Computer Networks 90 (2015) 14–33

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

PRINGL – A domain-specific language for incentive management in crowdsourcing☆

Ognjen Scekic*, Hong-Linh Truong, Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria

ARTICLE INFO

Article history:
Received 16 October 2014
Revised 23 February 2015
Accepted 15 May 2015
Available online 9 July 2015

Keywords:
Incentive management
Rewarding
Crowdsourcing
Socio-technical system
Social computing

ABSTRACT

Novel types of crowdsourcing systems require a wider spectrum of incentives for efficient motivation and management of human workers taking part in complex collaborations. Incentive management techniques used in conventional crowdsourcing platforms are not suitable for more intellectually-challenging tasks. Currently, incentives are custom-developed and managed by each particular platform. This prevents incentive portability and cross-platform comparison. In this paper we present PRINGL – a domain-specific language for programming and managing complex incentive strategies for socio-technical platforms in general. It promotes re-use of proven incentive logic and simplifies modeling, adjustment and enactment of complex incentives for socio-technical systems. We demonstrate its applicability and expressiveness on a set of realistic use-cases and discuss its properties.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Ever since the introduction of the term *crowdsourcing* in 2006 there has been a debate as to what exactly it should comprise (see [2]). When most of the community tacitly started applying the term to a family of micro-task platforms offered through an 'open-call' to anonymous crowds [3,4] a range of novel systems emerged attempting to leverage expert humans for more intellectually challenging tasks [5–8], by actively targeting preferred workers. These novel systems involve longer lasting worker engagement and complex collaboration workflows, often integrating the notion of team programmability. To highlight this distinction, some authors started naming these systems *socio-technical* or *social computing*. However, the principal trait of all these systems is that they need to manage interactions with and among human elements, referred to as *workers*, *agents*, *human services* or *peers*, performing different *tasks* (*jobs*) or *collaborative workflows* thereof.

While incentives were identified as one of the fundamental characteristics of conventional crowdsourcing systems [2], supporting more complex work patterns introduces novel challenges, with respect to finding, motivating and assessing (expert) workers executing them. Furthermore, in order to retain such workers the virtual labor market must be made more competitive and attractive [9]. In [9] the authors discuss the recent developments in the area and highlight a number of important research directions that need to be investigated in order to build such systems. *Incentive management* was identified as one of them. However, contemporary approaches to incentive management usually imply hard-coded, system-specific solutions (see Section 7). Such approaches are not portable, and prevent reuse of common incentive logic. That hinders cross-platform application of incentives and reputation transfer.

☆ Extended version of: O. Scekic, H.-L. Truong, S. Dustdar, Managing incentives in social computing systems with pringl, in: B. Benatallah, A. Bestavros, Y. Manolopoulos, A. Vakali, Y. Zhang (Eds.),Web Inf. Systems Engineering (WISE'14), Vol. 8787 of LNCS, Springer, 2014, pp. 415–424.

* Corresponding author. Tel.: +436504977327.
E-mail addresses: oscekic@dsg.tuwien.ac.at, oscekic@gmail.com (O. Scekic), truong@dsg.tuwien.ac.at (H.-L. Truong), dustdar@dsg.tuwien.ac.at (S. Dustdar).
URL: http://dsg.tuwien.ac.at (O. Scekic)

http://dx.doi.org/10.1016/j.comnet.2015.05.019
1389-1286/© 2015 Elsevier B.V. All rights reserved.

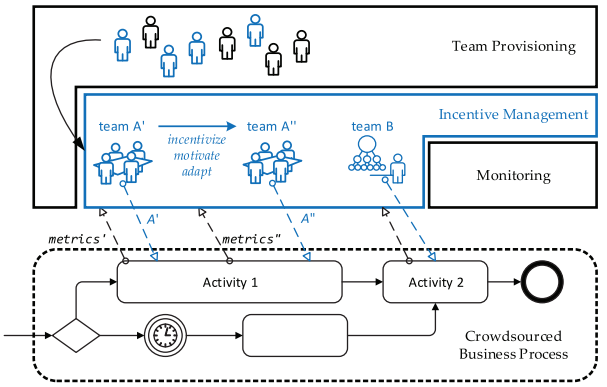


Fig. 1. Application context of incentive management systems.

Our ultimate goal is to develop a general framework for automated incentive management for the emerging crowdsourcing 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 externalized and offered as a service. Fig. 1 visualizes the context in which an incentive management framework is supposed to operate: A complex business process is being executed by employing crowdsourced team(s) of human experts to execute various workflow activities. The teams are provisioned by a dedicated service (e.g., Social Compute Unit – SCU [10,11]) that assembles teams of crowd workers based on required elasticity parameters, such as: 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 incentive management frameworks. They enact 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: $A' \rightarrow A''$).

Designing an incentive scheme is itself a challenging task performed by domain experts for a particular work pattern or company. As shown in [4,12] most real-world incentive strategies used in crowdsourcing environments can be composed of modelable and reusable bits of incentive logic. However, in Section 7 we also show that the efficacy of incentives can depend on multiple other factors, such as team size, cultural background, or knowledge of other participants. Therefore, the challenge is to design an incentive management framework capable of reusing existing and proven incentive mechanisms, but also allowing for easy tweaking to particular application contexts.

1.1. Contribution

The cornerstone of the previously described incentive management frameworks is the ***incentive programming model***, consisting of the two conceptual units:

(i) Incentive Model—supporting expression of a wide spectrum of incentive mechanisms suitable for crowdsourcing environments;

(ii) Execution Model—supporting enactment of the incentive mechanisms from *i*) onto crowd workers.

In this paper we present the programming model of PRINGL[1] – a novel domain-specific language (DSL) for modeling incentives for socio-technical and crowdsourcing systems. We describe PRINGL's modeling paradigm, and demonstrate its expressiveness by modeling a set of realistic incentive mechanisms. We then show how the modeled incentives can be enacted on a social-computing platform.

This paper substantially extends and refines our work presented in [1]. While the initial paper briefly presented the principal incentive elements, the extended version describes the entire incentive model, including a description of primitive elements and incentive operators (Section 4.1) and a significantly extended section on complex incentive elements (Section 4.3). Furthermore, we describe the allowed operations on the introduced incentive elements (Section 4.2.1), as well as the execution model (Section 4.4). In order to address the concerns of groundedness and applicability of our contribution, we present a substantial evaluation in Section 5, covering a set of realistic examples. The evaluation is based on the methodological approach outlined in the newly-added Section 2. Finally, the operational context and connectedness to our previous work are better explained (Section 3.2), and the Related Work (Section 7) extended.

The paper is organized as follows: Section 2 presents the research methodology. Section 3 gives an overview of PRINGL's architecture and intended use. In Section 4 the principal elements of PRINGL's incentive and execution model are presented. Section 5 shows how to model a set of realistic incentive schemes with PRINGL. Section 6 describes the implemented modeling tools. Section 7 presents the related work. Section 8 concludes the paper.

2. Methodology & background work

Our work is motivated by the lack of general and configurable incentive management solutions for (novel) types of crowdsourcing systems [9]. The purpose of any DSL is to allow the user to solve quickly and more easily some clearly identifiable problems in a domain, sacrificing in exchange the generality offered by a general-purpose programming language. In this case, the problem is to allow quick and uniform/portable modeling of commonly used incentive patterns identified in [12]. In order to design a useful DSL, we followed the general guidelines described in [13] to formulate design requirements, based on which we implemented and evaluated PRINGL's programming model.

As is common practice during the prototyping phase of DSL development, we evaluate the programming model qualitatively [13,14] – by establishing whether it is capable of meeting the formulated design requirements. Concretely, to establish that the offered functionality is grounded in reality, i.e., able to model a wide spectrum of real-world incentives, we show how PRINGL can be used to encode a number of example incentive schemes, chosen to cover most of the incentive patterns from [12] (Table 4). By describing and discussing the flexibility offered by PRINGL when modeling

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

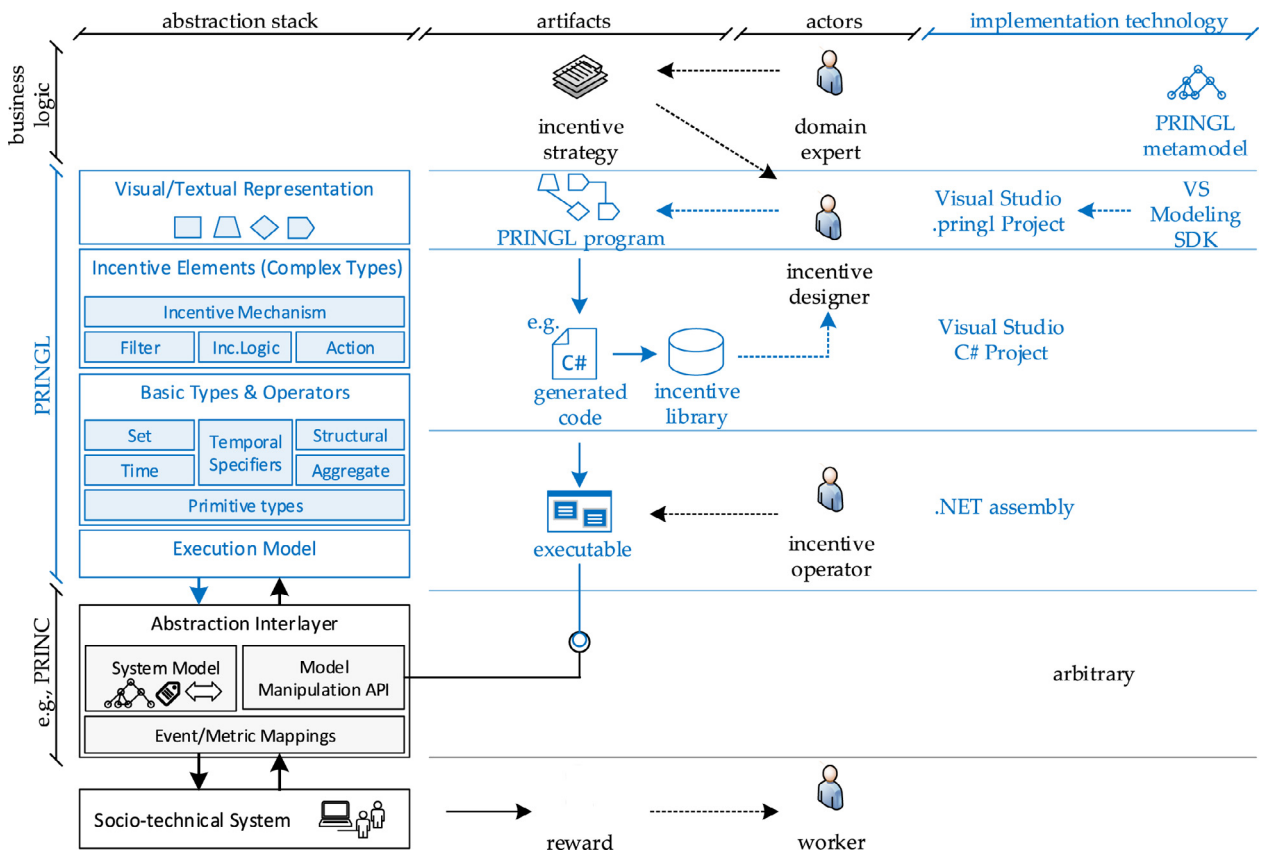


Fig. 2. A joint overview of PRINGL's programing model elements, architecture, users, operative environment and implementation. Implemented elements (Section 6) are marked in blue (lighter shaded). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article).

incentives in this example suite we argue for the usability aspect of PRINGL as well. Unfortunately, a fully rigorous usability claim could only be made after an extensive satisfaction survey of actual PRINGL users under real conditions, which is currently infeasible due to a lack of commercially exploitable end-to-end systems with incentive management capabilities. Finally, in Section 6 we present a prototype implementation of PRINGL's programming model, and use the implemented prototype to fully encode a complex incentive scheme from the example suite (Ex. 5), thus demonstrating the completeness of the proposed model and practically validating our implementation.

As shown in Section 7, previous research on incentives in crowdsourcing was mostly focused on concrete, application-specific incentive design and validation. To the best of our knowledge, there have been no previous attempts of formalizing a general and comprehensive approach to incentive management for crowdsourcing or socio-technical systems. A comparative evaluation of our research is therefore not possible. As the topic of this paper is not design or evaluation of particular incentive mechanisms on concrete crowdsourcing platforms, our evaluation does not include experimental or simulation testing of incentive application.

The basic incentive modeling concepts that were used to design PRINGL were inspired by, or based upon concepts previously introduced in the set of our background papers: in [15] we presented a possible model of the abstraction interlayer (Section 3.2); the basic functionality of the interlayer's incentive modeling capabilities were simulated and tested in [16]; and finally, in [10,17] we present components of a framework that allows provisioning and communication with collectives of human workers. We are currently working on integrating PRINGL and the aforementioned components into a single end-to-end socio-technical system with incentive management capabilities.

3. PRINGL overview

3.1. Users

PRINGL is a domain-specific language intended to be used by two types of *users* (Fig. 2): (a) *incentive designers* – domain experts that design and implement incentive scheme for an organization; and (b) *incentive operators* – organization members responsible for managing the every-day running and adaptation of the scheme.

An incentive designer (the Designer) is a multidisciplinary domain expert in the areas spanning management, economy, game theory and psychology. The Designer is hired by the crowdsourcing platform to design a set of appropriate incentive mechanisms for the given business model of the platform, taking into consideration context-specific

properties pertinent to the targeted population of workers. An example of how this process is performed for two different experimental platforms can be found in [18,19]. The role of an incentive operator (the Operator) has not been defined in the existing literature, as its existence is subject to the existence of the novel type of incentive management platforms that we describe in this paper. While a Designer can be a person external to the socio-technical platform, the Operator is a member of the management of the socio-technical platform in charge of monitoring the application of incentives and taking operative decisions on adaptations of various incentive parameters.

While Designers may need to concern themselves with implementation details of the underlying system in order to adapt general incentive mechanisms for it, Operators want to manage the incentive scheme by using a simple and intuitive user interface without knowing implementation internals. In Example 3 we showcase the parameters that are under Operator's control, while the whole of mechanism and the choice of which parameters are tweakable were defined by the Designer.

3.2. Operational environment

In order to enact a PRINGL-encoded incentive on a socio-technical platform (i.e., apply the incentives on real crowd workers), we need a simplified and uniform model of platform's workers, and the metrics and relationships that describe them. We call such a model together with the framework that manages it an ***abstraction interlayer*** (Fig. 2). More precisely, 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. The existence of an abstraction interlayer allows the incentive designer to write fully-portable incentives.

In [15] we presented a framework for low-level incentive management – PRINC. Although PRINC allowed monitoring of metrics and application of basic incentive mechanisms for socio-technical systems in general, it lacked a comprehensive, human-readable way of encoding incentive strategies, motivating us to design PRINGL. However, PRINC possesses all the characteristics of an abstraction interlayer. It features an abstract model (RMod) for representing the state of a socio-technical system, reflecting its quantitative, temporal and structural aspects. PRINC's mapping model (MMod) defines the mappings needed to properly express the platform-specific versions of metrics, actions, artifacts and attributes into their RMod cognates. Finally, PRINC takes care of exchanging messages with, and receiving update events from the underlying socio-technical platform, thus enabling the RMod abstract model to mirror the state of the underlying system. This in turn allows us to express incentive mechanisms decoupled from the underlying platform: to apply an incentive it suffices to alter the RMod state, while the task of mirroring this change onto the actual socio-technical platform is delegated to PRINC.

In this paper we assume the existence of PRINC as abstraction interlayer. The business logic code provided in the examples in Section 5 is C# code executable on PRINC. In theory, PRINGL can work without an abstraction interlayer. However, this would imply that all message handling with the underlying crowdsourcing system and complex monitoring logic would have to be written from scratch and placed into the incentive logic elements (Section 4.3). This contradicts one of the principal motives for introduction of PRINGL, and is more disadvantageous than building a completely system-specific incentive management solution.

3.3. Architecture

Fig. 2 shows an overview of PRINGL's architecture and usage. An incentive designer models an incentive scheme provided by a domain expert as a PRINGL program using PRINGL's visuo-textual syntax. The visually-expressed part of the syntax is completely system-independent, while system-specific business logic can be expressed as source code in an arbitrary programming language supported by the abstraction interlayer (see Section 4.3, Incentive Logic).

Starting from a PRINGL program the PRINGL code generator produces the following artifacts, encoded in a conventional programming language:

- An incentive model expressed in terms of incentive elements, basic PRINGL types and operators. This model also integrates the business logic code provided by the incentive designer. The incentive element definitions from this model can optionally be compiled into libraries for later reuse.
- Code for communication with the abstraction interlayer and application of the incentives.
- Code for manipulation of the incentive model.

These artifacts can be used to quickly build applications offering incentive management capabilities, e.g., a GUI-based application offering an incentive operator the possibility to change the runtime parameters. As previously explained, the abstraction interlayer takes care to communicate with the concrete socio-technical system, forward the rewarding actions and receive the updates.

3.4. Requirements

As PRINGL is a domain-specific language, the focus of the design requirements lies primarily on usability for its intended users (Section 3.1). In order to design an attractive language for the targeted users, the process was guided by the following requirements, formulated according to the guidelines outlined in [13]:

(a) *Usability* – provide an intuitive, user-friendly modeling DSL for incentive operators.
(b) *Expressiveness* – provide an expressive 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 labor or workers, and of underlying systems.

Table 1
Primitive types.

Type	Description
`Worker`	Represents an individual worker and his/her performance metrics.
`PoiT`	Represents a point in time. It can be instantiated by providing a fixed datetime or obtained as result of application of time operators.
`Interval`	Represents a named, addressable time interval. An interval can be: (a) *fixed*; and (b) adjustable. Fixed intervals have predefined starting and ending times, provided by two `PoiT`s, that cannot subsequently be altered. Adjustable intervals reflect the external system's changes intervals, e.g., deadline extensions (cf. *iterations* [15]). Changes are allowed to affect only points in future.
`Collection<T>`	An iterable collection of a primitive type T is also considered a primitive type.

Table 2
Built-in operators.

Operators	Description
Set operators	Union, intersection and complement on `Collection<T>`.
Time operators	Return `Collection<PoiT>` specifying times in which an action is expected. When working with adjustable intervals, their use guarantees that external changes will be observed. Commonly used with temporal specifiers.
Temporal specifiers	Instruct execution environment when to perform certain actions or evaluate predicates. As such, they cannot be directly used in user-provided programming code, but are rather offered as a choice through a visual GUI element (drop-down box), where needed. Internally, they are represented as built-in functions that operate on a collection of `PoiT`s provided by the environment at runtime.
Structural operators	Perform structural queries/modifications by examining/re-chaining relationships between worker nodes in the abstraction interlayer's (graph) model by using *graph transformations* [20].
Aggregation operators	Perform calculations on performance metrics or events over a `Collection<PoiT>`s in a fashion similar to SQL aggregate functions. The collection of time points over which they calculate is provided by the runtime environment at each invocation. A number of context-dependent restrictions apply on where they can be used.

4. Programming model

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 [4,12] 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 realistic incentive schemes (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 incentive scheme represents the whole of business logic needed for managing incentives in an organization. The scheme is expressed in PRINGL as a number of prioritized *incentive mechanisms* representing a PRINGL program. Each mechanism can them be further decomposed into a number of constituent incentive elements described in the following subsections. The designer programs new incentive elements or reuses existing ones from an incentive library to compose new, more complex ones. The following sections describe the incentive elements and operations on them. Due to spatial and readability constraints, the elements are not always fully described. For the same reason, the explanation of the code generation process is out of the scope of this paper. For more information the reader is referred to the supplement materials[2].

4.1. Primitive incentive elements & operators

From business logic perspective, primitive incentive elements represent the basic entities (workers, relationships and time units) that we use when composing incentive rules. From programming language perspective, they can be considered as atomic types that are used in user-provided or library code that specifies business logic. We use the two term: 'type' and 'incentive element' interchangeably. Apart from the four conventional primitive types: `string`, `bool`, `int` and `double`, PRINGL defines the types shown in Table 1. They do not have a direct visual representation. Only primitive elements can be used as inputs and outputs of *complex incentive elements* (Section 4.2). PRINGL provides a number of a useful operators for manipulating these types (Table 2)[2].

4.2. Complex incentive elements

Complex types enable PRINGL's core functionality and are represented by corresponding graphical elements. Their key property is that more complex types can be obtained by visually combining simpler ones. Visual, rather than purely textual representation was chosen to allow users to build up

[2] Extended descriptions are provided as supplement materials at http://dsg.tuwien.ac.at/research/viecom/PRINGL/

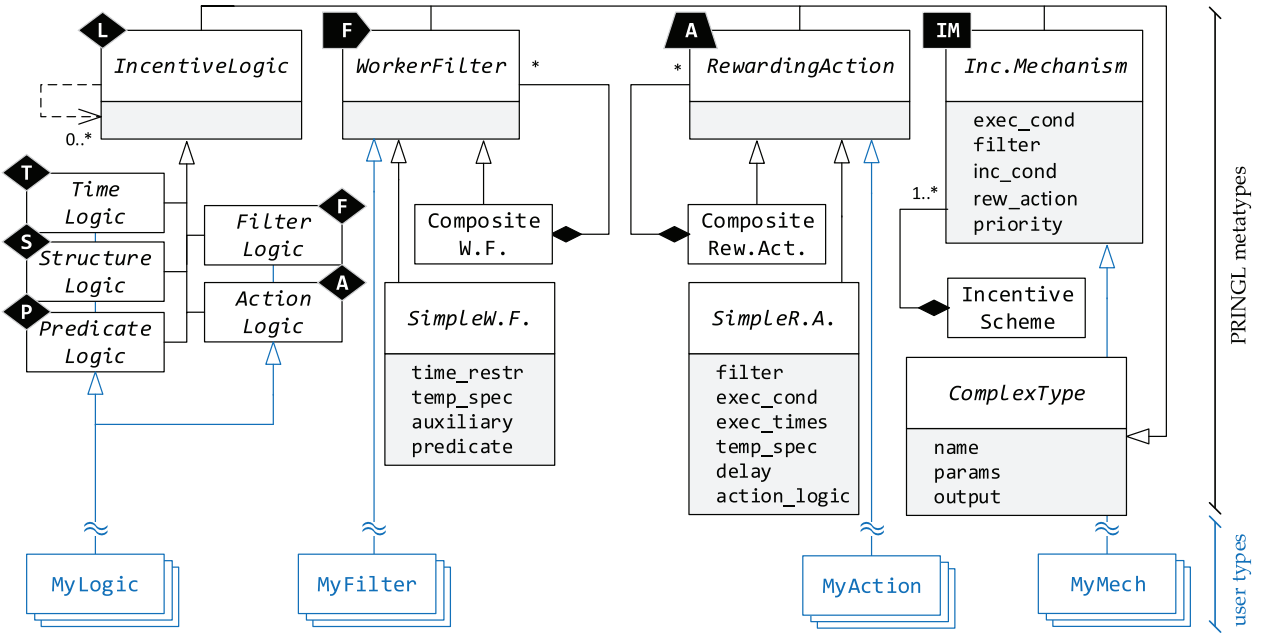


Fig. 3. Complex incentive elements class hierarchy.

complex incentive schemes by visually suggesting and restricting the choice of the possible components, thus facilitating the process of construction of incentive mechanisms. Complex incentive elements are managed through the following operations:

4.2.1. Operations on complex incentive elements

Definition – Complex types are defined by inheriting the following abstract metatypes: `IncentiveLogic`, `WorkerFilter`, `RewardingAction` and `Incentive Mechanism` (Fig 3). A new complex type inherits the predefined, addressable *fields* from the metatype it redefines. In order for a type definition to be complete, the fields must be filled out with appropriate values. Some fields are filled out automatically by PRINGL depending on the context where they are used (auto parameters); others must be filled out by the user (user-fields). The user-fields are: (a) `name`, which specifies the name of the new complex type; (b) arbitrary number of primitive-type input parameters (`params`) that can be used in evaluations and passed to other incentive elements; (c) type-specific fields[2], specifying how a particular functionality of the newly defined complex type is going to be executed – by indicating another incentive element to invoke, or by providing an executable code snippet. Definition is performed through appropriate graphical constructs being placed onto the working area. A new type definition retains its parent-metatype's graphical representation. For the non-auto input `params` (b), PRINGL visually exposes appropriate number of GUI form fields accepting the inputs that are to be filled out manually by the user. The input can contain expressions with primitive types and/or references to other accessible fields. To fill out type-specific fields (c), the user is expected to visually link the appropriate incentive element type, thus effectively declaring/instantiating it (see below).

Declaration/Instantiation – When defining new complex types, the user indicates (declares) which field/subcomponent instances will be required for PRINGL runtime to instantiate the newly defined object by placing the corresponding graphical (color-filled) element in the appropriate context within the working area, connecting it with appropriate connector from the parent type definition, and overriding parameter values from the parent type definition, if needed. The auto parameters are loaded at instantiation by PRINGL transparently to the user. Type instances are addressable objects that can be referenced (e.g., to read a field value) or invoked (see below) from other elements.

Indirect invocation – The `IncentiveLogic`, `WorkerFilter` and `RewardingAction` instances can also be 'invoked' just by being referenced from expressions in user-code. When the PRINGL code generator encounters an instance reference in an expression it transparently replaces it with an invocation of the default method for that type. Default methods for filters and rewarding actions return the resulting `Collection<Worker>`. The *default method* of a `IncentiveLogic` type is a function having input and output parameters as specified in its definition, and the user-provided code as the function body. The input parameters are provided by PRINGL runtime, so there is no need to pass any non-user parameters from the user code. Expressions containing indirect invocations can be used as field values (see Ex. 2, Fig. 11) or arbitrarily within the user-provided business-logic code in `IncentiveLogic` elements (see Ex. 3, Fig. 12, ①). Indirect invocation feature allows the user to pass instance references instead of output types of their default methods; for example, we can pass a filter instance to an `IncentiveLogic` element expecting a single input parameter of type `Collection<Worker>`. As these are common situations, indirect invocation helps cut down on verbosity of user code.

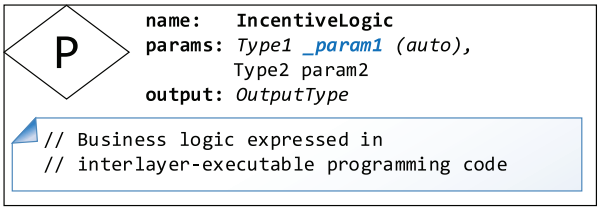


Fig. 4. Visual element representing an `IncentiveLogic` definition.

Static invocation – In addition to indirect invocation, `IncentiveLogic` elements can be invoked statically with arbitrary input parameters from the user code. In order to make the static invocation, the `IncentiveLogic` type name is appended with `.invokeWith([<param-list>])`; see Ex. 3, Fig. 12, ①.

4.3. Defining complex incentive elements

Incentive logic ◇. These constructs encapsulate different aspects of business logic related to incentives in reusable bits (e.g., determine whether a condition holds, read a metric value, or perform a simple action). They can be thought of as functions/delegates with predefined signatures allowing only certain input and 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: Action, Structural, Temporal, Predicate, Filter. Subtypes are needed to impose necessary semantic restrictions, e.g., the subtype prescribes different input parameters and allows PRINGL to populate some of them automatically[3]. Similarly, different subtypes dictate different return value types. These features encourage high modularization and uniformity of incentive logic elements. Incentive logic element definition is expressed in PRINGL with the visual syntax element shown in a Fig. 4, with appropriate subtype symbol shown in the upper left corner. As is the case with other incentive element definitions (presented in subsequent sections), the incentive logic element incorporates the distinguishing geometrical shape (diamond in this case), as well as auto-populated and user-defined parameters. Differently than other elements, it contains a field into which the Designer inputs executable code in a conventional programming language. The code captures the business logic specific to the incentive that is being modeled, but must conform to the rules imposed by the incentive logic subtype. As a shorthand, textual, inline notation for incentive logic elements we use a diamond shape surrounding the letter indicating the subtype, e.g., ◇T◇ for temporal logic.

[3] Marked with `auto` in figures

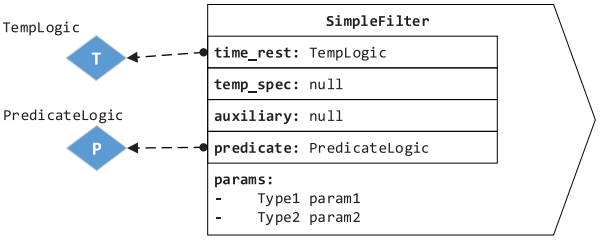


Fig. 5. Visual element used for `SimpleWorkerFilter` definition.

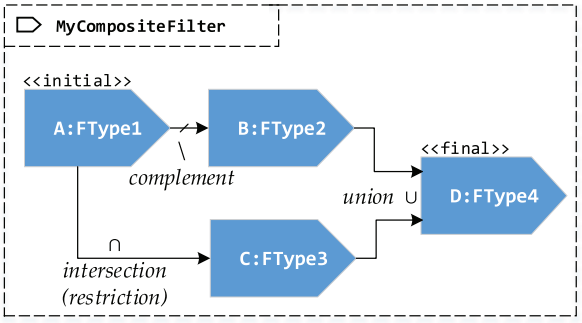


Fig. 6. An example `CompositeWorkerFilter` definition.

Worker filter ▷. The function of a `WorkerFilter` element 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 across different time points 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 workers satisfying the filter's predicate. Therefore, the functionality of a filter is to return a subset of workers from the input set, i.e., to perform a set restriction. Both `SimpleWorkerFilter` and `CompositeWorkerFilter` are subtypes of the abstract `WorkerFilter` metatype (Fig. 3), and can be used interchangeably where a worker filter is needed. A `SimpleWorkerFilter` element definition is expressed in PRINGL with the visual syntax element shown in Fig. 5, while a right-pointed shape ▷ is used as the inline, shorthand, textual denotation. Filter's type-specific fields are filled out visually by the user, by connecting them with appropriate incentive elements. The fields specify the time ranges over which to evaluate a worker (`temp_spec` and `time_rest` fields) and the predicate(s) (`predicate` and `auxiliary` fields) over metrics that need to hold.

In Fig. 6 we illustrate how a composite filter can be defined in PRINGL. It consists of graphical elements representing instances of previously defined, or library-provided `WorkerFilters`. The elements are connected with directed edges denoting the flow of `Workers`. There must be exactly one filter element without input edges representing the *initial filter*, and exactly one filter element without output edges representing the *final filter* in a composite filter definition. When a `CompositeWorkerFilter` is instantiated and executed, PRINGL provides the input for the initial filter, and returns the output of the final filter as the overall output of the

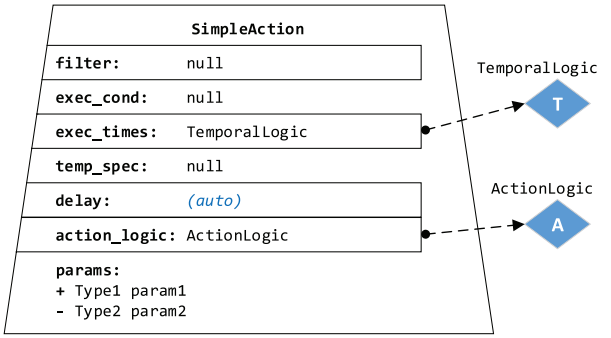


Fig. 7. Visual element used for `SimpleRewardingAction` definition.

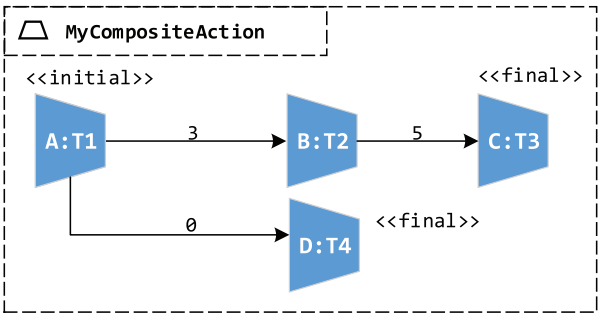


Fig. 8. An example `CompositeRewardingAction` definition with branch delays shown.

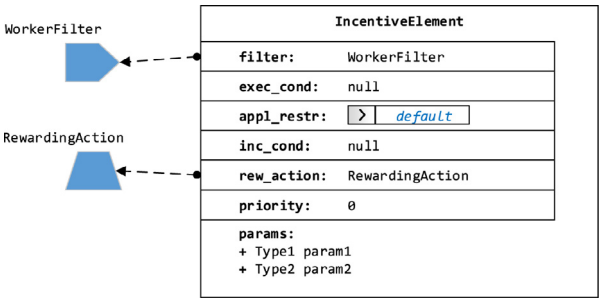


Fig. 9. An example `IncentiveMechanism` definition.

composite filter. As any other PRINGL composite type, a composite filter can also expose propagated or user-defined parameters.

A directed edge $\mathbb{A} \rightarrow \mathbb{C}$ implies that $\mathbb{C}$ takes as input $\mathbb{A}$'s output (the workers matching the criteria of $\mathbb{A}$). The output of $\mathbb{C}$ is a set containing workers fulfilling both filters' conditions, thus effectively representing $\mathbb{A} \cap \mathbb{C}$ operation. If an edge is marked as *negating* ($\nrightarrow$), then $\mathbb{A} \nrightarrow$ returns the set complement of $\mathbb{A}$'s input, i.e., $input(A) \setminus \mathbb{A}$. When multiple edges enter a single filter element, then the union ($\cup$) of workers coming over the edges is used as the input for the filter element. When multiple edges go out of a single element, then the same output set of workers is passed to each receiving end. Sometimes, we need a filter to forward a same set of workers to multiple filters or to collect workers from multiple filters without performing additional restrictions; the *pass-through* filter (predefined `PassThru` type) contains no logic, except for a predicate always returning `true`.

Rewarding action $\triangle$. Its function is to notify the abstraction interlayer (and consequently the crowdsourcing platform) 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. The rewarding actions can include, but are not limited to, the following: adjust reward rates (e.g., salary, bonus), assign digital rewards (e.g., points, badges, stars), suggest promotion/demotion or team restructuring, display a selected view of rankings to selected workers. The choice of the available actions is dependent of the set supported by the interlayer and the actual crowdsourcing platform. The abstraction interlayer is responsible for translating the action into a system-specific message and delivering it to the underlying crowdsourcing platform. PRINGL expects the underlying system to acknowledge via abstraction interlayer that the suggested action was accepted and applied to a worker, because its outcome may affect other incentive mechanisms. We use a trapezoid shape shown in Fig. 7 to denote the definition of a `SimpleRewardingAction`. For the shorthand notation, we use $\mathbb{A}$, both for simple and composite rewarding action elements.

In PRINGL's programming model the output of a `RewardingAction` is a `Collection<Worker>` containing affected workers, i.e., those to which the action was successfully applied. Informing the abstraction layer is performed a side-effect of executing the rewarding action. 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 used (`filter` field). 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). PRINGL's runtime also needs to determine the timing for action application (`temp_spec` and `exec_times` fields). We use temporal specifiers (see Section 4.1) to determine the exact time moment(s) of the time series. For defining incentives involving *deferred compensation* [12] we also need to specify an additional predicate that will be evaluated at the execution time establishing whether a worker fulfilled the reward criteria during the period from when the incentive was scheduled until the execution point (`exec_cond` field). The actual action to execute is determined by the `action_logic` field, pointing to a concrete $\diamond$ element.

Similarly to composite filters, a `CompositeRewarding Action` definition consists of graphical elements representing instances of previously defined `Rewarding Actions` (Fig. 8). Both `SimpleRewardingAction` and `CompositeRewardingAction` are subtypes of the abstract `RewardingAction` metatype (Fig. 3), and can be used interchangeably where a rewarding action is needed. The sub-elements are connected with directed edges denoting at the same time: a) *worker flow*; and b) *time delay*.

A `RewardingAction` returns *affected* workers and passes them over outgoing edges if it is member of a composite action. Affected workers are those workers on which the action was successfully applied by the underlying crowdsourcing system. The passing of workers is similar to that of

composite filters. The differences are explained in the supplement materials. Each edge can optionally specify a time delay as a non-negative integer without the unit. If omitted, zero is assumed. The actual unit is determined transparently to the user as the basic time unit of the abstraction interlayer. PRINGL forwards the delay value to the action that the edge points to.

Incentive mechanism ▭. `IncentiveMechanism` is the main structural and functional incentive element. It uses the previously defined complex types to select, evaluate and reward workers of the crowdsourcing platform. A complete *incentive scheme* can be specified by putting together multiple incentive mechanisms, prioritizing them, and turning them on/off when needed. As other complex types, incentive mechanism also has dedicated GUI elements for definition and instantiation (Fig. 9), as well as a shorthand notation used in this paper – ▭. Table 3 defines the functionality of ▭'s fields. We show examples of the usage of ▭s and other incentive elements in the following section.

4.4. Execution model

The execution of a PRINGL program (incentive scheme) is performed in cycles, as follows:

All ▭s are triggered for execution whenever a *triggering signal* from the abstraction interlayer is received. It is the responsibility of the Designer to ensure through priorities and execution conditions that a specific order of execution of ▭s is achieved. The order of execution of ▭s with the same priority is not predetermined. Execution conditions of the ▭s with higher priorities are evaluated first. Only after the higher-priority ▭s have executed are the conditions of lower-priority ones evaluated. This allows the higher priority mechanisms to preemptively control the execution of lower-priority ones by changing condition variables through side effects. The execution time of any single ▭ is limited by design to the time needed to pass the message to the underlying crowdsourcing platform. The execution of an ▭ begins by evaluating `exec_cond`. If true, the associated `filter` is passed the collection of all the workers in the system and invoked. The resulting workers are then passed to the `incentive_cond` to decide whether the execution should proceed with rewarding. If it returns true, `rew_action` is invoked. If the action does not override its `filter` field PRINGL passes the collection of workers returned by the ▭'s `filter` field.

A 𝔽 executes by checking for each worker from the input collection whether it fulfills the provided `predicate`. This is done for each `PoiT` returned by `time_restr` (◈). The results are then interpreted in accordance with the provided `temp_spec`. For example, if the specifier is `Once()` then it suffices that the worker fulfilled the predicate in at least one of the `PoiT`s in order to be placed in the resulting collection. In case of composite filters the constituent sub-filters are executed in the defined order. The initial filter receives the initial collection of workers from the environment, which is then passed on to subsequent filters. The resulting collection of workers from the final filter is returned as the overall result.

A simple ◬ is executed if the `exec_cond` (◈) returns true. In this case, the execution `PoiT`s for the action are obtained from `exec_times` (◈) and then interpreted in accordance with the `temp_spec`. Once the times are determined, the environment schedules the action in the abstraction interlayer (in our case PRINC's *Timeline*) and provides the actual code that performs the action from the `action_logic` (◈). However, during the entire runtime PRINGL keeps track of the scheduled action, in order to honor temporal specifications and to detect re-scheduling due to `Interval` redefinitions. The workers to which the action applies are taken from the associated `filter`. As explained, if the local filter is omitted, PRINGL assumes the workers from the parent ▭ 's `filter`.

The execution of a composite action starts by first breaking it into linear execution paths containing constituent simple actions. For each execution path PRINGL takes into account specified delays and adjusts the ◈ elements in constituent actions to account for provided delays, which are then (re-)scheduled with the abstraction interlayer. However, as in this case we need to pass worker sets between actions happening at different times PRINGL stores the intermediate results (worker sets) that actions scheduled for a future moment will collect when executed (memoization). In case more than one action is scheduled for execution at the same time, the order is unspecified.

Executing incentive logic elements ◈ equals to invoking the instance similarly to a conventional function. The

Table 3
Description of `IncentiveMechanism` fields.

Field	Description
`exec_cond`	An optional ◈ element used as execution condition for the entire mechanism. Used to check global and time constraints. The condition is commonly used to prevent unwanted multiple executions of the same mechanism. Defaults to true if omitted
`appl_restr`	Specifies how often a mechanism can be executed in a given interval. The runtime environment then alters the `exec_cond` accordingly, transparently to the user. This field can be used to turn mechanisms on or off to obtain different incentive scheme configurations
`filter`	An optional 𝔽 specifying the default target Workers for the ◬ specified in field `rew_action`. If not provided, if defaults to the collection of all the workers in the system. The filter is used to evaluate workers' past or current performance
`inc_cond`	An optional ◈ used to interpret the workers returned by the filter and decide whether to proceed with the rewarding. This condition is meant to be used when the evaluated and targeted worker groups are not the same. In that case, we need to decide whether the results of the evaluation performed through the filter should cause the invocation of the action(s). Returns true if omitted
`rew_action`	A mandatory ◬ assigning the reward or penalty
`priority`	An optional `int` indicating the priority of mechanism's execution. Zero by default

environment passes both the auto parameters and any user-defined ones. If user-defined parameters are omitted when a ◇ is invoked from the code by indirect invocation the parameters are obtained from the visually exposed parameter fields. However, when supplied, the arguments provided in the code override those provided in the fields. If the parameter value cannot be resolved in either way, the invocation fails.

Overall, PRINGL's execution is 'best effort'. This means that PRINGL expects the interlayer to pass to the underlying socio-technical system the rewarding actions to be taken, but will not expect them to be necessarily observed. Acknowledgments are used to keep track of successfully applied rewarding actions. If any error is encountered during the execution, the currently invoking incentive mechanism fails gracefully, but the execution of other mechanisms continues. The incentive scheme's execution needs to be stopped explicitly.

5. Evaluation

A domain-specific language (DSL) can be evaluated both quantitatively and qualitatively. *Quantitative analysis* of the language is usually performed once the language is considered mature [13], since this type of evaluation includes measuring characteristics such as productivity and subjective satisfaction, that require an established community of regular users [14].

During the initial development and prototyping phase, we use the *qualitative evaluation* [13], which, in general, can include: comparative case studies, analysis of language characteristics and monitoring/interviewing users. Analysis of language characteristics was chosen as the preferred method in our case, since it was possible to perform it on the basis of the findings gathered through analysis of numerous existing incentive models [12]. Due to difficulties in engaging a relevant number of domain experts willing to take part in monitoring we were unable to perform this type of user-based evaluation at this point. Comparative analysis was not applicable in this case, due to nonexistence of similar languages.

In order to qualitatively evaluate characteristics of PRINGL in Section 5.2 we constructed an example suite covering realistic incentive elements identified in [12]. By implementing the suite examples we showcase the various language characteristics necessary for a comprehensive coverage of the domain, thus demonstrating PRINGL's groundedness and expressiveness. Through discussion of particular implementation details, we demonstrate PRINGL's reusability and portability. While lacking the necessary conditions and metrics to conclusively show the usability of the language, the implemented set of examples allows us to argue for certain aspects of usability, such as 'usefulness' and 'portability' (from [14]).

5.1. Modeling real-world incentive elements

Paper [12] presents a review of literature on incentives and surveys existing incentive practices of 140 crowdsourcing companies and organizations. Based on the outcomes of this survey, the paper identifies the basic categories of incentives, and their key building elements – evaluation methods and rewarding actions. A short description is provided below:

Incentive categories

- *Pay per performance (PPP)* – workers are rewarded proportionally to the contribution. The contribution is calculated through context-specific metrics.
- *Quota system/Discretionary bonus* – the contribution is assessed at known time points or over predefined intervals. If the level of contribution exceeds a threshold for the monitored time frame, the worker is rewarded.
- *Deferred compensation* – a worker is promised a reward for current effort, but the actual rewarding action is applied after some time, and only if a condition is satisfied at that specified moment in future.
- *Relative evaluation* – a worker/artifact is evaluated with respect to other workers/artifacts within a specific group and rewarded according to the relative score.
- *Promotion* – a limited number of better positions are available for a group of workers to compete for (tournament theory). Involves a structural change reflecting the change in position and managerial relations.
- *Team-based compensation* – when individual contributions are not easily distinguishable, team members are equally compensated according to the overall, measurable success of the team as a whole.
- *Psychological incentives* – designed to act on human feelings. They usually provoke competitive reaction, but other consequences are possible as well, such as respect from colleagues, professional satisfaction, fear of dismissal. Psychological incentives need to be carefully tailored to suit the targeted social milieu.

Each of these incentive categories can be generalized and implemented to use different evaluation methods and/or rewarding actions. Sometimes, they can be interchangeable; in other cases, the context narrows down the choice. For example, if a crowdsourced worker is participating in a text-translation task, then the metric based on which the worker is evaluated in a PPP type of incentive can be obtained by monitoring the amount of translated text, or alternatively other translators can be asked to provide a vote on the quality of the translated text. As long at the evaluation can be quantified, the actual way how a particular evaluation is obtained remains transparent to the rest of the incentive. In the design-contests, on the other hand, the quantity of produced designs is largely irrelevant. The quality of artistic contribution can be evaluated only through human-based peer evaluation. The rewarding actions and evaluation methods encountered in practice and literature can be classified as follows ([12]):

Rewarding actions

- *Quantitative reward*– a quantitative change of the parameters targeting directly/indirectly worker's performance (e.g., salary increase, bonus, free days).
- *Structural change* – restructuring of collaboration, communication or management relationships in which worker takes part (e.g., change of collaborators/teams, delegation patters, promotion).

Table 4
Coverage of incentive categories, rewarding actions and evaluation methods by the provided examples.

	Ex. 1	Ex. 2	Ex. 3	Ex. 4	Ex. 5
Incentive category					
PPP			✓		
Quota/Discretionary			✓		
Deferred compensation	✓				
Relative evaluation			✓		
Promotion					✓
Team-based compensation		✓			
Psychological				✓	✓
Rewarding action					
Quantitative		✓	✓		
Structural					✓
Psychological				✓	✓
Evaluation method					
Quantitative			✓	✓	✓
Peer voting		✓			

- *Psychological action* – indirect motivation of workers by exposing them to information designed to increase competitiveness, collaboration, compassion, sense of belonging. Examples include: displaying rankings of similar/competing workers and digital badges.

Evaluation methods

- *Quantitative evaluation* – rating of individuals based on objective, measurable properties of their contribution. Does not require human participation. Fully implementable in software only.
- *Indirect evaluation* – rating of individuals calculated based on objective, relative evaluation of artifacts they produce with respect to artifacts produced by other participating individuals, under a closed-world assumption. Fully implementable in software only.
- *Subjective evaluation* – based on subjective opinion of a single peer-worker (e.g., manager, team leader), or statistically insignificant number of peers.
- *Peer voting* – based on aggregated votes of a statistically significant number of peer workers.

5.2. Examples

In this section, we present an example suite designed to cover most of the presented real-world incentive categories and their constituent parts (see Table 4)[4]. Due to spacing constraints, some examples are presented partially.

5.2.1. Example 1 – employee referral

A company introduces employee referral process[5] in which an existing employee can recommend new candidates and get rewarded if the new employee spends a year in the company having exhibited satisfactory performance.

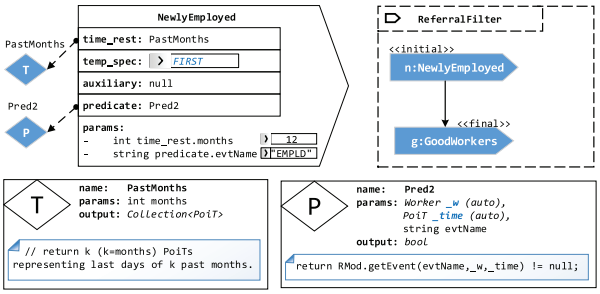


Fig. 10. A `CompositeWorkerFilter` for referral bonuses.

Solution: In order to pay the referral bonuses (deferred compensation) the company needs to: (a) identify the newly employed workers; and (b) assess their subsequent performance. 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 Fig. 10 we show how the new composite `ReferralFilter` is constructed. The 𝔻 instance `n:NewlyEmployed` makes use of: (a) ◈ `PastMonths` returning `PoiTs` representing end-of-month time points for the given number of months (12 in this particular case); and (b) predicate ◈ `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.

Discussion: The shown implementation fragment illustrates how easy it is to expand on top of the existing functionality. Under the assumption that there exists a metric for assessing the workers' performance, and that it can be queried for past values (cf. PRINC's `Timeline`), introducing the 'employee referral' mechanism is a matter of adding a handful of new incentive elements.

5.2.2. Example 2 – peer voting

Equally reward each team member if both of the following conditions hold: (a) each team member's current effort metric is over a specific threshold; and (b) the average vote of the team manager, obtained through anonymous voting of its subordinates, is higher than 0.5 [0–1].

Solution: As shown in Fig. 11 we compose the incentive scheme consisting of two 𝕀𝕄s – `i1:PeerAssessIM`, in charge of peer voting; and `i2:RewardTeamIM` in charge of performing team-based compensation. 𝕀𝕄 `i1` will execute first due to the higher priority, and set the global variable `done`, through which the execution of `i2` can be controlled (◈ `PeerVoteDone`). 𝕀𝕄 `PeerAssessIM` uses the 𝔻 `TeamMembers` to exclude the manager from the rest of team members. The `TeamMembers` is a composite filter composed of two subfilters 𝔻 `GetManager` 𝔻 `GetTeam`, borrowed from Ex.5, Fig. 15. The resulting workers are passed to 𝔸 `DoPeerVote` which performs the actual functionality of peer voting. The referenced rewarding action is simple; it just passes to ◈ `PeerVote` the workers that need to participate. The ◈ `PeerVote` is performed by dispatching messages to the workers and receiving and aggregating

[4] Note that the Indirect and Subjective evaluation methods have been omitted from Table 4. Former, because it implies use of sophisticated evaluation algorithms, but implementation-wise would not differ from the Quantitative evaluation. Latter, because is not easy to uniformly model in software, as it implies subjective human opinions that are unknown at design-time.

[5] http://en.wikipedia.org/wiki/Employee_referral

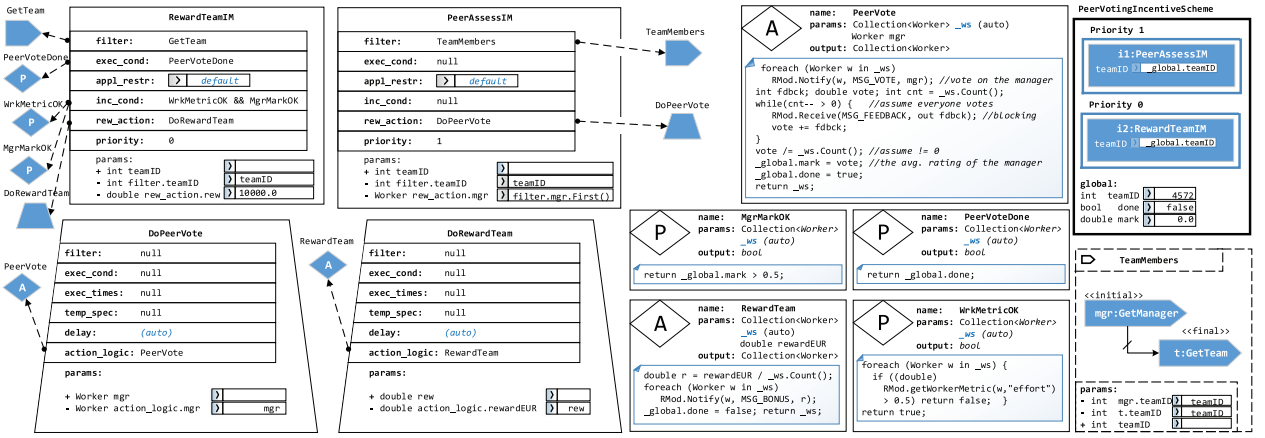


Fig. 11. An incentive scheme example combining peer voting and team-based compensation.

their feedback through the abstraction interlayer. Once the peer voting has been performed, the manager's assessment is stored in `_global.mark`, and the flag `_global.done` is set to allow execution of ▣ `i2`. Once set to execute, the ▣ `i2` first reads all the team members via ▷ `GetTeam`. Whether they ultimately receive the reward depends on the evaluation of the `inc_cond` field. The field contains a conjunction of two indirectly invoked ◇ elements (Section 4.2.1). The condition expresses the two constraints from the incentive formulated in natural language. If it resolves to 'true', the △ `DoRewardTeam` applies a predefined monetary reward, sharing it equally among all team members (via ◇ `RewardTeam`).

Discussion: The key question here is how to support incentives requiring direct human feedback, such as peer voting. Such interactions require support from the abstraction interlayer. To support this functionality, the abstraction interlayer can either rely on the functionality offered by the underlying crowdsourcing platform, or provide this functionality independently to safeguard the voting privacy and incite expression of honest opinions. In [17] we presented SMART COM – a framework for virtualization and communication with human agents. In this example we model the latter option in PRINGL, assuming the use of PRINC with SMARTCOM for interaction with workers.

5.2.3. Example 3 – bonus

Award a 10% bonus to each worker W that sometimes in the past 12 months had higher value of metric 'effort' than the average of workers related to W via relationship of type 'collab', and not rewarded in the meantime.

Solution: Fig. 12 shows the bottom-up implementation of this incentive (①–⑤). First, at level ① we define novel or context-specific business logic fragments as `IncentiveLogic` ◇ elements. This level relies on the abstraction interlayer to read the updated worker metrics, obtain data about recorded events, or send system messages. At ② we define new ▷ and △ types. Similarly, ▷ and △ definitions are further used for defining new composite filters and actions (③) and `IncentiveMechanisms` (④). By setting the parameter fields the designer specifies the necessary runtime parameters for different instances. Apart from constants, a field can contain references to other fields 'visible' from that element. The environment collects the field values (parameters) from all the constituent subcomponents and propagates them upwards, possibly until the top-most component's GUI form. Through the +/– symbols the designer controls whether to propagate a parameter and, thus, delegate the responsibility for filling it out to the upper level, or provide a value at the current level and hide it from upper levels. Parameter propagation is one of PRINGL's usability features. In Fig. 12 we show an example of parameter propagation (marked in orange/light shade). Element ◇ `PastProjects` (①) exposes the parameter `months`. The same parameter is then re-exposed by ▷ `BetterThanAvg` (②) that uses `PastProjects` as its time restriction. The parameter is further propagated up through ▷ `MyExampleFilter` until it finally gets assigned the value in ▣ `EndProjectBonus` (④).

Discussion: This incentive mechanism was chosen to highlight a number of important concepts. Every underlined term in the natural language formulation of this incentive mechanism is a specific value of a different parameter that can be changed at will. In PRINGL terms, this means that incentive operator can easily switch between different (library) incentive elements of the same type/signature and tweak the parameters to obtain different incentive mechanism instances. In this way, incentive designers or operators can adapt generic mechanisms to fit their needs. If we analyze the generic version of this incentive mechanism, we can see that it embodies the principles of pay-per-performance incentives, based on the value of a quantifiable metric, but coupled with the additional condition that is evaluated relatively to co-workers. In addition, the mechanism contains two temporal clauses ('in past 12 months' and 'in the meantime'), making it also a representative of a quota-system type of incentive.

The example also demonstrates reusability – the ◇ `PastProjects` is reused twice in two different ▷s. Also, steps Fig 12: ①–④ can be skipped altogether if the necessary type definitions are already available from the incentive library. As we can see, at levels ②–⑤ only visual programming is required. This means that there is no need to know any interlayer internals, apart from understanding the meaning of propagated parameters. So, if different platforms offer standardized implementations of the commonly used

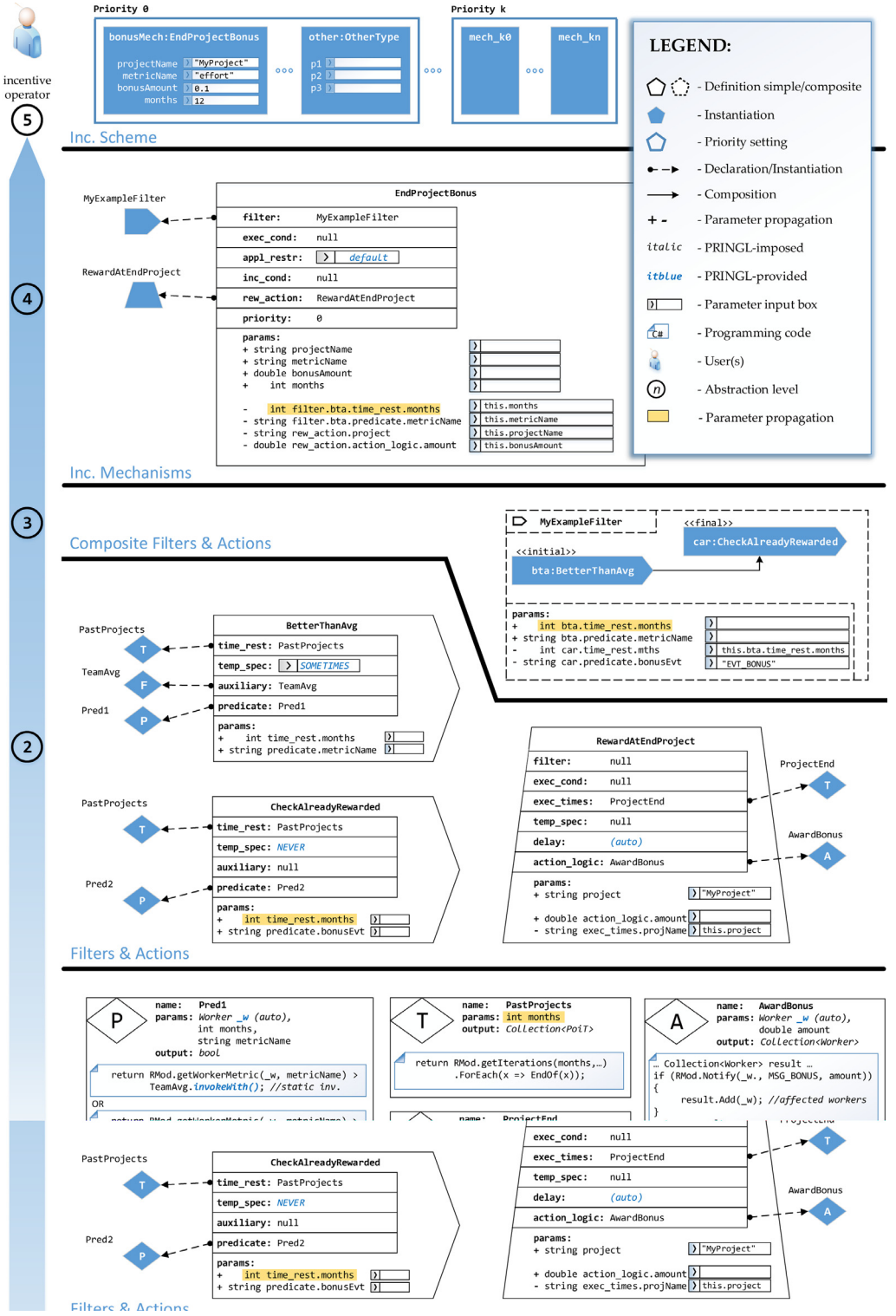


Fig. 12. Incentive scheme from Example 3, illustrating the decreasing of complexity going from modeling of (low-level) incentive elements by incentive designers to adjusting existing incentive schemes by incentive operators. (For interpretation of the references to colour in this figure in the text, the reader is referred to the web version of this article).

incentive logic, the incentive elements become completely portable.

5.2.4. Example 4 – rankings

Let us assume that the imaginary platform from Example 3 wants to extend the existing incentive scheme with an additional incentive mechanism in an (admittedly oversimplified) attempt to raise competitiveness of underperforming workers: *Show the list of the awarded employees and their performance (rankings) to those workers that did not get the reward through application of* EndProjectBonus *in Ex. 3 (Fig. 12).*

Solution: Fig. 13 shows the additional elements needed to support the new mechanism. The composite NonRewardedOnes reuses the existing MyExample Filter from Ex. 3 as initial subfilter, and returns the set complement, i.e., the non-rewarded workers to which the rankings need to be shown. In order to display the rankings, we copy-paste the existing RewardAtEndProject from Ex. 3 and change only the value of the field action_logic to point to the newly defined ShowRankings, also shown in Fig. 13. Let us name the newly obtained RankingsAtEndProject. In the same fashion, we copy-paste the existing EndProjectBonus from Ex. 3, make its filter and rew_action fields point to the newly defined NonRewardedOnes and RankingsAtEndProject, respectively. The obtained performs the requested functionality.

Discussion: This example shows a common, realistic scenario, where additional incentive mechanisms need to be added to complement the existing ones. In this case, the added mechanism acts on the underpeforming workers psychologically by showing them how they fare in comparison to the rewarded workers. Such mechanisms can be used to motivate better-performing underperformers ('lucky losers'), while having a de-motivating effect on the worst performing ones. As we have shown, such a mechanism can be easily and quickly constructed in PRINGL with a minimal effort.

5.2.5. Example 5 – rotating presidency

Teams of crowd workers perform work in iterations. In each iteration one of the workers acts as the manager of the whole team. This scheme motivates the best workers competitively by offering them a more prestigious position in the hierarchy. However, in order to keep team connectedness in a longer run, foster equality and fresh leadership ideas, a single person is prevented from staying too long in the managerial position. Therefore, in the upcoming iteration the team becomes managed by the currently best-performing team member, unless that team member was already presiding over the team in the past k iterations.[6]

Solution: For demonstration purposes, we are going to model on-the-spot all the type definitions necessary for implementing the rotating presidency incentive scheme. However, in practice it is reasonable to expect that a significant number of commonly-used type definitions would be available from a library, cutting down the incentive modeling time.

[6] An iteration can represent a project phase, a workflow activity or a time period.

Contrary to Example 3, this time we adopt a top-down approach in modeling. In order to express the high-level functionality of the rotating presidency scheme the Designer uses PRINGL's visual syntax to define an incentive scheme named RotatingPresidency (Fig 14, top) containing (referencing) two instances – i1 and i2, with the same priority (0). The RotatingPresidency scheme definition also contains a set of global parameters that are used for configuring the execution of the scheme: teamID uniquely defines the team that we want the scheme applied to, while iters specifies the maximum number of consecutive iterations a team member is allowed to spend as a manager. By choosing different parameter values an incentive operator (Operator) can later adjust the scheme for use in an array of similar situations in different organizations.

The two incentive mechanisms that the scheme references – i1 and i2, are instances of the types RewardBest and PreventTooLong, respectively (Fig 14, bottom). The RewardBest installs the best worker as the new manager if (s)he is not the manager already. The PreventTooLong will replace the current manager if the worker stayed too long in the position, even if the manager resulted again as the best performing team member. 'Installing' or 'replacing' a manager is actually performed by rechaining of management relations in the structural model of the team by applying appropriate graph transformations [21] through the abstraction interlayer.

When the incentive condition (inc_cond field) of PreventTooLong evaluates to true, this means that the actual manager occupied the position for too long, and that it should be now replaced by the second-best worker. PRINGL does this by invoking the specified RewSecondBest and passing it the collection of workers returned by the Candidates. The Candidates returns potential candidates for the manager position – the best performing Worker and the current manager. The same filter is referenced from both s.

Two rewarding actions are instantiated and invoked from the s. The RewBest monitors the 'effort' metric and rewards the best worker in the current iteration. The RewSecondBest replaces the current team manager with the second-best performing worker when needed. The inc_cond fields make sure that the two actions do not get executed in the same iteration.

We now show how the previously referenced filters are defined. We will first describe the definitions of the three simple filters (Fig 15, right) and then use them to visually assemble the definitions for another four composite filters (Fig 15, left).

- GetTeam: Returns all the workers belonging to the team with the specified teamID.
- GetBest: Returns the worker having achieved the highest value of the 'effort' metric by invoking the GetWrkBestMetric and then just formally matching it with the IsBest predicate. In this example we use the 'effort' metric [11], but any other compatible performance metric could have been used and exposed as a global parameter. This filter does not care to which team the evaluated worker belongs – if used independently, it would

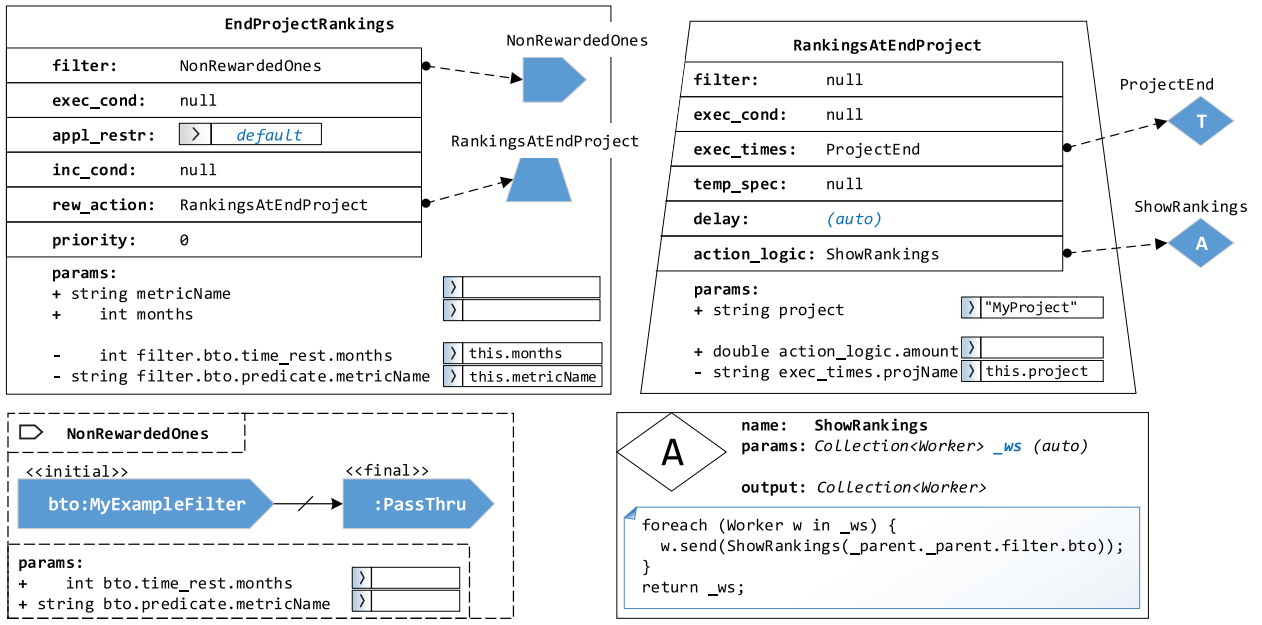


Fig. 13. Additional incentives elements needed to augment the incentive scheme from Example 3 (Fig. 12) in order to display motivational rankings to the non-rewarded workers from Example 3.

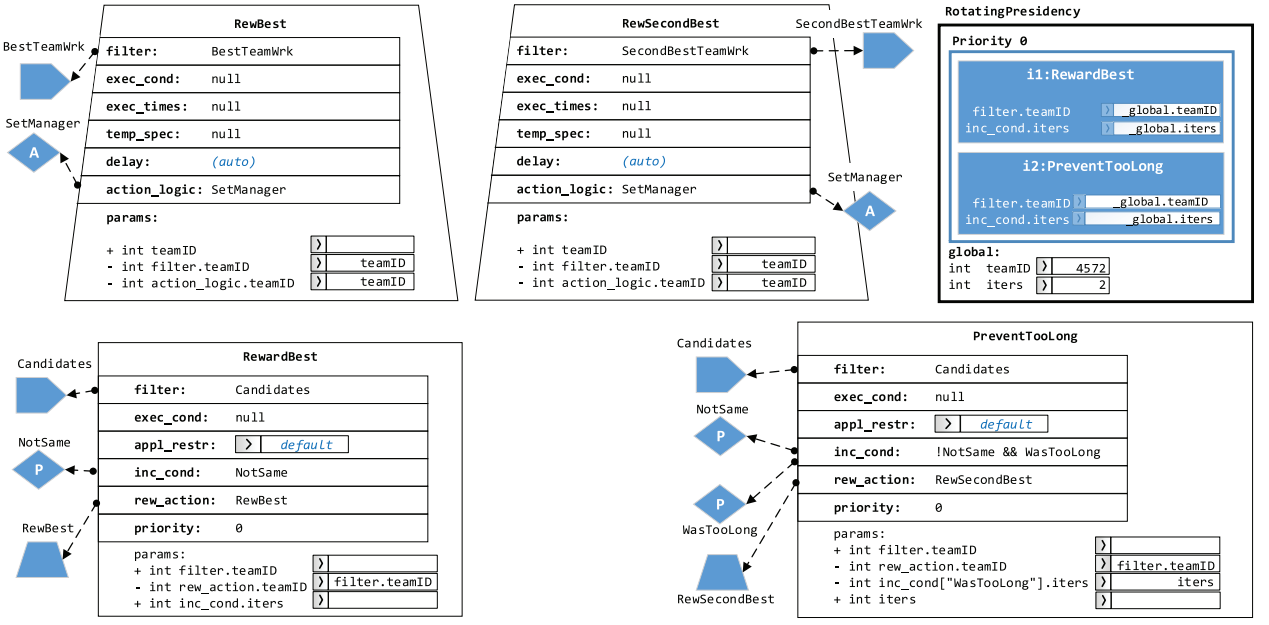


Fig. 14. Modeling the rotating presidency incentive scheme in PRINGL. Segment showing the incentive scheme (top right), rewarding actions (top center and left), and incentive mechanisms (bottom).

evaluate all the workers in the system. This is why we always use it in composite filters, where we initially restrict its input set with another filter.

- `GetManager`: Invokes a ◈ `GetMgrByRelations` that performs a graph query [21] on the team model through the abstraction interlayer to determine the manager within the provided input set of workers.

Composite filter type definitions are constructed visually. The following composite filters are defined:

- `CurrentMgr`: Returns the current team manager. The subfilter `a` returns all the workers belonging to the team with the given `teamID`, while the subfilter `b` uses managerial relationships to determine the manager among those workers.
- `BestTeamWrk`: Returns the best individual from a previously identified collection of team members.
- `SecondBestTeamWrk`: Returns the second best worker in the team. The subfilter `a` returns the best worker of the team and passes it forward to the subfilter `b` via a negated

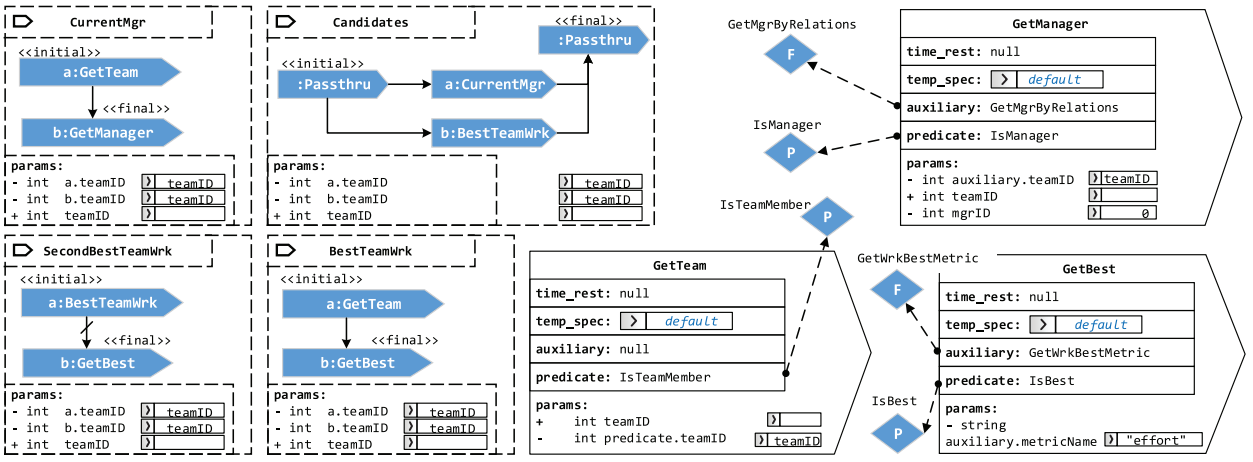


Fig. 15. Modeling the rotating presidency example: Segment showing simple filters (right) and composite ones (left).

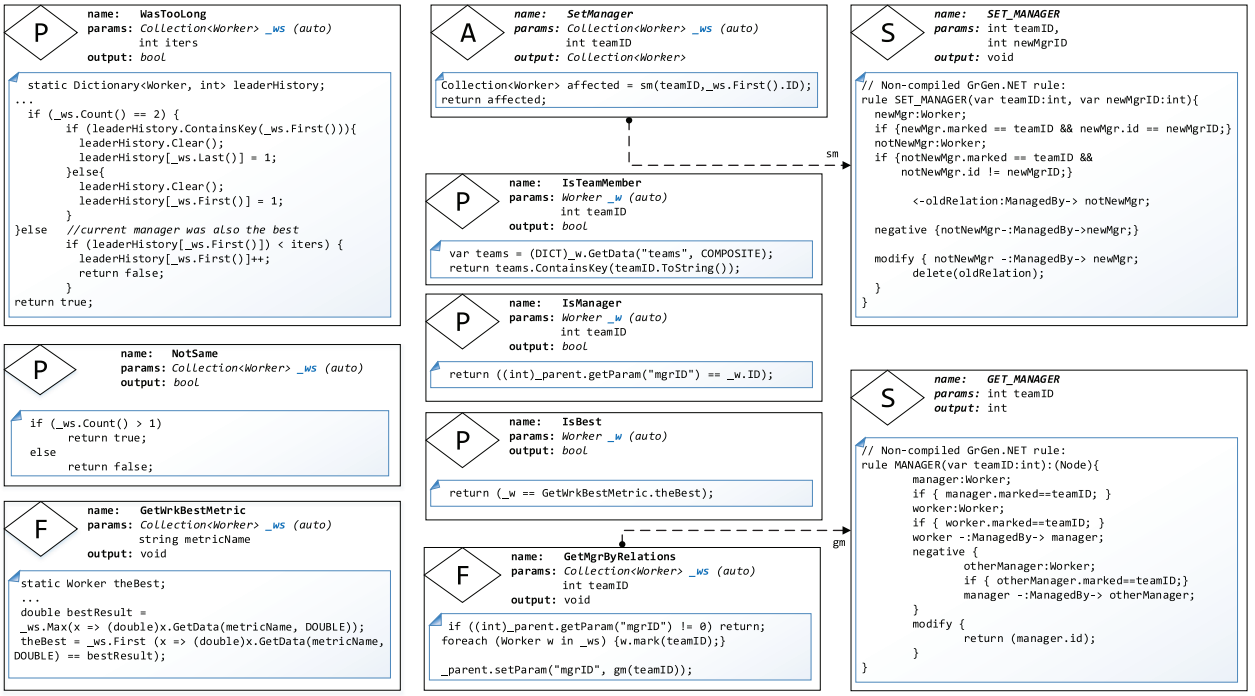


Fig. 16. Modeling the rotating presidency example: Segment showing the incentive logic elements.

edge (↛). This means that `b` now receives as input: *input(a)* \ `a`, i.e., in this particular case the collection of all workers belonging to the team minus the best worker. Subfilter `b` returns the best worker from this collection, and thus effectively the second best worker of the team.

- `Candidates`: This filter simply uses the previously defined filters `CurrentMgr` and `BestTeamWrk` and returns the set union of their results.

Incentive logic elements, shown in Fig. 16, contain the low-level business logic and code[7] that communicates with the abstraction interlayer. Designer takes care to implement incentive logic elements as small code snippets with intuitive and reusable functionality. A short description of the functionality of the employed elements is provided in Table 5.

Discussion: This example combines the promotion and psychological incentives. The promotion is performed through a structural rewarding action, and is designed to foster competitiveness and self-prestige. At the same time, team spirit and good working environment are being promoted by limiting the number of consecutive terms, thus giving a chance to other team members. This example shows a fully implemented and executable incentive scheme. Although the model may seem complex at the first glance, it is worth noting that the type definitions of the two actions (Fig 14, top)

[7] In this paper we use C# in all but ◈ elements, which are shown in the original GrGen.NET rule language: http://www.info.uni-karlsruhe.de/software/grgen/

Table 5
Incentive logic elements used in the rotating presidency example.

Element	Symbol	Description
`IsTeamMember`	◈	Determines whether a worker belongs to a team.
`IsManager`	◈	Checks if the currently evaluated worker has the ID previously determined to belong to the team manager by 𝔻 `GetMgrByRelations`.
`IsBest`	◈	Checks if the currently evaluated worker is the same as the one identified by the `GetWrkBestMetric`.
`NotSame`	◈	Determines if the input contains two manager candidates.
`WasTooLong`	◈	Keeps track of how many times a worker was in the manager position, and returns true if the worker is not supposed to become manager in the upcoming iteration.
`GetWrkBestMetric`	◈	Reads the value of the `'effort'` metric for each of the passed workers in `_ws` and updates the best worker.
`GetMgrByRelations`	◈	Invokes the read-only structural query ◈ `GET_MANAGER`.
`SetManager`	◈	Invokes the modifying structural query ◈ `SET_MANAGER`.
`GET_MANAGER`	◈	Contains a compiled non-modifying GrGen.NET graph query, here expressed in GrGen rule language. Matches and returns a node that other nodes point to via `ManagedBy` relations, but itself is not managed by another team member.
`SET_MANAGER`	◈	Contains a compiled modifying GrGen.NET graph query matching the old and the new manager, and re-chaining the `ManagedBy` relations to point to the new manager node.

are almost identical, differing only in the filter they use – with former using the 𝔻 `BestTeamWrk` and the latter the 𝔻 `SecondBestTeamWrk`. This means that once the Designer has modeled one of them, the other one can be created by copy-pasting and referencing a different filter. Similarly, if at a later time the underlying crowdsourcing platform decided to use a different ◈ to reward the best workers (e.g., to pay out money instead of rotating team managers) the Designer would only need to partially adapt the scheme by referencing a different ◈ from the ⚠'s `action_logic` fields. Such adaptations can also be performed by incentive operators with minimal understanding of the underlying code.

Filters like `GetTeam`, `GetBest` and `GetManager` perform very common incentive functionality. In practice, this means that such components could be readily available as library elements. Of course, if we need to use a company-specific flavor, we can easily replace the default one with a proprietary element. For example, a 𝔻 `GetManager` may be available with a default `auxiliary` field ◈ that looks for a manager in the team model by inspecting the node tags for a given manager tag. In that case, to adapt such a filter for our rotating presidency example the Designer would need to exchange the default, tag-based ◈ with a structural one, such as `GetMgrByRelations`.

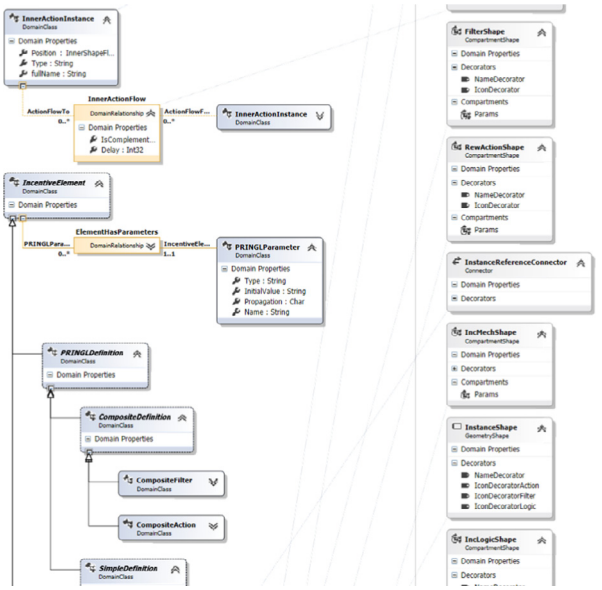


Fig. 17. Partial screenshot of the implemented PRINGL DSL metamodel.

6. Implementation

This section describes the prototype implementation of two entities: (a) Implementation of the PRINGL metamodel and the derived Microsoft Visual Studio PRINGL IDE; and (b) Implementation of Example 5 (Rotating Presidency) from Section 5.2.5 by using the tools from (a). The implementation of the rotating presidency example serves to evaluate: *(i)* Feasibility of implementation of *(a)*; and *(ii)* Fulfillment of the stated requirements from Section 3.4.

6.1. Metamodel implementation

Fig. 2 in Section 3 shows the overview of implemented components. PRINGL's language metamodel was implemented[8] in Microsoft's Modeling SDK for Visual Studio 2013 (MSDK) – Fig. 17. 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 (Fig. 18). 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#

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

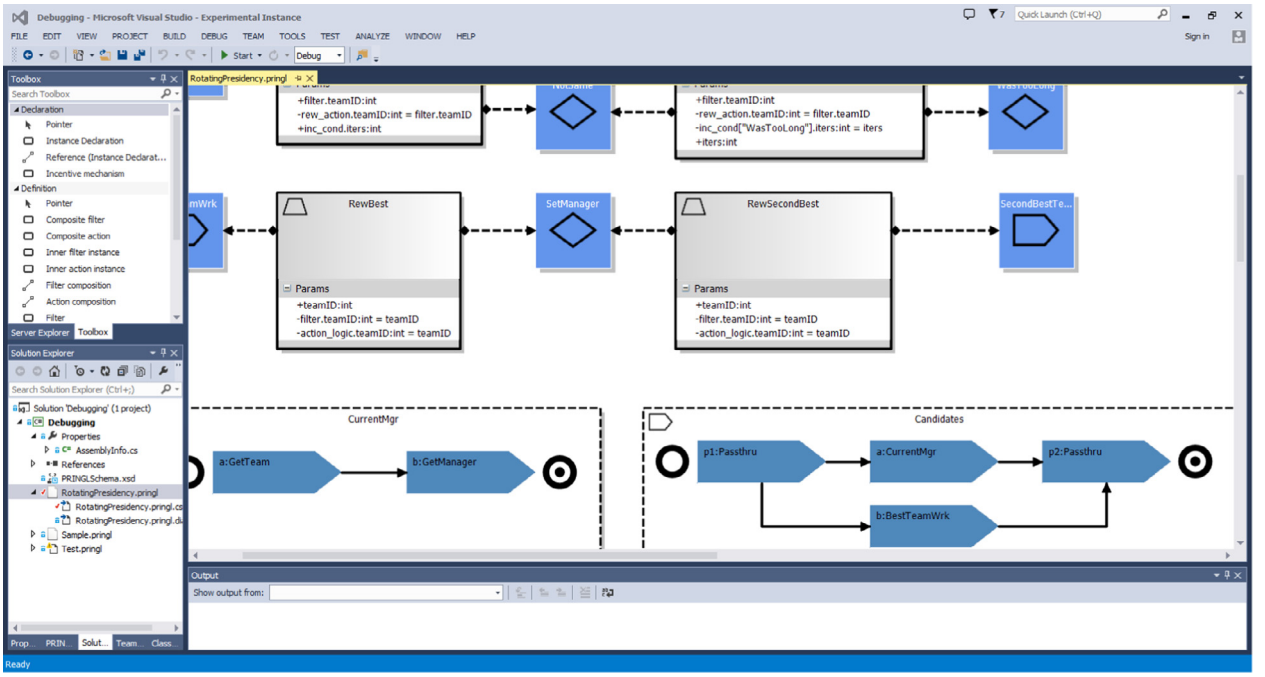


Fig. 18. Implementing the rotating presidency incentive scheme (Example 5) using generated PRINGL Visual Studio environment.

(.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).

6.2. Rotating presidency example implementation

Fig. 18 shows a screenshot of the implementation[8] of the rotating presidency example using the VS PRINGL IDE. The implemented incentive elements correspond to the individual element descriptions presented in Section 5.2.5 (Ex. 5). 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 a custom-made incentive management application using PRINC as the acting interlayer, as shown in Section 3.3.

The implemented example supports an arbitrary number and structure of `Workers` (represented as graph nodes) and their 'effort' metrics. Worker nodes are inter-connected with arbitrary-typed graph edges representing different relations. Our PRINGL-encoded incentive scheme will only consider the workers belonging to the team denoted by the `teamID` identifier, and only the managerial relations represented by `ManagedBy`-typed edges. Events notify PRINC when iterations end and 'effort' metrics change. The code generated from the implemented example monitors these events and executes the incentive mechanisms that make sure the best-performing worker is installed as the manager, but for not more than two consecutive iterations, subject to being replaced by the runner up in such a situation.

7. Related work

Previous research on incentives for socio-technical systems is dispersed and problem-specific, often spanning or originating from different areas, such as Management, Game Theory, Computer-Supported Collaborative Work, Human-Computer Interaction, Multiagent Systems. Due to this variety, the selection we present here is meant to give the reader an overview, rather than an in-depth coverage of the area. For further information, the reader is referred to [4,12,22,23].

The approaches in researching incentives in socio-technical systems can be roughly categorized in two groups. One group seeks to find optimal/appropriate incentives in formally defined environments through mathematical models and game-theoretical approaches [22–24]. The incentive is modeled as a monetary or quantifiable compensation to workers/agents to disclose their private information, and the proposed models are often simulated (e.g., [25]). Although successfully used in microeconomic models, these incentive models do not fully capture the diversity and unpredictability of human behavior that become accentuated in socio-technical systems [18]. In such cases the incentives predominantly help by identifying better workers, rather than increasing effort of the worse ones.

The other group examines the effects of existing or new incentives empirically, by running experiments or observing data from existing crowdsourcing platforms or social networks involving real human subjects. The number of topics here is more varied. In [26,27] the authors examine the effects of incentives by running experiments on existing crowdsourcing platforms and rewarding real human subjects with actual monetary rewards. In [28] the authors compare

the effects of lottery incentive and competitive rankings in a collaborative mapping environment. In [29] the authors analyze two commonly used approaches to detect cheating and properly validate crowdsourced tasks. In [6] the focus is on pricing policies that should elicit timely and correct answers from crowd workers. Paper [30] examines which psychological and monetary incentives are used to lure social network users to click on malicious links. In [31] the authors analyze how incentive schemes relying on peer voting can influence the decisions of workers from a crowdsourcing platform. The major limitation of this research approach [32] is that the findings are applicable only for a limited range of activities, considered as conventional crowdsourcing tasks, such as image tagging, multiple-choice question answering, text translation, or design contests. Furthermore, cultural background [33] can also skew the findings.

None of the two research approaches is suitable for evaluating PRINGL as we do not design nor evaluate particular incentive mechanisms. However, both approaches provide useful, generalizable findings that need to be taken into account when designing an incentive management system. For example, the finding that the transparency of actors and processes in a socio-technical system will likely improve the overall performance [34] for PRINGL translates to the requirement of portability and transparency of incentives. The findings of [35] indicate that for performing more intellectually challenging tasks smaller groups of expert workers may be more effective than web-scale crowd collectives. Again, this is in line with PRINGL's motivation of supporting novel socio-technical systems employing smaller teams of experts rather than large anonymous crowds only. Similarly, the aforementioned difference of effectiveness in different cultural backgrounds maps to the requirements of usability and expressivenes, to offer to incentive designers a tool for quick adaptations of general incentive mechanisms into the locally-effective versions. At the moment of writing, we are unaware of any similar languages or frameworks offering general incentive management functionality for socio-technical systems.

8. Conclusions and future work

In this paper we presented the programming model of PRINGL – a domain-specific language for programming incentives for socio-technical/crowdsourcing 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 [9].

Design of the language and its programming model was guided by the requirements obtained through an extensive survey of crowdsourcing techniques used in commercial environments. The model was evaluated qualitatively by modeling a suite of demonstrative examples selected to cover many realistic incentive categories. We implemented tools based on this model, supporting the creation of executable incentive schemes in PRINGL and evaluated them functionally on a realistic use case.

At this stage PRINGL is in an active state of development. Currently, PRINGL's default abstraction interlayer PRINC is undergoing a restructuring and integration with the human worker provisioning engine [10], worker orchestration components [36] and the virtualization and communication middleware SMARTCOM [17] in the context of the SmartSociety[9] socio-technical platform. The integration will allow building an end-to-end framework for applying PRINGL-modeled incentives on human crowd workers engaged in realistic execution scenarios. This is also a necessary precondition for running further quantitative evaluation of the usability of the language. Future work will see the integration of PRINGL's programming model into the general programming model of the SmartSociety platform.

Acknowledgments

This work is supported by the EU FP7 SmartSociety project under Grant no. 600854.

References

[1] O. Scekic, H.-L. Truong, S. Dustdar, Managing incentives in social computing systems with pringl, in: B. Benatallah, A. Bestavros, Y. Manolopoulos, A. Vakali, Y. Zhang (Eds.), Proceedings of the Web Information Systems Engineering (WISE'14), volume 8787 of LNCS, Springer, 2014, pp. 415–424.
[2] M. Hosseini, K. Phalp, J. Taylor, R. Ali, The four pillars of crowdsourcing: A reference model, in: Proceedings of the IEEE Internationl Conference on Research Challenges in Information Science (RCIS), 2014, pp. 1–12.
[3] A. Doan, R. Ramakrishnan, A.Y. Halevy, Crowdsourcing systems on the world-wide web, Commun. ACM 54 (4) (2011) 86–96.
[4] O. Tokarchuk, R. Cuel, M. Zamarian, Analyzing crowd labor and designing incentives for humans in the loop, IEEE Internet Comput. 16 (5) (2012) 45–51.
[5] S. Ahmad, A. Battle, Z. Malkani, S. Kamvar, The jabberwocky programming environment for structured social computing, in: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, in: UIST '11, ACM, 2011, pp. 53–64.
[6] D.W. Barowy, C. Curtsinger, E.D. Berger, A. McGregor, Automan: a platform for integrating human-based and digital computation, SIGPLAN Not. 47 (10) (2012) 639–654.
[7] P. Minder, A. Bernstein, Crowdlang: a programming language for the systematic exploration of human computation systems, in: K. Aberer, A. Flache, W. Jager, L. Liu, J. Tang, C. Guret (Eds.), Social Informatics, volume 7710 of LNCS, Springer, 2012, pp. 124–137.
[8] D. Miorandi, L. Maggi, Programming social collective intelligence, IEEE Technology and Society (forthcoming).
[9] A. Kittur, J.V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, J. Horton, The future of crowd work, in: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, CSCW '13, ACM, 2013, pp. 1301–1318.
[10] M.Z.C. Candra, H.-L. Truong, S. Dustdar, Provisioning quality-aware social compute units in the cloud, in: proceedings of the 11th International Conference on Service Oriented Computing (ICSOC 2013), Springer, Berlin, Germany, December 2-5, 2013.
[11] M. Riveni, H.-L. Truong, S. Dustdar, On the elasticity of social compute units, in: M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, J. Horkoff (Eds.), Advanced Information Systems Engineering, volume 8484 of LNCS, Springer, 2014, pp. 364–378.
[12] O. Scekic, H.-L. Truong, S. Dustdar, Incentives and rewarding in social computing, Commun ACM 56 (6) (2013) 72.
[13] P. Mohagheghi, Ø. Haugen, Evaluating domain-specific modelling solutions, in: J. Trujillo, G. Dobbie, H. Kangassalo, S. Hartmann, M. Kirchberg, M. Rossi, I. Reinhartz-Berger, E. Zimnyi, F. Frasincar (Eds.), Advances in Conceptual Modeling Applications and Challenges, volume 6413 of LNCS, Springer, 2010, pp. 212–221.

[9] http://www.smart-society-project.eu/

[14] A. Seffah, M. Donyaee, R.B. Kline, H.K. Padda, Usability measurement and metrics: a consolidated model, Softw. Qual. Control 14 (2) (2006) 159–178.
[15] O. Scekic, H.-L. Truong, S. Dustdar, Programming incentives in information systems, in: C. Salinesi, M.C. Norrie, Pastor (Eds.), Advanced Information Systems Engineering, volume 7908 of LNCS, Springer, 2013, pp. 688–703.
[16] O. Scekic, C. Dorn, S. Dustdar, Simulation-based modeling and evaluation of incentive schemes in crowdsourcing environments, in: R. Meersman, H. Panetto, T. Dillon, J. Eder, Z. Bellahsene, N. Ritter, P. De Leenheer, D. Dou (Eds.), On the Move to Meaningful Internet Systems: OTM 2013 Confs, volume 8185 of LNCS, Springer, 2013, pp. 167–184.
[17] P. Zeppezauer, O. Scekic, H.-L. Truong, S. Dustdar, Virtualizing communication for hybrid and diversity-aware collective adaptive systems, in: Proceedings of the 10th International Workshop on Engineering Service-Oriented Applications, WESOA'14, Springer, 2014. p. Forthcoming
[18] D.D. Fehrenbacher, Design of Incentive Systems, Contributions to Management Science, Springer, 2013.
[19] Smartsociety consortium, deliverable 5.3 - specification of advanced incentive design and decision-assisting algorithms for cas, http://www.smart-society-project.eu/publications/deliverables/2015.
[20] L. Baresi, R. Heckel, Tutorial introduction to graph transformation: a software engineering perspective, in: A. Corradini, H. Ehrig, H.-J. Kreowski, G. Rozenberg (Eds.), Graph Transformation, volume 2505 of LNCS, Springer, 2002, pp. 402–429.
[21] E. Jakumeit, S. Buchwald, M. Kroll, GrGen. NET, Int J. Softw. Tools Technol. Transf. 12 (3) (2010) 263–271.
[22] J.-J. Laffont, D. Martimort, The Theory of Incentives, Princeton University Press, New Jersey, 2002.
[23] J. Witkowski, Y. Bachrach, P. Key, D.C. Parkes, Dwelling on the negative: incentivizing effort in peer prediction, in: Proceedings of the First AAAI Conference on Human Computation and Crowdsourcing, AAAI, Palm Springs, CA, USA, 2013, pp. 190–197.
[24] M. Bloom, G. Milkovich, The relationship between risk, incentive pay, and organizational performance, Acad. Manag. J. 41 (3) (1998) 283–297.
[25] N. Peled, Y.K. Gal, S. Kraus, A study of computational and human strategies in revelation games, Auton. Agents Multi-Agent Syst. 29 (1) (2015) 73–97.
[26] G. Little, Exploring iterative and parallel human computation processes, in: Ext. Abstracts on Human Factors in Comp. Sys., CHI EA '10, ACM, 2010, pp. 4309–4314.
[27] W. Mason, D.J. Watts, Financial incentives and the "performance of crowds", in: Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP '09, ACM, 2009, pp. 77–85.
[28] S.D. Ramchurn, T.D. Huynh, M. Venanzi, B. Shi, Collabmap: crowdsourcing maps for emergency planning, in: Proceedings of the 5th ACM Web Science Conference, Paris, France, 2013, pp. 326–335.
[29] M. Hirth, T. Hoßfeld, P. Tran-Gia, Analyzing costs and accuracy of validation mechanisms for crowdsourcing platforms, Math. Comput. Model. 57 (1112) (2013) 2918–2932.
[30] T.-K. Huang, B. Ribeiro, H.V. Madhyastha, M. Faloutsos, The socio-monetary incentives of online social network malware campaigns, in: Proceedings of the ACM Conference on Online Social Networks, COSN '14, ACM, 2014, pp. 259–270.
[31] H. Rao, S. Huang, W. Fu, What will others choose? how a majority vote reward scheme can improve human computation in a spatial location identification task, in: B. Hartman, E. Horvitz (Eds.), Proceedings of the First AAAI Conference on Human Computation and Crowdsourcing, HCOMP, Palm Springs, CA, USA AAAI, 2013.
[32] E. Adar, Why i hate mechanical turk research (and workshops), in: Proceedings of the CHI'11 Workshop on Crowdsourcing and Human Computer, ACM, Vancouver, Canada, 2011.
[33] M. Gunkel, Country-Compatible Incentive Design, DUV, Wiesbaden, 2006.
[34] S.-W. Huang, W.-T. Fu, Don't hide in the crowd!: increasing social transparency between peer workers improves crowdsourcing outcomes, in: Proceedings of the SIGCHI Conference on Human Factors in Computer Systems, CHI '13, ACM, 2013, pp. 621–630.
[35] D.G. Goldstein, R.P. McAfee, S. Suri, The wisdom of smaller, smarter crowds, in: Proceedings of the ACM Conference on Economics and Computation, EC '14, ACM, 2014, pp. 471–488.
[36] Smartsociety consortium, deliverable 6.2 - static social orchestration: implementation and evaluation, http://www.smart-society-project.eu/publications/deliverables/, 2015.



Ognjen Scekic is a research assistant at the Distributed Systems Group, Institute of Information Systems, Vienna University of Technology, where he is working towards his PhD in the context of socio-technical systems, with particular focus on incentive management and programming models for human-based services. He received his M.Sc. degree in Computer Science from the School of Electrical Engineering, University of Belgrade, Serbia. His research interests include distributed and parallel computing and collective adaptive systems (CAS).



Hong-Linh Truong is an assistant professor for Service Engineering Analytics at the Distributed Systems Group, Institute of Information Systems, Vienna University of Technology. He received his Habilitation in Practical Computer Science, from Vienna University of Technology, and a PhD in Computer Science from the same university. His research interests include distributed and parallel systems with an applied, systems-oriented focus, from parallel and elastic cloud computing, to context-aware computing and socio-technical services.



Schahram Dustdar is a full professor of computer science and head of the Distributed Systems Group, Institute of Information Systems, at the Vienna University of Technology. Dustdar is an ACM Distinguished Scientist and IBM Faculty Award recipient. He received his Habilitation degree in Computer Science at Vienna University of Technology, and received his M.Sc. and PhD. degrees in Business Informatics from the University of Linz, Austria. His research interests include service-oriented architectures and computing, cloud and elastic computing, socio-technical and adaptive systems.