amazon Mechanical Turk) or the country, or cloud data center locations hosting SBS (e.g., Amazon EC2 EU site). The $path$ can indicate further dependencies in ctx, such as the same city or the same server rack in a data center.

Brokering: Delegation and forwarding relations are simply represented by $delegate(U, task, V)$ and $forward(U, task, V)$ where $task$ is a given task that needs to be delegated or forwarded. A variant of the delegation construct is $?delegate(U, task, x)$ which finds a appropriate unit x that U can delegate task t to. Pseudo code generated for $delegate(U, task, x)$ are given as follows:

```
for u: listUnit()
     for f:u.listFunction()
          if ((f.input == task.in) && f.output == task.out) {
                u.execute(task);
                U.waitUntil(task.finished == true);
                U.addInput(task.out);
                return;
           }
```

Social Relations: The construct $socialrelation(U, V, ctx, path)$ returns a distance between U and V (HBS only) via social relations in a given social context, denoted by $(cxt, path)$. It can also be used to establish a social relation constraint between U and V. The context ctx is a social network (e.g., Linkedin) and $path$ is a specific group in that network (e.g., data scientist). A negative distance (e.g., -1) indicates that there is no social relation found between U and V, whilst a value of 0 indicates that they belongs to the same given social group (e.g., in data science group on Linkedin). On

the other hand, a positive value indicates that they are related via some third parties who are directly related with them, e.g., A is a Linkedin colleague of B, B is a colleague of C, then the distance between A and C is 1. In order to find a HBS that is socially related to a given HBS within a specified distance, one can use the construct $?socialrelation(U, x, distance, ctx, path)$. The pseudo algorithmic code for $?socialrelation(U, x, distance, ctx, path)$ construct is as follows:

```
?socialrelation(U, x, distance, ctx, path) {
  for hbs:listHBS()
    //get the subgraph of the social network within a context
    Graph  socialNet = getSocialNetwork(ctx);
    //find the distance
    int d = social.findDistance(U, hbs, path);
    if d <= distance
        x.addElement(hbs);
  return x;
}
```

Elasticity: Elasticity construct can be used for different purposes. In the simplest case, the construct $?elasticity(U, elasticityReq, x)$ returns a new unit x that offer similar functions as unit U does but guarantees the elasticity requirement $elasticityReq$:

```
for v:listUnit() {
    boolean result = similarity(U,v,''function'')
    if (result)
         ElasticityCapability elasCap = v.
             getElasticityCapability();
         result=result && (matches (elasCap,elasticityReq);
    if (result) return x ;
}
```

Elasticity construct $?elasticity(Func, elasticityReq, x)$ returns a (new) unit x that offers function $Func$ as long as the elasticity requirement is met.

5 Illustrating Examples

To illustrate the "expressiveness" of our programming models, we use an illustrative application which is based on a real-world simulation application. Consider a multi-scale simulation application that utilizes different software as simulation solvers and visualization services. Typically, the simulation application includes several components, each of which is a SBS unit performing a particular task. These components can be used to pre-process data, execute solver engines, post-process results and analyze final results. In such an application, the quality of input, and the intermediate and final resulting data is crucial. Therefore, several components for evaluating quality of data (QoD) can be introduced into the application. Currently, such QoD evaluation components are rarely designed in the application. When redesigning the simulation application with QoD evaluation components, we face a problem: evaluating QoD cannot be done fully by SBS. We need to augment the application with human-based services to carry

out runtime quality evaluation. Furthermore, whether the employment of software or human-based service units for QoD evaluation is dependent on runtime aspects. Based on this application, we present how our programming elements and high-level constructs can be useful for implementing complex tasks using cloud APIs for SBS and HBS units.

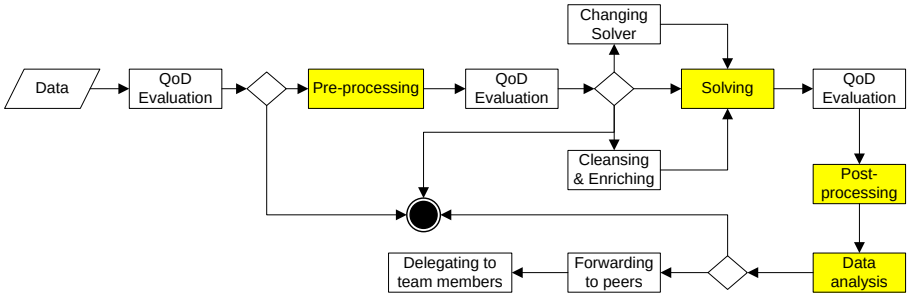

Fig. 2. Expected simulation components and their interactions using both SBS and HBS

Figure 2 describes expected simulation components and control flows using both SBS and HBS. Typically, only four main components are described in the simulation, namely `pre-processing`, `solving`, `post-processing`, and `data analysis`. However, by employing QoD-aware activities, we can introduce several new components for evaluating QoD and utilizing QoD to control the simulation. In the following we will illustrate how our programming elements and constructs can simplify the development of such new components and their interactions. For the sake of simplicity, we will not show the whole applications but illustrate main parts. *QoDEvaluation* components can be implemented different: (i) only SBS is needed, for example, in the QoDEvaluation step before `pre-processing`, (ii) SBS or HBS is used interchangeably, for example, in the *QoDEvaluation* after `pre-processing`, or (iii) only HBS is used, e.g., in *QoDEvaluation* after `solving`.

Programming elasticity and collectiveness in solving steps: Using different constructs, the programmer can invoke different types of units to deal with different situations. The following code excerpt shows examples of using ICU to check why the data is bad or to find solvers that can handle dirty data as long as they meet cost and quality requirements:

```
Double qodPreProcessedData =
    (Double)qodEvalUnit.execute("qodEvaluate", params1);
//get an ICU to check why data is bad
if (qodPreProcessedData < 0.5) {
//initiate a new unit
  ICU dataScientist = new ICU();
//create a dropbox place for sharing data
  DropboxAPI<WebAuthSession> scuDropbox = null;
  // ....
  DropboxAPI.DropboxLink link = scuDropbox.share("/hbscloud");
//ask the cloud of HBS to invoke the ICU
  VieCOMHBS vieCOMHBS = new VieCOMHBSImpl();
```

```
  vieCOMHBS.startHBS(dataScientist);
  HBSMessage msg = new HBSMessage();
  msg.setMsg("pls. use shared dropbox for communication " +link.
      url);
  vieCOMHBS.sendMessageToHBS(dataScientist, msg);
} else if (qodPreProcessedData < 0.7) {
//in this case, we just need a software to clean the data
     SBS dataCleansing = new SBS("datacleaner");
    //...
} else if (qodPreProcessedData < 0.9) {
  //specify some static properties of the solver
  SBS solverUnit2 = new SBS("solver");
  solverUnit2.capabilities.put("DIRTY_DATA", Boolean.valueOf(
      true));
  //specify expected cost and accuracy support
  CostModel costModel = new CostModel();
  costModel.price = 100; //max in EUR
  costModel.usageTime = 1000 * 60 * 60; //1 hour
  Quality quality = new Quality();
  quality.name = Quality.ACCURACY;
  quality.value = 0.95; // minimum value
  ArrayList nfps = new ArrayList();
  nfps.add(quality); nfps.add(costModel);
  //find solvers met quality and cost needs
  SBS elasticSolverUnit = (SBS)Relationship.elasticity(
      solverUnit2,nfps);        Object solverResult2 =
      elasticSolverUnit.execute("solving", params1);
} else {
       //....
}
```

Forwarding and delegating analysis request: After *post-processing*, in `data analysis`, an analyst can capture an unknown pattern which she can forward to her research connectors, who have a social relation to her in Linkedin. A professor may receive this pattern and he delegates the analysis tasks to his SCU, a set of graduate students. The following code excerpt shows the above-mentioned illustrative tasks:

```
ICU dataScientist = new ICU();
//....
ICU fUnit = new ICU();
Relationship.socialrelation(dataScientist, fUnit,1,"Linkedin:
    DataScienceGroup/TUWien");
Relationship.forward(data, fUnit);
//...
SCU studentSCU = new SCU();
//..
Relationship.delegate(data, studentSCU);
```

6 Conclusions and Future Work

In this paper, we investigate high level programming supports for solving complex problems using software-based and human-based compute units. We have presented a range of possible fundamental programming elements abstracting software and people and several possible high-level constructs. As the paper mainly discusses about high-level models and constructs, our validation is limited to illustrating examples and comparisons. We believe that programming elements and high-level programming constructs presented in this paper can be foundations for the development of domain-specific languages and software engineering processes for hybrid compute units. Our future work involves further developing our prototype and tooling support for the proposed high-level programming constructs.

References

1. Kittur, A., Smus, B., Khamkar, S., Kraut, R.E.: CrowdForge: crowdsourcing complex work. In: Proceedings of the 24th annual ACM symposium on User interface software and technology. UIST '11, New York, NY, USA, ACM (2011) 43–52
2. Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: Turkit: tools for iterative tasks on mechanical turk. In: Proceedings of the ACM SIGKDD Workshop on Human Computation. HCOMP '09, New York, NY, USA, ACM (2009) 29–30
3. Marcus, A., Wu, E., Karger, D., Madden, S., Miller, R.: Human-powered sorts and joins. Proc. VLDB Endow. **5** (2011) 13–24
4. Calheiros, R.N., Vecchiola, C., Karunamoorthy, D., Buyya, R.: The Aneka platform and qos-driven resource provisioning for elastic applications on hybrid clouds. Future Generation Comp. Syst. **28**(6) (2012) 861–870
5. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in time and space. In de Moor, O., Gottlob, G., Furche, T., Sellers, A.J., eds.: Datalog. Volume 6702 of Lecture Notes in Computer Science., Springer (2010) 262–281
6. Ahmad, S., Battle, A., Malkani, Z., Kamvar, S.: The jabberwocky programming environment for structured social computing. In: Proceedings of the 24th annual ACM symposium on User interface software and technology. UIST '11, New York, NY, USA, ACM (2011) 53–64
7. Minder, P., Bernstein, A.: Crowdlang: A programming language for the systematic exploration of human computation systems. In Aberer, K., Flache, A., Jager, W., Liu, L., Tang, J., Guéret, C., eds.: SocInfo. Volume 7710 of Lecture Notes in Computer Science., Springer (2012) 124–137
8. Truong, H.L., Dustdar, S., Bhattacharya, K.: Programming hybrid services in the cloud. In: 10th International Conference on Service-oriented Computing (ICSOC 2012), Shanghai, China (2012)
9. Fehling, C., Leymann, F., Ruetschlin, J., Schumm, D.: Pattern-based development and management of cloud applications. Future Internet **4**(1) (2012) 110–141
10. Dustdar, S., Guo, Y., Satzger, B., Truong, H.L.: Principles of elastic processes. IEEE Internet Computing **15**(5) (2011) 66–71