Managing Incentives in Social Computing Systems with PRINGL

Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria
{oscekic,truong,dustdar}@dsg.tuwien.ac.at

Abstract. Novel web-based socio-technical systems require incentives for efficient management and motivation of human workers taking part in complex collaborations. Incentive management techniques used in existing crowdsourcing platforms are not suitable for intellectually-challenging tasks; platform-specific solutions prevent both workers from comparing working conditions across different platforms as well as platform owners from attracting skilled workers. In this paper we present PRINGL, a domain-specific language for programming complex incentive strategies. It promotes re-use of proven incentive logic and allows composing of complex incentives suitable for novel types of socio-technical systems. We illustrate its applicability and expressiveness and discuss its properties and limitations.

Keywords: rewards, incentives, social computing, crowdsourcing.

1 Introduction

Human participation in web-based socio-technical systems has overgrown conventional crowdsourcing where humans solve simple, independent tasks. Emerging systems ([1, 2, 3]) are attempting to leverage humans for more intellectually challenging tasks, involving longer lasting worker engagement and complex collaboration workflows. This poses the problems of finding, motivating, retaining and assessing workers, as well as making the virtual labor market more competitive and attractive to workers. Paper [4] highlights a number of important research areas that need to be investigated in order to build such systems. Incentive management has been identified as one of the important parts of this initiative. However, contemporary incentive management in social computing systems usually imply a hard-coded and completely system-specific solution [5]. Such approach is not portable, and prevents reuse of common incentive logic. That hinders cross-platform application of incentives and reputation transfer.

Motivation. Our ultimate goal is to develop a general framework for automated incentive management for the emerging social computing systems. Such an *incentive management framework* could be coupled with different workflow or crowdsourcing systems, and, based on monitoring data they provide, would perform incentivizing measures and team adaptations. In this way, incentive management could be offered as a service in the cloud.

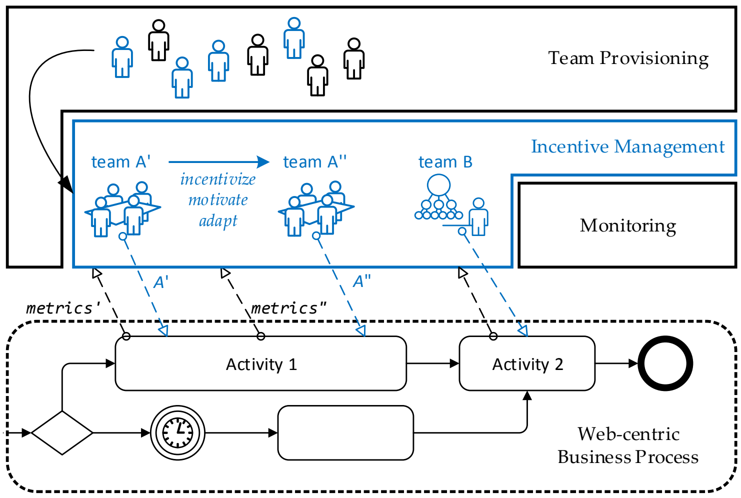


Fig. 1. Operational context of incentive management systems

Figure 1 visualizes the context in which an incentive management framework is supposed to operate: A complex business process execution employs crowd-sourced team(s) of human experts to perform various workflow activities. The teams are provisioned by a dedicated service (e.g., SCU [6]) that assembles teams based on required elasticity parameters, such as: worker skills, price, speed or reputation. However, choosing appropriate workers alone does not guarantee the quality of subsequent team's performance. In order to monitor and influence the behavior of workers during and across activity executions an incentive scheme needs to be enacted. This is the task of the incentive management framework. It enacts the incentive scheme by applying rewards or penalties in a timely manner to induce a wanted worker behavior, thus effectively performing runtime team adaptations (e.g., Fig. 1: $A' \rightarrow A''$).

Contribution. In [7] we presented a framework for low-level incentive management – PRINC. Although PRINC allowed monitoring of metrics and application of basic incentive mechanisms for social computing systems, it lacked a comprehensive, human-readable way of encoding incentive strategies, motivating us to design PRINGL[1] – a novel domain-specific language (DSL) for modeling incentives for socio-technical systems. In this paper we illustrate how real-world incentive mechanisms for social computing systems can be modeled in PRINGL.

Paper Organization. Section 2 gives an overview of PRINGL's design and intended usage. Section 3 introduces some of PRINGL's basic language constructs, describes the implemented language metamodel and discusses the advantages and limitations of the proposed approach. Section 4 presents the related work. Section 5 concludes the paper.

[1] **PR**ogrammable **IN**centive **G**raphical **L**anguage

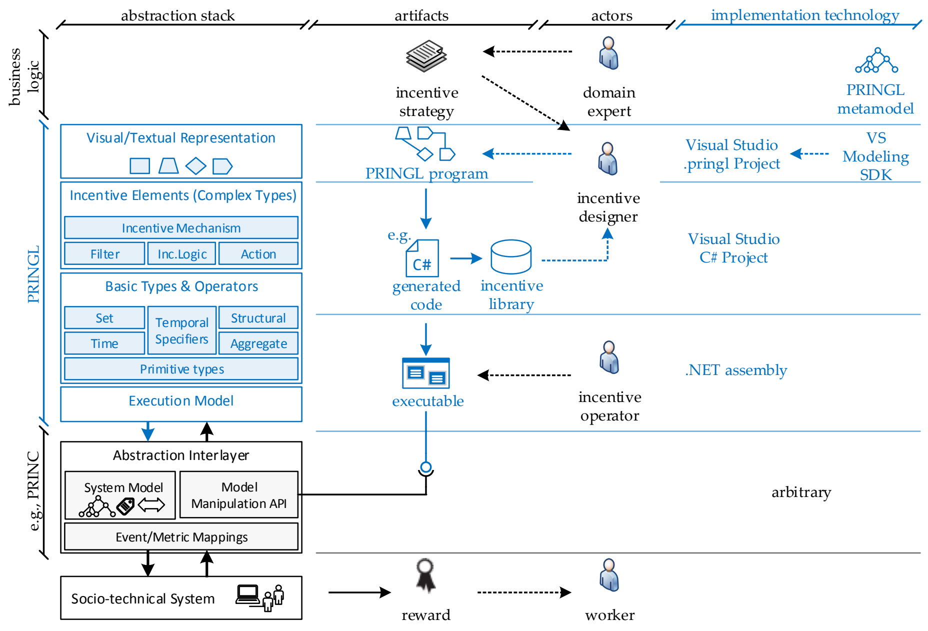


Fig. 2. Overview of PRINGL's architecture and usage

2 PRINGL Overview

Designing an incentive scheme is itself a challenging task usually performed by domain experts for a particular work type or company. However, as shown in [5, 8] most real-world incentive strategies used in social computing environments can be composed of modelable and reusable bits of incentive logic. PRINGL is a domain-specific language intended to be used by two types of *users* (Figure 2): a) *incentive designers* – domain experts that design and implement incentive strategies for different organizations (in particular crowdsourcing and socio-technical platforms); and b) *incentive operators* – members of the organizations responsible for managing the every-day running and adaptation of the scheme. While incentive designers may need to concern themselves with implementation details of the underlying system in order to adapt general incentive mechanisms for it, incentive operators want to manage the incentive scheme by using a simple and intuitive user interface without knowing implementation internals.

Figure 2 shows an overview of PRINGL's architecture and usage. An incentive designer models an incentive scheme using PRINGL's visual system-independent syntax. The PRINGL-encoded scheme gets translated into a system-specific executable able to exchange monitoring and incentive events with a social computing system through an *abstraction interlayer*. We use the term abstraction interlayer to denote any middleware sitting on top of a socio-technical system, exposing to external users a simplified model of its employed workforce and allowing monitoring of the workers' performance metrics. In [7] we presented an abstraction interlayer prototype, as part of the PRINC framework. For this paper, we re-use parts of the then implemented functionality (workers' structure model and timeline) to simulate an underlying social computing system (Section 3.2).

In order to build a language attractive for the targeted user types, PRINGL's design was guided by the following requirements: *a*) Usability – Provide an intuitive, user-friendly interface for incentive operators; *b*) Expressiveness – Provide an environment for programming complex real-world incentive strategies for incentive designers; *c*) Groundedness – Allow the use of *de facto* established terminology, components and methods for setting up incentive strategies; *d*) Reusability – Support and promote reuse of existing incentive business logic; *e*) Portability – Support system-independent incentive mechanisms, agnostic of type of labour or workers, and of underlying systems.

To meet the specified requirements PRINGL was conceived as a hybrid visual/-textual programming language, where incentive designers can encode core incentive elements, while incentive operators can provide concrete runtime parameters to adapt them to a particular situation. The language supports programming of the real-world incentive elements described in [5, 8] and allows composing complex incentive schemes out of simpler elements. Such a modular design also promotes reusability since the same incentive elements with different parameters can be used for a class of similar problems, stored in libraries and shared across platforms. PRINGL allows incentive designers to model natural-language, realistic incentive strategies (i.e., business logic) into a *platform-independent specification* through a number of incentive elements represented by a visual syntax (graphical elements with code snippets). The designer programs new incentive elements or reuses existing ones from an *incentive library* to compose new, more complex ones. Once the entire incentive scheme is specified, PRINGL translates it into a *platform-specific* code in a common programming language that can be further compiled into executable or library assemblies. The assemblies can then be used by incentive operators to execute and manage incentive enactment (Figure 2).

3 Modeling Incentives with PRINGL

The **incentive elements** are the basic functional units of a PRINGL program. Due to space constraints a detailed, conventional description of PRINGL's visual syntax and programming model cannot be presented here. Instead, in this section we briefly describe the functionality of the principal language constructs. The interested reader is encouraged to visit the PRINGL homepage[2] containing the full PRINGL specification, as well as other useful links and documents.

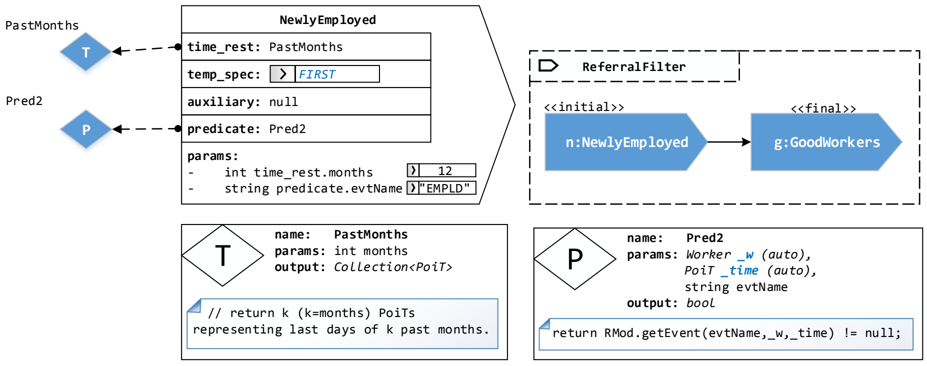
3.1 PRINGL Language Constructs

Incentive Logic. These constructs encapsulate different aspects of business logic related to incentives in reusable bits. They can be thought of as library-storable functions with predefined signatures allowing only certain input and

[2] `http://dsg.tuwien.ac.at/research/viecom/PRINGL`

output parameters. They are invoked from other PRINGL constructs, including other `IncentiveLogic` elements. Implementation is dependent on the abstraction interlayer, but not necessarily on the underlying socio-technical platform, meaning that many libraries can be shared across different platforms, promoting reusability of proven incentives, uniformity and reputation transfer. The Designer is encouraged to implement incentive logic elements as small code snippets with intuitive and reusable functionality. Depending on the intended usage, incentive logic elements have different subtypes: **A**ction, **S**tructural, **T**emporal, **P**redicate, **F**ilter. The subtype prescribes different input parameters and allows PRINGL to populate some of them automatically (marked with `auto`). Similarly, different subtypes dictate different return value types. These features encourage high modularization and uniformity of incentive logic elements. Incentive logic elements are denoted by a diamond shape surrounding the letter indicating the subtype, e.g., ◇T and ◇P in Fig. 3, bottom.

Worker Filter. Its function is to identify, evaluate and return matching workers for subsequent processing based on user-specified criteria. The criteria are most commonly related (but not limited) to worker's past performance and team structure. The workers are matched from the input collection of `Workers` that is provided by the PRINGL environment at runtime. By default, all the workers in the system are considered. The output is a collection of matching workers. We use a right-pointed shape F▷ to denote filters (Fig. 3, top left). A `CompositeWorkerFilter` definition consists of graphical elements representing instances of previously defined `WorkerFilters` (Fig. 3, top right).



```
// return k (k=months) PoiTs
representing last days of k past months.
```

```
return RMod.getEvent(evtName,_w,_time) != null;
```

Fig. 3. A `CompositeWorkerFilter` for referral bonuses

Example: A company wants to introduce employee referral process[3] in which an existing employee can recommend new employee candidates and get rewarded if the newly employed candidates spend a year in the company having exhibited satisfactory performance. In order to pay the referral bonuses the company needs

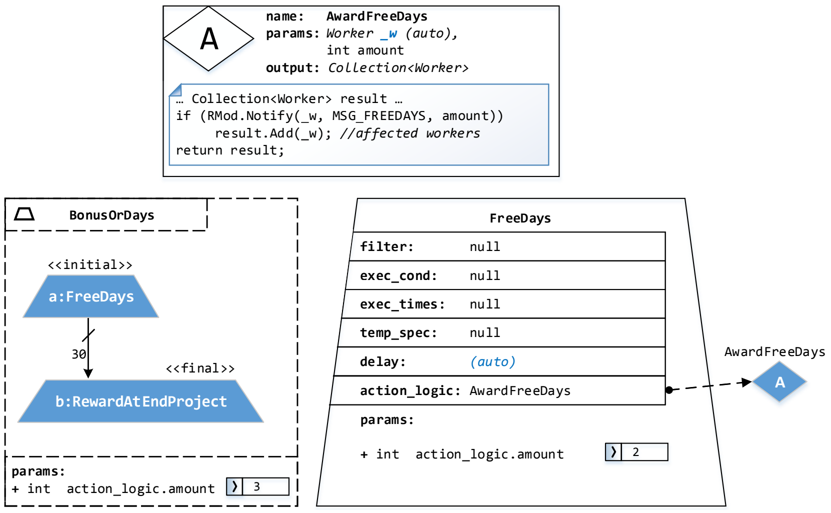
[3] `http://en.wikipedia.org/wiki/Employee_referral`

to: a) identify the newly employed workers; and b) asses the worker performance of those workers. Let us assume that the company already has the business logic for assessing the workers implemented, and that this logic is available as the library filter `GoodWorkers`. In this case, we need to define one additional simple filter `NewlyEmployed`, and combine it with the existing `GoodWorkers` filter. In Figure 3 we show how the new composite `ReferralFilter` is constructed. The $\boxed{F}\!\rhd$ instance `n:NewlyEmployed` makes use of: a) $\lhd\!T\!\rhd$ `PastMonths` returning time points representing end-of-month for the given number of months (12 in this particular case); and b) predicate $\lhd\!P\!\rhd$ `Pred2` checking if the employee got hired 12 months ago. `Pred2`'s general functionality is to check whether the abstraction interlayer (RMod) registered an event of the given name at the specified time.

Rewarding Action. Its function is to notify the abstraction interlayer that a concrete action should be taken against specific workers at a given time, or that certain specific actions should be forbidden to some workers during a certain time interval. In order to perform the action, the runtime environment needs to know to which workers the action applies, so a worker filter needs to be applied. In some cases, the workers that are rewarded/punished may be the same as initially evaluated ones. In that case we can reuse the original filter used for evaluation. In other cases, workers may be rewarded based on the outcome of evaluation of other workers (e.g., team managers for the performance of team members). The runtime also needs to determine the timing for action application. We use temporal specifiers (see PRINGL specification[4]) to determine the exact time moment(s). The output of a `RewardingAction` is a `Collection<Worker>` containing affected workers, i.e., those to which the action was successfully applied. To execute the action PRINGL needs to invoke the appropriate action in the abstraction interlayer which will then send out a system-specific message to the underlying system. We use a trapezoid shape /A\ to denote `RewardingAction` elements (Fig. 4, bottom right). Similarly to composite filters, a `CompositeRew-ardingAction` definition consists of graphical elements representing instances of previously defined `RewardingActions` (Fig. 4, bottom left).

Example: Consider a company that wants to reward workers either with free days or with a monetary reward. The choice is left to the worker. Free days are offered first. Only workers that refuse the free days will be given monetary rewards. We define a new composite rewarding action `BonusOrDays` (Figure 4) that, for the sake of demonstration, assumes the existence of a `RewardAtEnd-Project` action to award monetary bonuses, as well as a newly-defined action `FreeDays` to award free working days to the workers. The output of `a:FreeDays` is the set of workers who accepted the 3 free days offered. However, due to a complement edge ($\not\twoheadrightarrow$) connecting `a` and `b`, the output set of `a` is subtracted from the original input set. Therefore, the input of `b:RewardAtEndProject` are only those workers who declined to accept working days as award, and want to be evaluated at the end of project and paid a bonus according to their performance.

Incentive Mechanism. This is the main structural and functional incentive element used to express complex incentive schemes. It combines the previously



```
… Collection<Worker> result …
if (RMod.Notify(_w, MSG_FREEDAYS, amount))
     result.Add(_w); //affected workers
return result;
```

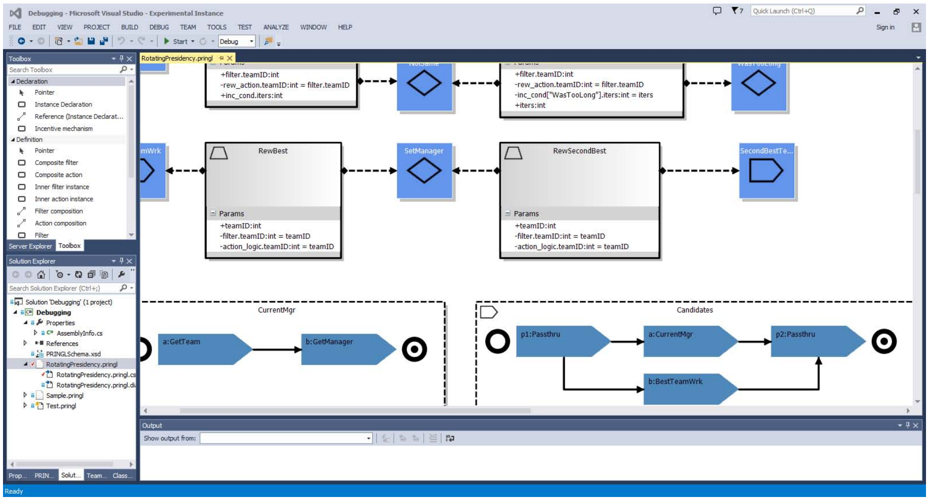
Fig. 4. A `CompositeRewardingAction` letting the workers choose one of the rewards

defined constructs (incentive elements) to select, evaluate and reward workers. As a self-sufficient and independent unit, it does not have any inputs or outputs. It can be stored and reused through instantiations with different runtime parameters. It also has dedicated GUI elements for definition and instantiation, as well as a shorthand notation – $\boxed{\text{IM}}$. Due to spatial restrictions, a full example of design and usage of incentive mechanisms is provided in supplement materials[4].

3.2 Implementation

Figure 2 (Section 2) shows the overview of implemented components. PRINGL's language metamodel prototype was implemented[4] in Microsoft's Modeling SDK for Visual Studio 2013 (MSDK). MSDK allows defining visual DSLs and translating them to an arbitrary textual representation. Using MSDK we generated a Visual Studio plug-in providing a complete IDE for developing PRINGL projects. In it, an incentive designer can create a dedicated Visual Studio PRINGL project and implement/model real-world strategies using the visuo-textual elements introduced in this paper (see Figure 5). The graphical elements provided in the implemented Visual Studio PRINGL environment, although not as visually appealing as those presented in this paper, functionally and structurally match them fully. PRINGL models are stored in `.pringl` files that get automatically transformed to the corresponding C# (.cs) equivalents. The generated code can then be used in the rest of the project as regular C# code or compiled in .NET assemblies (e.g., libraries or executables).

[4] Source code, screenshots and additional info available at:
`http://dsg.tuwien.ac.at/research/viecom/PRINGL/`



+filter.teamID:int
-rew_action.teamID:int = filter.teamID
+inc_cond.iters:int

+filter.teamID:int
-rew_action.teamID:int = filter.teamID
-inc_cond["WasTooLong"].iters:int = iters
+iters:int

+teamID:int
-filter.teamID:int = teamID
-action_logic.teamID:int = teamID

+teamID:int
-filter.teamID:int = teamID
-action_logic.teamID:int = teamID

Fig. 5. Screenshot. Implementing a realistic incentive scheme using PRINGL Visual Studio environment.

Figure 5 shows a screenshot of the implementation of the rotating presidency example[4] using the VS PRINGL IDE. The entire scheme was modeled using the generated PRINGL tools, demonstrating the feasibility of the proposed architectural design. The C# code obtained from the implemented model can be used to produce arbitrary incentive management applications, using PRINC as the acting interlayer (see Section 2).

3.3 Discussion

Advantages. In case of a conventional social computing platform (cf. [8]), the business logic necessary for enacting a platform-specific incentive scheme would likely need to be implemented and tested anew; subsequent changes would require changing the source code. With PRINGL, however, the incentive designer is likely to implement and test the basic incentive elements only once. PRINGL encouraging a modular design of incentive schemes composed of many small, easily testable components. With the basic elements available, composing complex ones can be done in a matter of minutes by visual modeling, copy-pasting and simple editing of fields and parameters. Once defined, the incentive elements can be stored in libraries and shared across different social-computing platforms promoting reusability and portability. PRINGL's composite actions enable the incentive designer to create tailored rewarding actions for different personality types or worker roles by combining a number of available rewarding mechanisms. As an additional benefit, by using standardized PRINGL incentive elements, comparing incentives across different social computing platforms becomes much easier. This is one of the fundamental requirements necessary to establish fair working conditions and sustainable virtual careers of crowdsourcing workers [4]. PRINGL's programming model was designed to support modeling of real-world incentive strategies from [5], thus addressing the desired Groundedness and Expressiveness requirements from the Section 2.

Limitations. So far PRINGL has been tested only in simulation environment with simple provisioning engines. However, it is important to point out that our goal is *not* to invent novel incentive mechanisms, nor to compare or improve existing ones. Rather, the focus is on functionally validating PRINGL's design and expressiveness in modeling documented, existing incentive mechanisms, thus not requiring evaluation with human subjects. As there are no known similar languages, a comparative qualitative evaluation was not possible. Also, at this moment PRINGL is limited to supporting worker-centric incentives only. Incentive adaptations currently require human intervention.

4 Related Work

Previous research on incentives for socio-technical systems is dispersed and problem-specific. It can be roughly categorized in two groups. One group seeks to find optimal incentives in formally defined environments through precise mathematical models [9]. Although successfully used in microeconomic models, these incentive models do not fully capture the diversity and unpredictability of human behavior that becomes accentuated in socio-technical systems. The other group examines the effects of incentives by running experiments on existing crowdsourcing platforms and rewarding real human subjects with actual monetary rewards. For example, in [10] the authors examine the effects of incentives by running experiments on existing crowdsourcing platforms and rewarding real human subjects with actual monetary rewards. In [11] the authors compare the effects of lottery incentive and competitive rankings in a collaborative mapping environment. In [2] the focus is on pricing policies that should elicit timely and correct answers from crowd workers. The major limitation of this research approach ([12]) is that the findings are applicable only for a very limited range of simple activities, such as image tagging and text translation. Two surveys of commonly used incentive techniques today can be found in [5, 8]. To the best of our knowledge, there have been no previous attempts of formalizing a general approach to incentive management for socio-technical systems.

5 Conclusions and Future Work

In this paper we introduced a domain-specific language named PRINGL for programming incentives for socio-technical systems. PRINGL allows the incentives to stay decoupled of the underlying systems. It fosters a modular approach in composing incentive strategies that promotes code reusability and uniformity of incentives, while leaving the freedom to incentive operators to adjust the strategies to their particular needs helping cut down development and adjustment time and creating a basis for development of standardized but tweakable incentives. This in turn leads to more transparency for workers and creates a basis for an incentive uniformity across companies; a necessary precondition for worker reputation transfer. In future, we plan to include support for artifact-centric incentives, and integrate PRINC into a general programming model for Hybrid Collective Adaptive Systems (HDA-CAS).

Acknowledgements. This work is supported by the EU FP7 SmartSociety project under grant №600854.

References

1. 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 2011, vol. 53 (2011)
2. Barowy, D.W., Curtsinger, C., Berger, E.D., McGregor, A.: Automan: A platform for integrating human-based and digital computation. SIGPLAN Not. 47(10), 639–654 (2012)
3. 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 2012. LNCS, vol. 7710, pp. 124–137. Springer, Heidelberg (2012)
4. Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., Horton, J.: The future of crowd work. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW 2013, p. 1301 (2013)
5. Scekic, O., Truong, H.L., Dustdar, S.: Incentives and rewarding in social computing. Communications of the ACM 56(6), 72 (2013)
6. Dustdar, S., Bhattacharya, K.: The social compute unit. IEEE Internet Computing 15(3), 64–69 (2011)
7. Scekic, O., Truong, H.-L., Dustdar, S.: Programming incentives in information systems. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 688–703. Springer, Heidelberg (2013)
8. Tokarchuk, O., Cuel, R., Zamarian, M.: Analyzing crowd labor and designing incentives for humans in the loop. IEEE Internet Computing 16(5), 45–51 (2012)
9. Laffont, J.J., Martimort, D.: The Theory of Incentives. Princeton University Press, New Jersey (2002)
10. Mason, W., Watts, D.J.: Financial incentives and the "performance of crowds". In: Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP 2009, pp. 77–85. ACM, New York (2009)
11. Ramchurn, S.D., Huynh, T.D., Venanzi, M., Shi, B.: Collabmap: crowdsourcing maps for emergency planning. In: Proceedings of 5th ACM Web Science Conference, Paris, France, pp. 326–335 (May 2013)
12. Adar, E.: Why i hate mechanical turk research (and workshops). In: Proc. of CHI 2011 Workshop on Crowdsourcing and Human Comp. ACM, Vancouver (2011)