FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Automated Incentive Management for Social Computing

Foundations, Models, Tools and Algorithms

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Dipl.-Ing. Ognjen Šćekić
Registration Number 1028158

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Second advisor: Dr. Hong-Linh Truong

External reviewers:
Prof. Fausto Giunchiglia. University of Trento, Italy.
Prof. Stuart Anderson. University of Edinburgh, UK.

Vienna, 15th January, 2016

Ognjen Šćekić	Schahram Dustdar

Declaration of Authorship

Dipl.-Ing. Ognjen Šćekić
Gasgasse 2/7079, 1150 Wien, Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 15th January, 2016

Ognjen Šćekić

Acknowledgements

I would hereby like to thank all those people who helped and supported me throughout my doctoral studies and writing of this thesis. Firstly, my supervisors Prof. Dr. Schahram Dustdar and Dr. Hong-Linh Truong for their daily guidance and collaboration. Secondly, my friends and colleagues from the Distributed Systems Group, foremost Stefan Nastić, Sanjin Sehić, Christoph Mayr-Dorn, and Philipp Zeppezauer for inspired discussions and concrete suggestions. Thirdly, all the people from various organizations who helped me financially and organizationally: the Vienna PhD School of Informatics for supporting me financially and for providing an interesting and creative environment during the initial stages of the studies, as well as all the colleagues and the Secretariate of the School; all the staff of the Distributed Systems Group; the EU FP7 SmartSociety research project for the financial support in the second part of the studies and the fellow researchers from the project; the FoCAS EU Coordination Action for financing a number of research and collaboration visits. Finally, but most importantly, I would like to thank my family – mother Mladenka, father Milan, brother Igor, and my fiancée Katarina for the wholehearted encouragement and for always being there for me.

Kurzfassung

Die menschliche Teilnahme in sozio-technischen Systemen überfordert herkömmliche Crowdsourcing Mechanismen, in denen Menschen üblicherweise einfache und unabhängige Tasks lösen. Neuartige Systeme hingegen, versuchen Menschen für intellektuell herausfordernde Aufgaben zu nutzen. Diese Aufgaben beinhalten länger anhaltende Beschäftigung und komplexe Muster zur Zusammenarbeit. Kontrollierbarkeit solcher Systeme erfordert verschiedene direkte und indirekte Methoden zur Beeinflussung der beteiligten Menschen. Konventionelle Organisationen, wie Unternehmen oder Institutionen, benutzen seit Jahrzehnten Anreize, um die Interessen der Arbeitnehmer und der Unternehmen aneinander anzupassen. Da Kooperationen, die mit sozio-technischen Plattformen verwaltet werden, immer komplexer werden und damit herkömmliche Ansätze in ihrer Komplexität übersteigen, gibt es einen Bedarf fortgeschrittene Anreiz-Techniken in virtuellen Umgebungen anzuwenden.

Allerdings sind bestehende Anreiz-Management-Techniken, die bereits in Crowdsourcing bzw. sozio-technischen Plattformen angewendet werden, nicht für die oben beschriebenen (komplexen oder intellektuell herausfordernden) Aufgaben geeignet. Zusätzlich nutzen bestehende Plattformen individuell entwickelte Lösungen. Dieser Ansatz ist jedoch nicht übertragbar und verhindert die Wiederverwendung von gemeinsamer Anreizlogik und Reputationsübertragung. Folglich wird dadurch verhindert, dass Arbeitnehmer verschiedene Plattformen vergleichen können, wodurch die Wettbewerbsfähigkeit des virtuellen Arbeitsmarktes behindert wird und dieser damit weniger attraktiv für qualifizierte Arbeitskräfte ist.

Diese Arbeit präsentiert eine Reihe von Modellen und Werkzeugen für programmierbares Anreiz-Management in sozialen Computing-Plattformen. Insbesondere werden die folgenden Punkte vorgestellt:

(i) Eine umfassende, multidisziplinären Analyse der bestehenden Literatur über Anreize, sowie eine umfangreiche Studie von realen Anreiz-Praktiken im sozialen Computing-Umfeld,

(ii) Ein Basismodell für Anreize, das in sozio-technischen Systemen angewendet werden kann,

(iii) PRINC - ein Modell und Framework zur Ausführung programmierbarer Anreizmechanismen, um Anreize mittels einem Service-Modell anzubieten.

(iv) PRINGL - eine domänen-spezifische Sprache zum Kodieren komplexer Anreizstrategien für sozio-technische Systeme. Die Sprache fördert einen modularen Ansatz beim

Aufbau von Anreizstrategien, reduziert Entwicklungs- und Anpassungszeit, und schafft eine Grundlage für die Entwicklung von standardisierten, aber anpassbaren Anreizen.

Die vorgestellten Werkzeuge sollen System- und Anreizentwicklern eine komplette Umgebung zur Modellierung, Verwaltung, Ausführung und Anpassung von verschiedenen realistischen Anreizmechanismen, in einer Privatsphäre erhaltenden Art und Weise, ermöglichen. Keine vergleichbaren Systeme waren zum Zeitpunkt des Schreibens dieser Arbeit bekannt.

Abstract

Human participation in socio-technical systems is overgrowing conventional crowdsourcing where humans solve simple, independent tasks. Novel systems are attempting to leverage humans for more intellectually challenging tasks, involving longer lasting worker engagement and complex collaboration patterns. Controllability of such systems requires different direct and indirect methods of influencing the participating humans. Conventional human organizations, such as companies or institutions, have been using incentives for decades to align the interests of workers and organizations. With the collaborations managed by the socio-technical platforms growing ever more complex and resembling, or even surpassing in complexity, the conventional ones, there is a need to apply advanced incentivizing techniques in the virtual environment as well.

However, existing incentive management techniques in use in crowdsourcing/socio-technical platforms are not suitable for the described (complex or intellectually-challenging) tasks. In addition, existing platforms currently use custom-developed solutions. This approach is not portable, and effectively prevents reuse of common incentive logic and reputation transfer. Consequently, this prevents workers from comparing different platforms, hindering the competitiveness of the virtual labor market and making it less attractive to skilled workers.

This research presents a complete set of models and tools for programmable incentive management for social computing platforms. In particular, it introduces:

(i) A comprehensive, multidisciplinary review of existing literature on incentives as well as an extensive survey of real-world incentive practices in social computing milieu,

(ii) A low-level model of incentives suitable for use in socio-technical systems

(iii) PRINC – a model and framework for execution of programmable incentive mechanisms, allowing the offering of incentives through a service model.

(iv) PRINGL – a high-level domain-specific language for encoding complex incentive strategies for socio-technical systems, encouraging a modular approach in building incentive strategies, cutting down development and adjustment time and creating a basis for development of standardized but tweakable incentives.

The tools are meant to allow system and incentive designers a complete environment for modeling, administering/executing and adjusting a whole spectrum of realistic incentive mechanisms in a privacy-preserving manner. No known comparable systems were known to exist at the time of writing of the thesis.

Contents

List of Figures

List of Tables

Publications

Parts of the work presented in this dissertation were published in the following publications:

1. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Incentives and rewarding in social computing. *Comm. of the ACM*, 56(6):72, 6 2013

2. Ognjen Scekic, Mirela Riveni, Hong-Linh Truong, and Schahram Dustdar. Social interaction analysis for team collaboration. In Reda Alhajj and Jon Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*. SpringerScience+Business Media, NewYork, 2014

3. Ognjen Scekic, Christoph Dorn, and Schahram Dustdar. Simulation-based modeling and evaluation of incentive schemes in crowdsourcing environments. In Robert Meersman, Hervé Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Deijing Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conf.s*, volume 8185 of *LNCS*, pages 167–184. Springer, 2013

4. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Modeling rewards and incentive mechanisms for social bpm. In Alistair Barros, Avigdor Gal, and Ekkart Kindler, editors, *Business Process Management*, volume 7481 of *LNCS*, pages 150–155. Springer Berlin Heidelberg, 2012

5. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Programming incentives in information systems. In Camille Salinesi, Moira C. Norrie, and Óscar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908 of *LNCS*, pages 688–703. Springer, 2013

6. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Managing incentives in social computing systems with pringl. In Boualem Benatallah, Azer Bestavros, Yannis Manolopoulos, Athena Vakali, and Yanchun Zhang, editors, *Web Inf. Systems Engineering (WISE'14)*, volume 8787 of *LNCS*, pages 415–424. Springer, 2014

7. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Pringl – a domain-specific language for incentive management in crowdsourcing. *Computer Networks*, 9 July 2015

8. Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Virtualizing communication for hybrid and diversity-aware collective adaptive systems. In *Proc. of 10th Intl. Workshop on Engineering Service-Oriented Applications*, WESOA'14, pages 56–67. Springer, 11 2014

9. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Supporting multilevel incentive mechanisms in crowdsourcing systems: an artifact-centric view. In *Cloud-based Software Crowdsourcing*, pages 95–114. Springer, 2015

10. Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. A collaboration model for community-based software development with social machines. In *Proc. of the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Miami, FL, USA, October 2014

11. O. Scekic, T. Schiavinotto, D.I. Diochnos, M. Rovatsos, H.-L. Truong, I. Carreras, and S. Dustdar. Programming model elements for hybrid collaborative adaptive systems. In *Proc. of the 1st IEEE International Conference on Collaboration and Internet Computing (CIC'15)*, Hangzhou, China, October 2015

CHAPTER 1

Introduction

1.1 Motivation

Incentives help align interests of employees and organizations. They first appeared with the division of labor and have since followed the growth in complexity of human labor and organization. Modern working environments are especially attractive to unfair gainful activities due to existing diversity of working roles, scale of workforce and complexity of tasks performed collectively. In such environments, incentives are being increasingly used to prevent the various types of occurring dysfunctional behavior. This is evidenced by the fact that most big or medium-sized companies employ some incentive measures; e.g., over 80% in the US [Feh13, Ch. 1]. Furthermore, numerous studies have shown effectiveness [Pre99] of different incentive mechanisms and their selective and motivating effects [Laz07].

We have recently witnessed the evolution of conventional *crowdsourcing systems* [HPTA14, MD15] and appearance of novel types of social computing systems, attempting to leverage human experts for more intellectually challenging tasks [ABMK11, BCBM12, MB12, MM, BBC+14, TDKC15], often 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 compared to conventional crowdsourcing, some authors started naming these systems *socio-technical*. 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. In this thesis the terms 'social computing' and 'socio-technical' are used interchangeably to describe the whole class of similar systems involving groups of humans providing effort in work processes managed by a software platform.

The novel social computing systems resemble in complexity the conventional (non-virtual) working environments (Figure 1.1), but still employ a limited number of simple incentive mechanisms, suitable for crowdsourcing platforms employing anonymous crowds

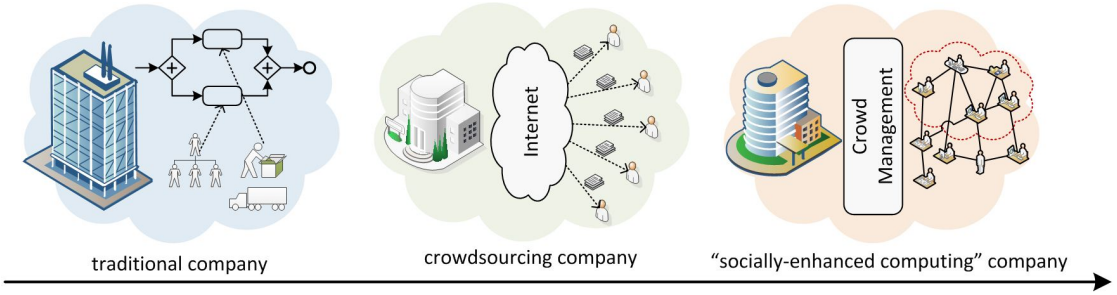

Figure 1.1: Evolution of Social Computing: As working patterns become more complex and reminiscent of traditional companies, social computing platforms need advanced organizational structure and ‘crowd’ management capabilities, including automated incentive management.

for one-off task executions. This can be perceived as a big opportunity to offer novel ways of handling incentives and rewarding. While incentives were already identified as one of the fundamental characteristics of conventional crowdsourcing systems [HPTA14], 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 [KNB+13].

In [KNB+13] 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* (cf. [MW09, Lit10]) was identified as one of them. However, contemporary approaches to incentive management usually imply hard-coded, system-specific solutions (see Chapter 2). Such approaches are not portable, and prevent reuse of common incentive logic. That hinders cross-platform application of incentives and reputation transfer [KNB+13]. Additionally, in social computing environments there is a need to combine, personalize and frequently adapt incentive mechanisms [Vas12], which is not straightforward in current approaches. Furthermore, the modeling of incentives is performed by multidisciplinary domain experts (*incentive designers*), often lacking the knowledge of the technical internals of the social computing platform. On the other hand, the platforms developers lack the domain knowledge necessary to understand the provided incentives, leading to a discrepancy between modeling and implementation processes.

By designing a comprehensive incentive model and developing an execution framework that can be coupled with different socio-technical systems it would be possible to perform advanced incentivizing measures and indirect team adaptations. The design of the incentives could be performed by the domain experts, while the execution would be handled by the framework, allowing a decoupling of the incentivizing functionality from the platform, and offering it as a service.

1.2 Research Problem

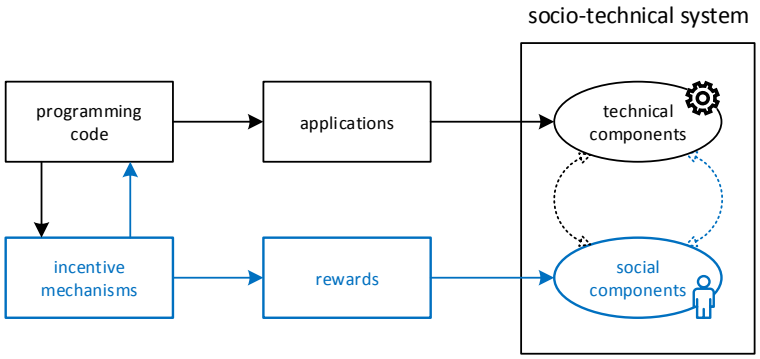

Figure 1.2: Controlling socio-technical systems requires influencing both its technical and social components.

Figure 1.2 shows the envisioned control loop for socio-technical systems, aiming to close the existing gap where socio-technical systems are controlled predominantly by programming elements suitable for software services and machine control units only (e.g., activities, constraints, APIs). Instead, we propose targeting the social component (human participants) with mechanisms tailored to act on psychological level. In turn, the social component is motivated to better use and control more effectively the technical component from within the system (e.g., motivating human participants to provide votes on other participants via the system and confide in system's reputation calculations). Furthermore, once the incentive controllability components are in place, they can be coupled in a feedback loop with the existing software software control components, exchanging state updates and suggested rewarding actions. We explain this in more detail in subsequent sections.

1.2.1 Application Context

Figure 1.3 visualizes the application context of an *incentive management framework*: 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 platform (e.g., Social Compute Unit (SCU) [CTD13, RTD14], SmartSociety [SMS+15]) that assembles teams of crowd workers based on required functionality, collaboration patterns and elasticity parameters, such as: price, speed or reputation. However, choosing appropriate workers alone does not guarantee the quality of subsequent team's performance. In order to monitor and influence the behavior of workers during and across activity executions an incentive scheme needs to be enacted.

This is the task of the envisioned incentive management framework. It enacts the incentive scheme by applying rewards or penalties in a timely manner to induce a

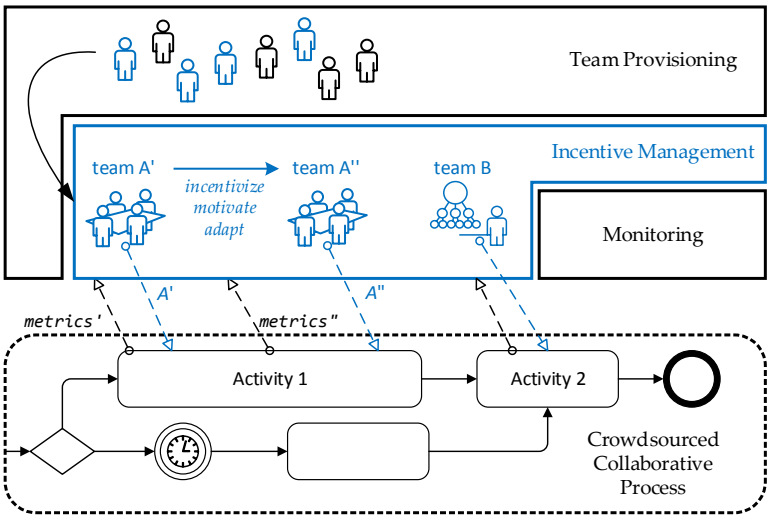

Figure 1.3: Application context of incentive management systems.

wanted worker behavior, thus effectively performing runtime team adaptations (e.g., Fig. 1.3: $A' \rightarrow A''$). Most real-world incentive strategies can be composed of modelable and reusable bits of incentive logic ([STD13a, TCZ12]). However, the efficacy of incentives can depend on multiple other factors, such as team size, cultural background, or knowledge of other participants ([Feh13]). The scheme is usually a result of a prior assessment or case study of the particular application scenario, but needs subsequent adaptations and adjustments [Vas12]. Therefore, the challenge is to design an incentive management framework capable of combining and reusing existing and proven incentive mechanisms, but also allowing for easy tweaking to particular application contexts. We make a strong case for this in Section 3.4.3

Prior to enactment, an incentive scheme must be modeled and encoded by an incentive designer. The *incentive model* used in the process needs to be based on widely-adopted incentive practices in both traditional companies as well as in contemporary social computing environments to allow for expressing of realistic incentives covering a wide array of incentivizing use-cases.

As the incentive designer cannot be assumed to be a software developer with knowledge of particular social computing platform APIs, a Domain-Specific Language (DSL) is to be provided for encoding the incentives. The DSL allows the designer to provide reusable domain-specific expertise in a portable, platform-agnostic fashion.

1.2.2 Research Questions

Based on the described application context, we can break down the necessary steps in designing the envisioned incentive management framework into the following research questions:

1. Identify fundamental incentive elements used in conventional companies/organizations and in contemporary social computing environments.
2. Design an incentive model comprising the incentive elements and constructs for their composition.
3. Design a programming model for application of incentive mechanisms based on the incentive model.
4. Design a supporting framework for coupling with social computing platforms and executing/applying the incentives.
5. Design a DSL for facilitating the encoding and description of incentive schemes suitable for social computing platforms.

1.3 Scientific Contributions

The goal of this thesis is to respond to the previously formulated research questions by presenting the results of the research leading to development of models, techniques and components of a general framework for automated incentive management for the emerging social computing systems. Concretely, the following contributions are presented:

- A comprehensive, multidisciplinary review of existing literature on incentives as well as an extensive survey of real-world incentive practices in social computing milieu. Related publications: [STD13a]
- A low-level model of incentives suitable for use in socio-technical systems. Related publications: [SDD13]
- An execution model, set of primitives and data structures for scheduling and execution of incentive mechanisms, allowing the offering of incentives through a service model; integrated into PRINC – a framework for incentive management for social computing platforms. Related publications: [STD13b]
- PRINGL – a high-level domain-specific language for encoding complex incentive strategies for socio-technical systems, encouraging a modular approach in building incentive strategies, cutting down development and adjustment time and creating a basis for development of standardized but tweakable incentives. Related publications: [STD14b, STD15b]

Overall, the goal of the thesis is to present a set of software tools for incentive designers and operators that will allow them to capture the range and diversity of incentive mechanisms currently in use in complex work settings. The tools are presented as a unified platform for incentive management in Part II of the thesis. The platform is intended to couple with different socio-technical systems. The platform's components are based on the listed contributions. An additional component – SmartCOM (Chapter 6)

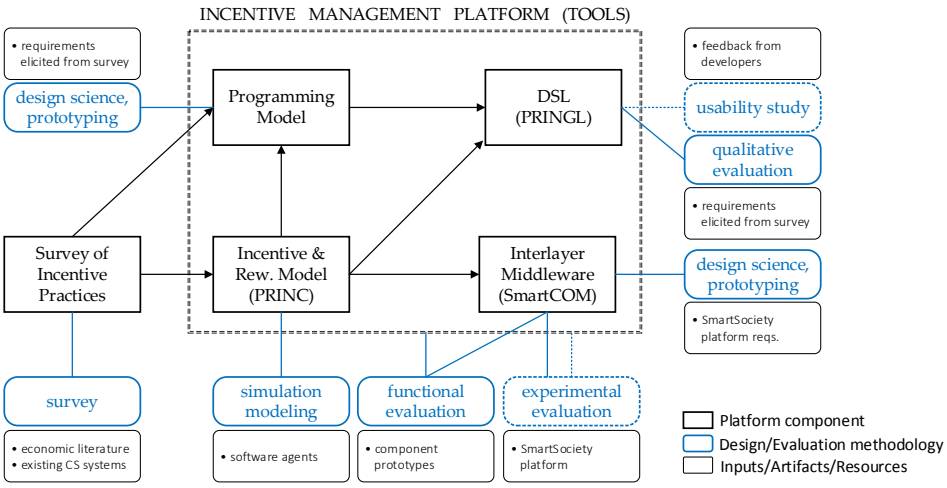

Figure 1.4: Overview of the components, contributions and design/evaluation methodologies used. Dashed lines represent planned/future activities.

is technically not considered a contribution of this thesis. However, being a fundamental part of the overall platform, for completeness purposes it was necessary to include the details relevant to its design and functioning in the incentive management context.

It is important to emphasize that the goal of the thesis is not to design new incentive mechanisms nor evaluate the effectiveness of existing ones.

1.4 Research Methodology

The work presented in this dissertation is motivated by the lack of general and configurable incentive management solutions for (novel) types of social computing systems [KNB+13]. As shown in Chapter 2, previous research on incentives in social computing 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 socio-technical systems.

In order to determine the appropriate incentive model granularity and expressiveness grounded in existing incentivizing practices, an extensive survey of existing literature and incentive schemes used in crowdsourcing companies was performed (Chapter 3). The resulting incentive model and its elements were validated through simulation modeling (Chapter 4). Simulation modeling is used in the computational social sciences to explore theoretical ideas in the context of synthetic populations, particularly where real studies would be impractical [GT05b, MN10]. Recently, this has been applied to crowdsourcing, in order to generalize results which otherwise would be tied to a particular situation [BFGK13]. Advantages and limitations of this approach are discussed in Section 4.3.

A programming model and the supporting software framework were then designed around the incentive model to support its operation with arbitrary socio-technical

platforms. The effort followed the general guidelines of the "design science" research methodology [HMPR04]. This included the design of specific artifacts, including supporting data structures, algorithms and mapping models for interaction with the underlying socio-technical platforms (Chapter 5), which were developed following a prototyping approach. Note that this is a widely accepted methodology in computer science when new designs and models are proposed, with experimental evaluation often lacking [TLPH95]. The design requirements were formulated to target the identified shortcomings of existing incentivizing approaches in socio-technical platforms. The prototypes were evaluated functionally, as comparative evaluation was not possible due to non-existence of similar systems.

In order to allow the use of the framework to domain experts, a DSL was developed. 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 challenge was to allow quick and uniform/portable modeling of commonly used incentive patterns identified in Chapter 3. In order to design a useful DSL, we followed the general guidelines described in [MH10] to formulate design requirements, based on which we implemented and evaluated the programming model. As is common practice during the prototyping phase of DSL development, we evaluated the programming model qualitatively [SDKP06], with particular focus on groundedness, expressiveness and reusability.

An interlayer middleware SmartCOM, originally planned for the purposes of communication and virtualization in the SmartSociety platform, was developed from the very beginning also to be able to serve to the incentive management platform, providing the functionalities of direct incentive messaging, coupling with arbitrary platforms, and managing sensitive worker data (Chapter 6). The middleware was evaluated both functionally (through unit and integration testing) and experimentally (through experiments to test the scalability as well as in-field runs integrated with the SmartSociety platform).

Figure 1.4 shows an overview of the described methodological approach.

1.5 Organization of the Thesis

This dissertation includes the contributions from original research papers that were published during the author's doctoral studies. The individual contributions have been re-worked, extended and presented in a unified context. Corresponding papers are referenced throughout the thesis, when relevant research components are presented. A comprehensive list of relevant author's publications is presented at the beginning of the document.

The thesis is organized as follows:
Chapter 2 introduces the key theoretical concepts related to incentives, as well as an overview or related work. Chapter 3 presents a survey of existing, real-world practices in conventional companies and crowdsourcing platforms, allowing us to identify particular incentive elements (subcomponents) and model them. The modeling process is described in Chapter 4. The remaining chapters present the design of key components designed to be

interoperable and jointly provide a usable platform for automated incentive management. In Chapter 5 the theoretical model introduced in the previous chapter is embodied into an executive framework that can be coupled with external socio-technical, workforce management platforms. The process of coupling, and the middleware that was designed for this purpose is presented in Chapter 6. A programming model and a domain-specific language capable of encoding incentive mechanisms executable on the incentive framework are presented in Chapter 7. Finally, Chapter 8 recapitulates the initial research challenges, describes the proposed solutions and discusses their advantages and limitations, as well as the future research outlook.

CHAPTER 2

Theoretical Background & Related Work

2.1 Theories of Motivation and Incentives

2.1.1 Intrinsic and Extrinsic Motivation

The fundamental concept related to incentives and rewards is the concept of *motivation*. Motivation has been the topic of interest of psychology for decades, with different theories emerging in different epochs. The resulting corpus of research lead to the commonly accepted view of today [RD00], where motivation is classified into two major types – *intrinsic* and *extrinsic*.

Intrinsic motivation is described as the driving force attracting individuals to perform an activity for the inherent satisfaction associated purely with the act of performing that activity. Extrinsic motivation, on the other hand, is the driving force pushing individuals to perform an activity in which they find no inherent interest or satisfaction, but which is associated with external rewards substituting the missing inherent satisfaction.

Intrinsic motivation is powerful and stable, often associated with curiosity, creativity, competitiveness, playfulness and volunteering. However, it can also be highly dependent on different social and environmental factors. On the other hand, extrinsic motivation is exerted directly by an intervention strategy (e.g., incentive), making it more controllable, but also more volatile, as its effects are gone when the intervention is absent. Extrinsic motivation is important for performing activities which individuals consider important and necessary but not inherently enjoyable (e.g., learning, working). The different motivation types, and how they influence human behavior, are introduced and researched as part of the *Self-Determination Theory (SDT)* [DR85].

From the operational aspect (which is of particular interest to us), the two motivation types can be treated through the notion of *reward*. While the concept of reward is inherent to the definition of extrinsic motivation, in case of intrinsic motivation the reward can be

considered as providing the opportunity to perform an activity or get better at it. Same rationale is also adopted by the *Operant Theory* [Ski54]. This operationalization allows us further to define *incentivization* as the process through which motivation is fostered by application or provisioning of rewards.

As a practical example, this means that by incentivization we consider both a promise of payment of a monetary reward (extrinsic motivation); as well as providing an opportunity for an amateur astronomer to voluntarily participate in a citizen-science galaxy identification program (intrinsic motivation). In both cases incentivization is influencing individual's *locus of control* – in former case the locus is external; in the latter the locus is internal.

2.1.2 Principal-Agent Theory

Historically, there has been much more commercial interest in investigation of extrinsic intervention strategies. The practical concept of 'incentives' appeared together with the division of labor. Delegating productive tasks to others (workers) meant it was necessary to make sure that workers, pursuing own interests, did not work against owner's interests. Incentives served primarily this purpose – to align interests of the owner and the workers. This meant that the most important extrinsic operational models and theories were originally developed by economists. An overview of historical development of the notion of incentives and rewarding in economic thought can be found in [LM02].

The predominant model of incentives used in economy was set out in the *Principal-agent Theory* (also known as *Agency Theory*) [BM98, LM02]. The theory introduces the role of *principal*, corresponding to a manager in a traditional firm, who delegates tasks to a number of *agents*, corresponding to employees under his supervision. It is assumed that the agents seek to minimize their effort and risks while maximizing their compensation. The principal wants to minimize the costs of agents' labor and maximize profits. Therefore, their interests diverge.

Every agent is unique, possessing unique qualities and properties. For the same task different agents will put in different levels of effort, and will value that effort at a different cost[1]. Additionally, every agent can, unobserved by the principal, perform certain actions that go against principal's interests. The theory implies that the agents know their personal values for these properties (*adverse selection*) and know their intentions on committing hidden actions (*moral hazard*). The fact that they remain secret to the principal is called *information asymmetry* or *information gap*.

We assume that the principal knows just the statistical distribution of these values. If the principal knew entirely the private information (*signals*) for every agent, then each agent could be offered the ideal *contract* from principal's point of view, i.e., paying him just as little as it takes for him to perform the work with the given effort. However, this not being the case, some agents get overpaid while others get underpaid, effectively

[1] All the costs and prices, as well as amounts of incentives are expressed as numerical quantities. It is assumed that these numerical values include and represent also any other properties that the agents and principal value, like risk, free time, gratification due to pleasant working environment, personal satisfaction, prospect of promotion, fear of dismissal, etc.

inducing them not to accept working on the task. The principal would want to know and measure as many agent signals as possible, because the more insight he gets into agent's capabilities and behavior, the more chances he has of setting up a better contract and maximizing the profit (*Informativeness Principle*). So, the principal offers the agent an incentive to disclose part of this information in order to compile a better-suited contract.

Incentives are an additional expense for the principal. However, if, based on the new information, the principal can offer better contracts and consequently make more profit by filtering out and motivating quality agents, then the investment in the incentives will pay off. Therefore, the incentive designer is faced with an optimization problem that involves human agents, whose individual behaviors cannot be foreseen. A way to solve this problem lies in assembling an incentive strategy comprised of a number of simple incentives whose effects on the majority of agents can be predicted closely enough, and then adapt the strategy based on the concrete, context-specific feedback. In a traditional company, that would mean that a manager would offer a combination of wage increases, free days, promotions, bonuses and other benefits to the workers that achieve higher levels of some wanted property (e.g., productivity, quality, knowledge, leadership). Increased expenses for the principal are compensated not only by increased productivity, but in fact much more by the selective effects of the incentives [Laz07]; by investing into incentives the management gains the knowledge on which workers can produce more value to the company and therefore should be stimulated to stay in the company.

The fundamental difficulty when applying the theory in practice lies in precisely defining and subsequently measuring the different qualities of agents and their performance (signals). As we previously mentioned, the Informativeness Principle states that each contract should be designed with as many signal measurements taken into consideration as possible. As it is in the interest of the agents to keep some signals private to them, measuring those signals becomes the major obstacle. In practice, working assumes performing a lot of complex and interrelated tasks, and often collaborating and depending upon different people, so the principal problem translates to the inability of effectively assessing the quality of a particular worker's performance in a dynamic and complex environment due to impossibility of quantifying and measuring all of the signals. This is even more accentuated in social computing environments, where contracts (in the sense of Agency Theory) are more persistent than the signals sets that need to be considered, calling for frequent contract and incentive adjustments. Additionally, it also causes a number of behavioral responses (*dysfunctional behavior*) in agents meant exclusively to increase rewards while damaging the overall performance levels.

The agency theory implies a fully rational, self-interested agent, who always acts in his/her best interest (so called *homo oeconomicus*). The incentive is always monetary and acting on extrinsic motivation. In practice, this is not always a sufficiently accurate model. For this reason, additional *decision-making theories* and multidisciplinary frameworks were developed, taking into consideration various determinants of behavior, including additionally factors of intrinsic motivation, environmental and social factors, and assuming agents with bounded rationality. In [Feh13, Vas12] a comprehensive overview of different incentive theories and decision-making frameworks is presented. However, while providing

more realistic and nuanced behavioral models, these frameworks are less suited for technical abstraction and exploitation. The agency theory remains, therefore an important theory that is practically applicable in appropriate working environments.

Both the agency theory, as well as the more complex decision-making frameworks state that the effect of incentives on agent's behavior is exhibited dually, through *selective (sorting) effects* and *performance effects*. Selective effects are defined as the act of revealing more precise details of an agent's qualities, shortcomings and performance parameters through monitoring the application of incentives or through agent self-selection. Performance effects are changes in agent's performance caused by behavior modified through application of incentives.

Depending on the incentive and the application context one effect type may be more expressed than the other. Also, one type may be valued more than the other by the principal. For example, in piecework productive activities performance effects are usually valued more, while in engineering disciplines discovery of creative workers may be in a long run more profitable than the rewarding of the currently more productive ones.

2.1.3 Efficacy of Incentives

Although it sounds a commonplace that offering monetary rewards to someone should gratify that person and induce him/her to perform better, different research efforts demonstrated through empirical studies that it may not always be the case. For example, in [FJ01] the authors empirically conclude that in some cases the monetary rewards actually decrease intrinsic motivation. On the other hand, in [Laz00] for example, we encounter strong empirical evidence that in specific cases monetary rewards do significantly increase performance.

However, all the studies conclude that, depending on the environment, there always exist types of incentives that can provide the necessary motivation. With some simple, repetitive tasks, paying for performance increases productivity [Laz00, MW09, MKC+13]. Professionals that value a humanistic impact of their work (volunteers, community workers, firemen, doctors, scientists, etc.) may be intrinsically motivated by having their positive contribution of their work to the society shown [HS96, HA07]. In companies with lengthy and complex tasks promotions and/or team-based rewards are effective [VCV06, KO02]. Finally, as mentioned before, sometimes the sorting effects of incentives are much more useful to the principal than possibly moderate performance effects.

The expertise on the expected effects of particular incentives is based on empirical data usually formulated as very general claims about behavior of agents under certain conditions and then proven by different empirical methods (see [Feh13, Ch. 6] for an overview). However, as in many areas dealing with human behavior, absolute and quantifiable incentives cannot be given in advance, but rather must be adapted for a given collaborative environment after a careful study of the context and the habitual/cultural background of the participants.

While choosing appropriate incentive strategies and adapting them to fit to specific types of labor may prove a challenging task, it is a conclusive fact that incentives can

exhibit considerable selective or performance effects on workers, corroborated in practice by the fact that most traditional businesses employ incentive schemes [Feh13, Ch. 1].

2.2 Related Work

Most related work in the area of rewarding and incentives originates from the economics, organizational science, psychology, and applied research, mostly for military purposes. It can be used to classify and substantiate the basic rewarding approaches, and expected outcomes, and to simulate the responses to our incentive strategies. There is only a small number of computer science papers that threat the topics of incentives and rewarding, usually within particular application contexts (e.g. peer-to-peer networks, agent-based systems). However, to the best of our knowledge, no other computer science paper threats the topic in a comprehensive manner. In fact, most papers completely disregard the existing theoretical foundations of incentives, and are concerned with solving only the particular problem, as we will show in the rest of this section. The work [Vas12] is a notable exception, discussing incentives designed to motivate participation in different social computing platforms and relating them to the leading behavioral theories, and presenting a vision for the future developments in this research area.

In [SHY+08a] the aim is to maximize p2p content sharing. Therefore, they define roles of (content) forwarders and receivers. Forwarder gets a reward when the receiver reacts in some way to the content being forwarded. They then define the prices of forwarding, and receiving actions and assign the incentives based on that. Many other papers similarly identify certain behavioral patterns and develop particular solutions to prevent unwanted behavior or enhance existing algorithms to optimize certain metrics ([Kau11]).

The paper [FGP+09] discusses ways of modeling and implementing adaptable agent-based systems. Each agent can be modified by adding or removing "modules" that the agent consists of. Cause for agent adaptability is usually a role change within an organization.

In [Lit10] the authors are trying to determine quality of work achieved when a task is done iteratively compared to when it is done in parallel. The difference is that in iterative processes (when applicable), workers are shown previous attempts by other workers, which can influence their work in positive or negative way. They conduct experiments with real workers on Amazon Mechanical Turk, and prove statistically that, when applicable, the iterative approach yields better (more accurate) results. The quality of the work in their experiments is quantifiable, or voted by the crowd. This is an important finding, since it justifies the choice of iterative execution model that we adopt.

In [MW09] two basic findings seem robust, and can be used as general conclusions when modeling behavior of contributors: "First, that paying subjects elicited higher output than not paying them (where increasing their pay rate also yielded higher output); and second, that in contrast to the quantity of work done, paying subjects did not affect their accuracy. Although surprising, this latter result may be related to an "anchoring effect" in that subjects' perception of the value of their work was strongly correlated with their actual pay rate."

In [MKC+13] the authors compare the performance between paid and volunteering workers by running experiments on well-known commercial platforms (Amazon Mechanical Turk, Zooniverse and Planet Hunters) and analyze different reasons for improved or worsened performance. Interestingly, although the experiments bear much resemblance to psychological experiments intended to measure intrinsic motivation in the context of SDT, no reference to those experiments is made.

In [WD99] the authors investigate whether self-governing and self-coordinated human teams (without a centralized authority) can be stable if individual members of such teams follow appropriate rules. In [YST+10] the authors seek to maximize the extension of social network by motivating people to invite others to visit more content (i.e. give contribution measured in number of pages), and evaluate a number of concrete rewarding schemes (e.g. Dynamic Differentiated Rewarding Scheme). In [HHTG13] the authors analyze two commonly used approaches to detect cheating and properly validate crowdsourced tasks. In [BCBM12] the focus is on pricing policies that should elicit timely and correct answers from crowd workers. Paper [HRMF14] examines which psychological and monetary incentives are used to lure social network users to click on malicious links. In [RHF13] the authors analyze how incentive schemes relying on peer voting can influence the decisions of workers from a crowdsourcing platform.

The major limitation of these research approaches (see [Ada11]) 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, difference in cultural background [Gun06] can also skew the findings. However, the results of the listed papers, taken together, can provide some 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 [HF13] for us translates to the requirement of portability and transparency of incentives. The findings of [GMS14] 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 our 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 expressiveness, to offer to incentive designers a tool for quick adaptations of general incentive mechanisms into the locally-effective versions.

Overall, in this thesis we take a different approach. Instead of investigating or designing concrete incentives we design a general model and framework for defining, adapting and monitoring incentive mechanisms, where supported mechanisms are elicited from a broad survey of literature and practice in social computing companies and volunteering organizations.

Part I

Modeling Incentives

CHAPTER 3

Existing Incentive and Rewarding Practices

3.1 Classification of Incentive Mechanisms

The incentive mechanisms classification presented in this section covers most known classes of incentives in general used in different types of human organizations – companies, non-profit (voluntary) organizations, engineering/design teams and crowdsourcing systems. Different organizations employ different (combinations of) incentive mechanisms to stimulate specific responses from agents. The classification is derived by the author from: a) a multidisciplinary review of relevant domain literature cited throughout this section; and b) a survey of existing practices in social-computing platform presented in Section 3.4. Note, for generality purposes, the following descriptions use the terms 'principal' and 'agent' from the well-know 'principal-agent theory' (Section 2.1). The theory introduces the role of *principal*, corresponding to an owner or a manager of the organization, who delegates tasks to a number of *agents*, corresponding to employees (workers) under his supervision. The principal offers the agents an incentive to disclose part of their personal performance information (*signal*) in order to compile a better-suited contract.

- **Pay-per-performance (PPP)** – PPP is one of the most commonly used incentive mechanisms. The guiding principle states that every agent should be compensated proportionally to his contribution. Labor types where quantitative evaluation can be applied are particularly suitable for employing this mechanism.

  A typical representative of the PPP incentive is the *wage*. As shown in Equation (3.1), the wage (w) usually consists of a fixed compensation amount (salary, w_0) and a variable amount (w_{inc}). The variable amount depends on measurable signals (s_i). Every signal is scaled by its weight coefficient (λ_i). Coefficient values depend not only on the actual importance that the principal attaches to a particular signal, but also on the accuracy of measurement that can be achieved.

$$w = w_0 + w_{inc}$$

$$w_{inc} = \beta \cdot \left[\lambda_1 s_1 + \lambda_2 s_2 + \cdots + \left(1 - \sum_{i=1}^{n-1} \lambda_i \right) s_n \right] \tag{3.1}$$

As the Informativeness principle suggests, the more signals we include, the more accurate evaluation we obtain. However, each signal value contains an intrinsic, normally-distributed measurement error. The incentive designer needs to take this into account. A lower value of the coefficient reflects the inaccuracy of measurement. In addition, each signal measurement has costs associated to it. In theory, the additional money needed for paying the rewards is provided from the additional profits obtained from the increased productivity. Therefore, designing an effective PPP incentive requires finding a proper trade-off between the costs of measurement and the accuracy obtained. A signal value can also represent a mark based on a subjective performance evaluation by a supervisor.

In practice, this type of incentive strategies shows significant, verifiable productivity improvements of 25-40% when used for simple, repetitive production tasks, both in traditional companies ([Laz00]), as well as with Human Intelligence Tasks (HITs) on Amazon's Mechanical Turk platform ([MW09]). Other studies, cited in [Pre99], conclude that about 30-50% of the productivity gain is due to the filtering and attraction of better workers, thanks to the *selection effect* of this kind of incentive. This is an important finding, because it explains why even with relatively small amounts of incentives it is possible to achieve higher profits. In fact, increasing the amount of incentives for the same effort over time can lead to the *anchoring effect*, causing the agents to overestimate personal qualities. Keeping them reasonably low enables the selection effect, while not producing the anchoring effect.

Problems that characterize the application of PPP are typically: measurement inaccuracy, choice of signals and multitasking (see Section 3.3). This is in accordance with the newly-observed results from [MW09], which confirm that the quality of work does not increase if the productivity is the only signal evaluated. Additionally applying some aggregate measures of performance can help alleviate these problems. Another problem that may arise due to an implemented PPP scheme is the decreased solidarity among workers, potentially hampering the transfer of know-how and experience among workers. Again, the countermeasures are similar to the ones for multitasking, especially team-based strategies including apprentice relations, where the team gains are related to the professional progress of novices.

As already explained, the context in which a particular incentive strategy is implemented can determine its effectiveness. As demonstrated in [HS96, HA07] PPP may not be an appropriate strategy to choose in cases where the agents are highly interested in the quality of the output. That includes domain experts from almost any area, who due to their expertise can usually earn enough money, so their

primary motivation is not the monetary reward, but the quality of the product, or the reputational gain. PPP is also not suited for large, distributed, team-dependent tasks, where measuring individual contributions is inherently difficult. However, it is frequently used to complement other incentive schemes.

- **Quota-systems & Discretionary bonuses** –

  These mechanisms are tightly related to the PPP mechanism. The conditions for applying them are the same as in case of PPP. We observe and measure the same signals. What is different is that instead of rewarding agents proportionally to their productivity, the principal sets a number of performance-metrics thresholds. When an agent reaches a threshold it is given a one-off, predefined bonus. Quota-systems evaluate at predefined moments whether a performance signal surpasses a threshold (e.g., yearly bonuses). On the other hand, discretionary bonuses are paid whenever an agent reaches a performance level for the first time (e.g., upon reaching a landmark number of customers).

  Being exposed to the issues of inaccurate measurements, these incentive strategies also suffer from appearance of multitasking. Additionally, two other phenomena have been observed [Pre99]:

  – The amount of effort is always dropping after an evaluation if the agent perceives the time until the next evaluation as long enough;
  – When performance level is close to an award-winning quota motivation is significantly higher than the motivation of agents who have already exceeded the quota or feel they have no realistic chances of achieving it (on time).

  Therefore, the evaluation intervals and the quotas should be set in such a way that they can be reachable with a reasonable amount of additional effort, albeit not too easily. It is clear that these two parameters are highly context-dependent, and therefore can be determined only after observing historical records of employee behavior in a particular setup. Ideally, these parameters should be dynamically adjustable.

- **Deferred compensation** – This mechanism is similar to a quota system, in that an evaluation is made at predefined points in time. The subtle but important difference is that deferred compensation takes into account three points in time (t_0, t_1, t_2). At t_0 an agent is promised a reward after successfully passing a deferred evaluation at t_2. The evaluation takes into account the period of time $[t_1, t_2]$ and not just the current state at t_2. In case $t_1 = t_0$ the evaluation covers the entire interval.

  Deferred compensation is typically used for incentivizing agents working on complex, long-lasting tasks. The advantage is that it allows assessing more objectively an agent's performance from a time distance. At the same time, the agent is given enough time $[t_0, t_1]$ to adapt to the new conditions, and then to prove the quality of his work over a period of time $[t_1, t_2]$. The disadvantage of this mechanism is

that it is not always applicable, since agents are not always in a situation to wait long periods for a significant part of their compensation. A common example of this mechanism is the referral bonus. Referral bonus is a reward for employees for recommending or attracting new, suitable employees or partners to the organization.

- **Relative evaluation** – Although this mechanism can have many variations, the common underlying principle is that an entity is evaluated with respect to other entities within a specified group. The entity can be an agent (human), or an artifact (movie, document, product). The relative evaluation is used mainly for two reasons:
  - By restricting the evaluation to a closed group of entities (individuals), it removes the need of setting explicit, absolute performance targets in conditions where such targets cannot be easily set, due to dynamic and unpredictable nature of the environment.
  - It has been empirically proven that people respond positively to competition and comparison to others. (e.g., in [TZ09]).

  Much of the initial success of Amazon and eBay can be attributed to the usage of good *reputation systems* [RKZF00], which in turn rely on relative evaluations of products by the customers.

- **Promotion** – Empirical studies [VCV06] confirm that a prospect of a promotion increases motivation. A promotion is the result of competition for a limited number of predefined prizes. The prize is usually a higher position in the organization's hierarchy, bringing along higher pays, more decision-making power, more respect and esteem, although other prizes are also possible. Often, the benefits enjoyed by the agent after a promotion are disproportionately higher after a promotion compared to the benefits on the previous position. The reason for this is not to reward fairly the person currently holding the position, but rather to make the future contenders for that position more competitive. In fact, the more an agent moves up the hierarchy, the more the rewards become disproportionate to personal abilities and productivity, moving away from the PPP principles and focusing on the competitiveness.

  Promotion is usually treated under the tournament theory ([Laz07]), although other models also exist. The advantage of promotions is that they also eliminate centrality bias and force positive selection, as management cannot select inappropriate persons to advance, as that would mean transferring a great responsibility to unreliable persons, and ultimately produce greater costs to the principal. The drawback is that by valuing individual success, its application can de-motivate agents from helping out each other and engaging in collaborations. Promotion often incorporates subjective evaluation methods, although other evaluation methods are possible.

	Usage environments			Application considerations			
	Traditional Company		Social Computing	Positive application conditions	Negative application conditions	Advantages	Disadvantages
Mechanism	SME	Large enterprise					
Pay-per-performance	++	+++	+++	*quantitative evaluation possible*	*large, distributed, team-dependent tasks; measurement inaccuracy; when favoring quality over quantity*	*fairness; effort continuity*	*oversimplification; decreased solidarity among workers*
Quota Sys. / Disc. Bonus	+	+++	+	*recurrent evaluation intervals*	*when constant level of effort is needed*	*allows peaks/intervals of increased performance*	*effort drops after evaluation*
Deferred Compensation	+	+++	+	*complex, risky, long-lasting tasks*	*subjective evaluation; short consideration interval*	*better assessment of achievements; paying only after successful completion*	*workers need to accept taking the risk and waiting for compensation*
Relative Evaluation	+	++	+++	*cheap group evaluation method available*	*subjective evaluation*	*no absolute performance targets; eliminates subjectivity*	*decreases solidarity; can discourage beginners*
Promotion	++	+++	+	*need to elicit loyalty and sustained effort; when subjective evaluation is unavoidable*	*flat hierarchical structure*	*forces positive selection; eliminate centrality bias*	*decreases solidarity*
Team-based Compensation	+	++	+	*complex, cooperative tasks; inability to measure individual contributions*	*when retaining the best individuals is priority*	*increases cooperation and solidarity*	*disfavors best individuals*
Psychological	+	+	++	*stimulate competition; stimulate personal satisfaction*	*when cooperation needs to be favored*	*cheap implementation*	*limited effect on best and worst workers (anchoring effect)*

Table 3.1: Left: Adoption of incentive mechanisms in different business environments (+ : low, ++ : medium, +++ : high). Right: Different application considerations.

- **Team-based compensation** – This mechanism is used when the contribution of individual agents in a team environment cannot easily be identified. With this mechanism, the entire team is evaluated and rewarded. The reward is then split among the team members. Team-based compensation is susceptible to different dysfunctional behavioral responses. Worse performing agents are effectively hiding within the group. At the same time, the performance of the better performing agents is 'diluted'. Furthermore, teams often exhibit the *free-rider phenomenon* [Pre99] – a situation in which individuals waste more resources (time, money, materials, equipment) than they would if individual expenses could be measured. The consequence is that the total expenses of a team surpass the summed up expenses of independent individuals. Minimizing these negative effects is the primary challenge when applying this mechanism [KO02]. The most common variants are the *team-level compensation* and *profit sharing*.

  When the team-level compensation is used, then the entire team is treated as an individual. After evaluation, the team is compensated by a (usually) monetary reward, that is then equally split among all team members. However, the scenario in which a reward is equally divided among members can lead to the dysfunctional behaviors we described above. In some cases, the better-performing team members will themselves naturally exert pressure on the free-riders, and thus weaken their negative effects. However, in cases where this does not happen, an attempt to differentiate individual efforts can be made. Peer voting is the most effective group evaluation mechanism in such cases, and it may be employed to differentiate agents and split the reward accordingly. This is clearly an example of a hybrid approach combining the idea of a team-based incentive, together with an incentive strategy targeted at individuals to eliminate dysfunctional behavior. Some studies (e.g., [PCE10]) have shown that hybrid incentive strategies are indeed more effective than the pure team-based compensation.

  The decision on the reward amount can be a matter of subjective or quantitative evaluation. Even with a constant high level of effort, the performance of the team can vary throughout its lifetime, depending on the compactness and interconnectedness of the group and the task that the team is working on. So, finding appropriate reward amounts becomes a difficult task [HDG00]. One way to avoid having to decide on the amount of compensation in cases of unknown outcome of the collaborative effort is to tie it to the profit the company (or a company section) makes. This strategy is called *profit sharing*.

  Quantitative or subjective evaluation is usually used, often in combination with peer voting. The incentive action is usually a monetary reward, divided among team members equally or according to individual ratings.

- **Psychological incentive mechanisms** – Psychological incentives are the most elusive, making them hard to define and classify, since they often complement other mechanisms or even occur within them. They are mostly meant to target intrinsic motivation of individuals. They can be operatively described as mechanisms that

must: a) relate to human emotions; b) be advertised by the principal; c) be perceived by the agent.

The already mentioned incentive strategy of Stack Overflow, apart from being an example of relative evaluation strategy at the same time employs a number of psychological incentive mechanisms, like: status, points, badges. They serve not only to attest to the quality of contributors and answers, but also to motivate further contributions. If the points and statuses were not shown to the others and advertised, but rather used for evaluation only, we would still have the indirect evaluation, but would miss out on the possibility to motivate users. Similarly, psychological incentive mechanisms can be coupled with a quota system. We already mentioned how agent's productivity/motivation rises upon nearing a bonus quota. Even though an agent is well-aware of the quota he is trying to achieve, the principal nonetheless advertises how 'little' it is still left to achieve the goal to further boost the motivation. Acting upon human fear is also a tactics commonly (mis)used (e.g., threat of dismissal or downgrading). The threat of being dismissed or downgraded is a powerful motivator, although very stressful for agents and causing different types of unforeseeable dysfunctional behavior.

Perception of the incentive by the agent affects its effectiveness. As the perception is context-dependent, choosing an adequate way of presenting the incentive is not a trivial decision. For example, choosing and advertising the employee of the month in societies where the sense of common good is highly valued can be very effective. In more individually-oriented environments it is the competition that drives the performance. A principal may choose to exploit this fact by showing performance comparisons to (targeted) agents.

Psychological incentives have long been used in video games to elicit player dedication and motivation. Today, the same techniques (gamification) are used to make boring tasks (product reviews, customer feedback) appear more interesting and appealing. As a large number of business models of Internet-based companies depend on the revenues obtained through placing targeted advertisements, incentivizing customers to provide accurate product reviews and leave feedback becomes fundamentally important.

3.2 Composition of Incentive Mechanisms

In practice, employing a single incentive mechanism is usually not enough. Most organizations need to combine different incentive strategies to target different work roles and employees with different statuses. If we look at an engineering company, it is quite common for it to be organized in teams at the lowest level. Such a company would typically have teams of engineers developing the products, testing teams, marketing teams, sales teams, IT teams, customer support teams, etc. In addition, there are employees responsible for providing other services necessary for the running of the company (finance, HR, security, transport, supplies). A number

of teams is forming a unit responsible for a family of related products, or a number of related projects. Different teams are led by managers, who in turn respond to higher-positioned managers (e.g., project managers, product managers, division managers). Within each team engineers can have the ranks, like: junior engineer, senior engineer, distinguished engineer. More experienced engineers usually act as team leads, i.e., highly experienced professionals who drive the technical aspect of product development.

Such a company may decide to use a profit-sharing scheme at the division level – i.e., reward with a bonus all the teams in the division by a profit share if the financial results of the division are positive. In addition to that, individual efforts within a team can be stimulated by individual incentive strategies. In some teams where the contribution can be easily established (e.g., number of customer cases solved in case of customer support) PPP may be used to filter and keep the best employees. In other teams, we can use a combination of the subjective evaluation and peer voting to assess the contributions and adjust the variable part of the salary. Team members are given an opportunity to advance in ranks and into managerial positions and keep advancing further if they accomplish certain goals. Every promotion brings along an increase in pay but also in responsibility. Top managers are evaluated exclusively with respect to the success of the company, and the payment of bonuses may be deferred. Additionally, the company may decide to give out referral bonuses, and award "employee of the year" awards.

Composing incentive mechanisms is often not simply wanted by an organization to improve performance, but also required to prevent dysfunctional behavior (possibly arising from application of original incentive mechanisms). At the end of Section 3.4.2 we describe the former incentive strategy of the company Locationary, which nicely showcases this: the originally introduced PPP was causing too many non-profitable contributions by the crowd workers; subsequently a profit-sharing team-based mechanism was composed into the scheme to influence the quality of the contributions.

Table 3.1 presents a condensed view of different usage environments and application considerations of the incentive mechanisms we described.

3.3 Identifying Composing Parts of Incentive Mechanisms

The related work we analyzed (Chapter 2) has not gone past the level of granularity of incentive mechanisms. We believe that this in great measure prevents development of generic handling of incentives in information systems. The goal of this section is to identify finer-grained building elements that can be individually modeled and used in information systems to compose and encode the incentive mechanisms.

By analyzing the previously described incentive mechanism categories in Section 3.1 we can identify the following *incentive elements*, i.e., the atomic subcomponents in terms of which all mentioned incentive mechanisms can be expressed (Figure 3.1):

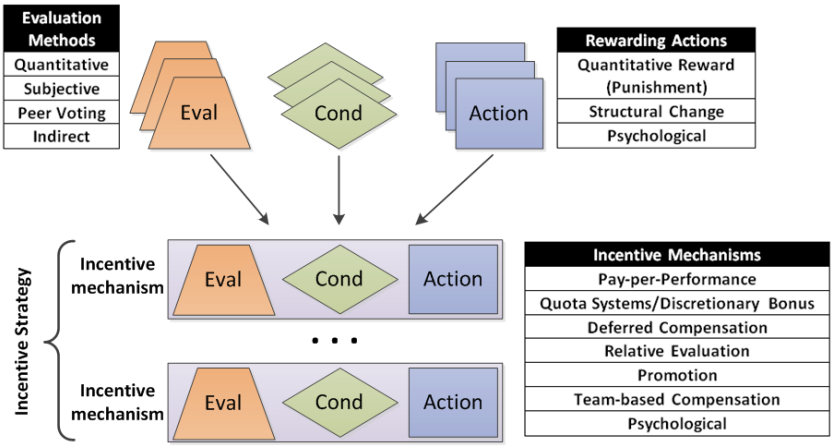

Figure 3.1: An entire incentive strategy of an organization can be composed using smaller, modelable components – incentive elements.

1. **Evaluation method** – provides inputs (signals) on agent performance to the incentive mechanism. Those inputs are evaluated in the logical context defined in the incentive condition.

2. **Incentive condition** – contains the business logic of the incentive mechanism, i.e., the logical rules for application of certain rewarding actions.

3. **Rewarding action** – a concrete activity exerted upon targeted agents meant to influence their *future* behavior. Represents the outcome of the incentive mechanism.

3.3.1 Evaluation Methods

Individual Evaluation Methods

As the name suggests, these methods are used to evaluate agents individually, i.e. not explicitly conditioning their scores with the scores or opinions of other agents.

Quantitative evaluation represents the rating of individuals based on measurable properties of their contribution. Quantitative evaluation is attractive because it does not require human participation and can be entirely implemented in software. It is considered to be a precise and fair method. However, as it is not suitable for all purposes, it is often combined with other methods.

Some labor types are suitable for precisely measuring the individual contributions of an agent (e.g., OCR correction, image labeling). In this case the agent can simply be evaluated on the number of units processed. But apart from the most primitive labor types, evaluating an agent's performance requires evaluating different performance aspects (i.e., measurable signals), the most common being: productivity, effort and quality of product. Different metrics are usually taken into consideration with different

weights, depending on their importance and measurement accuracy. For example, in case of a product assembly line, the metric can be the number of units assembled, but also (with lower importance) the quality of assembled products, since the quality of work of a particular worker cannot always be precisely established. In other cases, e.g., in case of telemarketing where different phone agents are covering different neighborhoods, towns or ethnic groups, effort of agents may be highly valued compared to the number of units sold, because the success of sales of a particular product may depend on the geographical location of the area, wealth, climate or local habits. Effort is always highly valued metric in cases where agents are not working under equal conditions.

Problems that arise here are the measurement inaccuracy and the difficulty of choice of proper signals and weights. An additional problem is the phenomenon called *multitasking*. In spite of its counterintuitive name, it refers to agents putting most of the effort into tasks that are subject to incentives, while neglecting other tasks, subsequently damaging overall performance [HM91]. Principal can fight this kind of misbehavior by additionally employing subjective evaluation.

Subjective Evaluation Many aspects of human work are not quantifiable. The reasons can be:

- there are no clear outputs to evaluate;
- contribution has properties understandable and valuable to humans only;
- tasks are too complex to be clearly defined.

For example, whether a logo design is good or not is ultimately a matter of aesthetic preference of the customer. In such cases we need to substitute an objective measurement with a human, subjective assessment of the quality of work. In this case a human acts as a mapping function that quantifies human-oriented work aspects by wrapping together all the undefinable signals into one subjective assessment signal. Subjective evaluation is a widely used evaluation mechanism. Its advantages are simplicity and low cost, but its implementation as a human-based task makes it inherently imprecise and prone to dysfunctional behavioral responses.

Some of the phenomena that characterize this evaluation method that have been observed in practice [Pre99] include:

- *Centrality bias* – ratings concentrated around some average value. Not enough differentiating of 'good' and 'bad' workers.
- *Leniency bias* – discomfort to rate 'bad' workers with low marks.
- *Rent-seeking activities* – actions taken by employees with the particular goal of increasing the chances of getting better rating from the manager, often including personal favors or unethical behavior.
- *Embellishment* – tendency of managers to rate subordinates better than deserved if manager's own reputation or team-bonus depend on it.

- *Theft* – tendency of managers to consistently give lower scores to subordinates to save budgeted money if the compensation of the employees depends on the scores.

Centrality bias and leniency bias can be prevented by checking if the ratings follow a distribution with specified parameters. However, this is not always preferable in practice, because it could motivate managers to perform data fitting. The usual solution for these, and also for other listed problems, is to make the ratings of managers subject of incentives as well. In practice, it means punishing a manager if his ratings of individual subordinate employees significantly or consistently differ from the ratings of other managers who worked with them, or from the ratings of their co-workers.

Group Evaluation Methods

Group evaluation methods evaluate an agent by aggregating assessments of community members.

Peer Evaluation (Peer Voting) is an expression of collective intelligence where members of a group evaluate the quality of other members. In the ideal case, the aggregated, subjective scores represent a fair, objective assessment.

The better the voters know the object of the vote, the better they can judge it. With the voting group large enough, this method eliminates or alleviates the problems that subjective and quantitative evaluation suffers from. Centrality and leniency biases are alleviated by the fact that the votes will be better distributed, as the aggregated scores cannot be subjectively influenced. Since there is no more a single voter who decides, activities that target single voter's interests, like embellishment, theft and rent-seeking are eliminated. Since a large number of different and professional peers evaluates different performance aspects, that leaves less space for multitasking activities.

This method also suffers from different weaknesses. In small, interconnected groups the voters can be unjust or lenient because of personal reasons. They can also feel uncomfortable and exhibit dysfunctional behavior if the person being judged knows their identity. Therefore, anonymity is often a favorable property in such cases. Another way of fighting these dysfunctional behaviors is to make voters subject to incentives: votes get compared, and those that stand out are discarded. At the same time one keeps track of agent's voting history to prevent consistent unfair voting.

When the community consists of a relatively small group of voted persons and a considerably larger group of voters, and both groups remain stable throughout the time, use of this method is particularly favorable. In that case, the voters have a good overview of much of the voted group. Since the relation voter-voted is unidirectional and will probably not change over time, voters do not have interest to exhibit dysfunctional behavior. This pattern is very common on the Internet today.

The method works as long as the size of the voted group remains small enough. As the voted group grows, voters become unable to catch up and acquire all the new facts necessary to pass fair judgments. Then they opt to rate better those persons or artifacts they know or feel have traditionally good reputation ([Pri76]). This phenomenon is known as *preferential attachment*, or colloquially "the rich get richer". It can be noticed on news

sites that attract high numbers of user comments. Newly arriving readers usually tend to read and vote the most popular comments only, leaving many interesting comments practically unvoted. Therefore, determining the group of voters and voted agents is crucial when designing instances of this incentive mechanism.

In traditional businesses, the major obstacle for applying this method was the cost, both in time and in money. Additionally, it was technically challenging, if not impossible, to apply this method often enough, and with appropriate voting groups. However, use of information systems, the Internet and social networks permitted a drastic decrease in application costs. A number of implementations already exist on the Internet (e.g., Like-button, binary voting, star voting, polls) but we lack a unified model able to express their different flavors and specify the voters and voted groups. Again, the act of voting is modeled as a human task, requiring active human participation.

Indirect Evaluation Since human performance is often difficult to define and measure, it is common to evaluate humans based on properties and relations among the artifacts they produce. As the artifacts are always produced to be consumed by others, the decision on their quality is left to the community.

The artifacts are connected by various relations among themselves (contains, refers-to, subclass-of, etc.), as well as with users (e.g., author, owner, consumer). The method of mapping properties and relations of artifacts to scores is non-trivial in general case. An algorithm tracks relations and past interactions of the agent or his artifacts with the artifact that is being evaluated and calculates the score. For example, in [Kau11, SHY+08b] the authors evaluate users of peer-to-peer networks by monitoring the content contributions of the users. Similarly, scientists can be evaluated by the number of publications in journals, which in turn are ranked by the impact factor, which depends on the number of citations scientists make. The well-known e-labor (freelance) platform UpWork[1] uses a proprietary algorithm for worker evaluation and ranking [DS14]. Usually, a tailor-made algorithm needs to be developed, or an existing one adapted to a particular environment. The major difference from peer evaluation is that here the agent does not actively evaluate the artifact, and hence the algorithm is not dependent on interacting with the agent.

Another efficient way of fighting this is by employing peer voting to evaluate artifacts. If we have favorable conditions for applying low-cost peer evaluation on artifacts then we eliminate the problem of dummy artifacts. It is also an added value for the accuracy of the algorithm. As the conditions for applying peer evaluations within Internet communities are usually favorable, this is a very commonly employed technique today.

Advantages and drawbacks of this method fully depend on the properties of the particular algorithm. If the algorithm is suitable it will exhibit fairness and prevent false results. The cost of this method also depends on the costs of developing, implementing and running the algorithm. A common problem is that users who know how the algorithm works may try to deceive it by outputting dummy artifacts with the sole purpose of increasing their scores. Detecting and preventing such attempts requires amending the algorithm, further increasing the costs.

[1] `https://www.upwork.com/` Result of merger between oDesk and Elance.

Table 3.2 lists some common application and composability considerations for evaluation methods presented here. It also indicates how drawbacks of a particular evaluation method can be alleviated by combining them with other methods.

		Application considerations			Composability			
Evaluation Methods		Advantages	Disadvantages	Active Human Participation	Issues	Alleviated by	Solving	Typical Usage
Individual	Quantitative	*fairness, simplicity, low cost*	*measurement inaccuracy*	*no*	*multitasking*	*peer evaluation; indirect evaluation; subjective evaluation*	*subjectivity-caused issues;*	*PPP; Quota Systems; Promotion; Deferred Compensation*
	Subjective	*simplicity, low cost*	*subjectivity; inability to assess different contribution aspects*	*yes*	*centrality bias; leniency bias; deliberate low-scoring; embellishment; rent-seeking activities*	*incentivizing the decision maker to make honest decisions (e.g. by peer evaluation)*	*multitasking*	*Relative Evaluation; Promotion;*
Group	Peer	*fairness; low cost in social computing environment*	*active participation required*	*yes*	*preferential attachment; coordinated dysfunctional behavior of voters*	*incentivizing the peers (e.g. also by peer evaluation)*	*multitasking; subjectivity-caused issues*	*Relative Evaluation; Team-Based Compensation; Psychological*
	Indirect	*takes into account complex relations among agents and their artifacts*	*evaluation algorithm cost of development and maintenance*	*no*	*depending on the algorithm in use; fitting data to the algorithm*	*peer voting; better implementation of the algorithm*	*subjectivity-caused issues; peer evaluation issues*	*Relative Evaluation; Psychological; PPP*

Table 3.2: Application and composability considerations for evaluation methods.

3.3.2 Rewarding Actions

In order to induce a future specific behavioral response from agents the principal must perform one or more **rewarding actions** over them.[2] The application of the actions is often colloquially called *rewarding* or *incentivizing*. A rewarding action can be one of the following types:

- Reward
- Structural change
- Psychological action

Rewards (but also: penalties, fines) can be modeled as quantitative changes in parameters associated with an agent or a group of agents. For example, a parameter can be the wage amount, which can be incremented by a bonus, or decreased by a penalty. Similarly, a parameter can be an agent's status, or a collection of objects in agent's possession (e.g., FourSquare motivates users by assigning them different badges for different check-in patterns)

Structural changes are an empirically proven [Laz07] motivator. A structural change does not imply strictly position advancement/downgrading in the traditional tree-like management structures. It also includes belonging to different teams at different times or collaborating with different people. For example, working in the team with a distinguished individual can diversify an agent's experience and boost his career. One way of modeling structural changes is by graph rewriting [BH02].

Psychological actions. Although all incentive actions work by exerting a psychological effect, what we denominate as psychological actions are only those in which an agent is influenced purely by being presented some information. For example, we may decide to show an agent only the results of a couple of better-ranking agents rather than the full rankings. That way, the agent will not know his position in the rankings, which can be beneficial in two ways – by preventing the "anchoring effect" [MW09] for agents in the top part of the rankings and by preventing discouragement of agents in the lower part. Psychological actions do not include any explicit parameter or position change, but the diversity of presentation options means that defining a unified model for describing different psychological actions is still an open challenge. Effects of these actions are hard to measure precisely, but apart from empirical evidence [FJ01], their broad adoption on the Internet today is another clear indication of their effectiveness.

Apart from the type of the rewarding action, another crucial aspect of the action's efficacy is the *timing* of the action (Figure 3.2). We can distinguish the moments: 1) when the action is announced/advertised to the agent; and 2) when the action is applied. The period between the two moments can be used to evaluate agent signals. The period spanning from the moment of the announcement and lasting possibly for an unspecified

[2] A *punishment* is simply a term used to describe a rewarding action meant to prevent a specific behavior instead of inducing one.

amount of time, but at least until the moment of the application of the action is called the *effectiveness range*.

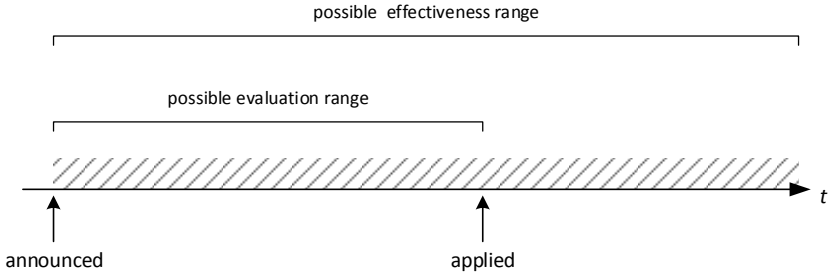

Figure 3.2: Application and effectiveness of rewarding actions.

3.3.3 Incentive Conditions

Incentive conditions state precisely how, when, and where to apply rewarding actions. Each consists of at most three components (subconditions):

- *Parameter component* expresses a subcondition in form of a logical formula over a specified number of parameters that describe an agent. For example: such a condition could filter out all the agents whose productivity is less than the team's average.

- *Time component* is used to formulate a condition over past behavior of an agent. For example: select all the agents who within last three months had an unsatisfactory productivity level.

- *Structure component* filters out agents based on the relations they take part in. This component can be used to select members of a team, or all the collaborators of a specific agent.

By using all three components at the same time we can specify a complex condition, e.g.: "incentivize the subordinates of a specific manager, who over the last year achieved a score higher than 60% in at least 10 months."

Incentive conditions are part of the business logic, and as such are stipulated by the domain experts empowered by the principal to manage the workforce. However, a small organization can take an advantage of some good practices and employ pre-made incentive models (patterns) adapted to fit particular organization's needs. Feedback information obtained through monitoring execution of rewarding actions can be used to adapt condition parameters.

3.4 A Survey of Incentive Mechanisms in Real-World Social Computing Platforms

3.4.1 Survey Criteria

In 2012[3] we surveyed over 1600 Internet-based companies and organizations that describe themselves using keywords such as 'social computing' or 'crowdsourcing'. We investigated their business models and contracts offered to users/participants/workers, as described on organizations' websites. The main goals of the survey were:

- To demonstrate that the classification we had established mostly based on the multidisciplinary literature survey is valid and applicable also for internet-based social-computing business models.
- To gain a better insight into the usage patterns of different incentive mechanisms, evaluation methods, rewarding actions, and combinations thereof.

We filtered the companies in such a way to exclude those that fulfill any of the following criteria:

- Crowd-funding websites, humanitarian & community-benefit sites or voluntary-contribution sites.
- Sites who only act as intermediaries to establish links between human service providers and consumers (often earning by charging commission for enabling secure transaction environment), except when they employed incentive mechanisms to increase the numbers and quality of participants.
- Sites which just provide a technical solution or environment to do the business.
- Sites that directly sell products created/owned by the crowd – e.g., stock photography.
- Sites that do not disclose or clearly/publicly state the incentive scheme.
- Sites that were not in one of the following languages: English, German, Spanish, Portuguese, Italian.

We also decided not to include companies employing gamification approaches into our classification part of the survey. There are several reasons for this decision:

- Every gamification approach can be considered a psychological strategy, with quantitative evaluation.
- A company rarely bases its principal incentive strategy on gamification only.
- Many companies employing gamification are not primarily social-computing companies, but traditional companies relying on gamification elements to attract users for performing uninteresting tasks.

[3]Please note that some information in this section may be outdated due to the highly dynamic nature of the social-computing market.

However, we acknowledge the growing importance of psychological incentive mechanisms and including gamification approaches. For example, SAP (within their SAP Community Network[4]) offers their employees to build up reputation by writing articles, guides, samples, answering question, etc. That not only helps other team members get useful information more easily, but also raises the reputation of the employee that transfers the knowledge. The contributors are scored, and score-boards are publicly available. Even though the score does not bring any concrete rewards, the reputation gained can implicitly bring better career advancement opportunities and higher respect from colleagues.

Another example of how gamification can incentivize employees is the use of games in teaching employees to better understand, use and represent the products of their own company (e.g., IBM Innov8[5]). While employees get distracted from dull tasks for some time and learn something in the process, the company gets better skilled and more competitive workers. Finally, yet another gamification example can be seen in a project run by the National Library of Finland[6], where contributors are engaged in a game, with the true purpose of correctly recognizing scanned material from the library archives.

3.4.2 Survey Results

After applying the filtering rules stated above, out of over 1,600 examined companies we identified 140 companies (8.75%) that employed and clearly described the rewarding/incentive practices (types of awards, evaluation methods, rules, conditions). We then classified them according to the previously described classification (Section 3.1).

The most surprising finding was that 59 of the 140 companies (42%) employed a very simple 'contest' business model employing relative evaluation incentive mechanism, meaning that a creative task is deployed to the crowd. Each crowd member (or entity) then submits a design. The best design is in the vast majority of cases chosen by subjective evaluation (85%). That was expected, since the company buying the design reserves the right to ultimately decide on the best design. In fact, in many cases, it is the only possible choice. The remaining 'contest' companies employ peer evaluation (10%) or quantitative evaluation (5%). When using peer evaluation, the company delegates the decision on the best design to the crowd of peers, while taking the risk of producing and selling the design. In some cases, e.g., programming contests, the artifacts are evaluated quantitatively, by automated testing procedures. It is worth noticing that using peer or quantitative evaluation produces quantifiable ratings of the users. In such cases, individuals are better motivated to take part in future contests even if they feel they cannot win, because they can use their ranking as a personal quality proof when applying for other jobs or just as a matter of prestige. We expect to see an increase in the latter two evaluation categories, as they help improve the quality of designs if the crowd is large, contains quality individuals, and is properly motivated. However, building up and

[4] http://scn.sap.com/

[5] http://www.ibm.com/innov8/

[6] http://www.digitalkoot.fi

managing such a crowd also implies the use of other incentive mechanisms. The contest model alone dissuades good (but not the best) agents, who rarely win the contests.

Apart from the 59 organizations running contests, relative evaluation is used by another 16 organizations, usually combined with various other mechanisms. This makes relative evaluation by far the most widely used incentive mechanism on the social computing market today (54%) (Table 3.3). This is in contrast with its use in traditional businesses, where it is used considerably less [Arm10], as the implementation costs are much higher.

Incentive Mech. Type	**No. of Companies**	**Percentage**
Relative Evaluation	75	54%
Pay-per-performance	46	33%
Psychological	23	16%
Quota Sys. / Disc. Bonus	12	9%
Deferred Compensation	10	7%
Promotion	9	6%
Team-based Compensation	3	2%

Table 3.3: Use of incentive mechanism categories by social computing companies.

The other significant group are the companies that pay the agents for completing human microtasks. We found 46 such companies (33%). Some of them are general platforms for submitting and managing any kind of human-doable tasks (like the emblematic Amazon Mechanical Turk[7]). Others offer specialized human services; most commonly: writing reviews, locating software bugs, translating or performing some simple, location-based tasks, etc. What all those companies have in common is the use of pay-per-performance mechanism (PPP). The tasks range from very simple (in majority of cases) to more imaginative and complex, like locating bugs. Quantitative evaluation is the method of choice in most cases (65%). Quantitative evaluation sometimes produces a binary output, e.g., when submitting successful/unsuccessful steps to reproduce a bug. The binary output allows expressing only two levels of the quality of work, so the agents are rewarded on a per-task basis for every successful completion. In that case, the company usually requires no entry tests for joining the contributing crowd. In other cases, the quality of work is not easy to establish and the output is proportional to the quantity of finer-grained units performed (e.g., word count in translation tasks) but the agents are usually asked to complete entry tests. Pay rate for subsequent work is determined by the test results. Other evaluation methods include subjective and peer/indirect evaluation, both at 17%. It is interesting to note that the peer evaluation for double-checking the results is not frequently employed, as companies find it cheaper to test the contributors once and trust their skills later on. However, as companies start to offer more complex

[7] `http://www.mturk.com/`

human tasks, quality assurance becomes imperative, so we expect so see a rise in peer and indirect evaluation. Eleven companies combined pure PPP with other mechanisms.

Only three companies employ a combination of 4-5 different mechanisms (Table 3.4). The most famous of them is uTest.com. As their business model requires them to have a large crowd of dedicated professionals, it becomes clear why they employ more than just simple PPP.

ScalableWorkforce.com is the only company in our study that advertises the importance of *crowd (workforce) management*. They offer the tools for crowd management on Amazon Mechanical Turk to their clients. Their tools allow for tighter agent collaboration (creating a sense of community among workers), workflow management, performance management and elementary career building.

No. of Inc. Mech.	**No. of Companies**	**Percentage**
1	116	83%
2	15	11%
3	6	4%
4	3	2%

Table 3.4: Number of incentive mechanisms used by social computing companies. Over 80% of the companies employ only one mechanism.

Twelve companies (8.5%) rely uniquely on psychological mechanisms to assemble and improve the agent community. The common trait is relying on indirect influence of rankings in agent's (non-virtual) professional life. For example, avvo.com attracts large communities of doctors and lawyers in the US who offer free responses and advice to people visiting the website. Quality and timeliness of professionals' responses affect their reputation rankings, which can be used as a prestige advertisement to attract actual paying customers to their private practices. Another very interesting example are companies like crowdpark.de or prediculous.com. They ask their users to 'predict' the future by placing bets on upcoming events using virtual currency. Users that have best predictions over time earn virtual trophies (badges), which is the only incentive for people to participate. The crowdsourced odds can be used to adjust odds in real betting.

Team-based compensation was used by only three companies we surveyed. For example, mercmob.com encourages formation of virtual human teams for various tasks. An agent expresses confidence in the successful completion of a task by investing part of a limited number of his 'contracts'. Once invested, the contracts are tied to the task, motivating the agents that accept the task to give their best to self-organize in a team and attract others to accomplish the task. If in the end the task is completed successfully each agent gets a monetary reward proportional to the number of invested contracts.

Discretionary bonuses or quota systems are used by eleven companies (8%). However, they are always used in combination with another mechanism – most commonly PPP (64%), as is also the case in traditional companies.

Deferred compensation is used by 7% of the companies, and usually as the only mechanism employed. Bluepatent.com is a company that crowdsources the task of locating 'prior art' for potential patent submissions. The agents (researchers) are asked to find and submit relevant documents proving existence of prior art. Deciding on the validity and usefulness of such documents is an intricate task, and hence the decision on the compensation is delayed until an expert committee decides on it. Advisemejobs.com pays out classical referral bonuses to the agents who suggest appropriate job candidates.

Only 7% of the companies offer some kind of career advancements, combined with other mechanisms. As the crowd structure is usually plain, the career advances usually mean a higher status, implying a higher wage. We have encountered only two cases where the advancement also meant some kind of structural change, with an agent taking responsibility of leading or supervising lower-ranked agents (e.g., uTest.com). In traditional companies the decision on a promotion of an employee is usually a matter of subjective evaluation by his superiors. With the promotion being the most commonly employed traditional incentive, the subjective evaluation is then also the most commonly used evaluation method. However, if we take out of the picture the companies running creative contests, where the artistic nature of the artifacts forces the use of subjective evaluation, we see that in the world of Social Computing this trend has reversed. Subjective evaluation trails behind quantitative and peer evaluation (Table 3.5). This is explained by the fact that the use of information systems enables cheaper measurements of different inputs and setting up of peer voting mechanisms.

Evaluation Method	**No. of Companies**	**Percentage**
Quantitative Evaluation	51	63%
Peer Voting + Indirect	35	43%
Subjective evaluation	14	17%

Table 3.5: Use of evaluation mechanisms (excluding companies running creative contests).

A small number of companies employ a combination of different incentive mechanisms. Locationary[8] was a company that used agents spread around the world to expand and maintain a global business directory by adding local business information.

Their strategy combined a number of incentive mechanisms: 1) 'lottery tickets' (a Quota system, also known as 'conditional PPP'); 2) team-based compensation (based on the 'shares' of added companies); 3) deferred compensation, based on the trust scores of the agents.

With every new entry added/corrected an agent wins 'lottery tickets' that increase the chances of winning a reward in a lottery. However, there is a minimum quota of tickets that represents the condition to enter the draw (hence 'conditional PPP'). Tickets are not tied to any particular directory entry. Agents are given different ticket amounts

[8]The incentive strategy was acquired before the company was taken over and integrated by Apple, Inc. `http://allthingsd.com/20130719/apple-acquires-local-data-outfit-locationary/`. The original URL was `http://www.locationary.com/`

for different actions (adding, editing or verifying different directory entry fields). The amount of tickets issued to an agent for editing an entry depends on how valuable the (accuracy of the) entry is to the company. For example, a Google street view URL is more valuable than the URL of the web page of the place. Similarly, fixing outdated/incorrect data is highly appreciated.

This mechanism incentivizes the increased activity of the agents, but also motivates them to cheat, as some people will start inputting invalid entries to increase their chances of winning. To counteract this caused dysfunctional behavior deferred compensation is used. The agents are only allowed to enter the prize draws if (apart from the ticket quota) their trust score is high enough. The trust metric plays a crucial role here. Trust is proportional to the percentage of the approved entries, and this metric discourages agents to cheat. The entries can be approved or disapproved only by other highly trusted agents (an example of peer evaluation). Trusted agents are motivated to perform validation tasks by getting more lottery tickets than they would get for adding/editing fields. On the other hand, cheaters are further punished by subtraction of lottery tickets for every incorrect data field they provided.

The incentive strategy described so far does a good job of attracting a high number of entries and keeping them fresh and accurate. However, it does not discriminate between the directory entries themselves. That means that it equally motivates agents to enter information on an insignificant local grocery store, as it motivates them to enter/update information on a high-profile company. As Locationary used to rely on advertising revenues, that meant that an additional incentive mechanism attracting higher numbers of profitable entries needed to be included on top of the strategy described so far. The team-based compensation plays this role. Locationary used to share 50% of the revenues originating from a directory entry with the agents holding 'shares' of that entry. Shares were given to the people who were first to provide new/additional information on the entry. Again, cashing out was permitted only to the trusted agents.

This example demonstrates how different mechanisms are used to target different necessities, and how they need to be composed to achieve their full effect. In Table 3.6 we list some examples of companies employing different evaluation methods within different incentive mechanisms.

3.4.3 Survey Conclusions

With creativity contests and microtask platforms dominating the landscape of Social Computing today we see that the organizational structure of agents is usually flat or very simple. Hierarchies and teams of agents usually do not exist. In such environment, most Social Computing companies need to use only one or two simple incentive mechanisms. Promotion, commonly used in traditional companies, is rarely found within Social Computing companies. The reason is the short-lived nature of transactions between agents and the Internet companies. For the same reason, team-based compensation is also poorly represented. The idea of building a "career in the cloud" is still considered in theoretical domain.

	Quantitative	Subjective	Peer	Indirect
Pay-per-performance	mturk.com	content.de	crowdflower.com	translationcloud.net
Quota/Discretionary Bonus	gild.com		carnetdemode.fr	
Deferred Compensation	advisemejobs.com	bluepatent.com	crowdcast.com	
Relative Evaluation	netflixprize.com	designcrowd.com	threadless.com	topcoder.com
Promotion	utest.com	scalableworkforce.com	kibin.com	
Psychological Incentives	crowdpark.de	battleofconcepts.nl	avvo.com	
Team-based Compensation		mercmob.com	geniuscrowds.com	

Table 3.6: Examples of companies employing different evaluation methods (columns) within different incentive mechanisms (rows) at the time the survey was compiled. Note: mechanisms presented here may not represent the only or primary mechanisms that the company uses.

On the other hand, most traditional companies combine different, elaborate mechanisms to elicit particular responses from agents and retain the quality workers [Pre99]. The mechanisms complement themselves to mutually cancel out individual drawbacks. In many cases the (more complex) mechanism combinations arose only after the practical use of simpler combinations exposed the weaknesses which the agents would exploit to their benefit. We encountered such experiences also in the surveyed social computing companies, with, e.g., the Locationary incentive scheme being a very illustrative example thereof.

Our survey shows that as the price of application of quantitative, peer and indirect evaluation has lowered, relative evaluation and PPP became the most popular incentive mechanisms among Social Computing companies. Subjective evaluation, although in total numbers well represented, is found largely within companies that base their business model on organizing creativity contests. Psychological incentives and gamification approaches are gaining ground. We expect them to achieve their full potential as amplifiers for other incentive mechanisms.

The envisioned growth in complexity of business processes and organizational structures for Social Computing will require novel, automated ways of handling behavior of agent crowds. That is why we perceive a necessity to develop models of incentive mechanisms and incentive management frameworks fitting existing business models and real-world socio-technical systems, capable of supporting complex, composable incentive mechanisms.

Such frameworks need to be able to monitor worker crowds and perform runtime applications and adaptations of incentive mechanisms to prevent diverse negative effects we described (e.g., free-rider problem, multitasking, biasing, anchoring, preferential attachment), switching when needed between different evaluation methods, rewarding actions and incentive conditions in runtime, while minimizing overall costs. This way, particular worker groups and behaviors can be efficiently targeted.

Additional benefits would include the following:

- Historical data can be used to detect performance bottlenecks, preferable team compositions, optimal wages, etc. Additionally, we can make predictions and

choose optimal composition of incentive mechanisms for the future. This opens up a possibility for novel ways of achieving indirect, automated team adaptability through application of incentives.

- For certain business models, application of proven incentive patterns cuts costs in both time and money. The incentive patterns can be tweaked to fit particular needs based on feedback obtained via monitoring.
- By generalizing and formally modeling incentive mechanisms, we can encode them in a system-independent manner. That way, they become portable and reusable on different underlying systems, without having to write system-specific programming code again.
- The management of rewarding and incentives can be offered remotely as a Web Service.

CHAPTER 4

Modeling Incentives for Use in Socio-Technical Systems

In Chapter 3 we performed the classification of incentive mechanisms by analyzing described incentive practices in the literature and surveying their practical application in the social computing domain. This allowed us to identify the composing elements of incentive mechanisms and their typical usage patterns.

This chapter builds on the first three chapters to develop actionable incentive models that allow us to:

(a) Model the previously described incentive mechanisms

(b) Model the application of incentive mechanisms from (a) and responses to it.

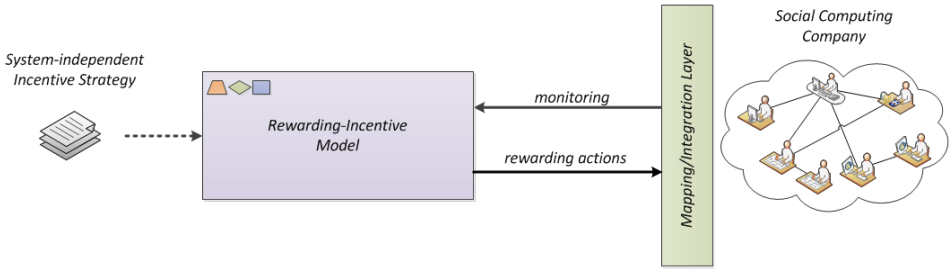

Figure 4.1: A conceptual illustration of a system capable of translating portable incentive strategies into concrete rewarding actions for different socio-technical platforms. The system corresponds to the lower section of Figure 1.2, representing the part of the control loop affecting the human component of a socio-technical system.

The models introduced in this chapter will allow us to build frameworks operating on these models, and offering the functionality of incentive management to third parties (Figure 4.1).

4.1 Comprehensive Incentive Model

Application of incentives always requires two interested parties - an *authority* (principal) and a *worker* (agent). The authority is interested in stimulating, promoting or discouraging certain behavioral responses in workers. The incentive exhibits its psychological effect by promising the worker a reward or a punishment based on the actions the worker will perform. The wish to get the reward or escape the punishment drives the worker's decisions on future actions. The reward (punishment) can be material or psychological (e.g., a change of status in a community – ranking, promotion). The type, timings and amounts of reward need to be carefully considered to achieve the wanted effect of influencing a specific behavior in a planned direction. In addition, introduction of incentives introduces additional costs for the authority who hopes to compensate for them through the newly arisen worker behavior (e.g., increased productivity).

However, as soon as an incentive mechanism is introduced, it produces dysfunctional behavior responses in the worker population. The workers, being rational agents, adapt to the new rules and change their working patterns, trying to exploit the new incentive to profit more than the rest of the population. The authority compensates for this by introducing other incentive mechanisms targeting the dysfunctional behavior, further increasing the authority-side costs, and causing new types of dysfunctional behavior. However, once the proper combination of incentive mechanisms is put in place and calibrated, the system enters a stable state. The problem with the crowdsourcing/social computing processes is that the system may not stay long in a stable state due to an unforeseen change in worker participation or collaboration pattern. Therefore, the incentive setup needs to be reconfigured and re-calibrated as quickly as possible, in order to avoid incurring high costs to the authority. This feedback control-loop involving the authority and the worker represents the actual incentive mechanism that we are interested in modeling.

Modeling an incentive mechanism, therefore, always involves modeling both the authority and the worker side, as well as the possible interactions between them. In Figure 4.2 we show an abstract representation of the model of incentive mechanism.

Workers differentiate from each other by having different sets of personal characteristics (e.g., accuracy, speed, experience). The characteristics are determined by a private set of variables stored in the *internal state* S. The internal state also contains records of worker's past actions. The internal state is private to the worker, and is used as one of the inputs for the *decision-making function* f_a that describes the worker's choice of the next action to perform.

In majority of cases the internal state variables are normally distributed across the worker population. Occasionally, certain variables can be intentionally given predefined values to simulate a certain type of behavior, or a specific class of workers. This can also

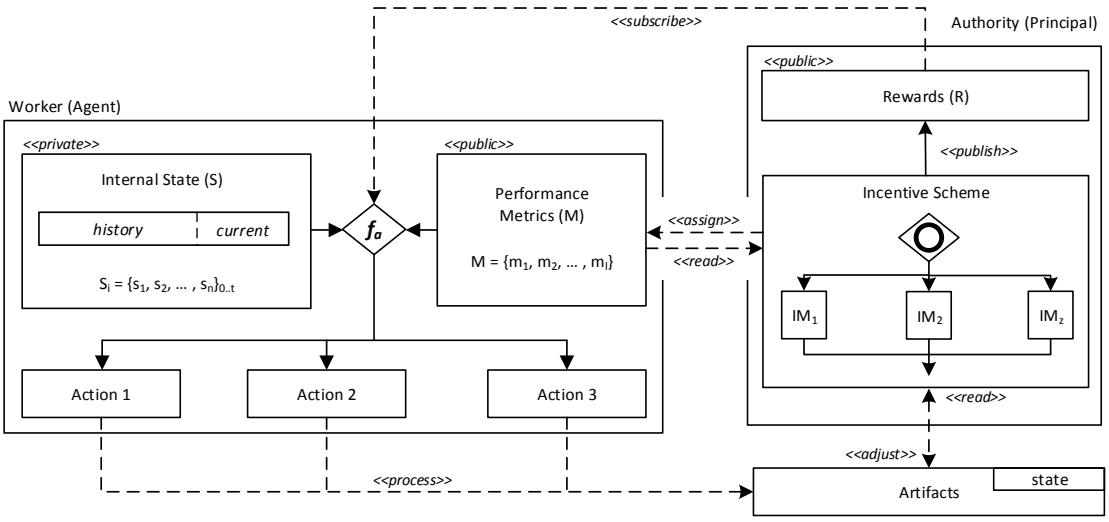

Figure 4.2: Incentive mechanisms need to capture the interaction between workers (agent) and authority (principal).

be used to simulate changes in behavior after unconstrained interactions among workers; for example, after an interaction with another worker a worker may be "persuaded" to decrease his performance levels by lowering his internal *effort* metric. The algorithms modifying the internal state S are not prescribed by our model, and are freely chosen by the system designer.

Apart from the internal state, each worker is characterized by the publicly exposed set of *performance metrics* M that are defined and constantly updated by the authority for each worker. The performance metrics reflect the authority's perception of the worker's past interactions with the system (e.g., trust, rank, expertise, responsiveness). Knowing this allows the worker to decide better on his future actions. For example, knowing that a poor reputation will disqualify him from getting a reward in future may drive the worker to work better or to quit the system altogether. It also allows him to compare with other workers. Therefore, the set of performance metrics is another input for the decision-making function f_a.

The third input for the decision-making function f_a is the set of promised *rewards (punishments)* R. Rewards are expressed as publicly advertised amounts/increments in certain parameters that serve as the recognized means of payment/prestige within the system (e.g., money, points, stakes/shares, badges). They are specified per action, per artifact, and per performance metrics (or a combination thereof), thus making them also dependent on a particular worker. For example, a reward may promise an increase of at least 100 points to any/first worker who performs the action of rating an artifact. The amount of points can then be further increased or decreased depending on the worker's reputation.

Workers interact with the authority solely by performing actions over *artifacts* (K)

offered to the worker population by the authority. A worker's behavior can thus be described as a sequence of actions in time, $A_t \in A = \{A_0, ..., A_n\}$, interleaved with periods of idling (idling being a special-case of action). The set of possible actions is the same for every worker. However, the effects of the execution of an action may be different, depending on the worker's personal characteristics from the internal state S. For example, a worker with innate precision and bigger experience can improve an artifact better than the worker not possessing those qualities.

As previously stated, worker's next action is selected through the use of a decision-making function $f_a = f(S, M, R)$ potentially considering all of the following factors: a) the statistically or intentionally determined personality of the worker; b) historical record of past actions; c) authority's view of one's own performance; d) performance of other workers; and e) promised rewards, with respect to the current state of one's performance metrics. The decision-making function is context-dependent, and defined by the model designer based on the observed/expected worker behavior.

From a social computing perspective, the authority's motivation for offering artifacts for processing to the worker crowd is to exploit the crowd's numerosity to either achieve higher quality of the artifacts (e.g., in terms of accuracy, relevance, creativity), or lower the cost (e.g., in terms of time or money). This motivation guides the authority's choice of incentive mechanisms. Authority has at its disposal a number of *incentive mechanisms* IM_i. Each one of them should be designed to target/modify only a small number of very specific parameters. Thus, it is the proper addition or composition of incentive mechanisms that allows the overall effect of an incentive scheme, as well as fine-tuning and runtime modifications.

An incentive mechanism IM takes as inputs: 1) the current state of an artifact K_i; 2) the current performance metrics of a worker M_j; and optionally 3) the output from another incentive mechanism returning the same type of reward – ${R'}_{a_k}$. The output of an incentive mechanism is the amount/increment of the reward R_{a_k} to offer to the worker M_j for the action a_k over artifact K_i.

$$IM : (K_i, M_j, {R'}_{a_k}) \rightarrow R_{a_k} \tag{4.1}$$

The true power of incentive mechanisms lies in the possibility of their combination. The reward (f_R) can be calculated through a number of additions (+) and/or functional compositions ($\circ$) of different incentive mechanisms. For example, a worker may be given an increment in points for each time he worked on an artifact in the past. Each of those increments can then be modified, depending on how many other workers worked on that same artifact. In addition, the total increment in points can be further modified according to the worker's current reputation. The finally calculated increment value represents the promised reward. The set of finally calculated rewards per worker $R_w = \{f_{R_1}, ..., f_{R_z}\}$ is then advertised to the workers, influencing their future behavior, and closing the feedback loop. The major difficulty in designing a successful incentive scheme lies in properly choosing the set of *incentive parameters* (performance metrics, incentive mechanisms, and their compositions). Often, the possible effects when using one set of parameters

are unclear at design time, and an experimental or a simulation evaluation is needed to determine them.

The comprehensive model presented so far describes the general incentivization loop. While useful in abstracting the way we can think of and represent the application of incentives in an information system, it is not practically usable, since it abstracts the human actor in the loop in a too generic way. This is why we use this simple model as the starting point for defining two practically useful (actionable) models: In Section 4.2 we define the *Rewarding Model (RMod)* allowing the authority to monitor and *apply* incentive mechanisms described in Chapter 3 upon a workforce model. In Section 4.3 we present a methodology and a *Simulation Model* for simulating the application of incentives and behavioral responses.

4.1.1 Definitions

Now we can define the key terms related with incentive management:

Worker (Agent)
: A human which is the target of incentives.

Authority (Principal)
: The entity engaging the workers for productive working purposes, administering incentives upon them.

Incentive
: Any activity or scheme employed by the authority to stimulate (motivate) increased level of certain work-related activities (e.g., productivity, speed, quality of work, number of participants) or to discourage certain activities (e.g., drop-out rate), before the actual execution of those activities.

Reward
: Any kind of recompense for worthy services rendered or retribution for wrongdoing exerted upon workers during the execution of the activity or after its completion. A reward can be made equivalent of an economic value (money or physical goods), or a social status like prestige, rank, or expertise.

Incentive Mechanism
: A concrete rule for assigning/applying the rewards targeting a specific (group of) workers, based on certain logical, temporal and spatial criteria; A concrete implementation of an incentive for a given application context.

Incentive Element
: An atomic component (construct) in terms of which incentive mechanisms can be expressed (as defined in Section 3.3).

Incentive Scheme
: Combined global effect of the application of a set of incentive mechanisms.

4.2 Rewarding Model

In the comprehensive model presented in the previous section, the authority reads worker's performance metrics and changes in artifact states associated with a worker as inputs for the incentive scheme. However, the model presented there says nothing further about how the authority is able to interpret those inputs and output concrete rewards. In this section, we investigate the authority's internal model for representing the workers, encoding incentive mechanisms and representing rewarding actions. The model is named the *Rewarding Model (RMod)*.

For the authority (principal) the RMod represents the following aspects of a real-world incentive loop:

State
: Represents quantitative state of the both the incentivized socio-technical system as well as the internal business logic state needed for making incentive decisions. This includes global attributes and individual worker attributes representing different worker performance metrics (QoS).

Time
: Expressed as a collection of time-annotated records of past and future worker interactions supporting various time conditions and constraints. The notion of timing is fundamental when dealing with incentives, as worker evaluation in most cases depends on the history of past behavior. Similarly, a reward may be scheduled for a future moment if the performance metrics in the upcoming period meet the expectations.

Structure
: Allows representation and manipulation of various types of relationships among workers. Workers are often stimulated just by being placed into position to collaborate with people they find most comfortable working with. In fact, proper team composition can be vital to a process success, and can often be subject to changes during the process execution. Finally, promotions, as one of the most widely used incentive mechanisms, imply a clear hierarchical change.

The authority employs a group of workers to perform a complex process, consisting of multiple tasks. It is assumed that the complete task lifecycle management (e.g., splitting into subtasks, task descriptions, task assignment, task negotiation and agreement) is under the control of the authority. A worker is assigned a (sub)task to perform in a given time and agrees to be subject of incentive evaluations. Concretely, the authority and the worker agree that the worker may be subject to rewards/penalties in some predetermined cases. Workers can work individually on assigned tasks, in a formalized organization (team, collective, SCU) or relationship with the authority (e.g., be employed, be part of teams, have managers). Authority's entire knowledge on the progress of the task is obtained by periodic *messages* (updates) that it receives from the workers and subsequent

reasoning over that data. The application of rewards to the workers is similarly abstracted as legally-binding messages to the worker.

Task is the basic working unit. Workers are rewarded for working on a particular task within the task's timeframe, although the outcome of the evaluation can also depend on the history of previous contributions. Therefore, the lifetime of a worker is not related to the duration of the task. The authority maintains its own view of the workers and the relations between them in a *community graph*. The nodes in the graph represent the workers, while the edges represent different real-world relationships among the workers. For example, they can represent records of past collaborations, notion of trust[SSD10], dependencies, managerial relations, etc. In addition, each node is described by a set of attributes. The attributes may represent task-specific (short-lived) or permanent records of worker's performance. This is the most general representation possible. However, in practice the model is to be coupled with a real-world socio-technical platform (e.g., *SmartSociety*[1]), so the nodes and relations need to be mapped to entities in that platform (cf. Sec. 5.2.2).

The model assumes an iterative task execution. *Iteration* length is measured in clock ticks. *Clock tick* is the basic unit of time measurement. Worker's progress is read upon iteration expiry so the model can obtain up-to-date the QoS metrics. Iteration is the basic time unit when monitoring and evaluating task execution. Iteration cycle length is tunable to allow better runtime adaptability, as the iteration length can be a significant factor when evaluating results and can affect the performance of the team.

In order to model history of past behavior, as well as scheduling of future performance evaluations and rewarding actions, we include in the model the notions of *timeline* and *event*. The timeline is a time-stamped collection of past and future event records. An **event** is an object encapsulating an executable action and a timestamp. Events are interpreted by the socio-technical platform as instructions or suggestions to the platform itself or particular workers. For example, an event could notify a worker that he needs to increase the QoS level of his service in future iterations, or face penalties. Similarly, it could instruct the platform to dissolve a team, invite new workers, or terminate contracts with the others. Events can be generated by the platform itself or originate from an incentive mechanism in RMod. They can target individual workers or groups of workers, depending on the query that forms part of the action contained in the event object. An event can also target global properties of the system itself.

An event can be in two states: scheduled and past. *Scheduled events* are used to enforce/influence future worker behavior. They contain information to execute performance measurements, evaluations or concrete rewarding actions in a specified moment in the future. Scheduled events can be canceled or re-scheduled when needed. The timestamp can be expressed either in iterations or clock ticks. Time expressed in clock ticks is fixed, whereas time expressed in iterations is automatically recalculated to an appropriate clock tick if the iteration duration is altered. This can be useful in many real-world situations. For example, a Christmas bonuses is to be paid out on a fixed date, while if a project stage is prolonged due to some unexpected events, we would want to reschedule the

[1] `http://cordis.europa.eu/project/rcn/106959_en.html`

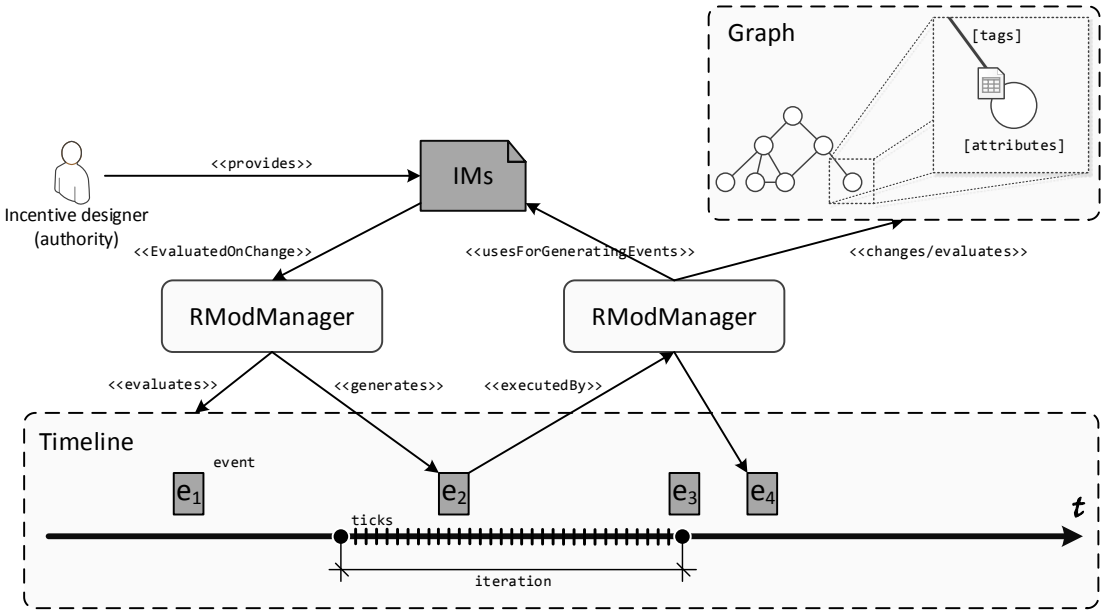

Figure 4.3: Components and interactions in RMod

current iteration and perform the rewarding only at its end. When the time to execute an event is reached, the contained action is executed and the results stored back in the event, which is then archived and put into past state. After that point, the purpose of the **past event** is to serve as a historical reference for future evaluations of workers. An event execution can generate new events, or perform modifications of the team structure and worker attributes. Events are initially generated by executing *rewarding actions*, that are parts of the enforced incentive mechanisms. Figure 4.3 describes a typical working cycle of our RMod. incentive mechanisms (IMs) provide the necessary logic for performing worker evaluations and rewarding actions. At every clock tick IMs get evaluated. Only the IMs that fulfill a logical condition will be triggered to execute. The IM examines the current state of the model, and if a rewarding action needs to be performed produces one or more event objects. The rewarding action contained in the event will include the business (incentive) logic specified in the IM. The events then get stored into the timeline. When the appropriate time comes, the events get executed, modifying the attributes and the graph structure, and possibly spawning new events. The *RModManager* boxes in Figure 4.3 represent the system that implements the functionalities and manipulates RMod.

The RMod allows us to express various basic incentive mechanisms, like:

- *"At the end of iteration, award each contributor who scored better than the average score of his neighbors in that iteration."*
- *"Reward every worker (contributor) who within the last"* `n` *"iterations scored a score"* `t` *"or greater in at least"* `k` *"iterations"* ($\texttt{k} \leq \texttt{n}$).

- *"Assign the person with most check-ins at a place a 'Mayor' badge."*
- *"Unless the productivity increases to a level"* `p` *"within"* `n` *"next iterations, replace team's current manager with the most-trusted of his subordinate workers."*

Furthermore, the model allows for easy composition of different incentive mechanisms, a feature necessary to target different dysfunctional behaviors of workers.

4.2.1 Formal Definition

The RMod contains a set of graph nodes $n \in N$. Each worker $w = (n, Q_w) \in W$ is associated with a set of quantitative attributes Q_w. Quantitative attributes are represented as simple data types (numerics, strings) or collections of them. The complete quantitative state of the model Q is represented by the union of global attribute data and all individual worker attribute sets.

$$Q = \bigcup_i Q(w_i) \cup Q_{global} \tag{4.2}$$

Relationship r is defined as triple (s, t, θ_r), where $s, t \in W$ and $\theta_r \in \Theta$ is the type of the relation. Relation type θ_r belongs to the context-specific set Θ of possible relation types. Relation types represent different notions of relations meaningful to humans, like management, friendship or trust. Union of all relationships is denominated R. The structural component of the model is represented by the community graph $G = (W, R)$. The graph G can be a multigraph, meaning that multiple edges of the same direction between two workers are possible. However, there may be no two relationships of the same type with the same source and target workers:

$$\neg\exists_{r_1, r_2 \in R} \{s(r_1) = s(r_2) \wedge t(r_1) = t(r_2) \wedge \theta_1 = \theta_2\} \tag{4.3}$$

This means that when we fix the type of the edges to θ, the graph containing only edges of that type is again a simple graph (θ*-typed view* of the graph). Depending on the context, we switch between different views in our model. At any time, a single graph transformation[BH02] operates only over a single-typed graph.

In time t_i the state of the model is represented by the triple $s_i = (t_i, G_i, Q_i)$. We mark the current state of the model with s_0. The history of the states of the model, from the moment of t_{start} until the present t_0 is represented by the time-ordered sequence $H = (s_t | t < 0)$.

An action is a function transforming the current quantitative and structural state of the model into a new state $a : (G, Q) \to (G', Q')$. An event is a triple $e = (t, a, p), e \in E$ where t is the scheduled execution time, a is the associated action, and p is the execution priority. At a discrete moment in time t_i a number of event actions may be scheduled to be applied over the current state to transform the state s_i into s_{i+1}.

$$(t_i, G_i, Q_i) \xrightarrow{a_{i_1} \circ a_{i_2} \circ \ldots \circ a_{i_p}} (t_{i+1}, G_{i+1}, Q_{i+1}) \tag{4.4}$$

The order of application is governed by the action priorities. In case of equal priorities, it is assumed that the actions are commutable, meaning that as long as their execution is serialized they can be executed in any order leaving the model in a well-defined state.

Events can have an external origin or can be generated as the result of evaluation of a rewarding action (see below). Events are called past or future depending on the execution time of event actions. Set of all future events is $E_f = \{e|time(e) > t_0\}$.

A rewarding action l is a function reasoning over the collection of past states of the model (H), current state of the model (s_0), and scheduled future events (E_f) to produce new events scheduled for future.

$$\begin{aligned} l : (H, s_0, E_f) &\rightarrow \{e' \in E|time(e') > t_0\} \\ E_f &\rightarrow E_f \cup e' \end{aligned} \tag{4.5}$$

For a worker, a reward represents a change in the values of some of his quantitative parameters (e.g., wage, bonus, rank) and/or the change in the relations the worker is involved in (e.g. promotion, change of the team). Therefore, we first define a worker's *rewarding set*. Rewarding set elements are context-specific.

$$RS_{w_i} = (Q^{\mathcal{R}}_{w_i}, R^{\mathcal{R}}_{w_i}) : Q^{\mathcal{R}}_{w_i} \subseteq Q_{w_i}, R^{\mathcal{R}}_{w_i} \subseteq R_{w_i} \tag{4.6}$$

A *reward* to a worker w_i is then defined as any change of his rewarding set:

$$\mathcal{R}_{w_i} : RS_{w_i} \rightarrow {RS'}_{w_i} \tag{4.7}$$

4.2.2 Prototype Implementation & Evaluation

We present here a prototype implementation of RMod and the system that manipulates it. In the prototype the workers are represented as nodes of an internal graph representation. Each worker is represented a class inherited from `Worker`, such as `SCUWorker` or `HumanProvidedService`. These classes are meant to allow an integration of our model with different underlying socio-technical platforms offering human-provided services. Each `Worker` object contains a set of local attribute data. The attribute types and initial data are user-specified, and provided as inputs. There is also a class responsible for handling system-level quantitative data, named `GlobalAttributes`. A `Relation` contains references to a source and a target `Worker` nodes, and a set of tags that determine its type.

An `Event` encapsulates all the information necessary to perform an evaluation or rewarding. In order to do that, an event must be able to identify which nodes will be the subjects of rewarding and actually perform the rewarding based on the outcome of condition evaluation. It is important to notice that the conditional evaluation may involve reasoning over nodes that do not belong to the target group, e.g., if we want to reward a team manager based on the evaluation of his team members. Similarly, attributes that are used in condition evaluation can be completely different than those altered when rewarding a worker (e.g. we reason over number of units produced to increase the wage). That is why each `Event` has to contain two data sets. In our case we

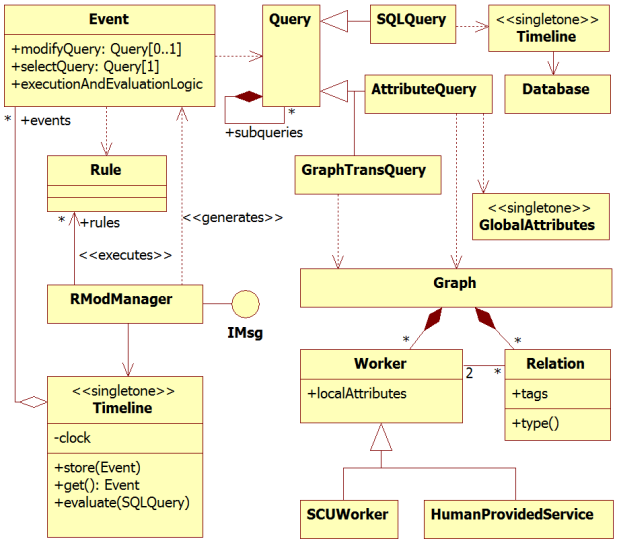

Figure 4.4: Partial UML class diagram of the model prototype.

achieve this by having `Event` contain up to two instances of the `Query` class (`selectQuery` and `modifyQuery` fields). Another reason is due to a property of graph transformation systems.

A `Query` can read current attribute data from `Worker` objects, past data from the database, or current worker structure (relations among nodes). That is why it contains a number of subqueries and a logic in which order to execute them. This logic originates from the rules. A subquery can be: a) SQL query; b) graph transformation query; or c) attribute query. If a query modifies in any way structure or parameters, its outcome is persisted in the database.

The timeline is implemented by using a database instance for storing and querying past events, and a separate in-memory structure for keeping and manipulating future events. Once persisted in the database, the event objects are immutable. The class `Timeline` is a façade for providing the unified view of the timeline in our model. Additionally, it is in charge of producing clock ticks, delimiting iterations and ordering events happening at the same moment.

The `RModManager` is in charge of the overall functioning of the prototype. It provides an interface to the underlying socio-technical platform. The interface is established via XML messages. The `RModManager` translates the incoming messages into appropriate `Event` objects that are executed at the next clock tick, taking precedence over previously stored events. This is the way for the external system to modify the internal model state. On the other hand, all the changes to the state of the model that are caused by internal events are reported to the underlying system using the same message interface. The underlying system is expected to understand the meaning of the messages.

The *rule execution engine* forms part of the `RModManager`. The rule engine is a prototype replacement simulating the working of the fully-fledged incentive mechanism

execution engine. Rewarding rules are provided by the prototype user (i.e., authority). Result of a rule execution is the generation of either a returned/stored value (following a non-modifying evaluation), or a number of `Event` objects stored in `Timeline`. A rule can be marked for evaluation either at each iteration cycle or upon receiving a message coming from the underlying system that changes the internal state of our model. For the prototype evaluation purposed, we only emulate rule execution by feeding pre-generated events into the system. The described class structure is presented in Figure 4.4.

The prototype of the model is capable of encoding simple incentive mechanisms and simulate rewarding actions. The described model and system were implemented[2] in C#, using a Microsoft SQL Server database as storage. Structural modifications are performed through graph transformations[BH02] (graph rewriting). Graph queries are performed using GrGen.NET[JBK10][3].

In order to facilitate performing structural transformations, each instance of `Worker` and `Relation` were mapped to a corresponding node and edge in GrGen's internal representation. For the prototype evaluation pre-compiled graph transformation patterns are used, which are able to capture structural requirements of the incentive mechanisms that needs to be performed. When performing a graph transformation, if more subsets in the graph match the pattern to be replaced (modified) then an arbitrary match is picked as the target subset. To avoid this ambiguity, we first run a GrGen matching query that does not perform any graph transformations. Then our system decides which of the matching subsets should be modified. Those graph elements are additionally marked, and a new, modifying query is issued, that will target exactly the subset(s) we want.

The prototype has the following inputs:

- A one-time specification of the graph model and the initial team configuration provided using GrGen's domain-specific language [JBK10].
- A set of global and individual worker attributes, with initial values, provided at startup.
- A set of rewarding rules that are being evaluated and executed throughout the running time.
- Messages coming from the underlying system that change the internal model state.

The output of the prototype are the messages instructing the underlying socio-technical platform to perform a rewarding action.

Scenario I - Customer Case Solving Teams

Let us consider an incident management system that utilizes SCU [DB11] for solving possible software/hardware defects associated with a distributed IT system. The IT

[2] `https://github.com/tuwiendsg/PRINC`

[3] GrGen is a powerful and versatile system for performing algebraic graph transformations, implementing single and double pushout (SPO and DPO) formalisms. It provides a domain-specific specification language allowing declarative graph pattern matching and rewriting.
URL: `http://www.info.uni-karlsruhe.de/software/grgen/`

system can be represented as a dependency graph of distributed IT components. This scenario is inspired by real situations faced by IBM engineers[4]. Figure 4.5 helps describe the scenario.

Each unit $A, B, \ldots, G$ has an assigned group of maintenance engineers g_i. When device A fails, a case is opened, and a first SCU reaction team is formed, consisting of maintenance engineers assigned to A, i.e., $\{g_1, g_2\}$. SCU_1 team performs the maintenance work on A during a time period. The team members are paid for taking part in a customer case according to the pay-rate RM_1. If the team is unable to fix the component in a predefined time, additional engineers, assigned to devices that depend on A are included to the team: $SCU_2 = SCU_1 \cup assignedTo(E, F)$. The new engineers are paid according to the pay-rate RM_2. Similarly, a third layer of engineers can be activated ($SCU_3 = SCU_2 \cup assignedTo(B, C, D)$), with the pay-rate RM_3.

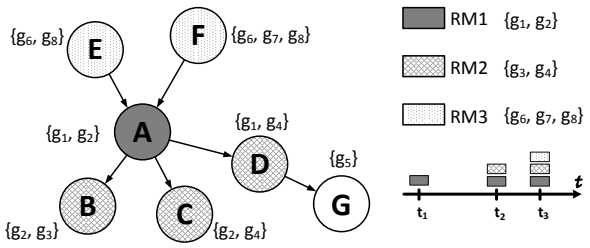

Figure 4.5: Composing rewarding mechanisms in an IT incident management system.

In the rules, we have the following graph patterns specified: `LAYER0(Node N)` returning engineers assigned to that node; `LAYER1(Node N)` returning engineers of the dependent nodes; `LAYER2(Node N)` returning engineers of the nodes `N` depends of. The rule `ASSIGN(Node, Workers)` assigns the engineers to a case, notifies them, and generates a new event that is meant to perform a payment to the assigned engineers. The rule `PROGRESS(Case)` evaluates the quantitative data obtained from engineers to determine the completion percentage of the case. The rules `PAY_RM1(Node)` and `PAY_RM2(Node)` perform the actual payments according to the pay-rates RM_1 and RM_2, respectively.

When an IT case is opened, a notification is received and iteration `I1` is initialized. Let the iteration be initialized to the average case duration for that particular component type, e.g., `END(I1) = 100`, where `END(Iteration)` is a built-in primitive to automatically map time values from iterations to clock ticks.

```
->MSG: CASE:C765; COMPONENT:A; SCU:g1,g2; ITERATION:I1,100;
```

An initialization rule immediately schedules the following events:

If at `t = 80` we detect that the case is not progressing as expected, the new set of engineers is assigned to the case and the iteration is prolonged to the 120% of the initial value. That also triggers a generation of the new event `EvtID = 20`. At this point, events 18 and 20 are both scheduled to happen at `t = 120`, effectively composing the two rewarding mechanisms `RM1` and `RM2`.

[4]The author would like to thank Dr. Kamal Bhattacharya, at the time from IBM Research–India for sharing the data for the incident management scenario.

```
EvtID = 18; t = END(I1);
    {if (PROGRESS(C76) == 1) PAY_RM1(A);}
EvtID = 19; t = 0.8 * END(I1);
    {if (PROGRESS(C76) < 0.8) {
        ASSIGN(C76,LAYER1(A)); END(I1) = 1.2 * END(I1);}}
```

```
    {if (PROGRESS(C76) == 1) PAY_RM2(A);
    else ASSIGN(C76, LAYER2(A)); }
    /* etc... */
```

This scenario shows how the RMod prototype composes different incentive mechanisms. In this case, the focus is placed on evaluation and composition. The only changes performed are the changes in the domain of quantitative data (different pays).

Scenario II - Rotating Presidency

A number of workers (employees) is split in teams, led by a single team leader (or manager). The manager post is not permanent, but instead, upon each iteration the best performing team member up to then becomes the team manager for the upcoming period, while the current manager reverts to a regular team member. A single employee cannot hold the leader position for more than two consecutive turns. This incentive mechanism is intended to increase competitiveness and have the team run by the most appreciated member.

We assume that there is a quantitative measure of worker's quality, e.g., an average of peer-voted scores received so far. Votes describing worker performance are fed via input messages a couple of times within a single iteration and stored in the database. The contents of the messages can be simulated to favor particular workers. An example of an incoming message looks like this:

```
->MSG: TASK:T843; TEAM:T5; WORKER:75; SCORE:96;
```

We have the following graph matching patterns defined: `TEAM(Team)` returning a collection of `Workers` assigned to the team provided as input parameter; and `MANAGER(Worker[])` returning the `Worker` who is the manager of the team. We also have one graph transformation pattern defined: `SET_MANAGER(Team, Worker)` performing the "re-chaining" of relations in the graph to point to the new team manager. In addition, we have an evaluation rule: `BEST_OF(Worker[])` returning the best-scored `Worker` in the collection provided as input parameter. The scores are calculated by averaging values read from the database. `SCHD_EVT(Time, Rule)` is the built-in primitive for scheduling an event and integrating into it the logic contained in the provided `Rule`. The primitives (with variable number of arguments) `DB_READ(String, Time,...)` and `DB_WRITE(String, Time,...)` execute predefined SQL queries.

At the beginning of each new iteration (e.g., `I3`) an event that performs an evaluation at the iteration's end (e.g., `EvtID = 47`) is scheduled. The event contains the logic to perform evaluation and a possible manager change.

```
EvtID = 47; t = END(I3); {
    Worker currMgr = MANAGER(TEAM(T5));
    Worker bestWrk = BEST_OF(TEAM(T5));
    if (currMgr != bestWrk)
      SCHD_EVT(START(I4), SET_MANAGER(T5, bestWrk));
    else
      if (DB_READ('manager', I3, T5, currMgr) == 1)
        SCHD_EVT(START(I4), SET_MANAGER(T5, BEST_OF(TEAM(T5) - currMgr)); //
            replace with 2nd best
      else DB_WRITE('manager', I4, T5, currMgr);
}
```

A combined DB-graph query is then run that considers the history of all previous evaluations for all the workers in the team, and the previous positions the worker held. If the current leader fulfills the two criteria, it can remain the leader for another term. If not, it is replaced by the second best candidate from his team. The information about positional change (or not) is then stored in the database for future evaluations.

Although simple, this example allows us to test modeling of evaluations based on historical quantitative data (performance votes) and current structural data (the fact that a worker is a leader is determined through graph relations). Compared to Scenario I we perform not only quantitative, but also structural changes.

4.3 Simulation Model

The Rewarding Model (RMod) presented in the previous section represented the authority's simplified view on the workers in a socio-technical platform. The RMod is designed to be generic and simple enough to be able to encode most incentive conditions and rewarding actions described in Chapter 3. However, referring back to Figure 4.2 in Section 4.1, we see that in order to close the incentive loop we need also to be able to model and simulate the behavioral responses of workers to the applied incentives. Those responses are result of the complex traits of human nature, scenario-specific working environment and social characteristics of the worker community. Therefore, in order to model and simulate behavioral responses, a context-specific model emphasizing selected behavioral traits needs to be developed each time anew. In this section, we present a *simulation methodology* for developing scenario-specific *simulation models* based on *agent-based social simulation*.

As shown in [Feh13], most incentive mechanisms are developed based on empirical data obtained from different studies. However, the empirical findings are often context-specific, and when applied in different environments may yield different behavioral responses. This is especially true for incentive models that need to consider social characteristics of the worker community, such as coordinated group actions (e.g., worker resistance [Mum05], informal forum-based worker coordination [MHOG14] , social/regional/ethnic peculiarities, voluntary work [HS96], importance of reputation/flaunting [SKM04], or web-scale malicious behavior [Weg12]. An additional complication is that these phenomena

change often and characterize different subsets of the crowd differently in different moments. This makes development of appropriate mathematical incentive models difficult.

The major problem an incentive designer is faced with in this case is how to evaluate the developed incentive mechanisms aimed at targeting such disruptive or dysfunctional behaviors. The designer needs to consider factors, such as: emerging, unexpected and malicious worker behavior, incentive applicability, range of stability, reward fairness, expected costs, reward values and timing. Failing to do so leads to exploding costs and work overload, as the system cannot scale with the extent of user participation typical of social computing environments. Unbalanced rewards keep new members from joining or cause established members to feel unappreciated and leave. Ill-conceived incentives allow users to game the system, prove ineffective against vandalism, or assign too many privileges to particular members tempting them to abuse their power.

In this section we present a methodology for incentive designers for quickly selecting, composing and customizing existing, real-world atomic incentive mechanisms, and roughly predicting the effects of their composition in dynamic social-computing environments. The model and simulation parameters can be changed dynamically, allowing quick testing of different incentive setups and behavioral responses at low cost. Specifically, we employ principles of agent-based *social simulation* [MN09, GT05a], an effective and inexpensive scientific method for investigating behavioral responses of large sets of human subjects. In theory, social simulation approaches (such as ours) allow modeling of incentives and responses of workers of arbitrary complexity. In practice, the social phenomena listed above as impeding factors for the development of comprehensive mathematical incentive models pose at the same time big obstacles for developing comprehensive simulation models, requiring development of complex agent behavioral models. Nonetheless, as discussed in [GT05a, Vas12], the simulation approaches are a viable alternative to testing various behavioral responses in real communities when this is impossible due to time, cost or ethical reasons. All three limitations are especially accentuated when testing incentive effects and their different combinations. In this case, speed is preferred over accuracy and ethical considerations are an important feasibility factor. The simulation approach is therefore the method of choice in this case, offering fast experimental setups and circumventing ethical issues.

Social simulation originated in computational social sciences to explore theoretical ideas in the context of synthetic populations. Recently, this has been applied to crowdsourcing, in order to generalize results which otherwise would be tied to a particular situation [BFGK13]. However, unlike the usual approach where agents interact directly (and thus benefit from cooperative behavior or suffer from defective behavior), we introduce a provider that facilitates interactions and determines the benefits or costs of those interactions.

4.3.1 Example Scenarios

In order to better describe the methodology and subsequently evaluate it, we first present two exemplifying scenarios based on real-world crowdsourcing applications.

Citizen-driven traffic reporting

Local governments have a responsibility to provide timely information on road travel conditions. This involves spending considerable resources on managing information sources as well as maintaining communication channels with the public. Encouraging citizens to share information on road damages, accidents, rockfalls, or flooding reduces these costs while providing better geographical coverage and more up to date information[5]. Such crowdsourcing process, however, poses data quality related challenges in terms of assessing data correctness, completeness, relevance, and duplication.

Crowdsourced software testing

Traditional software testing is a lengthy and expensive process involving teams of dedicated engineers. Software companies[6] may decide to partially crowdsource this process to cut time and costs and increase the number and accuracy of detected defects. This involves letting the remote testers detect bugs in different software modules and usage environments and submitting bug reports. Testers with different reputations provide reports of varying quality and change the assigned bug severity. As single bugs can be reported multiple times in separate reports, testers can also declare two reports as duplicates.

The two scenarios exhibit great similarities. The expected savings in time and money can in both cases be outweighed by an incorrect setup and application of incentive mechanisms. Furthermore, the system could suffer from high numbers of purposely incorrect or inaccurate bug report submissions, driving the processing costs high. For the purpose of this paper, we join and generalize the two scenarios into a single, abstract one that we will use in our simulation setup:

The *Authority* seeks to lower the time and cost of processing a large number of *Reports* on various *Situations* occurring in the interest domain of the Authority. The *Workers* are independent agents, occasionally and irregularly engaging with the system managed by the Authority to perform one of the following *Actions*: *Submit* a new Report on a Situation, *Improve* an existing Report, *Rate* the accuracy and importance of an existing Report, inform the Authority of his belief that two existing Reports should be considered *Duplicates*. The Worker actions are driven by the combination of the following factors: a) possibility to earn *Points* (translating to increased chances of exchanging them for money); b) possibility to earn *Reputation* (translating to a higher status in the community); and c) the intrinsic property of people to contribute and help or to behave maliciously. In order to influence and (de-)motivate workers, the Authority employs a number of *Incentive Mechanisms*, collectively referred to as *Incentive Scheme*.

This scenario also needs to address the following challenges:

- *Crowdsourced report assessment.* The effort required for manual validation of worker-provided reports may easily outweigh the gained effort and cost reduction

[5]For a real world example visit the Aberdeen City Council's SmartJourney initiative at: `http://smartjourney.co.uk/`

[6]For example, www.utest.com

from crowdsourced reporting in the first place. Hence, workers need to be properly stimulated to supplement and enrich existing reports as well as vote on their importance, thereby lifting the verification burden off the authority. The system also needs to strike a balance not to collect too much information.

- *Worker reputation (trust).* A worker's reputation serves as one potential indicator for data reliability, assuming that reputable workers are likely to provide mostly accurate information. Subsequently, reports from workers with unknown or low reputation need to undergo more thorough peer assessment. The system must support continuous adjustment of workers' reputation.

- *Adjustable and composable incentive scheme.* An effective incentive scheme needs to consider all past citizen actions, the current state of a report and the predicted costs of processing a report manually in order to decide whether and how to stimulate workers to provide additional information. It also needs to correctly identify and punish undesirable and selfish behavior (e.g., false information, deliberate duplication of reports, intentional up/downgrading of reports).

The resulting complexity arising from the possible combination and configuration of worker behavior, incentive schemes, and processing costs requires a detailed analysis to identify a stable and predictable system configuration and its boundaries.

4.3.2 Modeling and Simulation Methodology

Our methodology for simulating worker participation and incentive mechanisms in crowdsourcing processes is depicted in Figure 4.6. It consists of four basic steps, usually performed in multiple iterations:

(i) defining a domain-specific meta-model by extending a core meta-model;
(ii) capturing worker's behavioral/participation patterns and reward calculation into an executable model;
(iii) defining scenarios, assumptions, and configurations for individual simulation runs; and
(iv) evaluating and interpreting simulation results.

These steps are described in more detail below.

We use the DomainPro.[7] modeling and simulation tool suite in each of the outlined methodology steps to design and instantiate executable models of incentive mechanisms and run simulations of those models. The tool allows creating custom simulation languages through metamodeling and supports agent-based and discrete event simulation semantics (see [DEM12]). However, the overall approach is generic and can be easily applied using a different modeling and simulation environment.

[7] Tool available at request from:
`www.quandarypeak.com`

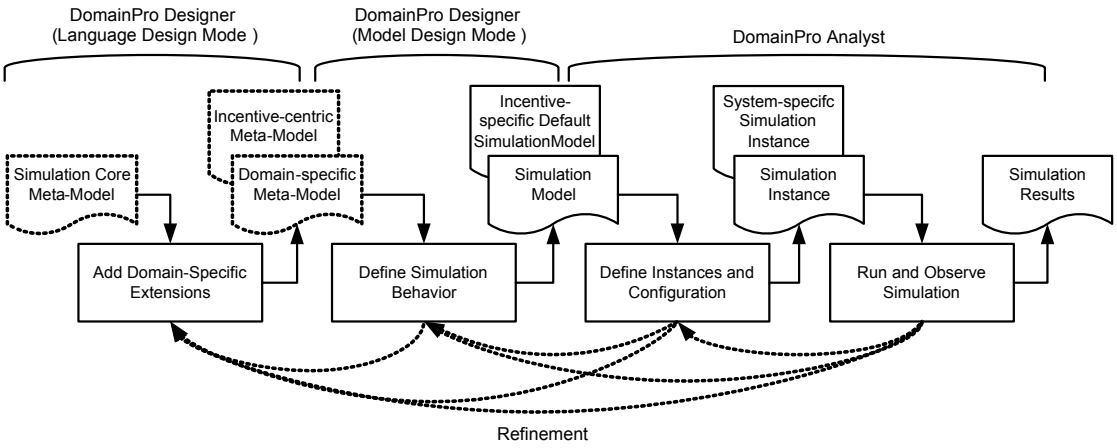

Figure 4.6: The methodology of simulation design and development.

The simulation core meta-model is implemented in the DomainPro Modeling Language. Optional extensions result in a domain-specific meta-model that defines which component types, connector types, configuration parameters, and links a simulation model may exhibit. In our case, we extend the core meta-model to obtain what we refer to as *incentive-centric meta-model* (Section 4.3.2). The obtained incentive-centric meta-model serves as the basis for defining the simulation behavior, i.e., the *executable simulation model* (Section 4.3.3). Obtaining the executable simulation model requires defining workers' behavioral parameters, authority's business logic (including incentive mechanisms and cost metrics), the environment and the control flow conditions between them. Finally, prior to each execution, the executable simulation model requires a quick runtime configuration in terms of the number of worker instances and monitored performance metrics (Section 4.3.4). During the execution, we do near real-time monitoring of metrics, and if necessary, perform simulation stepping and premature termination of the simulation run to execute model refinements.

The tool we use enables refinement at any modeling phase. A designer will typically start with simple meta- and simulation models to explore the basic system behavior. She will subsequently refine the meta-model to add, for example, configuration parameters and extend the functionality at the modeling level. This enables testing simple incentive mechanisms first, and then extending and composing them once their idiosyncrasies are well understood.

Incentive-Centric Meta-Model

The derived meta-model (Fig.4.7) reflects the conceptual view of incentive mechanisms as presented in Section 4.1. A *ParticipationPattern* consists of *Actors* (*User*s or *Providers*) and the *InteractionObjects*. Actors exhibit *Behavior* that encapsulates different *UserActivities*. InteractionObjects contain *ObjActivities* that define allowed and reward-yielding activities on an InteractionObject. *InternalSequences*, *ExternalSequence*, and *Object-*

Sequence determine the control flow among activities by specifying trigger conditions. The *EnvGenerator* drives the simulation by controlling the generation of interaction objects (artifacts) for the *Workers* to act upon, and for the *Authority* to check and further process. The *AtomicData* within a *SimulationElementType* defines which data may be passed between UserActivities and/or ObjActivities when an *InternalSequence*, *ExternalSequence*, or *ObjectSequence* fires. While arbitrary data types can be passed along, only *AtomicData* of type int, double, long, or boolean may be used as observable metrics during simulation execution. The exact applicable metrics are defined later on, on the simulation instance level.

As previously outlined, the iterative nature of the modeling process usually requires extending the core meta-model with domain-specific elements, as the need for them is identified. The domain-specific extensions we introduce are highlighted in bold/blue in Figure 4.7.

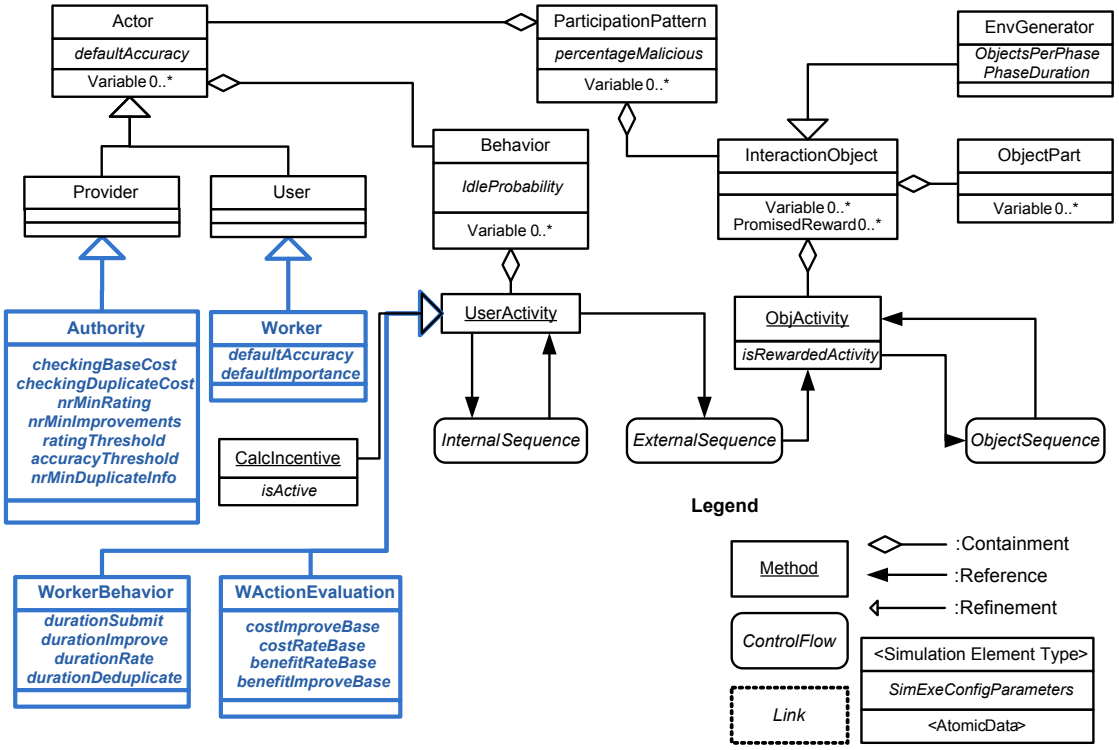

Figure 4.7: Simulation meta-model including domain specific extensions in bold/blue.

4.3.3 Simulation Model of the Real-World Scenario

In this section we derive an executable simulation model for evaluating the impact of various design decisions taken during the modeling of the case-study scenarios from Section 4.3.1. Specifically, the goals of the simulation are:

(i) prototyping and evaluating various incentive schemes;

(ii) determining the impact of malicious user behavior;

(iii) observing trends in processing costs, reward payments, report accuracy, and user activities.

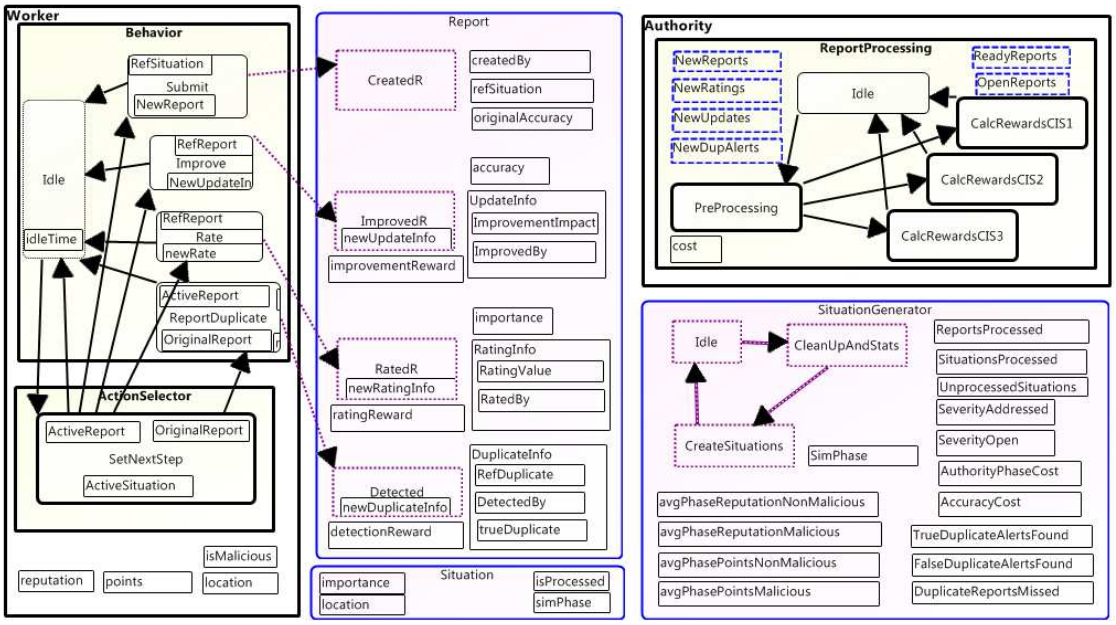

Figure 4.8: Partial screenshot of the implemented case-study simulation model in DomainPro Designer.

Figure 4.8 provides a partial screenshot of the case-study simulation model.

The simulation model comprises over 40 simulation parameters, determining various factors, such as: distribution of various personality characteristics in the worker population, injected worker roles (e.g., malicious, lazy), base costs for the authority, selection and composition of incentive mechanisms. Describing them all in detail/formally here would not be practical. Instead, the annotated source code of the model and the metamodel is provided here:[8]. The rest of this section is written in a narrative style.

Location and importance characterize a *Situation*. Situations can be generated with user-determined time, location and importance distributions, allowing us to concentrate more problematic (important) situations around a predefined location in selected time intervals, if needed. For the purpose of this paper, we generate situations with uniform probability across all the three dimensions. The *SituationGenerator* contains the activities for creating new situations and calculating phase-specific simulation metrics on cost, reputation, points, actions, and importance across reports, situations and workers.

[8] `http://tinyurl.com/incentives-sim-model`

The *Worker*'s *SetNextStep* activity represents the implementation of the worker's decision-making function f_a, introduced in Section 4.1. As previously explained, the worker here considers the next action to perform based on:

1. internal state (e.g., *location*), including innate, population-distributed or arbitrarily set personality characteristics (e.g, *laziness*, *isMalicious*);
2. current performance metrics (e.g., *reputation*, *points*);
3. advertised rewards (*detectionReward*, *ratingReward*, *improvementReward*).

Worker's location determines his/her proximity to a situation, and, thus, the likelihood to detect or act upon that situation (the smaller the distance, the higher the probability). However, two workers at the same distance from a situation will not equally likely act upon it. This depends on their personality, past behavior, and the number of points they currently have. Default behavior of workers is produced by normally distributing values of certain S state metrics, thus determining the "personality" of the worker. In this case, the likelihood to act upon a situation. However, different worker behaviors and personalities are obtainable through different *roles*. The simulation model does not prescribe the complexity of the roles. Instead, the incentive designer is free to implement them as necessary to simulated collective, disruptive of malicious behavior.

Points and *reputation* are the principal two metrics by which the authority assesses Workers in our scenario. In principle, points are used by the Authority as the main factor to stimulate activity of a Worker. The more points, the less likely will a worker idle. On the other hand, a higher reputation implies that the Worker will more likely produce artifacts of higher quality. Each new worker joining the system starts with the same default point and reputation values. Precisely how the two metrics are interpreted and changed thereafter depends on the incentive mechanisms used (see below).

As we are primarily interested in investigating how reputation affects (malicious) behavior, we characterize each agent by reputation metric, as laboratory experiments confirmed that reputation promotes desirable behavior in a variety of different experimental settings [WM00, MSK02, RM, SKM04].

The four *Behavior* activities produce the respective artifacts – *Reports*, *UpdateInfos*, *RatingInfos* and *DuplicateInfos*. Worker's internal state determines the deviations of accuracy, importance, improvement effect, and rating value of the newly created artifacts. The subsequently triggered *Report*-located activities (*CreatedR*, *ImprovedR*, *RatedR* and *Detected*) determine the worker action's effect on the two metrics that represent the artifact's state and data quality metrics at the same time – report *accuracy* and *importance*. We use Bayes estimation to tackle the cold-start assessment of report accuracy and importance, taking into account average values of existing reports and the reputation of the worker itself.

The produced artifacts are queued at the *Authority* side for batch processing. In *PreProcessing* activity we determine whether a Report is ready for being processed. This depends on the report's quality metrics, which in turn depend on the amount and value of worker-provided inputs.

Processing reports causes costs for the Authority. The primary cost factors are low quality reports and undetected duplicate reports. Secondary costs arise when workers focus their actions on unimportant reports while ignoring more important ones. Therefore, the Authority incentivizes the workers to submit required amounts of quality artifacts. As noted in Section 4.3.1, gathering as much inexpensive data from the crowd as possible was the original reason for the introduction of a crowdsourced process in the first place.

Our proof-of-concept simulation model for the given scenario defines three basic incentive mechanisms:

- IM_1**:** Users are assigned fixed amounts of points per action, independent of the artifact. Submitting yields most points.
- IM_2**:** The amount of points is increased before assignment, depending on the current quality metrics of the report. E.g., the fewer ratings or improvements the higher the increment in points.
- IM_3**:** Users are assigned a reputation. The reputation rises with accurately submitted reports, useful report improvements, correctly rated importance and correctly flagged duplicates.

In Section 4.3.4, we can compose these three mechanisms in different ways to produce different incentive schemes which we can run and compare.

For demonstration purposes we define only a single additional role - that of a malicious worker. Malicious worker behavior is designed to cause maximum cost for the Authority. To this end, we assume malicious workers to have a good perception of the actual situation characteristics. Hence, upon submission they will set initial report importance low and provide very inaccurate information subsequently. For important existing reports they submit negative improvements (i.e., conflicting or irrelevant information) and rate them low and while doing the opposite for unimportant reports.

4.3.4 Evaluation

For evaluating our approach, we keep using the case-study scenario from the previous sections and perform a set of experiments on it. All provided experimental data is averaged from multiple, identically configured simulation runs. Details on the experiment setup are followed by experiment result[9] presentation and gained insights.

Experiment Setup

Timing Aspects. We control the pace of the simulation by determining the amount of situations created per phase. Taking a reading of all relevant (i.e., experiment-specific) metrics at the end of each phase provides an insight on how these metrics change over time. All our simulations last for 250 time units (t), consisting of 10 phases of $25t$ each. Batch creation of situations is representative for real world environments such as bugs

[9]Data available here: `http://tinyurl.com/scekic-dorn-dustdar-coopis13`

that typically emerge upon a major software release or spikes in traffic impediments coinciding with sudden weather changes. Report submission takes $5t$, while improving, rating, and duplication flagging require only $1t$. The exact values are irrelevant as we only need to express the fact that reporting requires considerably more time than the other actions. Processing of worker-provided data on the provider side occurs every 1t. Note here, that for the purpose of the case study, we are only interested in the generic processing costs rather than the time it takes to process that data. Each report is assumed to cause 10 cost units for minimum quality (modeled as value of 0), and almost no cost when quality (through worker-provided improvements) approaches maximum (= 1). The higher the quality of received reports, the less reports are needed to persuade the principal to act (see below).

Scenario-specific thresholds. As we aim for high-quality data and significant crowd-base confirmation, the following thresholds need to be met before a report is considered for processing: at least three updates and high accuracy (> 0.75); or five ratings and medium importance (> 0.5); or four duplication alerts; or being reported by a worker of high reputation (> 0.8) and having high importance (> 0.7). Workers obtain various amounts of points for (correct) actions, the amount depending on the value of the action to the provider and the incentive scheme used.

Worker Behavior Configuration. A worker's base behavior is defined as 70% probability idling for $1t$, 20% submitting or duplication reporting, and 10% rating or improving. Obtained points and reputation increase the likelihood to engage in an action rather than idle. The base behavior represents rather active workers. We deliberately simulate only the top-k most involved workers in a community as these have most impact on benefits as well as on costs. Unless noted otherwise, $k = 100$ for all experiments.

Composite Incentive Schemes. The experiments utilize one or more of the following three *Composite Incentive Schemes – CIS*, introduced in Section 4.3.3:

- $CIS1 = IM_1$
- $CIS2 = IM_2 \circ IM_1 = IM_2(IM_1)$
- $CIS3 = CIS2 + IM_3 = IM_2 \circ IM_1 + IM_3$

CIS1 promises and pays a stable amount of points for all actions. CIS2 dynamically adjusts assigned points based on the currently available worker-provided data, but at least as high rewards as CIS1. CIS3 additionally introduces reputation calculation.

Experiments

Experiment 1: Comparing Composite Incentive Schemes.
Here we compare the impact of CIS1, CIS2, and CIS3 on costs, assigned rewards, report accuracy, and timely processing. Figure 4.9 displays incurred costs across the simulation duration. All three schemes prove suitable as they allow 100 workers to provide sufficient data to have 20 situations processed at equally high accuracy. They differ, however,

significantly in cost development (Fig.4.9 inset), primarily caused by undetected duplicate reports (on average 0.2, 0.25, and 0.4 duplicates per report per phase for CIS1, CIS2, and CIS3, respectively). CIS1 yields stable and overall lowest costs as the points paid induce just the right level of activity to avoid workers getting too active and thus causing duplicates. This is exactly the shortcoming of CIS2 which overpays workers that subsequently become overly active. CIS3 pays even more, and additionally encourages worker activity through reputation. The cost fluctuations are caused by the unpredictable number of duplicates (however remaining within bounds). Although more costly and less stable, CIS3 is able to identify and subsequently mitigate malicious workers (see Experiment 3 below).

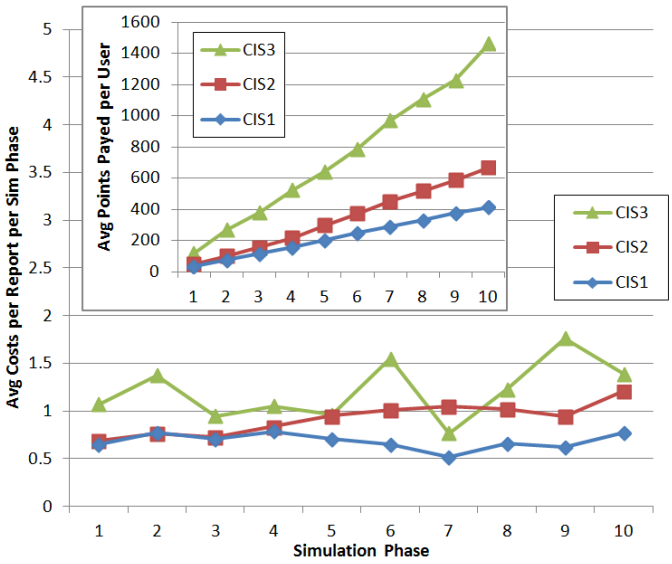

Figure 4.9: Incurred report processing costs for CIS1, CIS2, and CIS3. Inset: average paid points per worker.

Experiment 2: the Effect of Worker/Situation mismatch.
Here we analyze the effects of having too few or too many workers per situation. In particular, we observe per phase: the cost, points assigned, report importance (as reflecting situation importance), and reputation when:

(i) the active core community shrinks to 20 workers while encountering 50 situations (20u/50s);

(ii) a balance of workers and situations (100u/25s);

(iii) many active workers but only a few situations (100u/5s).

A surplus in situations (20u/50s) causes workers to become highly engaged, resulting in rapid reputation rise (Fig 4.11 bottom) coupled with extremely high values of accumulated

rewarding points (Fig 4.10 inset). Costs per report remain low as duplicates become less likely with many situations to select from (0.18 duplicates per report). Here, CIS3 promises more reward for already highly-rated reports to counteract the expected inability to obtain sufficient worker input for all situation (on average 22 reports per phase out of 50). Subsequently, the authority receives correct ratings for reports and can focus on processing the most important ones. Compare the importance of addressed situations in Figure 4.11 top. A surplus in active workers (100u/5s) suffers from the inverse effect. As there is little to do, reputation and rewards grow very slowly. Perceiving little benefit, workers may potentially leave while the authority has a difficult time distinguishing between malicious and non malicious workers. Configurations (100u/5s) and (100u/25s) manage to provide reports for all situations, therefore having average report importance remaining near 0.5, the average importance assigned across situations.

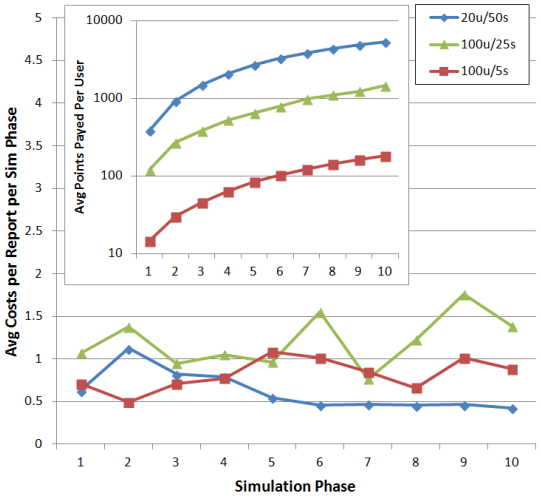

Figure 4.10: Costs per report incurred at various combinations of worker and situation count.

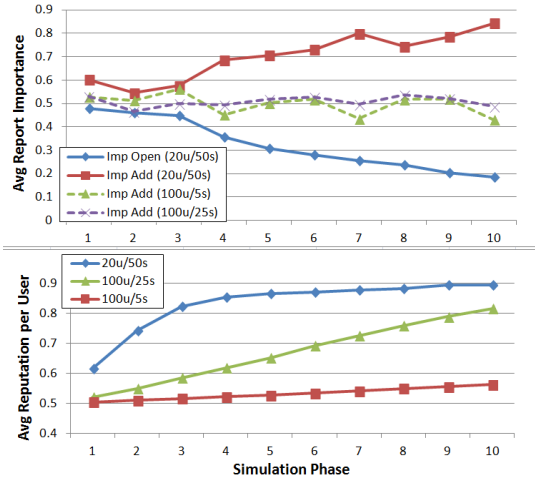

Figure 4.11: Reputation acquired by workers (bottom), and report **imp**ortance **add**ressed, respectively remaining **open** (top).

Experiment 3: Effect of Malicious Workers.
Here we evaluate the effects of an increasing amount of malicious workers on cost when applying CIS3. Figures 4.12 and 4.13 detail cost and reputation for 0%, 20%, 30%, 40%, and 50% malicious workers. All workers are considered of equal, medium reputation 0.5 upon simulation start. The drop in costs across time (observed for all configurations) highlights that the mechanism indeed learns to distinguish between regular, trustworthy workers and malicious workers. The irregular occurrence of undetected duplicates cause the fluctuations in cost apparent for 0% and 20% malicious workers. Beyond that, however, costs are primarily determined by low accuracy induced by malicious workers. CIS3 appears to work acceptably well up to 20% malicious workers. Beyond this threshold harsher reputation penalties and worker blocking (when dropping below a

certain reputation value) need to be put in place. In severe cases lowering the default reputation assessment might be applicable but requires consideration of side effects (i.e., thereby increasing the entry barrier for new workers).

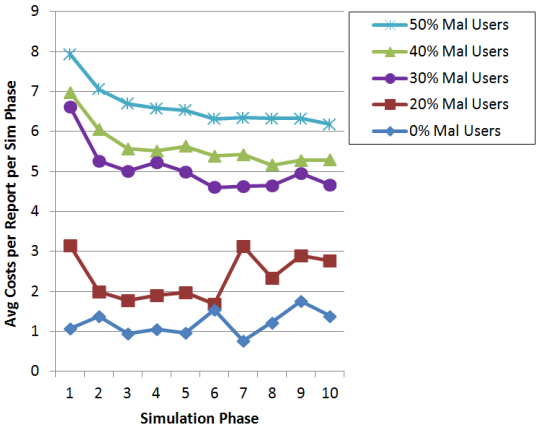

Figure 4.12: Costs per report incurred due to various level of malicious workers.

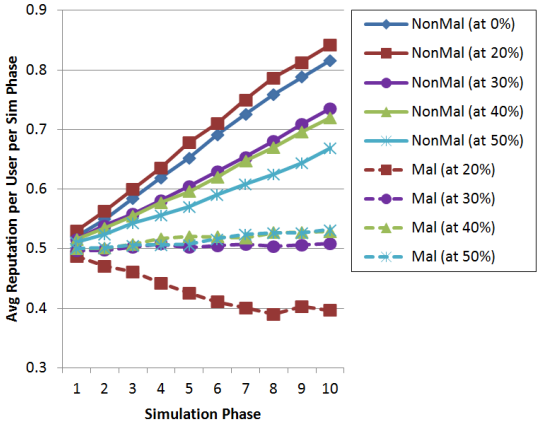

Figure 4.13: Average reputation acquired by malicious and non-malicious workers.

Limitations and Discussion

Simulations of complex socio-technical processes such as the use-case presented here can only cover particular aspects of interest, never all details. Thus any results in terms of absolute numbers are unsuitable to be applied directly in a real-world systems. Instead, the simulation enables incentive scheme engineers to compare the impact of different design decisions and decide what trade-offs need to be made. The simulation outcome provides an understanding what mechanisms might fail earlier, which strategies behave more predictably, and which configurations result in a more robust system design.

In particular, the presented comparison of CISs in Experiment 1 gives insight into the impact of overpaying as well as indicating that the CIS3 would do well to additionally include a mechanism to limit submissions and better reward the action of flagging the duplicates. Experiment 2 provides insights on the effect of having too few or too many workers for a given number of situations. It highlights the need to adjust rewards and reputation in reaction to shifts in the environment and/or worker community structure. Experiment 3 provides insight into the cost development in the presence of malicious worker and highlights the potential for mechanism extension.

Part II

Supporting Automated Incentive Management in Social Computing

CHAPTER 5

Executive Framework for Incentive Management

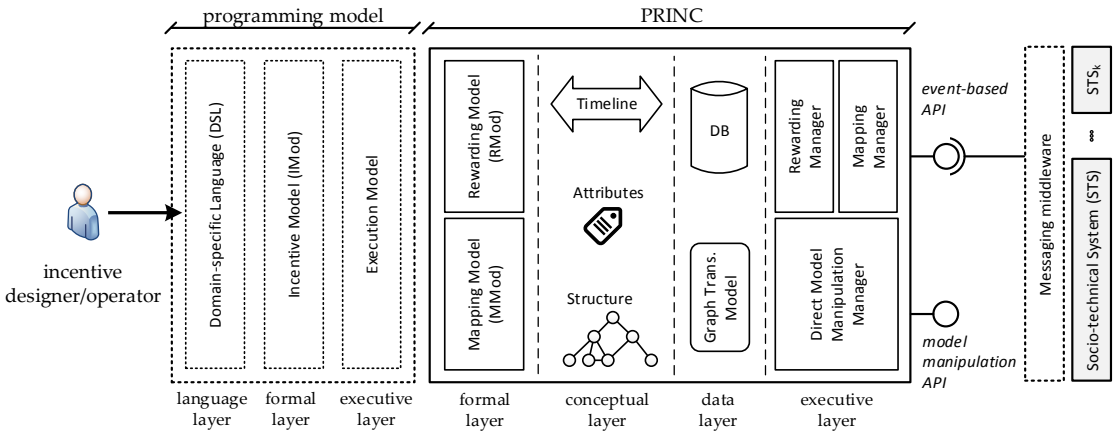

Figure 5.1: The incentive management platform. The PRINC *Framework*, presented in this chapter, is shown fully outlined. The remaining tools (dashed-outlined) are presented in subsequent chapters.

In Chapter 4 we introduced a *Rewarding Model (RMod)* capable of encoding incentives in form of scheduled application of rewarding actions applied over an abstract model. We then showed through simulation that the model was capable of detecting individuals with disfynctional behavior and reducing their influence. In this chapter we present the design of the executive framework making use of the introduced *RMod* and allowing application of rewarding actions on real-world socio-technical systems.

5.1 Usage Context

Figure 5.1 shows the architecture and the intended usage of this framework, named PRINC (standing for **PR**ogrammable **INC**entives). To better explain the intended usage PRINC is shown here together with the remaining incentive management tools presented in this thesis.

The incentive management platform shown in the figure is intended to be used by two types of *users*:

(a) *incentive designers* – domain experts that design and implement incentive scheme for an organization

(b) *incentive operators* – organization members responsible for managing the every-day running and adaptation of the scheme.

An incentive designer (*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 [Feh13, Del15].

The role of an incentive operator (*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.

To exemplify the expected type of functionality an Operator would be performing, let us assume the existence of a socio-technical platform offering the service a crowdsourced software development to its customers. The Operator's role would be to monitor the efficacy of incentive schemes in use and adjust them when needed. For example, the operator could learn that teams in which testers were incentivized to report more bugs throughout the entire development process resulted worse than those incentivized to report (less, but) more severe/dangerous ones in more mature product phases. Based on this experience, the Operator can adjust the scheme (e.g., bug thresholds and onus amounts) to put more emphasis on quality rather than on quantity as soon as the product has entered a fairly stable stage.

Both the Designer and the Operator can use the simulation modeling methodology introduced in Section 4.3 to aid the design, composition and adjustment of the incentive scheme. Operators in particular can benefit from the speed that the social simulation offers (compared to the conventional incentive mechanism design) when adaptations of the incentive scheme(e.g., parameter variations, turning on/off additional mechanisms) are necessary to counteract disruptive or newly-emerging dysfunctional behavior.

We will further showcase the distinction between the two roles in Chapter 7. At this point it suffices to stress that both Designer and Operator use a platform-independent,

largely declarative domain-specific language to encode/adjust incentive schemes that are provided as input to the incentive management platform. The schemes are then automatically translated to executable (imperative) code interacting with PRINC. This is done by scheduling a number of future rewarding actions to be executed on RMod, modifying the internal state of RMod, which is then propagated to the external system through a messaging middleware. At the same time, the state of RMod is updated via events originating from the external system.

5.2 Internal Architecture

The Rewarding Model (RMod) lies at the heart of PRINC framework and encodes the imperative, system-specific version of the incentive scheme. It constantly mirrors the state of the external system and executes incentive mechanisms on it. The *Mapping Model* (*MMod*) defines the mappings needed to properly interpret the system-independent version of the strategy in the context of a specific social computing platform (*external system*). The mapping itself is performed by the *Mapping Manager*.

The *Rewarding Manager* implements the RMod, performs and interleaves all event-based operations on RMod and ensures its consistency and integrity (e.g., by rejecting disallowed structural modifications or preventing modification of the records of past behavior).

The *Direct Model Manipulation Manager* (*D3M*) provides direct RMod manipulation functionalities without relying on the event mechanism and without enforcing any consistency checks. The direct access to the RMod is needed for offering the necessary functionalities internally within PRINC, but also to allow more efficient monitoring and testing. D3M is therefore used to load initial state of the system, and to save snapshots of the system's current state.

The communication between PRINC and the external system is two-way and message-based. The external system continuously feeds the framework with the necessary worker performance data and state changes and receives rewarding action notifications from PRINC. For example, PRINC may notify the external system that a worker earned a bonus, suggest a promotion or a punishment. Similarly, it may need to send an admonition message to the worker, or display him/her a motivating visual information (e.g., rankings). The external system ultimately decides which notifications to conform to and which to discard, and reports this decision back in order to allow keeping the RMod in consistent state.

The PRINC framework can host RMods for multiple external systems at the same time. As long as the communication middleware is capable of routing the event updates and rewarding action messages to proper destinations, we are able to effectively externalize the incentive management *as a service*. In Chapter 6 we present the design of one such middleware – SmartCOM.

5.2.1 Rewarding Model (RMod)

The full description and evaluation of the RMod is provided in Section 4.2. PRINC incorporates the described model and introduces additional functionality needed to exploit this functionality.

5.2.2 Mapping Model (MMod)

In order for PRINC to couple with an external socio-technical system the incentive designer and an integration engineer need to provide mappings that enable the application of generic incentive strategies on top of the particular context of the external system. These mappings are provided through the Mapping Model.

The functionalities of MMod include:

- *Definition of system-specific artifacts, actions, attributes and relation types.*

  These definitions inform PRINC of the unique names and types of different company-specific *artifacts*, *actions*, attributes and relation types that need to be stored and represented in PRINC for subsequent reasoning over conditions for applying rewards. Actions represent different events happening in the external system. Artifacts represent objects of the actions (Section 5.2.3). For example, a design company may want to define an artifact to represent the various graphical items that its users produce during design contests, and an action to denote the act of submitting a design artifact for evaluation.

- *Definition and parameterization of metrics, structural patterns and incentive mechanisms.*

  Metrics are attributes that are calculated by PRINC from other attributes provided by the external system. They are used to express different performance aspects of individuals or groups of workers. For example, a context-independent incentive strategy may rely on worker's trust metric in a reward application condition. However, for different companies, the trust metric is calculated differently. For example, the trust of a worker may depend on the percentage of the peer-approved tasks in the past (as in Section 5.3), or it may involve a calculation based on trust values of nearest neighbors.

  PRINC offers encoding predefined metrics as built-in (library) functions (e.g., trust, productivity, effort), thus cutting the time needed to adapt a generic incentive strategy to a particular scenario. Predefined metrics then only need to be parameterized. In cases where a built-in metric definition is unable to express a system-specific aspect, clients can provide their own definition. This is usually the case with company-specific *predicates*, which we can define in MMod. One common use of predicates is to define criteria of team membership. A criterion can be structural (e.g., all workers managed by 'John Doe'), logical (e.g., workers with the title 'Senior Java developer'), temporal (e.g., workers active in the past week) or composite (e.g.,'Senior Java developers' active in the past week).

In the same way, predefined structural patterns and entire incentive mechanisms can be parameterized in MMod. For example, the library pattern `COLLABORATORS (Worker W, RelationType RT, Weight w)` returns for a given `Worker` node `W` a set of workers that are connected with `W` via `RT`-typed relations, having the weight greater than `w`, where `w` is a client-provided value. In case a translated incentive strategy relied on using this library pattern, the client could be asked to provide only a value for `w`, while PRINC would initialize the other parameters during the execution.

- *Message mappings.*

  When a condition for performing a rewarding action is fulfilled, PRINC needs to inform the external system. For each rewarding action we need to specify the type of message(s) used to inform the external system and the data they will contain. The data contained may include metric values to be used as a justification for executing a reward/punishment, or a structural pattern suggesting a structural transformation to the external system. Also, we need to specify which messages PRINC expects to get as an answer to the suggested action. Only in case of a positive feedback will PRINC proceed to update its internal model. Otherwise, the rewarding action is ignored.

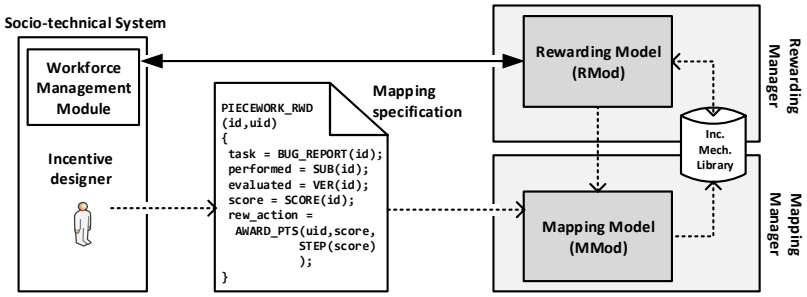

Figure 5.2: Adapting a general piece-work incentive mechanism for software testing company use-case.

Example.

A software testing company wants to setup quickly an incentive mechanism that awards every bug submitter a certain number of points for every verified bug. The amount of points assigned is company-specific and depends on bug severity. There is a number of real crowdsourcing companies that rely on such mechanisms (e.g., translation companies and design companies).

A pre-designed library incentive mechanism `PIECEWORK_RWD(···)` works with the concept of a 'task'. Once the task is 'performed', an 'evaluation' process on its quality is

started. The evaluation phase ends with obtaining a 'score'. The 'rewarding action' is then executed if a predicate taking the evaluation score as one of its input parameters returns true.

In this particular case the testing company can define an artifact named `BUG_REPORT` to represent a bug report in our system, containing a bug ID, severity, and other fields. The act of submitting a bug report can be defined as the `SUB(id)` action, the act of verifying a bug report as the `VER(id)` action. What is left to do is to simply map these actions, artifacts and metrics to the incentive mechanism parameters (Figure 5.2). In this case, the concept of 'task' is mapped to the `BUG_REPORT` artifact. Performing of the task is signaled by a message containing the `SUB(id)` action. The voting phase ends with the arrival of the `VER(id)` action. From then on, the corresponding score can be accessed as the metric `SCORE(id)`.

Assignment of rewards to the bug submitters can also be automatically handled by one of the library rewarding actions we indicate in the mapping. For example, the action `AWARD_PTS(userID, score, mappingFunction(score))` simply informs the company's system of how many points the user should be awarded, based on his artifact's score and a mapping function. The mapping function in this case can be a step function or a piecewise-linear function.

5.2.3 Interaction Interfaces

The framework provides two APIs for manipulation of the internal state: a) An API for direct manipulation of RMod and MMod (*DMMI*); and b) A message API for event-based RMod manipulation (*MSGI*), meant for the external system.

DMMI is intended for internal use within the PRINC framework. This API exposes directly the functionalities which are not supposed to be used during the normal operation of the framework since the consistency of the model's state cannot be guaranteed. External use should therefore be limited to handling uncommon situations or performing monitoring. MSGI is intended for exchange of notifications about external system state changes or suggested rewarding actions. (Un-)marshalling and interpreting of messages is handled by the Rewarding Manager. The functionalities offered by the APIs are summarized in

API	Functionality	Description
MSGI	State updates	Notify framework of external structural/temporal/attribute changes.
	Rewarding	Suggest a rewarding action to the external system.
	Notifications	Mutually exchange artifact, action and attribute updates/events.
DMMI	Database API	Manipulate DB records. Execute DB scripts.
	Rules API	Directly execute RMod rules and queries.
	Timeline API	Modify past and future iteration parameters.
	Structure API	Directly perform graph transformations.
	Mappings	Change mappings in runtime (dynamically).

Table 5.1: Functionalities exposed through the APIs

Table 5.1. Abstract representation of the message format is shown in Figure 5.3. This format can be used for both incoming and outgoing messages.

The *Action* defines the message identifier, type, timestamp and importance. In case of an incoming message, the type can represent the following: (a) A system-specific activity that needs to be recorded (e.g., task completion, sick leave) for later evaluation; (b) Update of an attribute (e.g., hourly wage offered); or (c) Update of the worker/team structure.

The *Artifact* specifies the object of the action. It contains the new value of the object that needs to be communicated to the other party. In case of an incoming message, the *Artifact* can correspond to: (a) an activity notification (expressed as an artifact defined in MMod); (b) an attribute update; (c) a structural update; or (d) an iteration update. In case of an outgoing message, the artifact can correspond to: (a) an activity notification; (b) a metric update; or (c) a rewarding action notification.

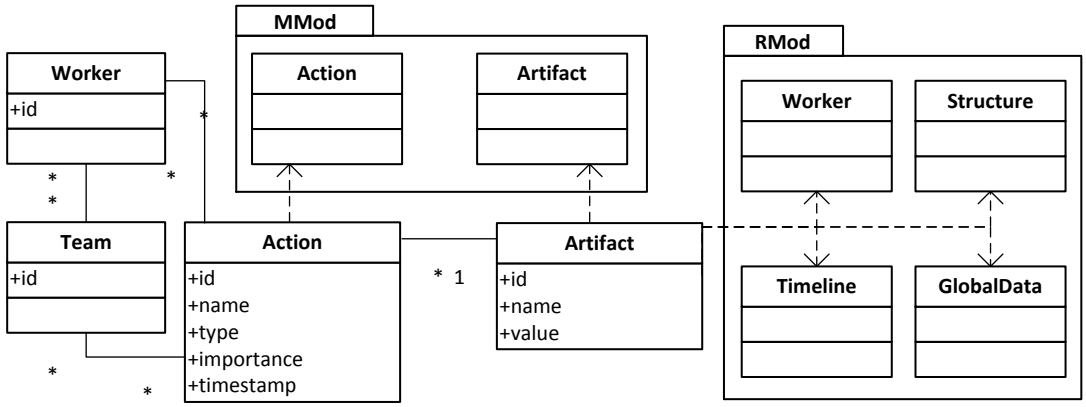

Figure 5.3: Abstract representation of the MSGI message format.

Structural updates can be expressed either as library-defined structural modification patterns or as completely new descriptions of the graph (sub)structure defined in a formal language. Iteration updates notify the system of the (re-)scheduling of future iterations and the duration changes of the currently active ones. *Worker* and *Team* parts of the message specify the workers the message applies to. As already explained, the team identifiers are defined in MMod and serve to target all individual workers fulfilling a condition, or as a simple shorthand notation.

5.3 Prototype & Evaluation

The implemented prototype[1] consists of the framework components framed with full borderlines in Figure 5.1. The prototype was implemented in C#, using Microsoft SQL Server database. Structural modifications are performed using the GrGen.NET [JBK10] library. GrGen is a versatile framework for performing algebraic graph transformations,

[1] `https://github.com/tuwiendsg/PRINC`

including a graph manipulation library and a domain-specific language for specifying declarative graph pattern matching and rewriting. We used a number of pre-compiled graph transformation patterns, which are able to capture structural requirements of the incentive mechanisms we intend to support.

The prototype uses the imperative rewarding rules and MMod mappings provided by the user via initialization scripts. For the proof-of-concept purposes, they are specified as C# code. The implemented message-based API supports binary[2] or XML messages, following the format presented in Section 5.2.3. Structural patterns can be expressed in GrGen's domain-specific language, or in the DOT language[3].

The prototype includes the RMod component already described and evaluated in Section 4.2. The remaining evaluation in this section focuses on Mapping Model. See Section 5.2.2. (MMod). It is intended to show how through MMod it is possible to use previously defined library incentive elements and parametrize them for use for a typical, real-world incentive scheme. A rich library of incentive elements can in this way be used as the foundation for the rest of the incentive management platform tools.

Out of the strategies surveyed in [STD13a] we decided to model and implement the following subset of the incentive strategy of the Locationary company (fully presented in Section 3.4.2). Locationary was a company that used to sell access to a global business directory. In order to have a competitive advantage over a number of companies already offering traditional and internet business directories they needed to maximize the number, accuracy and freshness of their entries. For this reason, they sought to incentivize workers spread around the world to add and actively update local business data.

For adding/updating the entries in the business directory the workers were being issued 'lottery tickets'. Tickets were allowing the workers to enter into occasional cash prize draws. Chances of winning were proportional to the number of tickets held. This mechanism incentivized the increased activity of the workers, but also encouraged dysfunctional behavior – the inputting of fake/incorrect data. The mechanism that was introduced to counteract this unwanted behavior was the 'deferred compensation'. The workers were only allowed to enter the prize draws if they had gathered enough tickets (ticket quota) *and* if their trust score was high enough. The trust metric was proportional to the percentage of the approved entries, motivating workers to enter correct data.

The primary reason for choosing these particular incentive mechanisms for showcasing the mapping and parameterization functionalities of MMod was that they demonstrate particularly well how through reusing and adapting a number of generic incentive elements effective incentive mechanism can quickly be put into place and combined together.

In Section 5.2.2 we described how a general rewarding mechanism for piece-work can be adapted to fit the needs of a software testing company. Here we use the same mechanism to reward workers with lottery tickets, and the same rewarding action `AWARD_PTS(···)` to simulate cash payouts.

A lottery is a frequently used mechanism when the per-action compensation amount is too low to motivate workers due to a high number of incentivized actions. Listing 5.1

[2] `https://github.com/google/protobuf`

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

```
% Library definitions in RMod
interface T_LOTTERY_TCKT    % Predefined artifact interface.
{
  id;
  uid;                      % Owner ID.
  value = 1;                % Ticket value. Default is 1.
}

LOTTERY                     % Predefined (library) incentive mechanism.
{
  id;                       % Auto-generated, or assigned during the runtime.
  tickets[];                % Collection of T_LOTTERY_TCKT objects.
  ...
  type;                     % To choose from various sub-types.
  timing;                   % Periodic, conditional or externally-triggered.
  numberOfDraws;            % How many tickets should be drawn.
  external_trigger;         % User-declared action triggering a lottery draw.
  ticketType;               % User-defined artifact that represents a ticket.
                            % Must be derived from the predefined
                            % T_LOTTERY_TCKT interface.

  rew_action;               % Action to execute upon each owner
                            % of a winning ticket.

  prize_calculation;        % Metric used to calculate the total reward
                            % amount for a draw. Usually proportional
                            % to the number of the tickets in the draw.

  entrance_cond;            % Predicate used to evaluate whether a worker
                            % is allowed to enter the draw.
  ...
}
```

Listing 5.1: Definitions of library incentives.

shows the pseudo-code declaration of a general lottery mechanism we implemented as part of our incentive mechanism library. In order to use this mechanism, we simply need to parameterize the general mechanism by providing the necessary mappings (values, metrics, actions and predicates), as shown in Listing 5.2. Once the incentive strategy is running, we can easily adapt it by changing which metrics, predicates and actions map to it.

This example also shows how we can combine different incentive mechanisms. For example, the predicate that controls worker's participation in a lottery draw requires the worker to possess a certain quota of tickets. The threshold is managed by another parameterized incentive mechanism, namely `LOTTERY_QUOTA`. To express trust we use one of the predefined metrics. The remaining mechanisms are similarly implemented, demonstrating that our approach is capable of functionally modeling realistic incentive strategies.

```
% User definitions in MMod
action RUN_LOTTERY(int id);
artifact LOCATIONARY_TICKET extends T_LOTTERY_TCKT {...};

metric CALC_PRIZE(int id, float prizePerTicket)              % A user-defined
{                                                            % metric.
  LOTTERY L = getLottery(id);
  return prizePerTicket * L.tickets.count;
}

predicate ENTER_LOTTERY_PREDICATE(int lotteryId, int userId) % User defined
{                                                            % predicate.

  return TRUST(userID) > 0.65    &&                          % Trust and
         LOTTERY_QUOTA(userId);                              % lottery quota
}                                                            % are library
                                                             % elements.

% User mappings in MMod
LOCATIONARY_LOTTERY = LOTTERY                                % Parameterizing
   {                                                         % a general inc.
     ...                                                     % mechanism.
     timing = "triggered";
     numberOfDraws = 1;
     external_trigger  = RUN_LOTTERY;
     ticketType = LOCATIONARY_TCKT;
     rew_action = AWARD_PTS(ticket.uid, amount, amount);     % Previously
                                                             % explained.

     prize_calculation = CALC_PRIZE(id, 0.0025);             % Here we use a
                                                             % custom metric.
     entrance_cond = ENTER_LOTTERY_PREDICATE(id, ticket.uid);
   }
```

Listing 5.2: Defining customized incentive mechanisms with library elements.

CHAPTER 6

Communication Middleware for Application of Incentives

In order to support coupling of the incentive management framework presented in Chapter 5 (see Fig. 5.1) with different underlying socio-technical systems, to support selective and safe accessing and management of personal worker data, and to support delivery of certain types of rewarding actions (e.g., motivating messages, display of rankings, badge delivery) directly to workers an additional communication and virtualization layer was needed. This layer was named SmartCOM *middleware*.

The middleware was designed[1] in the context of the EU FP7 "SmartSociety" project[2] to accomplish various goals, but here we present only the functionality and design traits relevant to the usage in the incentive management context:

(a) Message queuing, routing, transformation and delivery.

(b) Support for different messaging formats to support coupling with different platforms and but also allowing direct communication with human participants through popular protocols (e.g., email, Android notifications, Twitter).

(c) Privacy and anonymity isolation layer, decoupling PRINC from working with sensitive user data.

[1] **Disclaimer:** Parts of SmartCOM design and implementation were co-authored by Dipl.-Ing. Philipp Zeppezauer, as part of his master thesis work under author's co-supervision. The results were published in the joint publications [ZSTD14, ZST+14]. The material presented in this chapter is not claimed as a contribution of this thesis, but is presented for completeness purposes, since specifics relevant to SmartCOM's use in an incentive management context were not presented in the original publications. Parts of the material from the cited joined publications is used in this chapter.

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

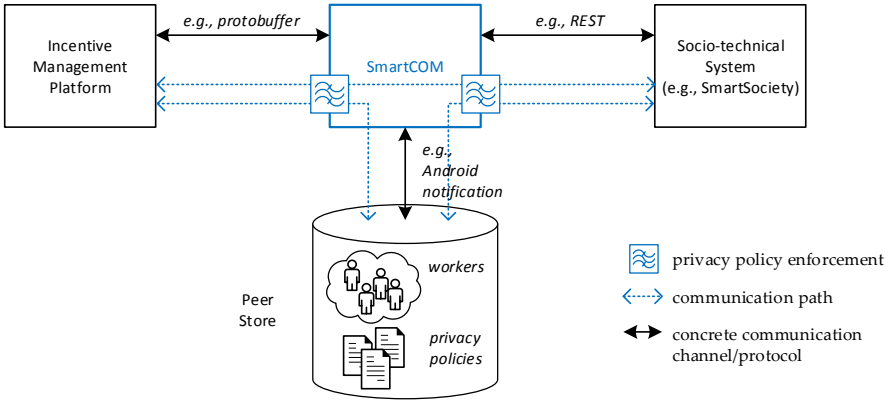

Figure 6.1: SmartCOM's application context.

Figure 6.1 shows the intended usage of the middleware in the incentive management context, performing the three stated functionalities.

Most of these functionalities, taken individually, can be compared with existing solutions. However, to the best of our knowledge, no existing system incorporates a similar union of functionalities as SmartCOM. Popular open-source and proprietary Enterprise Service Buses (ESBs) and integration frameworks provide the same support and flexibility for custom adapters (Sec. 6.1.2) as SmartCOM does. On the other hand, many ESBs lack the support of multi-tenancy (e.g., Apache ServiceMix[3] and JBossESB[4]) or do have restrictions on implementing custom adapters (e.g., JBossESB). Others do not support the dynamical enforcement of policies (e.g., WSO2 ESB [5]) and there is in general no support of the group-level addressing at all which is one of the key features of SmartCOM. Furthermore the support of humans interacting with the system is generally not considered. Some existing solutions (e.g., [ABS+14]) virtualize human peers as Web Services, restricting the use of other communication channels, protocols, or external tools. For example, Social Computing platforms like Jabberwocky [ABMK11] or TurKit [LCMG09] utilize human capabilities to solve problems. However, they rely on existing crowdsourcing platform's communication model with all the restrictions this brings along; for example, support for inter-peer communication and collaboration is very restricted.

In the remainder of this chapter, we present the design and implementation of key SmartCOM components and demonstrate how they fulfills the required functionalities. Due to its size, the full specification is provided as an external technical report[6]

[3] http://servicemix.apache.org/
[4] http://jbossesb.jboss.org/
[5] http://wso2.com/products/enterprise-service-bus/
[6] https://github.com/tuwiendsg/SmartCom/blob/master/doc/technical-report.pdf

6.1 Middleware Design and Architecture

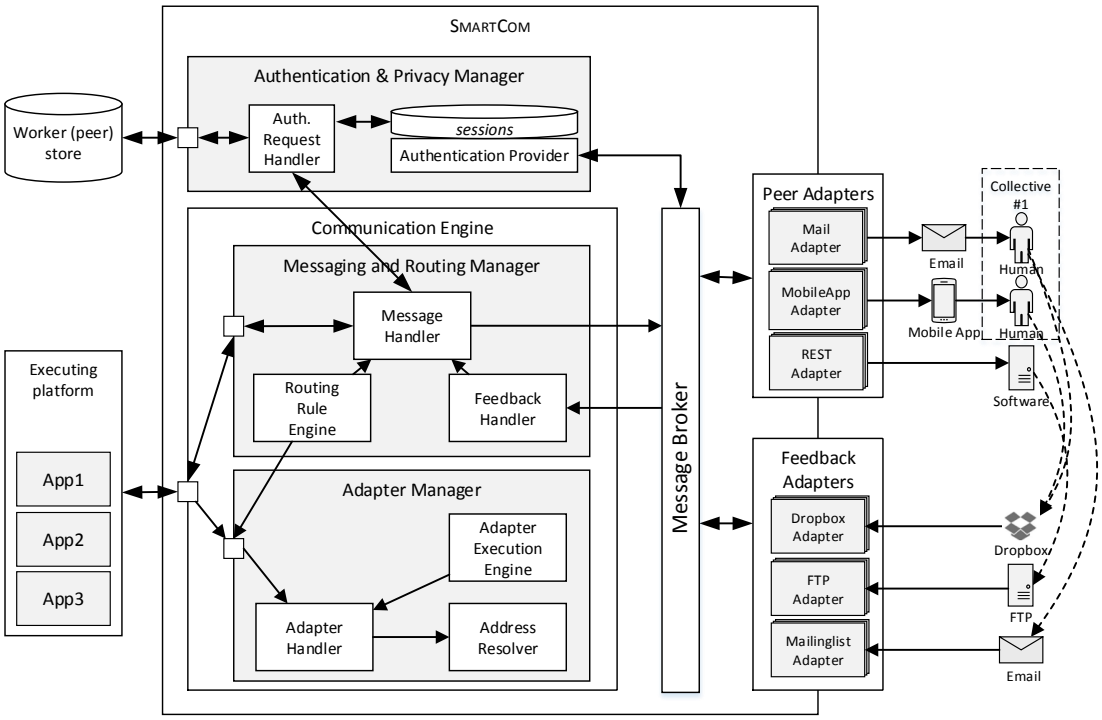

Figure 6.2: Internal architecture of SmartCOM middleware.

Figure 6.2 shows the internal architecture of the SmartCOM middleware. The primary function of the middleware is exchange of messages between *peers* and the *executing platform*, as well as among peers themselves. The term *peer* is used to denote both human entities (e.g., workers) and software entities (e.g., external Web Services) that act as communication endpoints (senders/receivers of messages). The *executing platform* (in our case both PRINC and the socio-technical system) is the software entity performing computational processes involving peers, for which SmartCOM provides communicational support. The term *collective* is used to denote a set of peers requiring multicast routing and delivery at given time. For example, in a incentive context, a collective can represent a team of workers that need be contacted concurrently via different communication channels and protocols to deliver informational or motivational messages.

The executing platform passes the messages intended for collectives to SmartCOM (i.e., to the *Communication Engine* component) through a public API. The task of the Communication Engine is to virtualize the notions of peers and collectives to the executing platform, determine the recipients and delivery routes and instantiate an 'adapter' to perform the delivery. The term *adapter* denotes the middleware component in charge of handling the communication.

6.1.1 Message Handling

Messages are handled by the Messaging and Routing Manager. Principal message handling and routing algorithms are described in Appendix A, Section A.I.

6.1.2 Adapters

In order to use a specific communication channel, an associated *adapter* needs to be instantiated. The communication between peers and the adapters is unidirectional — *output adapters* are used to send messages to the peers; *input adapters* are used to receive messages from peers. SmartCOM originally provides some common input/output adapters (e.g., SMTP/POP, Dropbox, Twitter). The role of adapters should be considered from functional and technical perspectives.

Functionally, the adapters allow for:

(a) Hybridity – by enabling different communication channels to and from peers;

(b) Scalability – by enabling SmartCOM to cater to the dynamically changing number of peers;

(c) Extensibility – new types of communication and collaboration channels can easily be added at a later stage transparently to the middleware's users.

(d) Usability – human peers are not forced to use dedicated applications for collaboration, but rather freely communicate by relying on familiar third-party tools.

(e) Load Reduction and Resilience – by requiring that all the feedback goes exclusively and unidirectionally through external tools first, only to be channelled/filtered later through a dedicated input adapter, the SmartCOM is effectively shielded from unwanted traffic load, delegating the initial traffic impact to the infrastructure of the external tools. At the same time, failure of a single adapter will not affect the overall functioning of the middleware.

Technically, the primary role of adapters is to perform the message format transformation. Optional functionalities include: message filtering, aggregation, encryption, acknowledging and delayed delivery. Similarly, the adapters are used to interface SmartCOM with external software services, allowing the virtualization on third party tools as common software peers. The *Adapter Manager* is the component responsible for managing the adapter lifecycle (i.e., creation, execution and deletion of instances), elastically adjusting the number of active instances from a pool of available adapters. This allows scaling the number of active adapter instances out as needed. This is especially important when dealing with human peers, due to their inherent periodicity, frequent instability and unavailability, as well as for managing a large number of connected devices, such as sensors. The Adapter Manager consists of following subcomponents:

- *Adapter Handler*: managing adapter instance lifecycle. It handles the following adapter types:

1. Stateful output adapters – output adapters that maintain conversation state (e.g., login information). For each peer a new instance of the adapter will be created;
2. Stateless output adapters – output adapters that maintain no state. An instance of an adapter can send messages to multiple peers;
3. Input pull adapters – adapters that actively poll software peers for feedback. They are created on demand by applications running on the HDA-CAS platform and will check regularly for feedback on a given communication channel (e.g., check if a file is present on an FTP server);
4. Input push adapters – adapters that wait for feedback from peers.

- *Adapter Execution Engine*: executing the active adapters.
- *Address Resolver*: mapping adapter instances with peers' external identifiers (e.g., Skype/Twitter username) in order to initiate the communication.

Input messages from peers (e.g., subtask results) or external tools (e.g., Dropbox file added, email received on a mailing list) are consumed by the adapters either by a push notification or by pulling in regular intervals (more details in Section 6.2). Principal adapter handling algorithms are described in [Zep14, ZST+14].

6.1.3 Privacy and Anonymity Features

The *Authentication Manager* is used to identify peers and verify the authenticity of their messages in the system while minimizing the exposure of personal data of the peers involved in communication. This means that the external software platforms, such as PRINC can delegate the full responsibility of authentication and privacy management to SmartCOM, and work only with human peer profiles that disclose as little information as required for the platform to perform the functionality. In case of incentive management, this means having the ability to work with a unique alias of the worker and read the required performance metrics and, possibly, interactions and relationships with other managed workers through their anonymized aliases.

In order to obtain anonymized human peer profiles SmartCOM relies on existence of an external peer store called *PeerManager*[7] that maintains and manages information about human- or machine-based peers in a privacy-preserving framework. PeerManager is able to store sensitive information about human participants for different applications/platforms concurrently, applying a *entity-centric semantic enhanced model* [GDM13] that defines an extensible set of entity schemas providing the templates for an attribute-based representation of peers' characteristics, which effectively allows it to expose peer profiles

[7]**Disclaimer**: PeerManager is used as an external tool obtained through collaboration on the joint research project 'SmartSociety'. It does **not** represent a contribution of this thesis nor of its author. Documentation available at: `http://www.smart-society-project.eu/publications/deliverables/d_4_2/`

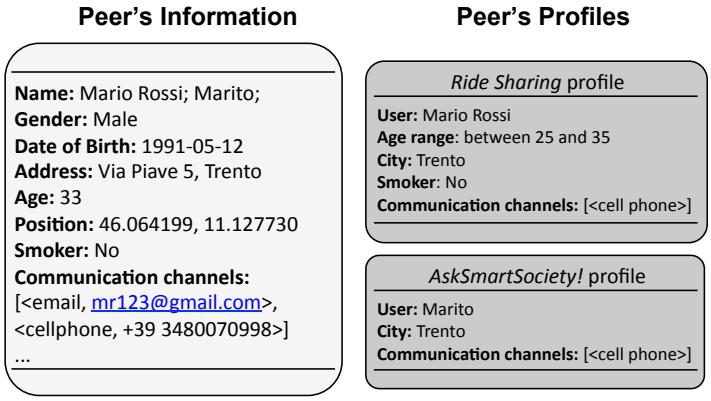

Figure 6.3: Simplified example of a peer with multiple profiles. Each profile is revealed to a different application.

The PM defines a privacy protection model that pays special attention to different privacy principles enacted by the EU Data Protection Directive 95/46/EC[8] affecting storage and processing of personal data. Specifically, the model defines privacy regulations and considerations described in [HJCA+15], such as purpose specification and binding, that are enforced upon search queries, allowing to reveal only partial or (semantically) obfuscated information, used for replying to specific information requests, thus enforcing data minimization. Figure 6.3 (from [SMS+15]) shows a simplified example of a human peer subscribed to participate in two platform applications, revealing different information (by using different profiles) in each case. This allows, e.g., a human peer to reveal its age range (as a way to obfuscate the exact date of birth), while the same information is completely hidden when participating in a different application.

By relying on PeerManager as the external peer store, the Authentication Manager is able to maintain multiple identities corresponding to different profiles of the same person, and route the messages to/from him/her accordingly. In an

Authentication request messages are collected by the *Authentication Request Handler*. After the successful authentication, the manager creates a security token that can be used by peers and SmartCOM to provide security features (e.g., message authentication or message encryption). This token is only valid a certain period of time. The time period between the creation of the token and the invalidating thereof is called *session*. The result of the authentication is passed to the Control Queue in form of a response message. The *Authentication Provider* can be used by the Messaging and Routing Manager to verify the authenticity of a message – if required.

The Authentication Manager uses a *Session Data Storage* to handle the sessions of peers. Sessions consist of a *session token* that can be used by peers to authenticate messages, and a timestamp. If a message arrives with a token of an invalid session, the peer has to be informed to renew its token. Such messages should be discarded or at

[8] http://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:31995L0046

least retained until the peer authenticates itself again.

Additional Privacy Functionalities

SmartCOM supports specifying and observing delivery and privacy policies on message, peer and collective level: *Delivery policies* stipulate how to interpret and react to possible communication exceptions, such as: failed, timed out, unacknowledged or repeated delivery. *Privacy policies* restrict sending or receiving messages or private data to/from other peers, collectives or external applications under different circumstances. Apart from offering predefined policies, SmartCOM also allows the users to import custom, application- or peer-specific policies. As noted, both types of policies can be specified at different levels. For example, a peer may specify that he can be reached only by peer 'manager' via communication channel 'email', from 9am to 5pm in collective 'Work'. The same person can set to be reachable via 'SMS' any time by all collective members except 'manager' in collective 'Bowling'. Similarly, a collective delivery policy stating that when sending instructions to a collective it suffices that the delivery to a single member succeeds to consider the overall delivery successful on the collective level. SmartCOM takes care of combining and enforcing these policies transparently in different collective contexts.

6.2 Implementation & Evaluation

SmartCOM prototype was implemented in the Java programming language. One can interact with it through a set of provided APIs. The prototype comes with some implemented standard adapters (e.g., Email, Twitter, Dropbox) that can be used to test, evaluate and operate the system. Additional third-party adapters can be loaded as plug-ins and instantiated when needed. SmartCOM uses MongoDB[9] as a database system for its various subsystems. Depending on the usage of the middleware, either an in-memory or dedicated database instances of MongoDB can be used. To decouple execution and communication we use Apache ActiveMQ[10] as the message broker. The source code is provided in SmartCOM's GitHub repository[11].

As the envisioned positioning of SmartCOM in the overall architecture of the incentive management platform requires that all information exchange takes place through it, the prototype was put through a performance evaluation to demonstrate that it is capable of withstanding high message loads (peaks) that could occur at specific times when an incentive may need be applied to a large group of workers simultaneously (e.g., a deadline).

The following performance evaluation was made on a 64-bit Intel Core2 Duo machine with 2x 2.53 GHz, 4.00 GB DDR2-RAM. The simulation configuration is as follows:

[9] `http://www.mongodb.org`
[10] `http://activemq.apache.org`
[11] `https://github.com/tuwiendsg/SmartCom`

- One implementation of a Stateless Output Adapter (one instance shared by all peers).
- 10 Input Push Adapter to receive input from peers.
- Output and Input Adapters communicate directly using a in-memory queue to simulate a peer with a response time of zero.
- Worker threads ('Workers') simulate the concurrently executing incentive mechanisms sending incentive messages (rewarding actions) to the 'peers' (simulated human workers).
- One million messages are sent for each evaluation test run to get a meaningful average number of messages sent/received.
- Only sent and received messages are considered as 'handled', no internal messages.

Figure 6.4 depicts the setup for the performance evaluation as described above.

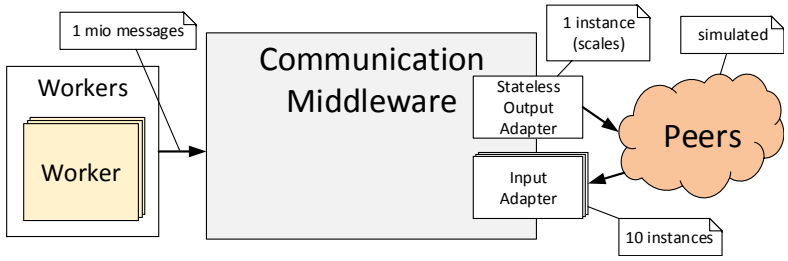

Figure 6.4: Setup for the performance evaluations.

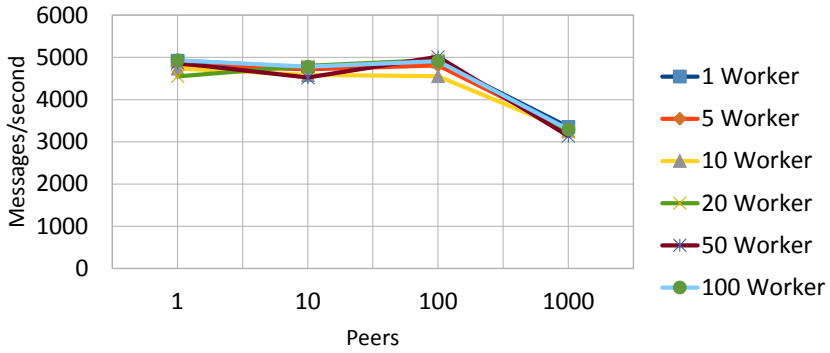

Figure 6.5: Simulated message throughput. 'Workers' are concurrent threads simulating concurrent applications of rewarding actions to human 'peers'.

The performance was evaluated for every combination of 1, 5, 10, 20, 50 and 100 worker threads sending $1 \cdot 10^6$ messages concurrently, uniformly distributed to 1, 10, 100, and 1000 peers waiting for messages and replying to them. Each test run was executed

10 times to obtain average throughput results. Figure 6.5 presents the results of the test runs. The test runs can be reproduced using the stated setup data to configure the Java application located at GitHub[12]. As can be seen, the average throughput remains between 5000 and 3000 messages per second. The performance decrease with higher amounts of peers is result of increased memory requirements rather than computational complexity. The limiting factor here is the used ActiveMQ message broker which only allows a maximum of approximately 20000 messages per second. The system has an upper bound of 5000 messages per second since each message is handled multiple times by the message broker and the SmartCOM. This limitation applies to a single SmartCOM instance, but multiple SmartCOM instances can be deployed to balance the load if needed, sharing the database and peer store access. The chosen numbers of worker threads and peers cover the reasonably expected maximum numbers of concurrently executing incentive mechanisms and concurrently targeted humans, respectively. Performance (scalability) is, thus, not expected to become a primary concern of SmartCOM, especially considering the inherent latency of human peers and variance of response times which are both much higher in real-world than in simulated conditions.

[12] `https://github.com/tuwiendsg/SmartCom/blob/master/smartcom-demo/src/main/java/at/ac/tuwien/dsg/smartcom/demo/PerformanceDemo.java`

CHAPTER 7

Programming Model and Domain-Specific Language for Incentive Management

In this chapter we present the programming and execution model, semantics and syntax of PRINGL[1] – a domain-specific language for modeling incentives for socio-technical systems. We describe PRINGL's modeling paradigm, and demonstrate its expressiveness by modeling a set of realistic incentive mechanisms.

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. 7.1). 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.

The PRINC framework presented in Chapter 5 lacked a comprehensive, human-readable way of encoding incentive mechanisms. However, PRINC possesses all the characteristics of an abstraction interlayer. It features the general 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 coupled with SmartCOM 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

[1] PRogrammable INcentive Graphical Language (PRINGL)

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 and SmartCOM.

In this chapter we assume the existence of PRINC as abstraction interlayer. The business logic code provided in the examples in Section 7.4 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 (Sec. 7.2.3). This contradicts one of the principal motives for introduction of PRINGL, and is more disadvantageous than building a completely system-specific incentive management solution.

7.1 Overview

Figure 7.1 shows an overview of PRINGL's architecture and usage in the overall context of the incentive management platform. PRINGL is designed to be used by the same types of users as PRINC (introduced in Section 5.1) – *incentive designers* and *incentive operators*. An incentive designer models an incentive scheme provided in natural language 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 Sec. 7.2.3, 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.

7.1.1 Requirements

As PRINGL is a domain-specific language, the focus of the design requirements lies primarily on coverage of the domain and usability to the stakeholders within that domain. The

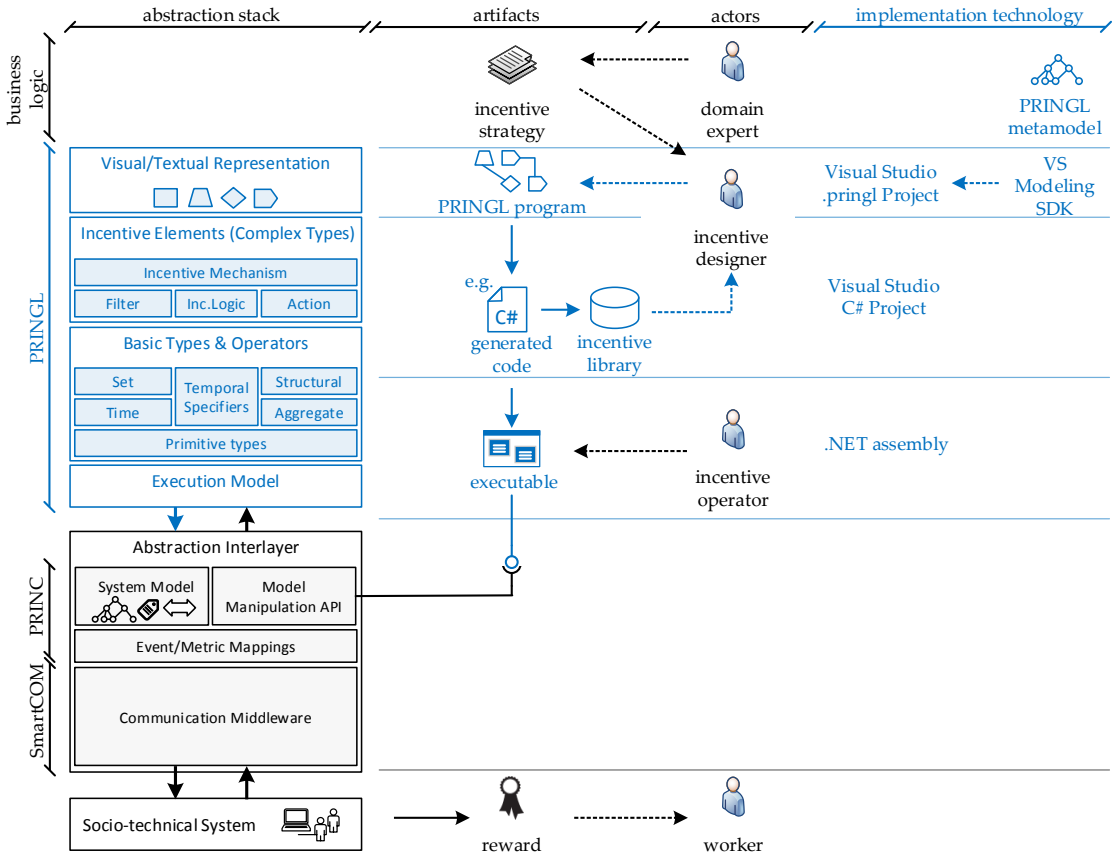

Figure 7.1: Incentive management platform tools, showing an overview of PRINGL's programing model elements, architecture, users, operative environment and implementation (marked in blue).

design of the language was guided by the following requirements, formulated according to the guidelines outlined in [MH10]:

1. *Usability* – provide an intuitive, user-friendly modeling DSL for incentive operators.
2. *Expressiveness* – provide an expressive environment for programming complex real-world incentive strategies for incentive designers.
3. *Groundedness* – allow the use of *de facto* established terminology, components and methods for setting up incentive strategies.
4. *Reusability* – support and promote reuse of existing incentive business logic.
5. *Portability* – support system-independent incentive mechanisms, agnostic of type of labor or workers, and of underlying systems.

7.2 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 [STD13a, TCZ12] and allows composing complex ***incentive schemes*** out of simpler elements. Such a modular design also promotes reusability since the same incentive elements with different parameters can be used for a class of similar problems, stored in libraries and shared across platforms.

PRINGL allows incentive designers to model natural-language, realistic incentive 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.

7.2.1 Primitive Incentive Elements & Operators

Primitive Incentive Elements

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 7.1. They do not have a direct visual representation. Only primitive elements can be used as inputs and outputs of *complex incentive elements* (Section 7.2.2).

PRINGL provides a number of operators for manipulating the introduced primitive types.

Built-in Operators

- *Set operators.* – Union, intersection and complement on `Collection<T>`.

- *Time operators.* If working with adjustable intervals, it is advisable to use operators wherever possible as they are evaluated at run time and guarantee that any external changes (e.g., deadline extensions) will be taken into account. A common use-case would see a user initializing an `Interval` from an iteration, and using interval

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* [STD13b]). 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 7.1: Primitive types.

operators to specify points in time in which an action is needed. Time operators are commonly used with temporal specifiers.

  - `StartOf(Interval i)` – returning the `Collection<PoiT>` containing a single time point representing the interval's currently expected starting time.
  - `EndOf(Interval i)` – returning the `Collection<PoiT>` containing a single time point representing the interval's currently expected ending time.
  - `PartOf(Interval I, double p)` – $p[0, 1]$ returning the PoiT at percentage p of the interval. $PartOf(i, 0) == StartOf(i) PartOf(i, 1) == EndOf(i)$
  - `MultiPoint(Interval i, int k)` – returns a `Collection<PoiT>` of points evenly distributed between StartOf() and EndOf().
  - `AllOf(Interval i)` – returns a `Collection<PoiT>` of points representing all time points (depending on the resolution of the underlying system) contained in the interval.

- *Temporal specifiers.* These are special operators used to instruct the 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 that is provided by the environment at runtime.

  - `Always(Collection<PoiT>)` – "at each `PoiT` in collection".
  - `Sometimes(Collection<PoiT>)` – "at least once in collection".
  - `Once(Collection<PoiT>)` – "exactly once in collection".
  - `Never(Collection<PoiT>)` – "never in collection".
  - `First(Collection<PoiT>)` – "oldest in collection".

  - `Last(Collection<PoiT>)` – "newest in collection".

- *Structural operators.* They perform structural queries/modifications by exa-mining/re-chaining the relationships between worker nodes in the abstraction interlayer (graph) model by using *graph transformations*[2] [BH02].

  - Querying:

    * `neighborsOf(Worker w, string relationType, int numHops, bool directed)` – returns a `Collection<Worker>` filled with workers `numHops` hops away from `Worker` w over un-/directed `relationType` relationships.
    * `managersOf(Worker w)` – returns `Collection<Worker>` filled with manager(s) of worker W. The relationship type representing the managerial relation is obtained from the abstration interlayer.
    * `subordinatesOf(Worker w)` – analogous to `managersOf`.

  - Modifying:

    * `changeManager(Worker w, string teamLabel)` – rechains the implicitly determined managerial relations within the members of the `tealLabel` team to point to the new manager.

- *Aggregation operators.* They perform calculations on performance metrics or events over a `Collection<PoiT>`s, in a fashion similar to SQL's aggregate functions. The collection of time points over which the operators calculate is provided by the runtime environment at each invocation. They can only be used in predicate logic blocks ⟨Ⓟ⟩ that are directly or indirectly reachable through declaration relationships originating from a `WorkerFilter` 𝔽⟩ element.

  - `@AVG(double` m`)` – returns the average value of the metric m over the given time point collection.
  - `@COUNT(string` *evt*`)` – returns the number of occurrences of event *evt* in the timespan delimited by the first and last `PoiT` in the given input collection.
  - `@MAX(double` m`)` – returns the largest value of the metric m over the given time point collection.
  - `@MIN(double` m`)` – returns the smallest value of the metric m over the given time point collection.
  - `@SUM(double` m`)` – returns the sum of the values of the metric m over the given time point collection.

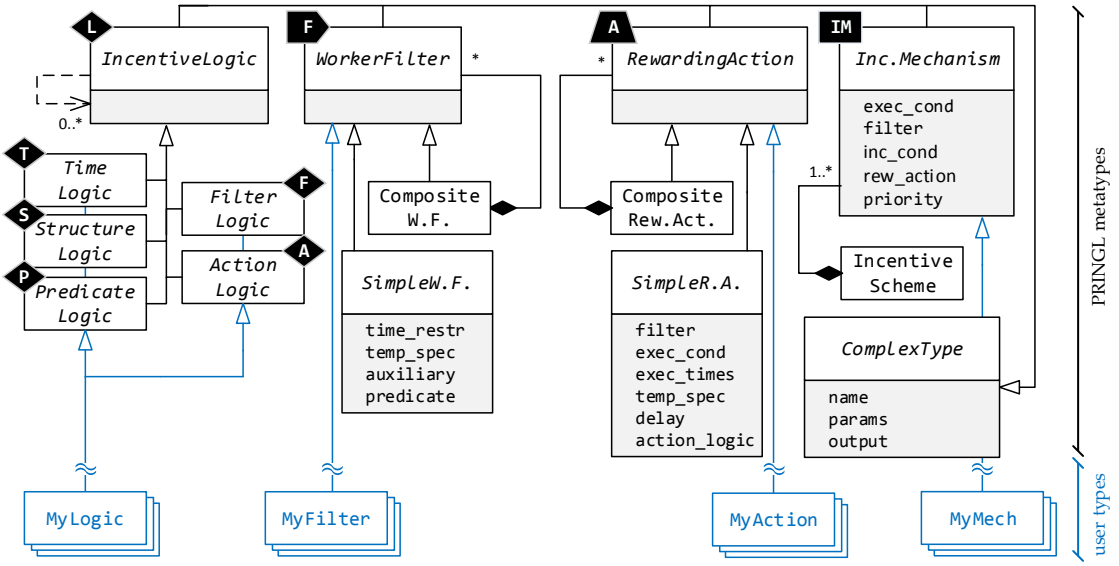

Figure 7.2: Complex incentive elements class hierarchy.

7.2.2 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 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:

Operations on Complex Incentive Elements

Definition – Complex types are defined by inheriting the following abstract metatypes: `IncentiveLogic`, `WorkerFilter`, `RewardingAction` and `IncentiveMechanism` (Fig 7.2). 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[3], 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

[2]Please note that the list of structural operators is non-exhaustive at the moment and serves purely for demonstrational purposes.

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 (blue-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. For example, in case of ⟨T⟩ (Section 7.2.3) all named `Interval`s and all workers are passed as input parameters and made available through predefined variable names (preceded with underscore). This removes the need of having to know how to access certain data from a type definition, thus making it self-contained and portable. The user-defined fields are initialized with values calculated from the expression contained in the type definition and values provided by the user or propagated from the composing elements. Type instances are addressable objects that can be referenced (e.g., to read a field value) or invoked (see below) from the programming code and 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. 7.12) or arbitrarily within the user-provided business-logic code in `IncentiveLogic` elements (see Ex. 3, Fig. 7.10, 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.

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. 7.10, ①.

7.2.3 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[4]. Similarly, different subtypes dictate different return value types. These features encourage high modularization and uniformity of incentive logic elements. Descriptions of the incentive logic subtypes are provided in Table 7.2. Incentive logic element definition is expressed in PRINGL with the visual syntax element shown in a Fig. 7.3, 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.,  for temporal logic.

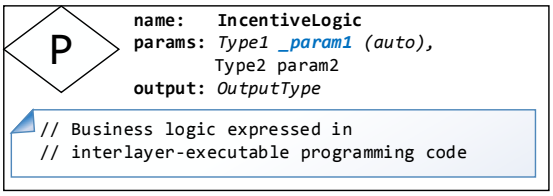

Figure 7.3: Visual element representing an `IncentiveLogic` definition.

[4]Marked with `auto` in figures

Subtype	Symbol	Environment-provided input	Allowed output	Intended usage
`TimeLogic`	⟨T⟩	all named Intervals, all Workers, reference to global state	`Collection <PoiT>`	To return time intervals/points at which a predicate should be evaluated or an action performed.
`StructureLogic`	⟨S⟩	reference to the structural model, reference to global state	`Collection <Worker>` for queries: found workers; for transformations: affected ones	To perform graph queries/transformations on the model representing workforce structure and relationships. A transformation ⟨S⟩ is only allowed to be invoked from ⟨A⟩. A query ⟨S⟩ can only be invoked from ⟨P⟩ and ⟨F⟩.
`PredicateLogic`	⟨P⟩	currently evaluated Worker, all Workers, currently evaluated `PoiT`, reference to global state	`bool`	To evaluate whether a predicate holds at given moment.
`FilterLogic`	⟨F⟩	currently evaluated Interval, all named Intervals, currently evaluated Worker, all Workers, reference to global state	arbitrary	To provide business logic for evaluating past worker performance.
`ActionLogic`	⟨A⟩	Workers to be rewarded/punished, reference to global state	`Collection <Worker>` (affected)	To perform rewarding actions over workers or global variables.

Table 7.2: `IncentiveLogic` subtypes

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 `Worker`s 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.

If we denote the input set of `Worker`s of a `WorkerFilter` ⟨X⟩ with I_x, and the output set with O_x, then the functionality of ⟨X⟩ can be defined as the function f_x:

$$
\begin{aligned}
f_x &: I_x \to O_x \\
I_x &= input(x) \\
O_x &= \{e \in O_x \mid e \in I_x \wedge p_x(e) = true\}
\end{aligned}
$$

where p_x is 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 metatype `WorkerFilter` (Fig. 7.2), 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 a Fig. 7.4, while a right-pointed shape ⟨F⟩ 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. Field descriptions are provided in Table 7.3.

Field	Description
`time_restr`	An optional ⟨T⟩ returning a collection of time points which should be considered when evaluating workers. If omitted, the default value is a collection containing only a single `PoiT` representing the present moment.
`temp_spec`	An optional temporal specifier (Section 7.2.1) determining how to interpret the filter predicate values across different time points. If unspecified, the predicate is evaluated only for the last (most recent) `PoiT` in the collection.
`auxiliary`	An optional ⟨F⟩ that is used to fetch some global metrics needed for worker evaluation, and possibly provide some intermediate results to be used for evaluating the filter predicate.
`predicate`	A required ⟨P⟩ providing the predicate that will be evaluated against each worker in specified time points.

Table 7.3: `SimpleWorkerFilter` fields.

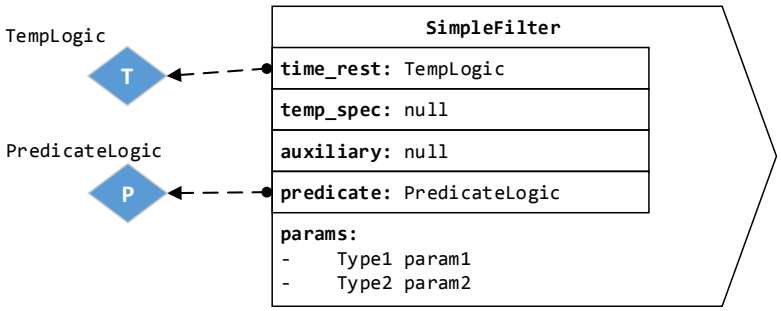

Figure 7.4: Visual element used for `SimpleWorkerFilter` definition.

Composite Filters

In Figure 7.5 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 composite filter. As any other PRINGL composite type, a composite filter can also expose propagated or user-defined parameters.

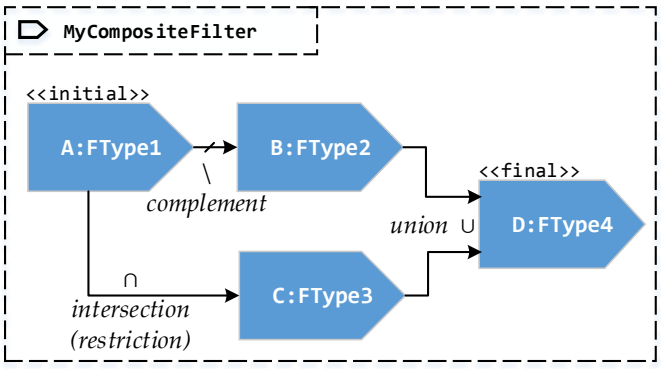

Figure 7.5: An example `CompositeWorkerFilter` definition.

A directed edge $\boxed{A} \rightarrow \boxed{C}$ implies that $\boxed{C}$ takes as input $\boxed{A}$'s output (the workers matching the criteria of $\boxed{A}$). The output of $\boxed{C}$ is a set containing workers fulfilling both filters' conditions, thus effectively representing $\boxed{A} \cap \boxed{C}$ operation. If an edge is marked as *negating* ($\nrightarrow$), then $\boxed{A} \nrightarrow$ returns the set complement of $\boxed{A}$'s input, i.e., $input(A) \setminus \boxed{A}$. 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 

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. 7.6 to denote the definition of a `SimpleRewardingAction`. For the shorthand notation, we use /A\, both for simple and composite rewarding action elements.

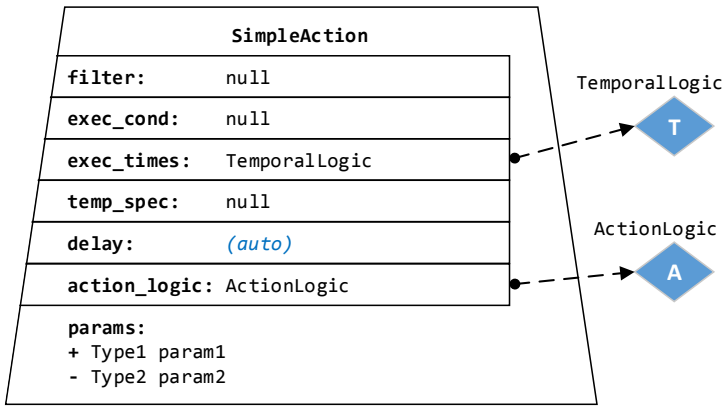

Figure 7.6: Visual element used for `SimpleRewardingAction` definition.

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

Field	Description
`filter`	An optional 𝔽⟩ determining the workers to which to apply the action. If omitted, the worker collection is by default provided by the runtime environment from the output of the original evaluation filters.
`exec_cond`	An optional ⟨ℙ⟩ establishing whether the currently evaluated worker earned the reward/punishment or not. If omitted, considered 'true' by default.
`exec_times`	An optional ⟨𝕋⟩ returning `Collection<PoiT>` determining the possible execution points. If omitted, the environment assumes current `PoiT` and executes immediately.
`temp_spec`	An optional temporal specifier further restricting the original collection of execution `PoiT`s. Defaults to `Always()` if omitted.
`delay`	A hidden parameter set by the environment and used for recalculating execution times in composite rewarding actions. It contains a non-negative integer time offset added to the execution `PoiT`s. The actual time unit is determined as the basic time unit of the underlying layer (an RMod *tick* in our case). The default value is zero.
`action_logic`	A mandatory reference to an ⟨𝔸⟩ element containing the system-specific business logic that invokes the rewarding action.

Table 7.4: `SimpleRewardingAction` fields.

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 Sec. 7.2.1) to determine the exact time moment(s) of the time series. For defining incentives involving *deferred compensation* [STD13a] 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 ⟨𝔸⟩ element. To execute the action PRINGL needs to invoke the appropriate action in the abstraction interlayer which will then send out a system-specific message to the underlying socio-technical platform. Field descriptions are summarized in Table 7.4.

Composite Actions

Similarly to composite filters, a `CompositeRewardingAction` definition consists of graphical elements representing instances of previously defined `RewardingAction`s. It must contain exactly one *initial action* a_0, and exactly k_0 *final actions*, where k_0 is the number of a_0's outgoing edges. The elements are connected with directed edges denoting at the same time: a) *Worker flow*; and b) *time delay*. There must be no cycles in the graph, i.e., the flow must be a tree with the root in the initial action, with each final action being the leaf. As any other PRINGL composite type, a composite action can also expose

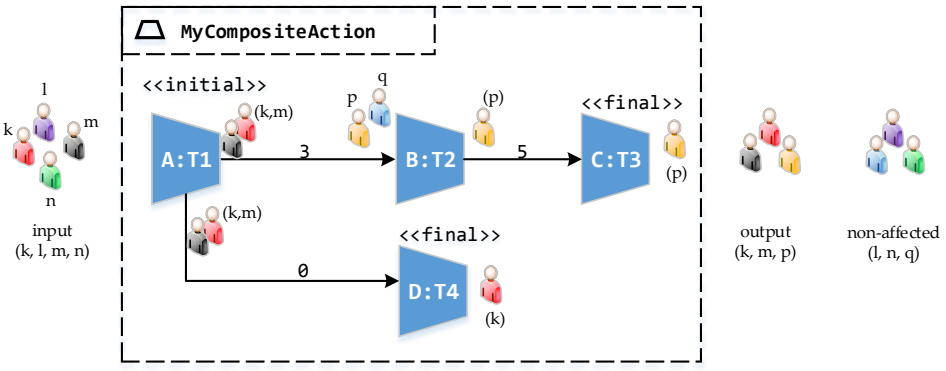

Figure 7.7: An example `CompositeRewardingAction` definition with branch delays shown.

propagated or user-defined parameters.

Worker flow. A `RewardingAction` returns *affected* workers and passes them over outgoing edges. Affected workers are those workers on which the action was successfully applied by the underlying system. The definition of a successful application is system-specific. Therefore PRINGL expects the underlying system to acknowledge via abstraction interlayer that the suggested action was accepted and successfully applied to a worker. The passing of workers is similar to that of composite filters. The two major differences are:

1. The absence of graph cycles prevents the union ($\cup$) operation on passed worker sets.

2. Any `RewardingAction` element can decide whether to use the provided input workers, or completely ignore them, and identify the input workers by itself. For example, a `SimpleRewardingAction` does it by initiating the optional `filter` field. This limitation allows the worker flow to be changed at arbitrary places in the composition.

Figure 7.7 shows an example of `CompositeRewardingAction` definition. It also shows an example of worker-passing. The initial action /A\ is given the set (k, l, m, n) as input. The execution of /A\ ends with successful rewarding of workers (k, m). This intermediate set is immediately added to the resulting output set. The same intermediate set of workers is passed to actions /B\ and /D\. Action /D\ ends with rewarding only one of those workers – (k). k is already part of the output, so nothing else happens on this execution branch. The action /B\, on the other hand, discards the input worker set (k, m), and determines its own input set (p, q). After execution, /B\ returns just (p), which is also added to the aggregate output set and passed further as input to /C\, which also happens to award p successfully.

Time delay. 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 point to.

If this action is a `SimpleRewardingAction`, this equals to adding the specified time offset to the hidden `delay` parameter. Later, when executing the action, PRINGL will add the value of the `delay` parameter to each `PoiT` returned by action's `exec_times` ⟨T⟩. If the delay is forwarded to a `CompositeRewardingAction`, then the delay is forwarded to its initial action.

The execution of a composite action starts by first breaking it into linear execution paths containing constituent simple actions. For each execution path PRINGL then takes into account specified delays for each simple action and immediately schedules it with the abstraction interlayer. However, as in this case we need to pass worker sets between actions happening at different times PRINGL needs to store 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.

Example

The notion of affected workers is important for incentivizing, because a choice on whether or not to perform a subsequent rewarding action may depend on whether previous actions were successfully applied. Consider a company that wants to reward workers either with free days or with a monetary reward. The choice is left to the worker. Free days are offered first. Only workers that refuse the free days will be awarded monetary rewards.

We define a new composite rewarding action `BonusOrDays` (Figure 7.8) that, for the sake of demonstration, assumes the existence of a `RewardAtEndProject` action (similar to the one from the original paper) to award monetary bonuses, as well as a newly-defined action `FreeDays` to award free working days to the workers.

The output of `a:FreeDays` is the set of workers who accepted the 3 free days offered. However, due to a complement edge (⇸) connecting `a` and `b`, the output set of `a` is subtracted from the original input set. Therefore, the input of `b:RewardAtEndProject` are only those workers who declined to accept working days as award, and want to be evaluated at the end of project and paid a bonus according to their performance.

Incentive Mechanism ▭

`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. As a self-sufficient and independent unit, it does not have any inputs or outputs. It can be stored and reused through instantiations with different runtime parameters. 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

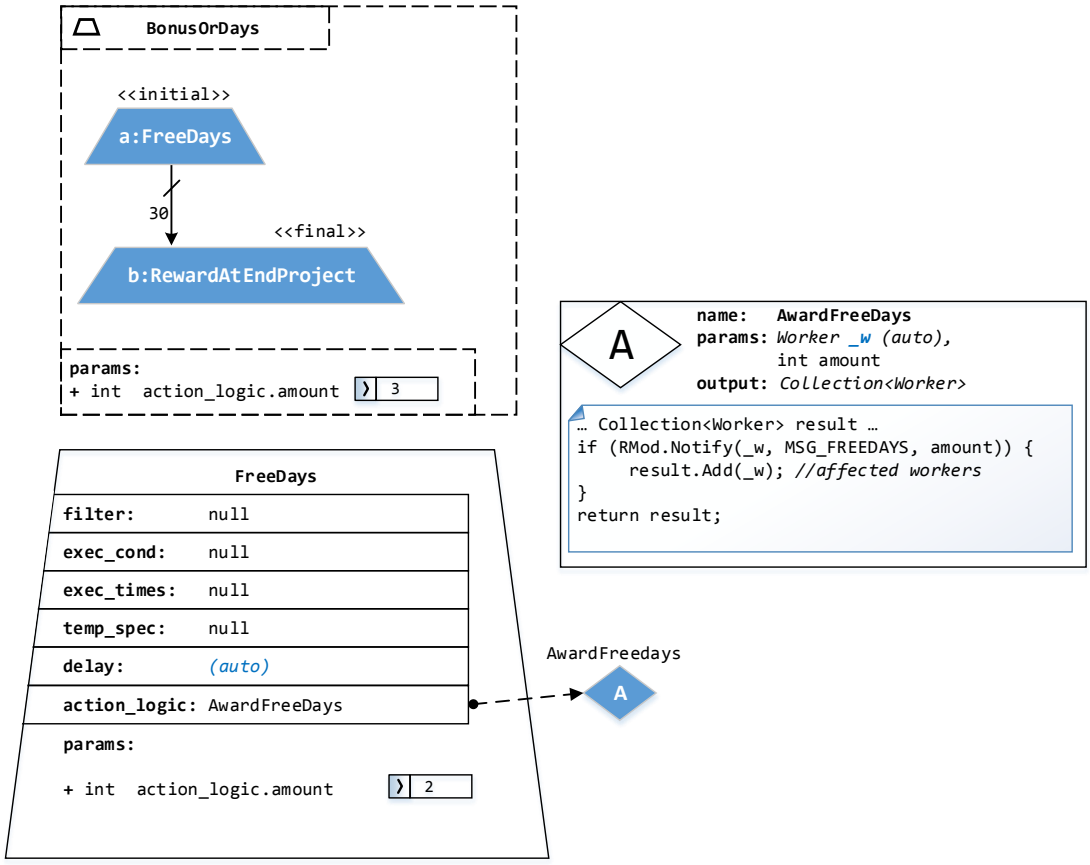

Figure 7.8: A `CompositeRewardingAction` letting the workers choose one of the rewards.

definition and instantiation (Fig. 7.9), as well as a shorthand notation used in this paper – IM. Table 7.5 defines the functionality of IM's fields. We show examples of the usage of IMs and other incentive elements in the following section.

Incentive Schemes (Incentive "Programs")

The *incentive strategy* is the whole of business logic needed for managing incentives in an organization. The strategy in PRINGL is built bottom-up, by first defining small, reusable chunks of business logic as different complex types. When compiled, the new types are stored to the *incentive library* (Fig. 7.1) with fully qualified names in hierarchical namespaces. From the library they can be instantiated with different parameters and reused in definitions of other complex types, including incentive mechanisms. The mechanisms are combined to obtain an *incentive scheme* (Fig 7.10, ⑤) – a set of high-level rules representing the incentive strategy. An incentive scheme is the equivalent of a visual DSL program. As any program, it can be run with different parameters and used on different systems with similar characteristics.

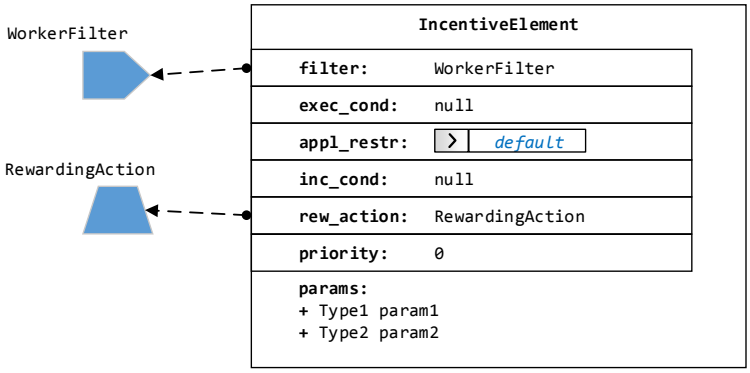

Figure 7.9: An example `IncentiveMechanism` definition.

Field	Description
`exec_cond`	An optional ⟨P⟩ 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 ⟨A⟩ 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 ⟨P⟩ 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 ⟨A⟩ assigning the reward or penalty.
`priority`	An optional `int` indicating the priority of mechanism's execution. Zero by default.

Table 7.5: Description of `IncentiveMechanism` fields.

Figure 7.10 shows how incentive strategies are constructed. First, missing or specific business logic fragments are defined and compiled into appropriate `IncentiveLogic` elements (Figure 7.10, ①). In the following steps, after being visually declared, these and other existing library elements can be instantiated for use in definitions of `SimpleWorkerFilters` and `SimpleRewardingActions` (Fig 7.10, ②). Similarly, filter and rewarding action type definitions are further used for defining new composoite filters and actions (③) and `IncentiveMechanisms` (④).

Moving up from step ① towards ⑤ the need of knowing PRINC/ PRINGL internals decreases and reduces to understanding the meaning of exposed runtime parameters on a purely visual dashboard. Steps ① - ④ can be skipped altogether if the necessary type definitions are already available from the library. The goal of PRINGL is exactly to promote the reusing of well-defined and common business logic related to incentive management.

Using the graphical elements the user specifies the necessary runtime parameters for

different instances he uses. The GUI environment collects the parameters from all the constituent sub-components and propagates them upwards, possibly until the top-most component's graphical form.The user sets through the GUI whether to propagate a parameter (`+`/`-` symbols, Fig 7.10), and therefore delegate the responsibility for filling it out to an upper level, or provide a value at the current level and hide it from upper layers. If a propagated parameter is supplied with different values on different levels, then the rule is that the topmost value overrides all the others. For example, if a parameter is propagated from the ⟨L⟩ level (①) to the incentive scheme level (⑤), then the value defined at the level ⑤ is used.

This is possible to do for all the elements that are used to perform predetermined roles (e.g., *rewarding action* of an IM, or the *auxiliary logic* of a F⟩). In this case the runtime environment itself creates the instances and can therefore pass the parameters from the GUI. When the environment does not control the creation of instances (e.g., when `IncentiveLogic` elements are declared for arbitrary use from code) the programmer must set them in the provided code directly.

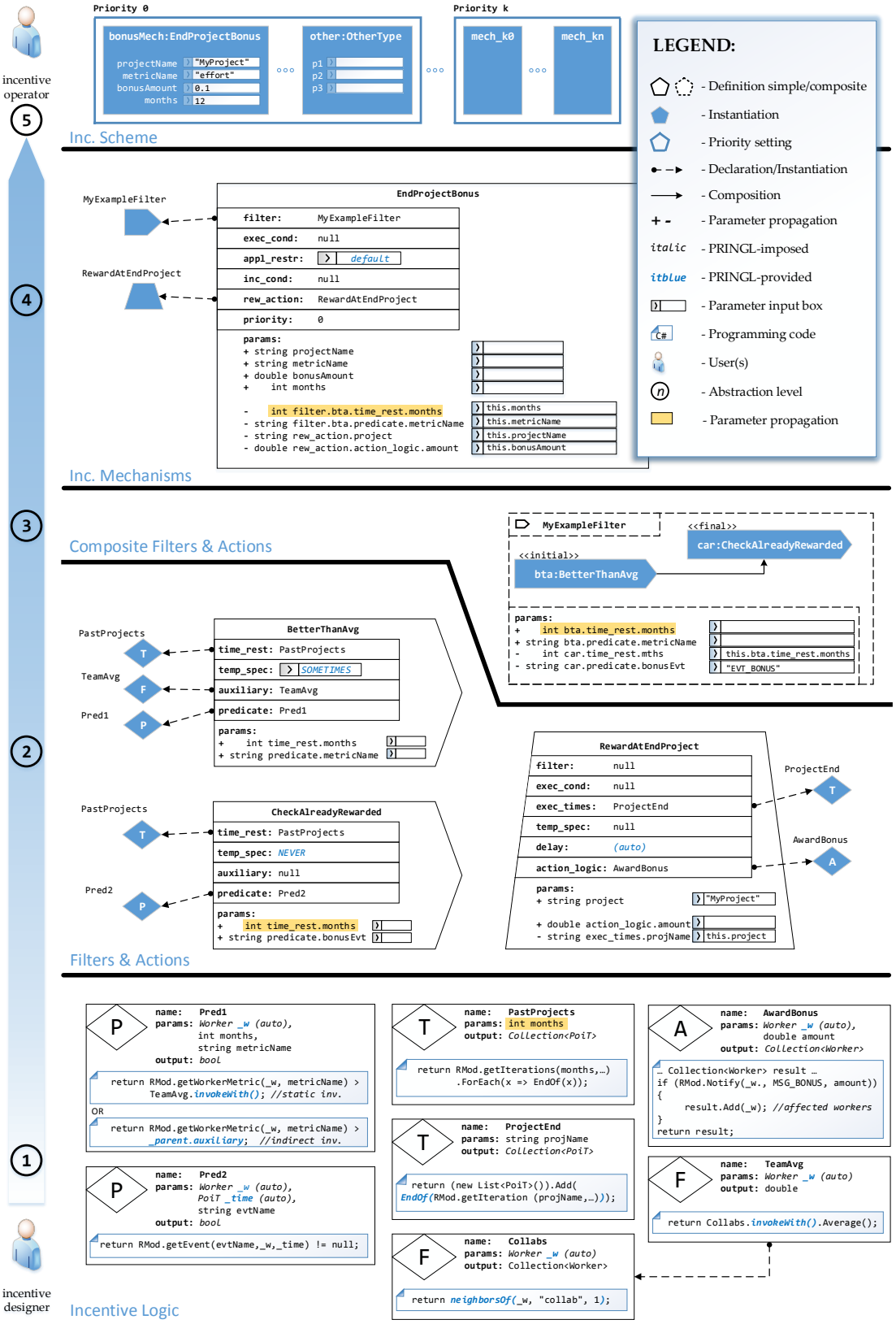

Figure 7.10: Incentive scheme from Example 3, illustrating the decreasing of complexity going from modeling of (low-level) incentive elements by incentive designers to adjusting existing incentive schemes by incentive operators.

7.3 Execution Model

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

All IMs 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 IMs is achieved. The order of execution of IMs with the same priority is not predetermined. Execution conditions of the IMs with higher priorities are evaluated first. Only after the higher-priority IMs 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 IM is limited by design to a maximum time T_{IM}^{max}. It is the time needed to pass the message to the underlying crowdsourcing platform. Therefore, a single execution cycle of an incentive scheme of n mechanisms can last at most $T_{sc}^{max} = n \times T_{im}^{max}$. It is necessary that $T_{sc}^{max} < T_{tck}$, where T_{tick} is the basic time of the abstraction interlayer (*tick* in case of PRINC). The execution of an IM 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 IM'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` (T). 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 sub-filter (marked `«initial»`) 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»` sub-filter is returned as the overall result. The `«initial»` filter is given different default inputs by the PRINGL environment depending on where the composite filter is instantiated. The anonymous `:Passthru` sub-filter instances are special PRINGL sub-filter types passing the union of workers from all input edges onto all output edges without performing any filtering.

A simple A is executed if the `exec_cond` (P) returns true. In this case, the execution `PoiT`s for the action are obtained from `exec_times` (T) 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` (A). 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 IM '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 ⟨T⟩ 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 ⟨L⟩ equals to invoking the instance similarly to a conventional function. The environment passes both the auto parameters and any user-defined ones. If user-defined parameters are omitted when a ⟨L⟩ 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.

Parameters are collected and propagated automatically from instances created to fulfill complex type field roles. In that case the runtime environment controls the instantiation and therefore knows to which instances to pass the parameters from the GUI. When the environment does not control the creation of instances (e.g., when `IncentiveLogic` elements are declared for arbitrary use from code) the programmer must set them in the provided code directly.

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.

7.4 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 [MH10], since this type of evaluation includes measuring characteristics such as productivity and subjective satisfaction, that require an established community of regular users [SDKP06].

During the initial development and prototyping phase, the common approach is to use the *qualitative evaluation* [MH10], 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 and presented in Chapter 3. 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.

The qualitative evaluation of PRINGL is performed with respect to the language requirements elicited in Section 7.1.1. We constructed an example suite covering realistic incentive elements identified in Chapter 3. By implementing and analyzing different incentive use-cases from the suite we showcase the usage of PRINGL and argue for the coverage of the requirements. Concretely, the requirements are evaluated as follows:

- The diversity of examples in the suite and the fact that they were obtained from the broad survey of realistic incentive practices testify for PRINGL's *groundedness* and *expressiveness*.
- Through elaborate discussion of particular implementation details of different suite examples 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 conclusively argue for certain aspects of usability, such as '*usefulness*' and '*portability*' (as defined in [SDKP06]).

Table 7.6)[5] shows the coverage of the chosen examples with respect to introduced incentive categories and their constituent parts. Some examples are presented partially to illustrate/highlight the claimed capabilities that the particular example is supposed to cover.

7.4.1 Example 1 – Employee Referral

A company introduces employee referral process[6] *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.*

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 Figure 7.11 we show how the new composite `ReferralFilter` is constructed. The 𝔽⟩ instance `n:NewlyEmployed` makes use of: a) ⟨𝕋⟩ `PastMonths` returning `PoiT`s representing end-of-month time points for the given number of months (12 in this particular case); and b) predicate ⟨ℙ⟩ `Pred2` checking if the employee got hired 12 months ago. `Pred2`'s general functionality is to check whether

[5] Note that the Indirect and Subjective evaluation methods have been omitted from Table 7.6. 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.

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

	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		✓			

Table 7.6: Coverage of incentive categories, rewarding actions and evaluation methods by the provided examples.

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.

7.4.2 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. 7.12 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 (◇P◇ `PeerVoteDone`). IM `PeerAssessIM` uses the F◇ `TeamMembers` to exclude the manager from the rest of team mambers. The `TeamMembers` is a composite filter composed of two subfilters F◇ `GetManager` F◇ `GetTeam`, borrowed from Ex.5, Fig. 7.15. The resulting workers are passed to /A\ `DoPeerVote` which performs the actual functionality of peer

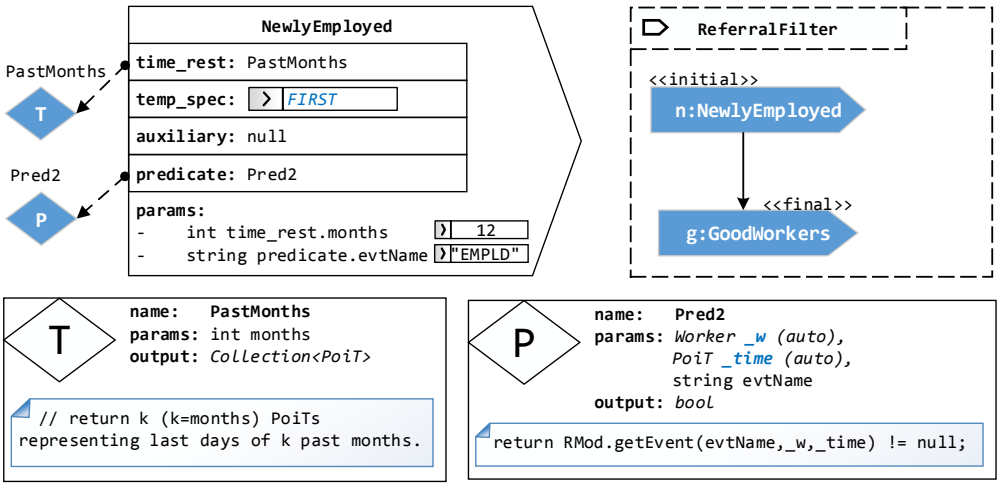

Figure 7.11: A `CompositeWorkerFilter` for referral bonuses.

voting. The referenced rewarding action is simple; it just passes to ⟨A⟩ `PeerVote` the workers that need to participate. The ⟨A⟩ `PeerVote` is performed by dispatching messages to the workers and receiving and aggregating 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 [IM] `i2`. Once set to execute, the [IM] `i2` first reads all the team members via [F] `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 ⟨P⟩ elements (Sec. 7.2.2). The condition expresses the two constraints from the incentive formulated in natural language. If it resolves to 'true', the /A\ `DoRewardTeam` applies a predefined monetary reward, sharing it equally among all team members (via ⟨A⟩ `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 [ZSTD14] we presented SmartCOM – a framework for virtualization and communication with human agents. In this example we model the latter option in PRINGL, assuming the use of PRINCwith SmartCOM for interaction with workers.

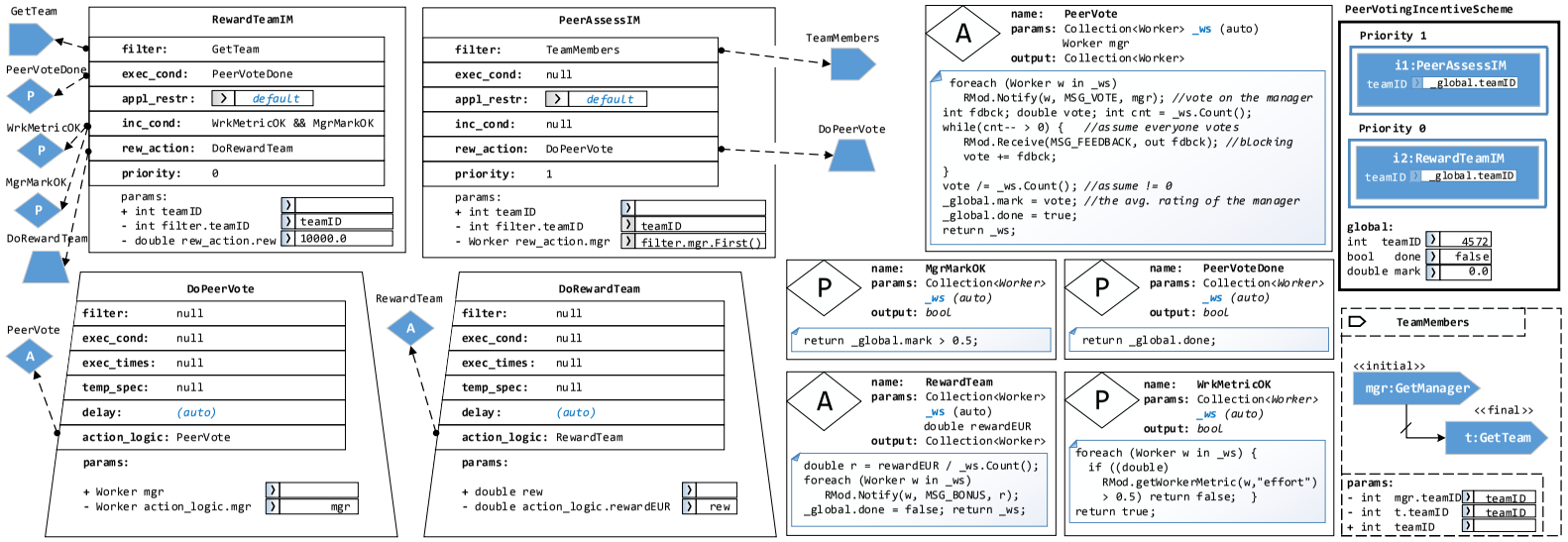

Figure 7.12: An incentive scheme example combining peer voting and team-based compensation.

7.4.3 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: Figure 7.10 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` ◁L▷ 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 𝔽▷ and /𝔸\ types. Similarly, 𝔽▷ and /𝔸\ definitions are further used for defining new composite filters and actions (③) and `IncentiveMechanism`s (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 sub-components 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. 7.10 we show an example of parameter propagation (marked in orange). Element ◁T▷ `PastProjects` (①) exposes the parameter `months`. The same parameter is then re-exposed by 𝔽▷ `BetterThanAvg` (②) that uses `PastProjects` as its time restriction. The parameter is further propagated up through 𝔽▷ `MyExampleFilter` until it finally gets assigned the value in 𝕀𝕄 `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 ◁L▷ `PastProjects` is reused twice in two different 𝔽▷s. Also, steps Fig 7.10: 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 incentive logic, the incentive elements become completely portable.

7.4.4 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 over-simplified) 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* IM `EndProjectBonus` *in Ex. 3 (Fig. 7.10).*

Solution: Figure 7.13 shows the additional elements needed to support the new mechanism. The composite F> `NonRewardedOnes` reuses the existing F> `MyExampleFilter` 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 /A\ `RewardAtEndProject` from Ex. 3 and change only the value of the field `action_logic` to point to the the newly defined <A> `ShowRankings`, also shown in Fig. 7.13. Let us name the newly obtained /A\ `RankingsAtEndProject`. In the same fashion, we copy-paste the existing IM `EndProjectBonus` from Ex. 3, make its `filter` and `rew_action` fields point to the newly defined F> `NonRewardedOnes` and /A\ `RankingsAtEndProject`, respectively. The obtained IM 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.

7.4.5 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.[7]

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.

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`

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

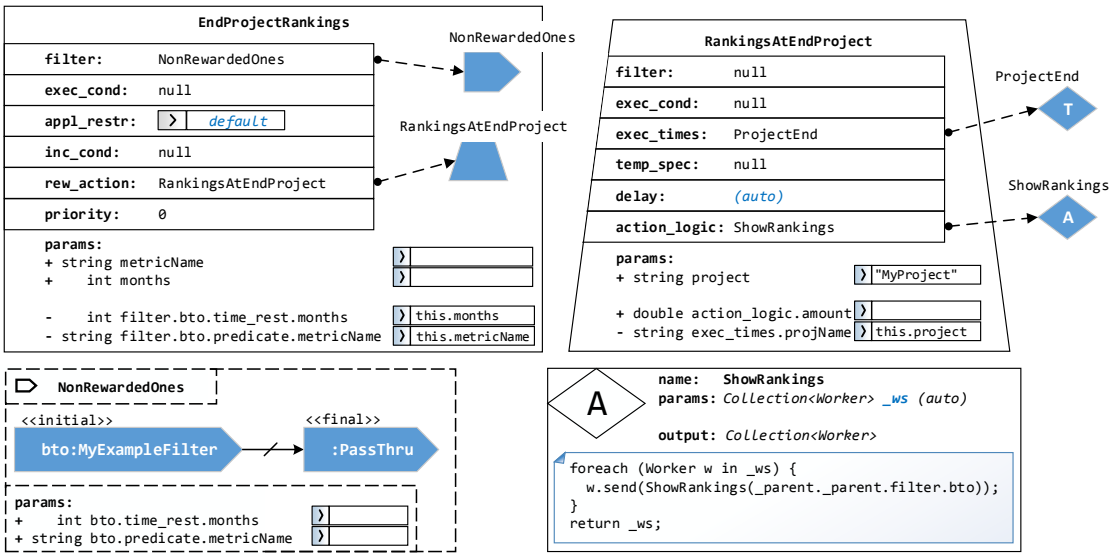

Figure 7.13: Additional incentives elements needed to augment the incentive scheme from Example 3 (Fig. 7.10) in order to display motivational rankings to the non-rewarded workers from Example 3.

(Fig 7.14, top right) containing (referencing) two IM 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 IM types `RewardBest` and `PreventTooLong`, respectively (Fig 7.14, bottom). The IM `RewardBest` installs the best worker as the new manager if (s)he is not the manager already. The IM `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 re-chaining of management relations in the structural model of the team by applying appropriate graph transformations [JBK10] through the abstraction interlayer.

When the incentive condition (`inc_cond` field) of IM `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 /A\ `RewSecondBest` and passing it the collection of workers returned by the F> `Candidates`. The F> `Candidates` returns potential candidates for the manager position – the best performing `Worker` and the current manager. The same filter is referenced from both IMs. However, the IM `PreventTooLong` invokes F> `Candidates` through a complex incentive condition field, referring to two <P> elements, which both

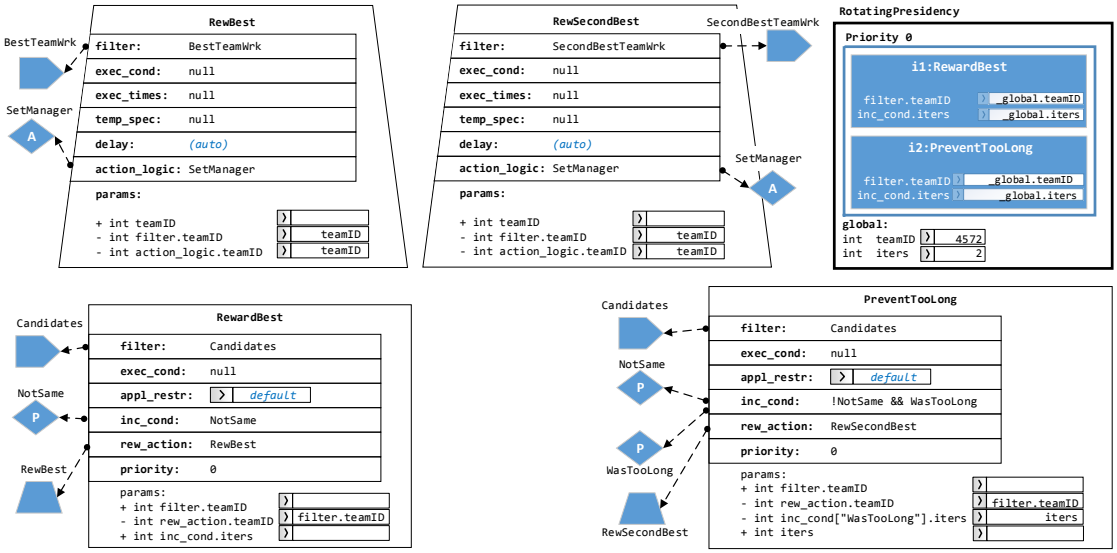

Figure 7.14: Modeling the rotating presidency incentive scheme in PRINGL. Segment showing the incentive scheme (top right), rewarding actions (top center and left), and incentive mechanisms (bottom).

need be visually declared. PRINGL allows this as a shorthand notation instead of forcing the user to create a container ⟨P⟩ element to perform the same logical function. In this case, the exposed parameters cannot be simply referenced by using the field name, but rather the parameters are accessed through an associative array (C# Dictionary) bearing the same name as the field, while the names of the used ⟨P⟩ elements serve as key names. For example, to access the ⟨P⟩ `WasTooLong`'s parameter `iters` from [IM] `PreventTooLong` where ⟨P⟩ `WasTooLong` is used in the `inc_cond` field, we must write: `inc_cond["WasTooLong"].iters` As it can be visually tiring to read the lengthy fully-qualified names of propagated parameters, we often stop propagating such parameters and propagate a new, local one with the same name, whose value we then copy to the long-named parameter (e.g., just `iters` instead of `inc_cond["WasTooLong"].iters`).

Both [IM]s get executed always as the nullified `exec_cond` fields default to `true`. However, [IM] `PreventTooLong`'s incentive condition (`inc_cond` field) contains: `!NotSame && WasTooLong`. It ensures that the /A\ `RewSecondBest` of [IM] `PreventTooLong` will never get executed at the same time as the /A\ `RewBest` of the [IM] `RewardBest`.

Two rewarding actions are instantiated and invoked from the [IM]s. The /A\ `RewBest` monitors the 'effort' metric and rewards the best worker in the current iteration. The /A\ `RewSecondBest` replaces the current team manager with the second-best performing worker when needed. The [IM] `inc_cond` fields make sure that the two actions do not get executed in the same iteration. The fact that a rewarding action instantiates its own `filter` means that it discards the workers passed to it by the PRINGL environment from the encompassing [IM]'s `filter` field and rewards those returned by the local filter.

In both actions most fields are nullified, meaning that the PRINGL execution envi-

ronment will assume the default field value. This means that the `action_logic` ⟨A⟩ `SetManager` will be unconditionally scheduled for execution.

We now show how the previously referenced filters are defined. We will first describe the definitions of the three simple filters (Fig 7.15, right) and then use them to visually assemble the definitions for another four composite filters (Fig 7.15, left).

- `GetTeam`: Returns all the workers belonging to the team with the specified `teamID`. The filtering is performed by running each of the workers from the input set against the `predicate` ⟨P⟩ `IsTeamMember` and including it in the output if fulfilling the predicate.

- `GetBest`: Returns the worker having achieved the highest value of the 'effort' metric by invoking the ⟨F⟩ `GetWrkBestMetric` and then just formally matching it with the `IsBest` predicate. In this example we use the 'effort' metric [RTD14], 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 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.

  In our example this filter encapsulates and hides the metric it uses for evaluating the workers. In principle, it would make sense to propagate the metric name upwards and thus make it user-settable, consequently making the whole scheme more general. However, for readability purposes we decided not to propagate this parameter in this example.

- `GetManager`: Invokes a ⟨F⟩ `GetMgrByRelations` that performs a graph query [JBK10] on the team model through the abstraction interlayer to determine the manager within the provided input set of workers.

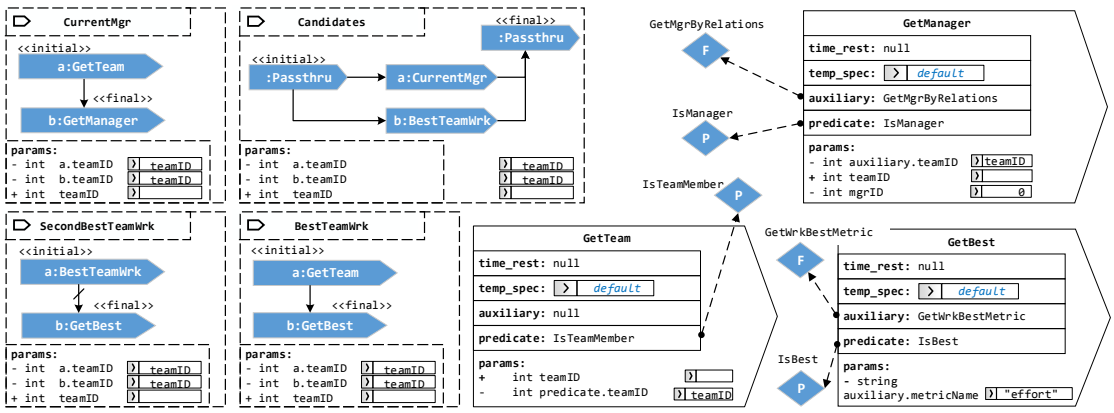

Figure 7.15: Modeling the rotating presidency example: Segment showing simple filters (right) and composite ones (left).

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

- `CurrentMgr`: Returns the current manager of the team. The 𝔽⟩ `a:GetTeam` returns all the workers belonging to the team with the `teamID`, while the 𝔽⟩ `b:GetManager` uses managerial relationships to determine the manager among those workers[8].
- `BestTeamWrk`: Returns the best individual from a previously identified collection of team members. The 𝔽⟩ `b:GetBest` determines what 'best worker' means in this case.
- `SecondBestTeamWrk`: As the name suggests, 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 edge ($\nrightarrow$). This means that `b` now receives as input: *input(a)* $\setminus$ `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.

[8]While managerial relations in principle need not be stored as a graph, and can thus be identified much more easily, we still use the graph managerial relations as an easily understandable example of how any graph-encoded structural property can be used in incentive management.

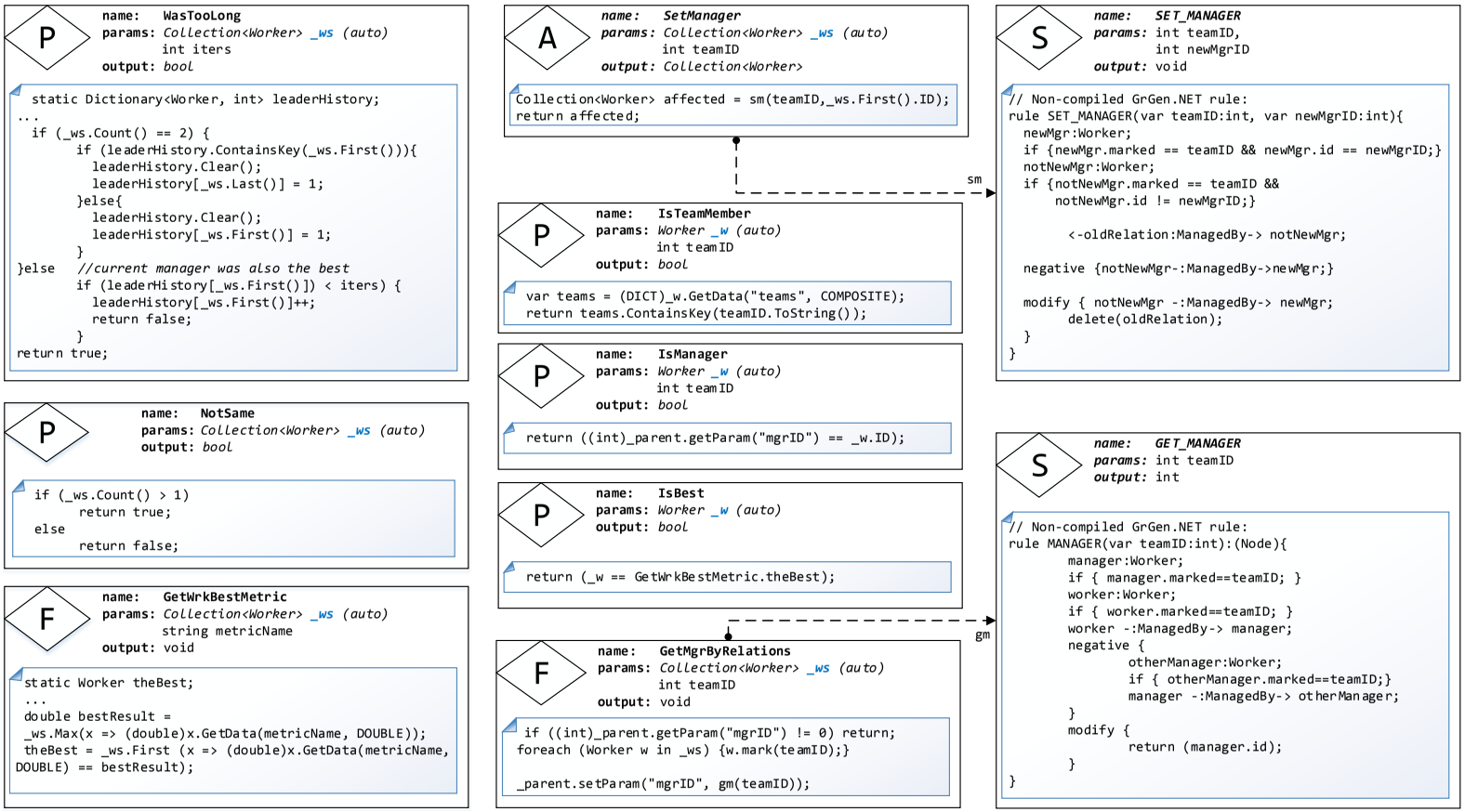

Figure 7.16: Modeling the rotating presidency example: Segment showing the incentive logic elements.

Incentive logic elements, shown in Figure 7.16, contain the low-level business logic and code[9] 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 7.7.

Element	Symbol	Description
`IsTeamMember`	⟨P⟩	Determines whether a worker belongs to a team.
`IsManager`	⟨P⟩	Checks if the currently evaluated worker has the ID previously determined to belong to the team manager by ⟨F⟩ `GetMgrByRelations`.
`IsBest`	⟨P⟩	Checks if the currently evaluated worker is the same as the one identified by the `GetWrkBestMetric`.
`NotSame`	⟨P⟩	Determines if the input contains two manager candidates.
`WasTooLong`	⟨P⟩	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`	⟨F⟩	Reads the value of the `'effort'` metric for each of the passed workers in `_ws` and updates the best worker.
`GetMgrByRelations`	⟨F⟩	Invokes the read-only structural query ⟨S⟩ `GET_MANAGER`.
`SetManager`	⟨A⟩	Invokes the modifying structural query ⟨S⟩ `SET_MANAGER`.
`GET_MANAGER`	⟨S⟩	Contains a compiled non-modifying GrGen.NET graph query, here expressed in GrGen rule language. The rule only considers the nodes marked by the `teamID` tag (see `GetMgrByRelations`). The rule matches and returns a node that other nodes point to via `ManagedBy`-typed relations, but itself is not managed by another team member.
`SET_MANAGER`	⟨S⟩	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.

Table 7.7: Incentive logic elements used in the rotating presidency example.

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 7.14, top) are almost identical, differing only in the filter they use – with former using the ⟨F⟩ `BestTeamWrk` and

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

the latter the 𝔽⟩ `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 ⟨𝔸⟩ 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 ⟨𝔸⟩ from the /𝔸\'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 𝔽⟩ `GetManager` may be available with a default `auxiliary` field ⟨𝔽⟩ 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 ⟨𝔽⟩ with a structural one, such as `GetMgrByRelations`.

7.5 Implementation

Figure 7.1 (Sec. 7.1) shows the overview of implemented components (outlined in blue). PRINGL's language metamodel was implemented in Microsoft's Modeling SDK for Visual Studio 2013 (MSDK). The metamodel is provided in Appendix B.I. Source code, screenshots and additional info is available here[10] 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 (Figure 7.17). The graphical elements provided in the implemented Visual Studio PRINGL environment, although not as visually appealing as those presented in this paper, functionally and structurally match them fully. PRINGL models are stored in `.pringl` files that get automatically transformed to the corresponding C# (`.cs`) equivalents. The generated code can then be used in the rest of the project as regular C# code or compiled in .NET assemblies (e.g., libraries or executables).

As a proof of concept, demonstrating the feasibility of implementation of the introduced programming and execution model, we implemented the "rotating presidency" example (Ex. 5) from Section 7.4.5. Figure 7.17 shows a screenshot of implemented rotating presidency example using the VS PRINGL IDE as well as the intended use of generated code artifacts. The implemented incentive elements correspond to the individual element descriptions presented in Section 7.4.5 (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.

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

The implemented example supports an arbitrary number and structure of `Worker`s (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. The PRINGL source code (.pringl file) for this example is provided in the Appendix B.II.

7.6 Discussion

In Sections 2.2 and 4.3 we explained the necessity of composition and frequent adjustments of incentive elements and mechanisms. In this respect, the roles of Incentive Designer and Incentive Operator may be seen as critical to the success of any future socio-technical platform. For companies concerned to control costs having an Operator in the loop controlling the cost of incentives and adapting them may be the decisive factor for the adoption of incentive management. An Operator can also provide early warning of dysfunctional behavior of workers.

PRINGL's design requirements were formulated to respond to these demands and make PRINGL a useful tool for the Incentive Designer/Operator. The evaluation performed so far indicates that PRINGL is able to fulfill adequately most of the listed requirements. However, it is also clear that the objective usability of the language needs to be tested and quantitatively proven in field. This would include training a number of incentive designers/operators to code in PRINGL and running a user study to gather subjective opinions about its applicability, usefulness, expressiveness, speed/ease of programming. As this is not possible without a fully functional socio-technical platform, we were prevented from performing it so far. However, we regard this as an important step in our future work.

The language itself is independent of the size of workforce and the number of incentive actions it needs to manage (as these are of concern to the underlying components, such as RMod), so scalability is not a potential issue. The evaluation of PRINGL's performance was not of interest to us at this phase, as it primarily depends on the technologies and components used in the incentive management platform, some of which are at a prototype phase. Furthermore, in absence of any related domain-specific languages or modeling approaches, no comparative analysis was possible.

The remaining limitations are discussed in Section 8.2 in relation to the overall incentive management platform.

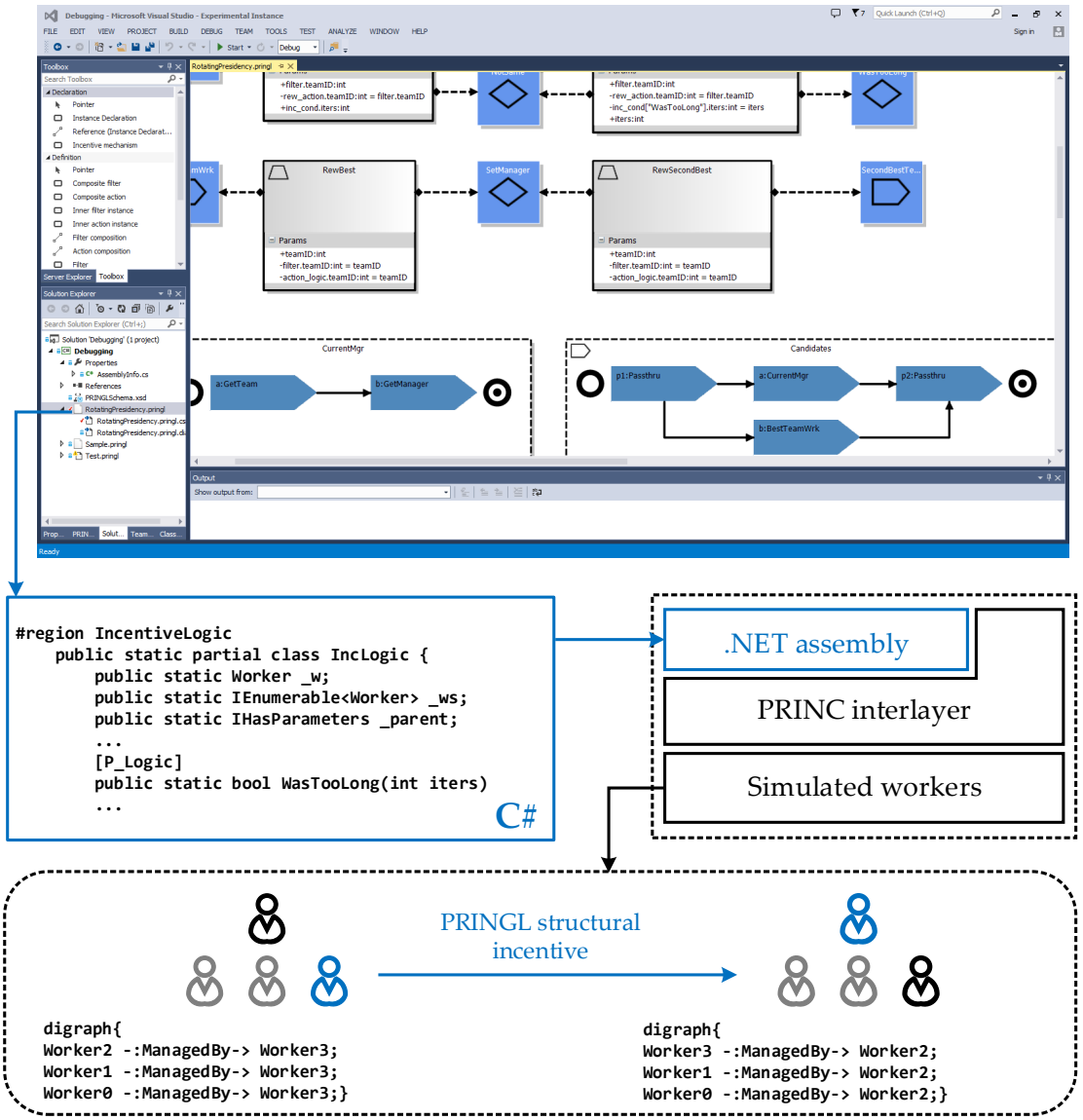

Figure 7.17: Implementing the rotating presidency incentive scheme (Example 5) using generated PRINGL Visual Studio environment. Generated C# code is performs calls to PRINC APIs, which ultimately perform structural changes on the worker graph (part of RMod).

CHAPTER 8

Conclusion & Research Outlook

8.1 Discussion

The remarkable success in widespread adoption and use of social networks made it evident that large numbers of geographically distributed people can successfully engage with information systems and other people using the information systems as intermediaries. Initially, the intermediating role was limited to facilitating communication between people only. With time, however, it became apparent that people could be engaged in (large-scale) productive and collaborative activities. The term *crowdsourcing* was coined to describe the process of performing tasks by splitting them into a number of independent subtasks and assigning them across a set of independent workers managed by an information system (crowdsourcing platform). Even though the precise definitions of 'crowdsourcing' may differ, in practice the term remained limited to (potentially large) but easily parallelizable task being performed by largely unqualified workers. The probable explanation for this is that the successful commercial exploitation of crowdsourcing required early adopters, and attracting paying customers was more likely if the prices for performing a task were considerably lower than engaging a team of professionals to perform it. This also made the associated risks of failure more acceptable to the customers. However, this also meant that when the total amount would get split among all the performing workers, the wages pro worker remained extremely below the average hourly wage for most parts of the world. This in turn meant that tasks processed via crowdsourcing platforms remained simple and non-challenging compared to types of tasks performed in traditional (non-virtual) companies and organizations, since the majority of workforce was not schooled/trained enough to perform such tasks. In addition, the platforms themselves did not offer the technical facilities that would enable more complex coordination, communication and socialization among workers, or any hierarchies or possibilities of advancement.

However, even with such simplistic workforce management, crowdsourcing systems started employing a number of incentive mechanisms, some of them being adapted forms

of traditional corporate incentives, the others being web-specific (e.g., FourSquare badges) (Section 3.4).

At the same time, the research community was active in trying to design systems that would go beyond crowdsourcing, engaging humans (together with software services) in more complex collaborative patterns. As outlined in the roadmap paper [KNB+13], a number of innovative properties were identified as necessary to be supported by the emerging systems if they are to become attractive to a wider and more competitive workforce. Systems like [ABMK11, MB11, TDKC15, SMS+15] are among the first examples of systems supporting some of these properties. They are often described with more generic terms, such as *socio-technical systems* or *(hybrid) collective adaptive systems* to emphasize support for a richer set of functionalities, types of workers/participants and collaborative patterns than those supported by crowdsourcing systems.

As the complexity of human participation increases, it exerts more influence on the socio-technical environment, because the overall execution depends ever more on human behavior which is inherently difficult to predict, formally describe and control. Furthermore, humans are highly adaptive, driven at different times and in different occasions by different motives (ranging from self-interest to charity). This poses a great challenge when trying to incorporate humans as processing units into existing orchestration and execution formalisms. While the existing approaches (Section 2.2) may work well for business processes which are prescribed and separated into single activities executed by roles, in cases where the collaboration is less constrained or not known in advance this may not be possible or effective. However, this is a problem that also traditional companies experienced when the increased complexity of labor made measuring the 'signals' difficult, triggering a number of disfynctional behaviors to emerge. We wrote about these in Section 3.1. The response in this case was the the development of a number of different incentive mechanisms and their combination into incentive schemes.

There is an ever-going discussion on the efficacy and justification of the use of incentives. As elaborated in Section 2.1.3 different empirical findings demonstrated that specific incentives do effectively block certain disfunctional behaviors. Also, it was shown that incentives exhibit secondary selective effects, making them even more important in large-scale collaborations as a testimony of the quality of workers' performance. The fact that around 80% of big and medium-sized US companies employ incentives [Feh13] is a good indicator of their effectiveness. On the other hand, our own survey (Section 3.4.2) showed that less than 10% of the surveyed web-based and crowdsourcing companies employed incentives. The majority of those that did employ them used a very limited number of incentive mechanisms.

The survey also showed that the current commercial landscape is dominated by crowdsourcing platforms dealing with simple tasks, which therefore require less complex incentives, because the signals are more easily measurable. This leads to a kind of a negative feedback loop, where the lack of incentive management tools limits the companies in offering more advanced incentive schemes which might motivate and attract more professional workers and enable processing of more complex tasks. In order to break the negative loop a company would have to develop not only the entire functionality for

complex task processing, communication,coordination and orchestration, but also the full incentive management functionality. All of the listed functionalities are challenging research problems, appreciated even more by the author of the thesis through his participation in the SmartSociety project. This thesis was an attempt to describe, analyze and provide solutions for the latter of the listed functionalities – the incentive management.

The main idea of the research effort behind this thesis was to bring the benefits of incentive management closer to the emerging socio-technical systems and provide a set of tools for defining and applying incentives, composed of existing/proven incentive elements, thus relieving the socio-techincal platforms (or the companies) of this task. The incentive elements were based on existing practices in traditional and crowdsourcing companies. They were designed to be as portable and as reusable as possible, allowing us to offer incentive management as an external service to various platforms. Once again, it must be emphasized that the focus was not on designing novel incentive mechanisms, nor evaluating the effectiveness of existing ones in concrete situations.

A prominent contribution of this thesis is the domain-specific language PRINGL, being the one part intended for interaction with users, and thus reflecting design decisions taken when modeling incentive elements. As such, it required most time to develop, and was, accordingly, given most space in the thesis. As explained in Section 7.4, the language was evaluated qualitatively through a number of real-world examples. A quantitative usability evaluation is expected to follow after the planned integration with the SmartSociety platform (Sec. 8.3). This will allow the full end-to-end evaluation. Currently, all the contributions of the thesis were implemented and evaluated at prototype level.

Another important aspect that is often neglected when dealing with incentives in the research domain is the notion of privacy and ethics. The envisaged complex socio-technical systems will inevitably deal with a lot of personal data and influence the professional careers of the participating workers. Therefore, providing methods to implement incentives following ethical guidelines while handling personal information transparently becomes of great importance. While former is a research problem that we intend to deal with in the future, the foundations for the latter were already set in this work through the introduction of the SMARTCOM middleware that was designed in the context of the SmartSociety platform. The use of such a middleware in the abstraction interlayer is crucial for offering incentives as a service, because it adds a guarantee that the externalized incentive mechanisms will not be able to misuse or access unwanted data, contributing potentially to an easier adoption of the concept of incentives-as-a-service in the future (in the same way as PayPal speeded up the adoption of online payments).

8.2 Limitations

8.2.1 Artifact-centric incentives

The expressiveness of the language and the incentive model coincides with the incentive elements identified in the survey. This means that although we can reasonably argue

that the introduced model covers most contemporary incentive mechanisms used both in traditional companies and crowdsourcing platforms, no claim of universal coverage can be made. Furthermore, as existing incentives used in socio-technical systems are derived from the incentives used in conventional companies they are always associated with humans that they target[1] (i.e., their behavior). While this approach works well for processes similar to traditional business processes and/or platforms that have the goal of establishing a long-term relationship with the workers (like SmartSociety), in cyber-physical environments the number of digital artifacts is often exceeding the number of participants. In such cases, interactions with workers may be highly irregular and transient. Therefore, incentives targeting workers might not achieve the expected result, or the choice of existing incentives becomes severely limited (mostly to Pay-Per-Performance incentives). Instead, in such environments, the digital artifact (and not the worker) is the long-lived entity, and can be used as the carrier of the incentive. We already identified and formulated this problem in [STD15a] and presented some ideas on how we can approach it. As stated in Section 8.3 this will be one of the topics of our long-term future work.

8.2.2 Automated incentive adjustment

Another potential limitation of the presented incentive management platform is that it is unable to automatically adjust incentives to the changes in monitored collaborative processes. Instead, the existence of an Incentive Operator as the ultimate source of expertise for the given application scenario is assumed. The Operator is provided with the necessary tools for incentive management (the platform) and is expected to perform the adaptation of incentive schemes through addition/removal (turning on/off) of specific incentive mechanisms and adjustment the parameter values. Certainly, as a general concept, designing an automated incentive scheme adjustment seems like a plausible idea. However, it is questionable how feasible it would be in practice.

If the adaptability were to be described with a set of (rule-based) adaptation policies, these would have to be based on some existing empirical data. However, no general quantifiable results are available (cf. [Feh13]). For example, it was observed that when approaching a quota threshold the performance of majority of workers increases, and then drops after surpassing the quota [Pre99], but one cannot make a general claim nor express in concrete terms how much the effort would change. Furthermore, these empirical findings are obtained under controlled circumstances with only one incentive in place. With multiple incentive mechanisms in place the findings could become irrelevant, or the experiments should be performed for the given combination of incentives.

An alternative idea would be to use machine learning from past worker behavior in similar circumstances to try to foresee the future behavior of workers and adjust the incentive mechanisms accordingly; or select workers that are expected to respond well to a fixed set of incentives. For example, for a fixed type of long-running tasks (such as software development), such an approach could be used on the same platform for new

[1] For that same reason, the $\mathbb{F}\rangle$ elements introduced in this thesis return only `Worker` objects.

tasks to be processed with new worker teams. As the work on a task is progressing, the incentive mechanisms can be adjusted based on previous experiences with similar tasks. Going back to the example used to introduce the role of Incentive Operator in Section 5.1, let us assume the task is to develop a new blogging web application. If the platform holds the records for a number of previous similar tasks (i.e., similarly-sized development teams working with same technologies under the same development methodology), in theory the decision to incentivize more quality rather than quantity after some point can also be made automatically. The necessary precondition for automated incentive adjustment is the existence of data-sets for training, which are currently missing, so this can be an area worthy of further investigation in the future.

8.2.3 Specificity of metrics

Portability of incentive mechanisms encoded in the incentive model proposed in this thesis is effectively limited by the portability/generality of metrics it relies upon. For example, the filter $\mathbb{F}\triangleright$`GetBest` introduced in Section 7.4.5 is portable as long as the effort metrics it relies upon is provided with consistent semantics by different platforms. This is a problem not specific to incentive management only. Other related areas dealing with context-specific, such as trust and reputation management, suffer from similar problems. The solution to this problem was often approached by building ontologies of trust-related entities and associated metrics (e.g., [BMP09, SNHB10]). The solution of this problem goes beyond the scope of this thesis. For the targeted scenarios, we assume the existence of fundamental performance and trust/reputation metrics, such as those introduced in [RTD14].

8.3 Future Work

In this section we briefly summarize the key points that were already identified throughout the thesis as the main directions of future work.

As a short-term goal, we foresee the addition of an incentives-as-a-service component to the SmartSociety platform, consisting of the PRINC frameworkk with a PRINGL-based frontend, connected with the SmartSociety platform over SMARTCOM middleware acting as the abstraction interlayer. The full integration will further require: *a)* the definition of incentive scheme(s) for one or multiple scenarios supported by the SmartSociety platform, derived from case studies performed by project partners [Del15]; *b)* integration of high-level constructs for controlling the interaction with the incentive service from the SmartSociety programming model that is currently under development [SSD+15]; and *c)* running an in-field functional evaluation of the fully integrated system.

Medium-term goals would include the full (more than a prototype) implementation of the PRINGL code generator, offering the complete described language expressiveness to the incentive Designer along a platform to run the modeled schemes. This will allow running a user study to evaluate the usability of PRINGL DSL quantitatively and measure subjective user satisfaction, as described in Section 7.4.

As long-term directions for further development, we foresee extending the incentive model and PRINGL's programming model to support definition and execution of artifact-based incentives, and investigating the feasibility of automated incentive adjustment. Another interesting research direction is the personalization of incentives. This is mostly interesting for psychological incentives, since intrinsic motivation may vary significantly across individuals, social and cultural groups, limiting the efficacy of psychological incentives.

Bibliography

[ABMK11] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. The jabberwocky programming environment for structured social computing. In *Proc. 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 53–64. ACM, 2011.

[ABS+14] Vasilios Andrikopoulos, Antonio Bucchiarone, Santiago Gomez Saez, Dimka Karastoyanova, and Claudio Antares Mezzina. Towards modeling and execution of collective adaptive systems. In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 69–81. Springer, 2014.

[Ada11] Eytan Adar. Why i hate mechanical turk research (and workshops). In *Proc. of CHI'11 Workshop on Crowdsourcing and Human Comp.*, Vancouver, Canada, 2011. ACM.

[Arm10] Michael Armstrong. *Armstrong's Handbook of Reward Management Practice: Improving Performance Through Reward*. Kogan Page Publishers, London, 3rd edition, 2010.

[BBC+14] Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Andrea Mauri, and Riccardo Volonterio. Pattern-based specification of crowdsourcing applications. In *Proc. 14th Intl. Conf. on Web Engineering (ICWE) 2014*, pages 218–235, 2014.

[BCBM12] Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor. Automan: A platform for integrating human-based and digital computation. *SIGPLAN Not.*, 47(10):639–654, October 2012.

[BFGK13] Alessandro Bozzon, Piero Fraternali, Luca Galli, and Roula Karam. Modeling crowdsourcing scenarios in socially-enabled human computation applications. *Journal on Data Semantics*, pages 1–20, 2013.

[BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Transformation*, volume 2505 of *LNCS*, pages 402–429. Springer, 2002.

[BM98] M.C. Bloom and G.T. Milkovich. The relationship between risk, incentive pay, and organizational performance. *The Academy of Management J.*, 41(3):283–297, 1998.

[BMP09] KamaljitKaur Bimrah, Haralambos Mouratidis, and David Preston. Trust ontology for information systems development. In Chris Barry, Michael Lang, Wita Wojtkowski, Kieran Conboy, and Gregory Wojtkowski, editors, *Information Systems Development*, pages 767–779. Springer US, 2009.

[CTD13] Muhammad Z. C. Candra, Hong-Linh Truong, and Schahram Dustdar. Provisioning quality-aware social compute units in the cloud. In *11th Intl. Conf. on Service Oriented Computing (ICSOC 2013)*, Berlin, Germany, December 2-5, 2013. Springer.

[DB11] Schahram Dustdar and Kamal Bhattacharya. The Social Compute Unit. *Internet Computing, IEEE*, 15(3):64–69, 2011.

[Del15] Smartsociety consortium, deliverable 5.3 - specification of advanced incentive design and decision-assisting algorithms for cas. `http://www.smart-society-project.eu/publications/deliverables/D_5_3`, 2015.

[DEM12] Christoph Dorn, George Edwards, and Nenad Medvidovic. Analyzing design tradeoffs in large-scale socio-technical systems through simulation of dynamic collaboration patterns. In Robert Meersman, Hervé Panetto, Tharam S. Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi, and Isabel F. Cruz, editors, *OTM Conferences (1)*, volume 7565 of *Lecture Notes in Computer Science*, pages 362–379. Springer, 2012.

[DR85] E.L. Deci and R.M Ryan. *Intrinsic Motivation and Self-Determination in Human Behavior*. Plenum Press, 1985.

[DS14] J.P. Diller and S.S.M. Song. Method of and a system for ranking members within a services exchange medium, April 15 2014. US Patent 8,700,614.

[Feh13] Dennis D. Fehrenbacher. *Design of Incentive Systems*. Contributions to Management Science. Springer, 2013.

[FGP+09] G Frackowiak, M Ganzha, M Paprzycki, M Szymczak, Y Han, and M Park. Adaptability in an Agent-Based Virtual Organization – Towards Implementation. In José Cordeiro, Slimane Hammoudi, Joaquim Filipe, Wil Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, and Clemens Szyperski, editors, *Web Information Systems and Technologies*, volume 18, pages 27–39. Springer Berlin Heidelberg, 2009.

[FJ01] B.S. Frey and R. Jegen. Motivation crowding theory. *Journal of Economic surveys*, 15(5):589–611, 2001.

[GDM13] Fausto Giunchiglia, Biswanath Dutta, and Vincenzo Maltese. From knowledge organization to knowledge representation. In *ISKO UK Conference*, 2013.

[GMS14] Daniel G. Goldstein, Randolph Preston McAfee, and Siddharth Suri. The wisdom of smaller, smarter crowds. In *Proc. ACM Conf. on Economics and Computation*, EC '14, pages 471–488. ACM, 2014.

[GT05a] N Gilbert and K Troitzsch. *Simulation for the social scientist*. Open University Press, McGraw-Hill Education, 2005.

[GT05b] Nigel Gilbert and Klaus Troitzsch. *Simulation for the social scientist*. McGraw-Hill International, 2005.

[Gun06] Marjaana Gunkel. *Country-Compatible Incentive Design*. DUV, Wiesbaden, 2006.

[HA07] Bernhard Hoisl and Wolfgang Aigner. Social Rewarding in Wiki Systems – Motivating the Community. In *Proceedings of HCI International - 12th International Conference on Human-Computer Interaction (HCII 2007)*, pages 362–371. Springer, 2007.

[HDG00] Robert L. Heneman, Katherine E. Dixon, and Maria T. Gresham. Team pay for novice, intermediate, and advanced teams. *Advances in Interdisciplinary Studies of Work Teams*, (7):141–160, 2000.

[HF13] Shih-Wen Huang and Wai-Tat Fu. Don't hide in the crowd!: Increasing social transparency between peer workers improves crowdsourcing outcomes. In *Proc. SIGCHI Conf. on Human Factors in Comp. Systems*, CHI '13, pages 621–630. ACM, 2013.

[HHTG13] Matthias Hirth, Tobias Hoßfeld, and Phuoc Tran-Gia. Analyzing costs and accuracy of validation mechanisms for crowdsourcing platforms. *Math. and Comp. Modelling*, 57(11–12):2918 – 2932, 2013.

[HJCA+15] Mark Hartswood, Marina Jirotka, Ronald Chenu-Abente, Alethia Hume, Fausto Giunchiglia, Leonardo A. Martucci, and Simone Fischer-Hübner. Privacy for peer profiling in collective adaptive systems. In *Privacy and Identity Management for the Future Internet in the Age of Globalisation*. Springer, 2015.

[HM91] B. Holmstrom and P. Milgrom. Multitask principal-agent analyses: Incentive contracts, asset ownership, and job design. *JL Econ. & Org.*, 7(January 1991):24, 1991.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[HPTA14] M. Hosseini, K. Phalp, J. Taylor, and R. Ali. The four pillars of crowdsourcing: A reference model. In *Research Challenges in Information Science (RCIS), IEEE Intl. Conf.*, pages 1–12, May 2014.

[HRMF14] Ting-Kai Huang, Bruno Ribeiro, Harsha V. Madhyastha, and Michalis Faloutsos. The socio-monetary incentives of online social network malware campaigns. In *Proc. ACM Conf. on Online Social Networks*, COSN '14, pages 259–270. ACM, 2014.

[HS96] J.J. Heckman and J. Smith. What do bureaucrats do? The effects of performance standards and bureaucratic preferences on acceptance into the JTPA program. *Advances in the Study of Entrepreneurship Innovation and Economic Growth*, 7:191–217, 1996.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen. NET. *Intl. J. on Software Tools for Technology Transfer*, 12(3):263–271, 2010.

[Kau11] Sebastian Kaune. *Performance and Availability in Peer-to-Peer Content Distribution Systems: A Case for a Multilateral Incentive Approach.* Phd thesis, TU Darmstadt, 2011.

[KNB+13] Aniket Kittur, Jeffrey V. Nickerson, Michael Bernstein, Elizabeth Gerber, Aaron Shaw, John Zimmerman, Matt Lease, and John Horton. The future of crowd work. In *Proc. of the 2013 Conf. on Computer supported cooperative work*, CSCW '13, pages 1301–1318. ACM, 2013.

[KO02] Máire Kerrin and Nick Oliver. Collective and individual improvement activities: the role of reward systems. *Personnel Review*, 31(3):320–337, 2002.

[Laz00] E.P. Lazear. Performance Pay and Productivity. *American Economic Review*, 90(5):1346–1361, 2000.

[Laz07] E.P. Lazear. Personnel economics: The economist's view of human resources. *Journal of Economic Perspectives*, 21(4):91–114, 2007.

[LCMG09] Greg Little, Lydia B. Chilton, Rob Miller, and Max Goldman. Turkit: Tools for iterative tasks on mechanical turk. In *In Human Computation Workshop (HComp2009)*, 2009.

[Lit10] Greg Little. Exploring iterative and parallel human computation processes. In *Ext. Abstracts on Human Factors in Comp. Sys.*, CHI EA '10, pages 4309–4314. ACM, 2010.

[LM02] Jean-Jacques Laffont and David Martimort. *The Theory of Incentives.* Princeton University Press, New Jersey, 2002.

[MB11] Patrick Minder and Abraham Bernstein. Crowdlang - first steps towards programmable human computers for general computation. In *In Proceedings of the 3rd Human Computation Workshop (HCOMP 2011), AAAI-Press*, San Francisco, CA, USA, January 2011.

[MB12] Patrick Minder and Abraham Bernstein. Crowdlang: A programming language for the systematic exploration of human computation systems. In Karl Aberer, Andreas Flache, Wander Jager, Ling Liu, Jie Tang, and Christophe Guéret, editors, *Social Informatics*, volume 7710 of *LNCS*, pages 124–137. Springer, 2012.

[MD15] Pietro Michelucci and Janis L. Dickinson. The power of crowds. *Science*, 351(6268):32–33, 2015.

[MH10] Parastoo Mohagheghi and Øystein Haugen. Evaluating domain-specific modelling solutions. In Juan Trujillo, Gillian Dobbie, Hannu Kangassalo, Sven Hartmann, Markus Kirchberg, Matti Rossi, Iris Reinhartz-Berger, Esteban Zimányi, and Flavius Frasincar, editors, *Advances in Conceptual Modeling – Applications and Challenges*, volume 6413 of *LNCS*, pages 212–221. Springer, 2010.

[MHOG14] David Martin, Benjamin V. Hanrahan, Jacki O'Neill, and Neha Gupta. Being a turker. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '14, pages 224–235, New York, NY, USA, 2014. ACM.

[MKC+13] Andrew Mao, Ece Kamar, Yiling Chen, Eric Horvitz, Megan E. Schwamb, Chris J. Lintott, and Smith Arfon M. Volunteering Versus Work for Pay: Incentives and Tradeoffs in Crowdsourcing. In *Proc. First AAAI Conf. on Human Computation and Crowdsourcing*, pages 94–102, Palm Springs, CA, USA, 2013. AAAI.

[MM] Daniele Miorandi and Lorenzo Maggi. Programming social collective intelligence. *IEEE Technology and Society*, page (forthcoming).

[MN09] Charles M. Macal and Michael J. North. Agent-based modeling and simulation. *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 86–98, December 2009.

[MN10] Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[MSK02] Manfred Milinski, Dirk Semmann, and Hans-Jurgen Krambeck. Reputation helps solve the 'tragedy of the commons'. *Nature*, 415(6870):424–426, January 2002.

[Mum05] Dennis K. Mumby. Theorizing resistance in organization studies: A dialectical approach. *Management Communication Quarterly*, 19(1):19–44, 2005.

[MW09] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *Proc. ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 77–85. ACM, 2009.

[PCE10] Matthew J. Pearsall, Michael S. Christian, and Aleksander P. J. Ellis. Motivating interdependent teams: Individual rewards, shared rewards, or something in between? *Journal of Applied Psychology*, 95(1):183–191, 2010.

[Pre99] Canice Prendergast. The provision of incentives in firms. *J. of economic literature*, 37(1):7–63, 1999.

[Pri76] Derek De Solla Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, 27(5):292–306, September 1976.

[RD00] Richard M. Ryan and Edward L. Deci. Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*, 25(1):54 – 67, 2000.

[RHF13] Huaming Rao, Shih-Wen Huang, and Wai-Tat Fu. What will others choose? how a majority vote reward scheme can improve human computation in a spatial location identification task. In Björn Hartman and Eric Horvitz, editors, *Proc. of the First AAAI Conf. on Human Computation and Crowdsourcing, HCOMP 2013, November 7-9, 2013, Palm Springs, CA, USA*. AAAI, 2013.

[RKZF00] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems: Facilitating trust in Internet interactions. *Communications of the ACM*, 43(12):45–48, 2000.

[RM] Bettina Rockenbach and Manfred Milinski. The efficient interaction of indirect reciprocity and costly punishment. *Nature*, 444(7120):718–723.

[RTD14] Mirela Riveni, Hong-Linh Truong, and Schahram Dustdar. On the elasticity of social compute units. In Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors, *Advanced Information Systems Engineering*, volume 8484 of *LNCS*, pages 364–378. Springer, 2014.

[SDD13] Ognjen Scekic, Christoph Dorn, and Schahram Dustdar. Simulation-based modeling and evaluation of incentive schemes in crowdsourcing environments. In Robert Meersman, Hervé Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Deijing Dou, editors, *On the Move to Meaningful Internet Systems: OTM 2013 Conf.s*, volume 8185 of *LNCS*, pages 167–184. Springer, 2013.

[SDKP06] Ahmed Seffah, Mohammad Donyaee, Rex B. Kline, and Harkirat K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Control*, 14(2):159–178, June 2006.

[SHY+08a] Kenichiro Sato, Ryo Hashimoto, Makoto Yoshino, Ryoichi Shinkuma, and Tatsuro Takahashi. Incentive Mechanism Considering Variety of User Cost in P2P Content Sharing. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5. IEEE, 2008.

[SHY+08b] Kenichiro Sato, Ryo Hashimoto, Makoto Yoshino, Ryoichi Shinkuma, and Tatsuro Takahashi. Incentive Mechanism Considering Variety of User Cost in P2P Content Sharing. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5. IEEE, 2008.

[Ski54] Skinner, b. f. science and human behavior. new york: The macmillan company, 1953. 461 p. *Science Education*, 38(5):436–436, 1954.

[SKM04] Dirk Semmann, Hans-Jürgen Krambeck, and Manfred Milinski. Strategic investment in reputation. *Behavioral Ecology and Sociobiology*, 56(3):248–252, July 2004.

[SMS+15] O. Scekic, D. Miorandi, T. Schiavinotto, D.I. Diochnos, A. Hume, R. Chenu-Abente, H.-L. Truong, M. Rovatsos, I. Carreras, S. Dustdar, and Giunchiglia F. Smartsociety – a platform for collaborative people-machine computation. In *Proc. of the 8th IEEE International Conference on Service Oriented Computing and Applications (SOCA'15)*, Rome, Italy, October 2015.

[SNHB10] Wanita Sherchan, Surya Nepal, Jonathon Hunklinger, and Athman Bouguettaya. A trust ontology for semantic services. *2014 IEEE International Conference on Services Computing*, 0:313–320, 2010.

[SRTD14] Ognjen Scekic, Mirela Riveni, Hong-Linh Truong, and Schahram Dustdar. Social interaction analysis for team collaboration. In Reda Alhajj and Jon Rokne, editors, *Encyclopedia of Social Network Analysis and Mining*. SpringerScience+Business Media, NewYork, 2014.

[SSD10] Florian Skopik, Daniel Schall, and Schahram Dustdar. Modeling and mining of dynamic trust in complex service-oriented systems. *Information Systems*, 35(7):735–757, November 2010.

[SSD+15] O. Scekic, T. Schiavinotto, D.I. Diochnos, M. Rovatsos, H.-L. Truong, I. Carreras, and S. Dustdar. Programming model elements for hybrid collaborative adaptive systems. In *Proc. of the 1st IEEE International Conference on Collaboration and Internet Computing (CIC'15)*, Hangzhou, China, October 2015.

[STD12] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Modeling rewards and incentive mechanisms for social bpm. In Alistair Barros, Avigdor Gal, and Ekkart Kindler, editors, *Business Process Management*, volume 7481 of *LNCS*, pages 150–155. Springer Berlin Heidelberg, 2012.

[STD13a] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Incentives and rewarding in social computing. *Comm. of the ACM*, 56(6):72, 6 2013.

[STD13b] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Programming incentives in information systems. In Camille Salinesi, Moira C. Norrie, and Óscar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908 of *LNCS*, pages 688–703. Springer, 2013.

[STD14a] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. A collaboration model for community-based software development with social machines. In *Proc. of the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, Miami, FL, USA, October 2014.

[STD14b] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Managing incentives in social computing systems with pringl. In Boualem Benatallah, Azer Bestavros, Yannis Manolopoulos, Athena Vakali, and Yanchun Zhang, editors, *Web Inf. Systems Engineering (WISE'14)*, volume 8787 of *LNCS*, pages 415–424. Springer, 2014.

[STD15a] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Supporting multilevel incentive mechanisms in crowdsourcing systems: an artifact-centric view. In *Cloud-based Software Crowdsourcing*, pages 95–114. Springer, 2015.

[STD15b] Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Pringl – a domain-specific language for incentive management in crowdsourcing. *Computer Networks*, 9 July 2015.

[TCZ12] O. Tokarchuk, R. Cuel, and M. Zamarian. Analyzing crowd labor and designing incentives for humans in the loop. *Internet Computing, IEEE*, 16(5):45–51, 2012.

[TDKC15] Stefano Tranquillini, Florian Daniel, Pavel Kucherbaev, and Fabio Casati. Modeling, enacting, and integrating custom crowdsourcing processes. *ACM Trans. Web*, 9(2):7:1–7:43, May 2015.

[TLPH95] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *J. Syst. Softw.*, 28(1):9–18, January 1995.

[TZ09] Anh Tran and Richard Zeckhauser. Rank as an Incentive: Evidence from a Field Experiment. 2009.

[Vas12] Julita Vassileva. Motivating participation in social computing applications: a user modeling perspective. *User Modeling and User-Adapted Interaction*, 22(1-2):177–201, 2012.

[VCV06] Marco Van Herpen, Kees Cools, and Mirjam Van Praag. Wage Structure and the Incentive Effects of Promotions. *Kyklos*, 59(3):441–459, August 2006.

[WD99] Crayton C. Walker and Kevin J. Dooley. The Stability of Self-Organized Rule-Following Work Teams. *Computational & Mathematical Organization Theory*, 5(1):5–30, 1999.

[Weg12] Ryan Wegner. *Multi-agent malicious behaviour detection*. Phd thesis, University of Manitoba, 2012.

[WM00] Claus Wedekind and Manfred Milinski. Cooperation Through Image Scoring in Humans. *Science*, 288(5467):850–852, May 2000.

[YST+10] Kazufumi Yogo, Ryoichi Shinkuma, Tatsuro Takahashi, Taku Konishi, Satoko Itaya, Shinichi Doi, and Keiji Yamada. Differentiated Incentive Rewarding for Social Networking Services. *2010 10th IEEE/IPSJ International Symposium on Applications and the Internet*, pages 169–172, July 2010.

[Zep14] Philipp Zeppezauer. Virtualizing communications for hybrid and diversity-aware collective adaptive systems, 2014. Parallelt. [Übers. des Autors] Virtualizing Communications for Hybrid and Diversity-Aware Collective Adaptive Systems; Wien, Techn. Univ., Dipl.-Arb., 2015.

[ZST+14] Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong, Dimitrios I. Diochnos, Michael Rovatsos, Tommaso Schiavinotto, Iacopo Carreras, and Daniele Miorandi. Smartsociety consortium, deliverable 7.1 - virtualization techniques and prototypes. `http://www.smart-society-project.eu/publications/deliverables/D_7_1/`, 2014.

[ZSTD14] Philipp Zeppezauer, Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar. Virtualizing communication for hybrid and diversity-aware collective adaptive systems. In *Proc. of 10th Intl. Workshop on Engineering Service-Oriented Applications*, WESOA'14, pages 56–67. Springer, 11 2014.

Appendices

A SmartCOM Algorithms

A.I Message Handling

Every incoming message, regardless of whether it is from an internal component, an application or a peer is handled by the *handleMessage* function. Algorithm A.1 depicts the principal routing procedure.

First, the message is assigned with a unique message identifier which is used to track the message within SmartCOM. Additional to the explicit receiver of the message (can also be empty), further receivers - if there are any - are determined by the Routing Rule Engine based on routing rules. Finally the presented algorithm forwards the message to the corresponding receivers.

Algorithm A.2 describes how the messages are forwarded to a collective while Algorithm A.3 describes how the messages are forwarded to single peers. The function calls *registerCollectiveMessageDeliveryAttempt* and *registerPeerMessageDeliveryAttempt* indicate the registration of a policy handler that observes whether a delivery policy has been enforced for an outgoing message or if there was an error during communication.

First, the algorithm retrieves the collective info from the peer store (*CollectiveInfoCallback* API). This object contains information about the delivery policy of the collective as well as the peers that are currently part of the collective. If required (indicated by the variable *createHandlers*) a collective message delivery attempt is registered. Thereafter the message is delivered to every peer which is currently part of the collective. Note that this membership is subject to constant change.

Peer info is retrieved first. It consists of the delivery policy, privacy policies and contact addresses for adapters (which are not used in this algorithm). First, the algorithm checks if a message is allowed to be sent to a peer at the moment based on its privacy policies. If required (indicated by the variable *createHandlers*) a peer message delivery attempt is registered. The list consists of Ids of adapters which can send the message to this peer. Finally, using the Message Broker the message is sent to the peer over each adapter (indicated by its Id) that has been returned previously by the routing engine.

Algorithm A.1: Handling of messages in the Messaging and Routing Manager.

```
1  Function handleMessage is
       input  : Message (msg)
2      if message Id is empty then
3          create unique message ID;
4      end
5      createPolicyHandlers = false;
6      if message receiver is not null then
7          add the message receiver to the receiver list;
8          createPolicyHandlers = true;
9      end
       /* check if there are further receivers                          */
10     get further receivers from the routing engine (based on routing rules);
11     add them to the receiver list;
12     if receiver list is empty then
13         send error message to NotificationCallback;
14         return;
15     end
16     for each receiver in receiver list do
17         if receiver is component then
18             forward message to component;
19             continue;
20         else if receiver is collective then
21             deliverToCollective(msg, receiver, createPolicyHandlers); // Alg. A.2
22         else
23             try
24                 deliverToPeer(msg, receiver, createPolicyHandlers);  // Alg. A.3
25             catch
26                 send error message to the sender of the message;
27         createPolicyHandlers = false;
28     end
29 end
```

Algorithm A.2: Sending messages to a collective.

```
1  Function deliverToCollective is
       input  : Receiver (collective)
                Message (msg)
                Boolean value whether to create delivery policy handler
       (createHandlers)
2      retrieve collective info (collInfo) from CollectiveInfoCallback;
3      if createHandlers then
           // to trace the enforcement of delivery policies
4          registerCollectiveMessageDeliveryAttempt(msg, collInfo.deliveryPolicy);
5      end
6      for each peer in collInfo.peers do
7          try
8              deliverToPeer(msg, receiver, createHandlers);      // see Alg. A.3
9          catch
10             enforceCollectiveDeliveryPolicy(new error message);
11             if collInfo.deliveryPolicy is TO_ALL_MEMBERS then
                   /* delivery failed because the massage could not be sent
                      to everyone                                          */
12                 break;
13             end
14     end
15 end
```

Algorithm A.3: Sending messages to peers.

```
1  Function deliverToPeer is
       input  : Receiver (peer)
                Message (msg)
                Boolean value whether to create delivery policy handler
       (createHandlers)
2      retrieve peer info (peerInfo) from PeerInfoCallback;
3      for each policy in peerInfo.privacyPolicies do
           // check if policy allows sending messages
4          if !policy.condition(msg) then
5              throw an exception;
6          end
7      end
8      if createHandlers then
           // to trace the enforcement of delivery policies
9          registerPeerMessageDeliveryAttempt(msg, peerInfo.deliveryPolicy);
10     end
11     determine list of adapters (adapterList) from routing engine;
12     if adapterList is empty then
13         throw exception;
14     end
15     for each adapter in adapterList do
16         send output message to adapter using the message broker;
17     end
18 end
```

B PRINGL Models

B.I PRINGL Metamodel

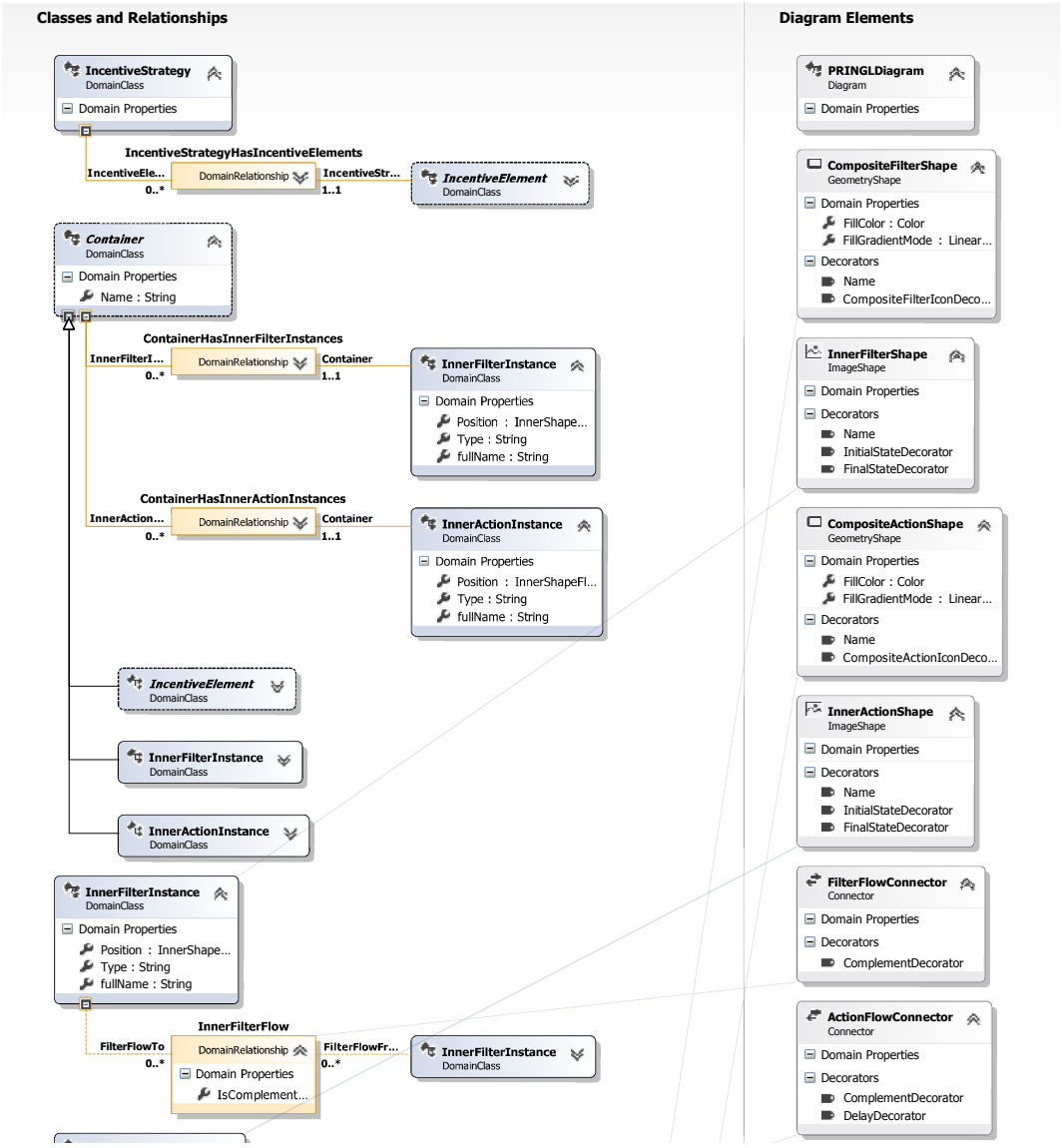

Figure B.1: Partial screenshot of the implemented PRINGL DSL metamodel. (upper section)

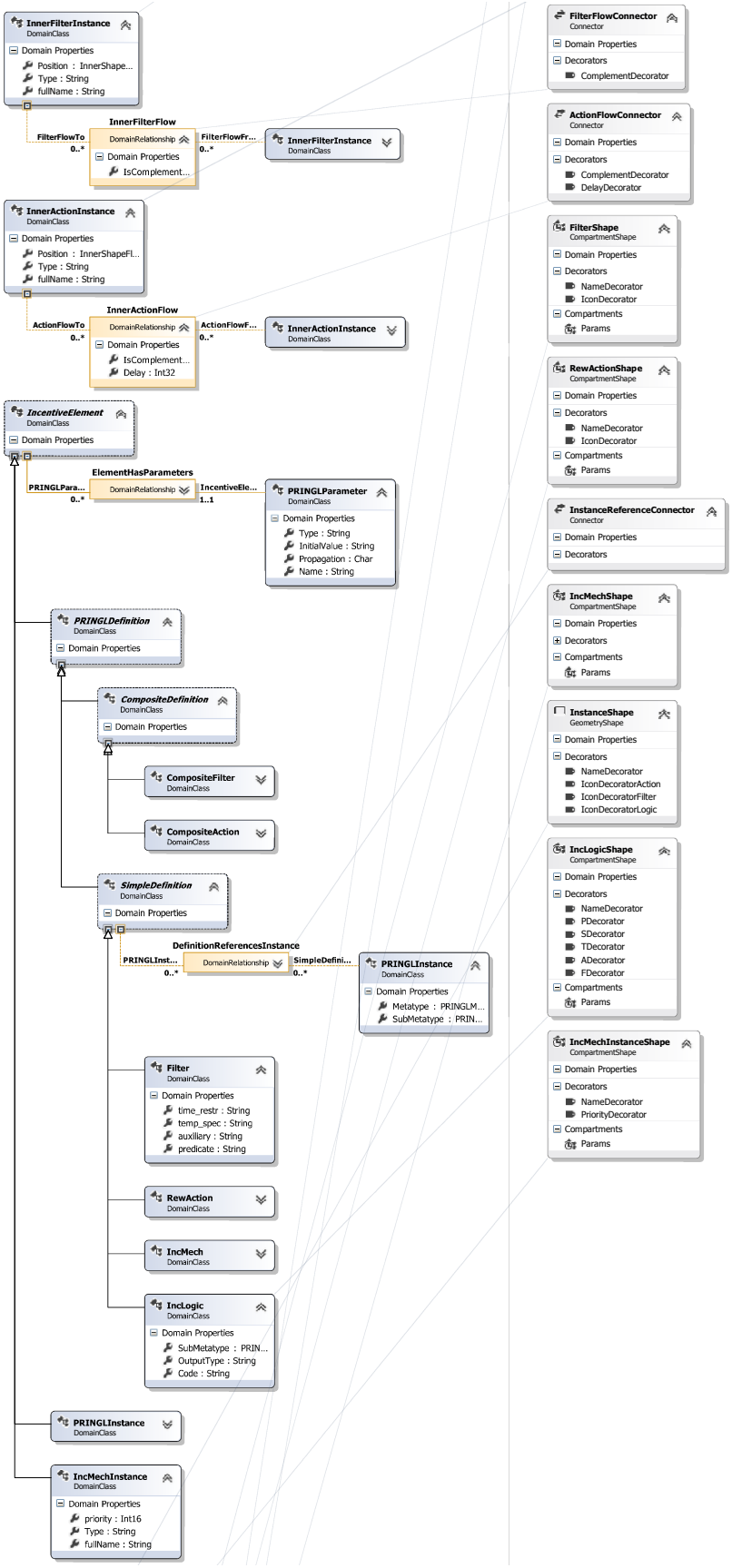

Figure B.2: Partial screenshot of the implemented PRINGL DSL metamodel. (middle section)

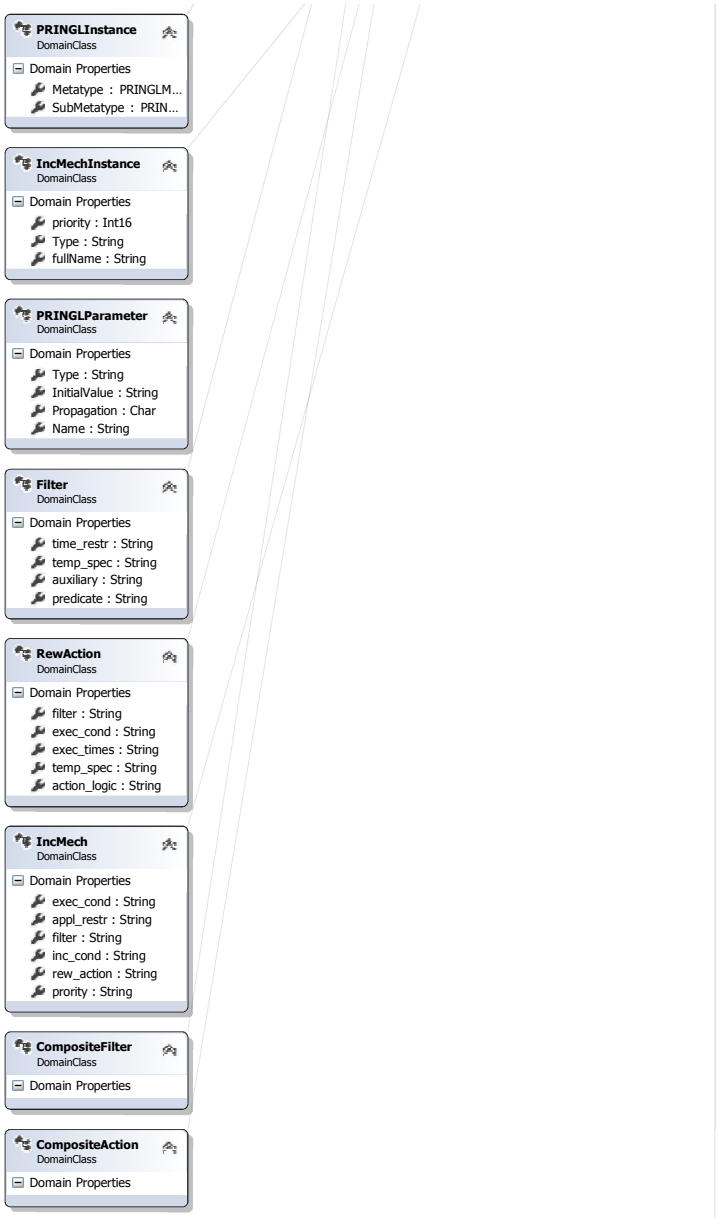

Figure B.3: Partial screenshot of the implemented PRINGL DSL metamodel. (lower section)

B.II "Rotating presidency" Model in implemented PRINGL

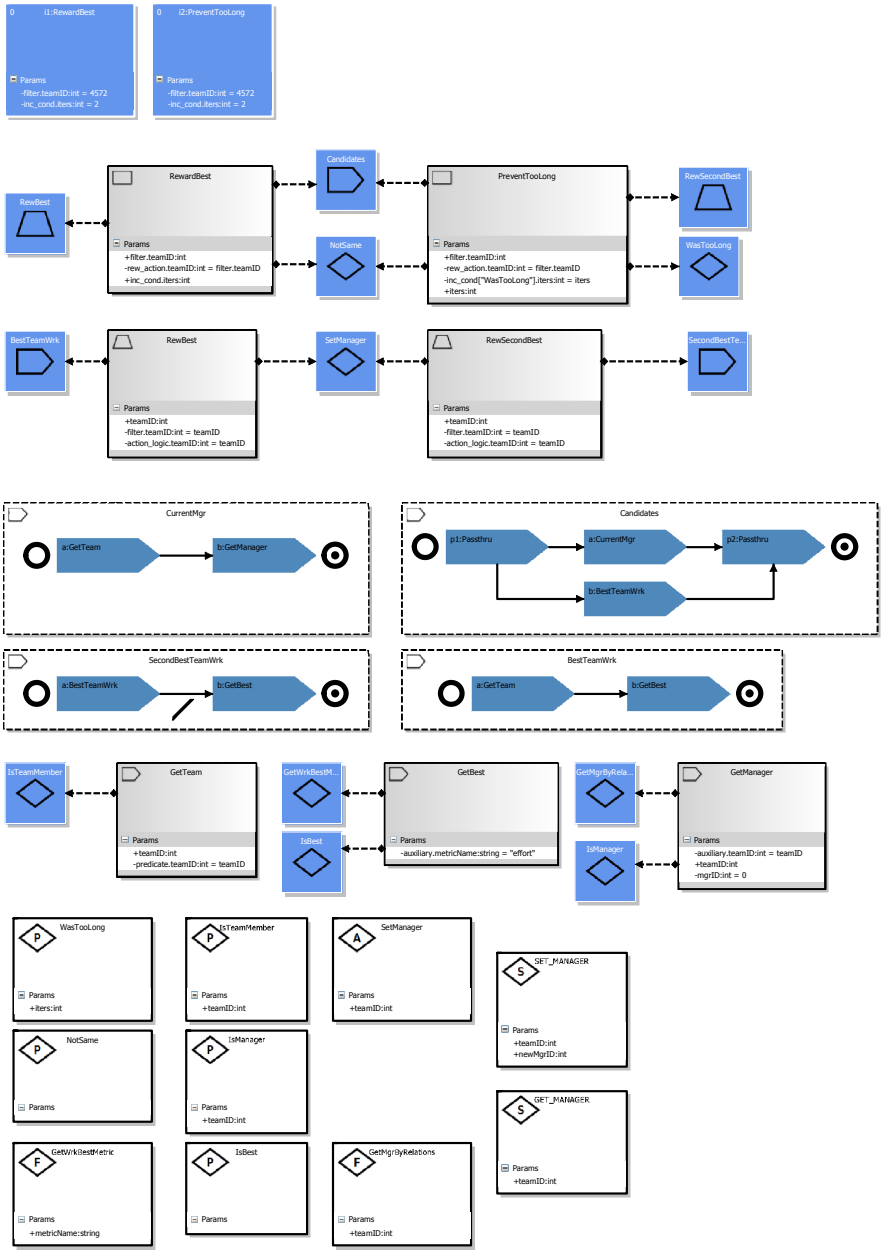

Figure B.4: Example 5from Section 7.4.5 modeled with implemented PRINGL Visual Studio plugin.

Listing B.L1: Example 5 from Section 7.4.5 – XML description of the incentive scheme

```
<?xml version="1.0" encoding="utf-8"?>
<incentiveStrategy xmlns:dm0="http://schemas.microsoft.com/VisualStudio
    ↪ /2008/DslTools/Core" dslVersion="1.0.0.0" Id="2dcbc39f-22ae-42b2-
    ↪ ae48-9fdcf45fdd58" strategyName="RotatingPresidency" xmlns="http://
    ↪ schemas.microsoft.com/dsltools/PRINGL">
  <incentiveElements>
    <incentiveStrategyHasIncentiveElements Id="41212765-aecd-4a15-b002
    ↪ -682b1ce2a4ae">
      <incMechInstance Id="492db129-8eed-4c73-b870-8f5328fc9131" name="i1
          ↪ " priority="0" type="RewardBest">
        <pRINGLParameter>
          <elementHasParameters Id="be223f5d-88bf-41ce-9ad7-fbc1e4e93815"
    ↪ >
            <pRINGLParameter Id="bc8aaf03-0978-4e16-8fc2-67bb6cb181a7"
                ↪ type="int" initialValue="4572" propagation="-" name="
                ↪ filter.teamID" />
          </elementHasParameters>
          <elementHasParameters Id="42b73150-99f1-47e6-9c08-165d5ad38b55"
              ↪ >
            <pRINGLParameter Id="2ab2c174-ef3f-4be0-9365-ae88dce0530d"
                ↪ type="int" initialValue="2" propagation="-" name="
                ↪ inc_cond.iters" />
          </elementHasParameters>
        </pRINGLParameter>
      </incMechInstance>
    </incentiveStrategyHasIncentiveElements>
    <incentiveStrategyHasIncentiveElements Id="4dfde389-870c-400f-83a5-
        ↪ cfd76c6b55ae">
      <incMechInstance Id="166df587-87ea-49ab-b958-370356c4a4e6" name="i2
          ↪ " priority="0" type="PreventTooLong">
        <pRINGLParameter>
          <elementHasParameters Id="68e68913-38c7-45eb-98e3-51249b3fa281"
    ↪ >
            <pRINGLParameter Id="8ecaa3a3-7a1b-4e90-a7bd-43865eebd762"
                ↪ type="int" initialValue="4572" propagation="-" name="
                ↪ filter.teamID" />
          </elementHasParameters>
          <elementHasParameters Id="0e38b6bc-9ccf-4510-83b2-bd1333c78862"
              ↪ >
            <pRINGLParameter Id="115030dd-b320-409c-837e-663cb27c2a53"
                ↪ type="int" initialValue="2" propagation="-" name="
                ↪ inc_cond.iters" />
          </elementHasParameters>
        </pRINGLParameter>
      </incMechInstance>
    </incentiveStrategyHasIncentiveElements>
    <incentiveStrategyHasIncentiveElements Id="4161a095-2604-42c5-92c3
        ↪ -0309d4ffd87e">
      <incMech Id="983fb58e-3bf5-43b6-87b4-3e2f056b04ba" name="RewardBest
          ↪ " exec_cond="null" appl_restr="default" filter="Candidates"
          ↪ inc_cond="NotSame" rew_action="RewBest" prority="0">
        <pRINGLParameter>
```

```
      <elementHasParameters Id="75ac443e-03c4-4e2e-a463-8ae372715489"
↪ >
        <pRINGLParameter Id="e3b30e64-68db-44d5-9bd5-34cc15003e33"
           ↪ type="int" propagation="+" name="filter.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="b37c8a35-91ff-40ad-97d8-891ed2ee7c9e"
          ↪ >
        <pRINGLParameter Id="9570b56f-afd6-4b40-9a5b-196a7da9217a"
           ↪ type="int" initialValue="filter.teamID" propagation="-"
           ↪  name="rew_action.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="3348f441-1209-4fe6-9241-34de87163352"
          ↪ >
        <pRINGLParameter Id="b8791eb5-499e-452d-99f7-9a850d3dfafb"
           ↪ type="int" propagation="+" name="inc_cond.iters" />
      </elementHasParameters>
    </pRINGLParameter>
    <pRINGLInstance>
      <definitionReferencesInstance Id="762f7518-3056-4f11-85b2-9
↪ d638999a1f6">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/Candidates" />
      </definitionReferencesInstance>
      <definitionReferencesInstance Id="1dd4b5c3-eef1-463f-a9dd-
          ↪ d29871d35d37">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/NotSame" />
      </definitionReferencesInstance>
      <definitionReferencesInstance Id="b83f51bb-1c6e-4027-ba4c-
          ↪ e3ecdaeda07c">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/RewBest" />
      </definitionReferencesInstance>
    </pRINGLInstance>
  </incMech>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="f7eeb5c0-4d5e-490f-ba9e-
    ↪ ea59c8b63320">
  <pRINGLInstance Id="5955aa51-145b-4732-8959-8b0815cf5e5f" name="
     ↪ Candidates" metatype="Filter" subMetatype="None" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="694b989a-ca3d-46c2-8d5b-8
    ↪ dec31232baf">
  <pRINGLInstance Id="f6a156ba-b7ee-4a22-9824-3e52f511fcab" name="
     ↪ NotSame" metatype="Logic" subMetatype="PredicateLogic" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="acd06ee4-8157-4a69-9eb4-3
    ↪ f20cbba0749">
  <pRINGLInstance Id="9a8fb4c3-7e30-47d2-a01f-5f497b6e4521" name="
     ↪ RewBest" metatype="Action" subMetatype="None" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="d7ca455a-e960-4541-bdb9-
    ↪ a36893497f3c">
```

```
  <incMech Id="74559555-719c-4ae1-9bff-762b124b008c" name="
     ↪ PreventTooLong" exec_cond="null" appl_restr="default" filter=
     ↪ "Candidates" inc_cond="!␣NotSame␣&amp;amp;amp;␣WasTooLong"
     ↪ rew_action="RewSecondBest" prority="0">
    <pRINGLParameter>
     <elementHasParameters Id="e6400fc9-e163-46f4-9472-11f36d138102"
↪ >
       <pRINGLParameter Id="0b2e032b-f888-4db3-91b1-9b62b06b0aff"
          ↪ type="int" propagation="+" name="filter.teamID" />
     </elementHasParameters>
     <elementHasParameters Id="d9596f1b-a820-49ce-9095-9fd6ddc2d58a"
        ↪ >
       <pRINGLParameter Id="ed8f055f-b7a8-48da-a3cf-9000c09f5b00"
          ↪ type="int" initialValue="filter.teamID" propagation="-"
          ↪  name="rew_action.teamID" />
     </elementHasParameters>
     <elementHasParameters Id="cb76ba3b-c915-4dfb-99fe-aab628a212c0"
        ↪ >
       <pRINGLParameter Id="ed987037-6d9c-4352-8793-3397511eb946"
          ↪ type="int" initialValue="iters" propagation="-" name="
          ↪ inc_cond[&amp;quot;WasTooLong&amp;quot;].iters" />
     </elementHasParameters>
     <elementHasParameters Id="0ba92c72-ce82-4604-8c17-6b99564eed60"
        ↪ >
       <pRINGLParameter Id="7cddbf96-2805-4b57-ab2c-386312e81d34"
          ↪ type="int" propagation="+" name="iters" />
     </elementHasParameters>
    </pRINGLParameter>
    <pRINGLInstance>
     <definitionReferencesInstance Id="75ef8e04-3d9c-4d41-915b-
↪ bcb13ffe7c70">
       <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/RewSecondBest" />
     </definitionReferencesInstance>
     <definitionReferencesInstance Id="b29c2345-449a-45ad-a4ce-
        ↪ b05500a05314">
       <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/NotSame" />
     </definitionReferencesInstance>
     <definitionReferencesInstance Id="6e4872ae-982e-4bac-964b-30
        ↪ e4de83047b">
       <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/Candidates" />
     </definitionReferencesInstance>
     <definitionReferencesInstance Id="4c1ee0d0-c8ed-4a5a-83c8-6222
        ↪ b938beec">
       <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/WasTooLong" />
     </definitionReferencesInstance>
    </pRINGLInstance>
  </incMech>
 </incentiveStrategyHasIncentiveElements>
 <incentiveStrategyHasIncentiveElements Id="4f8a63f0-cb53-46aa-aaca
```

```
    ↪ -8704533ffdef">
  <pRINGLInstance Id="78f8228f-6e57-4e9e-8f35-aede68e13937" name="
    ↪ RewSecondBest" metatype="Action" subMetatype="None" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="a816fc50-03ab-439d-9306-07
    ↪ ca2c2cceb9">
  <pRINGLInstance Id="41c29e95-c85d-4408-87dc-794bc4a03320" name="
    ↪ WasTooLong" metatype="Logic" subMetatype="PredicateLogic" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="d58172c9-437a-4bd7-b776-2
    ↪ f2b606a5fa5">
  <rewAction Id="5d9262d7-44c0-444d-b8ad-5ca9088b7504" name="RewBest"
    ↪  filter="BestTeamWrk" exec_cond="null" exec_times="null"
    ↪ temp_spec="null" action_logic="SetManager">
    <pRINGLParameter>
      <elementHasParameters Id="08e9fb91-ead7-478f-a05e-b8765422c72d"
↪ >
        <pRINGLParameter Id="3197a5b5-4f5f-4fd8-9604-301f7ba3915e"
          ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
      <elementHasParameters Id="fd6c0f77-485b-42ff-a6a9-faac49388aa1"
        ↪ >
        <pRINGLParameter Id="ba835691-c0e3-44c9-a22a-a36619ad55a8"
          ↪ type="int" initialValue="teamID" propagation="-" name="
          ↪ filter.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="fa532ac8-79b0-4ef4-ab13-83e60dc4eba8"
        ↪ >
        <pRINGLParameter Id="45e975fe-d985-46da-a5e1-e5775de8b889"
          ↪ type="int" initialValue="teamID" propagation="-" name="
          ↪ action_logic.teamID" />
      </elementHasParameters>
    </pRINGLParameter>
    <pRINGLInstance>
      <definitionReferencesInstance Id="94bc7ac2-8dbc-4aae-a669-4778
↪ bee9dd18">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/BestTeamWrk" />
      </definitionReferencesInstance>
      <definitionReferencesInstance Id="a76e3222-5b9e-4415-b7a3-
        ↪ ff9583dc248f">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
          ↪ fdcf45fdd58/SetManager" />
      </definitionReferencesInstance>
    </pRINGLInstance>
  </rewAction>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="645f8fe8-0de2-4d6c-84e4
    ↪ -564a9736eca9">
  <pRINGLInstance Id="4cbdd0c7-5cbe-4d43-9bbc-c8c54b1bc61e" name="
    ↪ BestTeamWrk" metatype="Filter" subMetatype="None" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="e529084c-b115-4273-a0fc-
```

```
    ↪ e35b12a177d2">
   <pRINGLInstance Id="4cdaa32b-e19f-4e08-b839-1df317aa727b" name="
      ↪ SetManager" metatype="Logic" subMetatype="ActionLogic" />
 </incentiveStrategyHasIncentiveElements>
 <incentiveStrategyHasIncentiveElements Id="6671d3ef-aa47-48c1-b232-
    ↪ c75a8d4ccc0f">
   <rewAction Id="c1392cd4-4496-45de-bfc7-d5beed730e02" name="
      ↪ RewSecondBest" filter="SecondBestTeamWrk" exec_cond="null"
      ↪ exec_times="null" temp_spec="null" action_logic="SetManager">
     <pRINGLParameter>
       <elementHasParameters Id="815a56ae-895e-4dea-acb1-d2b423c171ab"
↪ >
         <pRINGLParameter Id="f937be5f-576d-4100-b894-49ba64475de3"
            ↪ type="int" propagation="+" name="teamID" />
       </elementHasParameters>
       <elementHasParameters Id="f6380f21-9db7-4650-a877-16c91a0f5400"
          ↪ >
         <pRINGLParameter Id="2f1a5bce-9b96-4501-a89b-251e2b49dfbf"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ filter.teamID" />
       </elementHasParameters>
       <elementHasParameters Id="cc15012f-44a2-44dd-83ae-4e020ee7e41b"
          ↪ >
         <pRINGLParameter Id="c4c2e8e5-d6ee-499c-99b1-8625461147bb"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ action_logic.teamID" />
       </elementHasParameters>
     </pRINGLParameter>
     <pRINGLInstance>
       <definitionReferencesInstance Id="b66a6ef2-3ba3-4db8-b25c-
↪ e2c2b5d0aa30">
         <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
            ↪ fdcf45fdd58/SecondBestTeamWrk" />
       </definitionReferencesInstance>
       <definitionReferencesInstance Id="016d6e96-f1d2-4577-b6e0-3806
          ↪ c14dd2e4">
         <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
            ↪ fdcf45fdd58/SetManager" />
       </definitionReferencesInstance>
     </pRINGLInstance>
   </rewAction>
 </incentiveStrategyHasIncentiveElements>
 <incentiveStrategyHasIncentiveElements Id="e61721a4-1a9b-4bf4-b204-
    ↪ f98475b005f0">
   <compositeFilter Id="1acc4949-c752-4678-9ecd-05d945a265b2" name="
      ↪ CurrentMgr">
     <innerFilterInstances>
       <containerHasInnerFilterInstances Id="dd99c989-7f04-426a-a685-2
↪ d20ec000c7b">
         <innerFilterInstance Id="00806b4e-ad8d-4b54-a40f-3c007548ad7f
            ↪ " name="a" position="Initial" type="GetTeam">
           <filterFlowTo>
             <innerFilterFlow Id="7cdd4860-55b3-4578-988c-85260c216b73
```

```
↪ " isComplemented="false">
                <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                    ↪ ae48-9fdcf45fdd58/CurrentMgr/b" />
              </innerFilterFlow>
            </filterFlowTo>
          </innerFilterInstance>
        </containerHasInnerFilterInstances>
        <containerHasInnerFilterInstances Id="5ac9822e-727f-4863-b997-9
            ↪ e8142527f89">
          <innerFilterInstance Id="8b14fe56-10d3-45eb-8528-9e0cdfff2a7d
              ↪ " name="b" position="Final" type="GetManager" />
        </containerHasInnerFilterInstances>
      </innerFilterInstances>
      <pRINGLParameter>
        <elementHasParameters Id="ec1ac694-de2a-466f-ae73-8493b03ba9c6"
↪ >
          <pRINGLParameter Id="5ed80b21-b948-4993-bd48-c9e2ccb3679e"
              ↪ type="int" initialValue="teamID" propagation="-" name="
              ↪ a.teamID" />
        </elementHasParameters>
        <elementHasParameters Id="8462f902-a3d1-4ae9-b727-53dc4c193bea"
            ↪ >
          <pRINGLParameter Id="b2292a7a-2e81-49fc-a144-a39ffad93a62"
              ↪ type="int" initialValue="teamID" propagation="-" name="
              ↪ b.teamID" />
        </elementHasParameters>
        <elementHasParameters Id="5978c43a-1764-4710-bdc0-b8ec9b8621f1"
            ↪ >
          <pRINGLParameter Id="eca11e47-7255-410a-ac4a-9180f9505781"
              ↪ type="int" propagation="+" name="teamID" />
        </elementHasParameters>
      </pRINGLParameter>
    </compositeFilter>
  </incentiveStrategyHasIncentiveElements>
  <incentiveStrategyHasIncentiveElements Id="9b4fe392-fe20-4262-b34d-7
      ↪ bf5af106c51">
    <pRINGLInstance Id="7cc8a7c2-fbd8-45cc-8cbc-d1d4cf1a2dcb" name="
        ↪ SecondBestTeamWrk" metatype="Filter" subMetatype="None" />
  </incentiveStrategyHasIncentiveElements>
  <incentiveStrategyHasIncentiveElements Id="1680e455-1532-46c1-818c-
      ↪ dcf6f085d7d4">
    <incLogic Id="93164578-45ec-4943-b7a5-b8d13c703525" name="
        ↪ WasTooLong" subMetatype="PredicateLogic" outputType="bool"
        ↪ code="/*␣WasTooLong␣BEGIN␣*/␣static␣Dictionary&lt;Worker,␣int
        ↪ &gt;␣leaderHistory␣=␣new␣Dictionary&lt;Worker,␣int&gt;();␣if␣
        ↪ (_ws.Count()␣==␣2)␣{␣␣if␣(leaderHistory.ContainsKey(_ws.First
        ↪ ()))␣␣{␣␣␣leaderHistory.Clear();␣␣␣leaderHistory[_ws.Last()]␣
        ↪ =␣1;␣␣}␣␣else␣␣{␣␣␣leaderHistory.Clear();␣␣␣leaderHistory[_ws
        ↪ .First()]␣=␣1;␣␣}␣}␣else␣{␣␣if␣(leaderHistory[_ws.First()]&lt
        ↪ ;␣iters)␣␣␣{␣␣␣leaderHistory[_ws.First()]++;␣␣␣return␣false;␣
        ↪ ␣}␣}␣return␣true;␣␣/*␣WasTooLong␣END␣*/">
      <pRINGLParameter>
        <elementHasParameters Id="fc915d78-341b-4284-b4a8-72c928724d9c"
```

```
↪ >
        <pRINGLParameter Id="fa29533d-0b8e-418d-853e-89e7b3b8cfca"
            ↪ type="int" propagation="+" name="iters" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="35262008-5091-4fa4-8461-70
    ↪ f9a28859b6">
  <compositeFilter Id="bf7bc8f5-430f-4f46-a7cc-c2fcaf514976" name="
      ↪ SecondBestTeamWrk">
    <innerFilterInstances>
      <containerHasInnerFilterInstances Id="64c8d46d-0373-447b-8ee1
↪ -77ebb51a1c53">
        <innerFilterInstance Id="2d2efc5d-e3d4-42f9-873d-f3ead2b2c60e
            ↪ " name="a" position="Initial" type="BestTeamWrk">
          <filterFlowTo>
            <innerFilterFlow Id="499560b6-9353-4642-bf27-4fcadb919d31
↪ " isComplemented="true">
              <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/SecondBestTeamWrk/b" />
            </innerFilterFlow>
          </filterFlowTo>
        </innerFilterInstance>
      </containerHasInnerFilterInstances>
      <containerHasInnerFilterInstances Id="4098e334-e849-4eea-ad35-
          ↪ a37adf8a25c7">
        <innerFilterInstance Id="830facad-2983-437d-87bc-deb68a618177
            ↪ " name="b" position="Final" type="GetBest" />
      </containerHasInnerFilterInstances>
    </innerFilterInstances>
    <pRINGLParameter>
      <elementHasParameters Id="7c3c9090-650d-4d6f-a58c-ed1aed9c28da"
↪ >
        <pRINGLParameter Id="4f34c4a7-1f01-4625-8105-4ce54cadba7a"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ a.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="0c7e7391-b6f5-4f71-8c12-0dba6a416944"
          ↪ >
        <pRINGLParameter Id="24dd52ef-c5e0-494e-8715-5819d34c1357"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ b.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="7fb2292a-1da0-40a2-a3b5-0b588b2eed60"
          ↪ >
        <pRINGLParameter Id="bc65ff5b-56cf-4079-be78-f34f6658178c"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </compositeFilter>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="1fa479d0-6437-4e7d-a321-
```

```
    ↪ ee944ff00437">
   <compositeFilter Id="34c89378-9a70-454d-adaf-05a257d66ced" name="
      ↪ Candidates">
     <innerFilterInstances>
       <containerHasInnerFilterInstances Id="fb1c052e-1c80-4b11-8abe-
↪ abdea2ccc029">
         <innerFilterInstance Id="8a851d05-eb76-4fb6-b9a0-c23e95a62484
            ↪ " name="p1" position="Initial" type="Passthru">
           <filterFlowTo>
             <innerFilterFlow Id="c4923a7a-a7b0-4623-bfc3-a105c81b0aaf
↪ " isComplemented="false">
               <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/Candidates/a" />
             </innerFilterFlow>
             <innerFilterFlow Id="8a3d0bd2-09d7-4423-8941-87b76afdcadf
                ↪ " isComplemented="false">
               <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/Candidates/b" />
             </innerFilterFlow>
           </filterFlowTo>
         </innerFilterInstance>
       </containerHasInnerFilterInstances>
       <containerHasInnerFilterInstances Id="f24db1bf-f122-4509-87e6-
          ↪ d880585fe037">
         <innerFilterInstance Id="ba972da3-deeb-4162-b174-3a7905baf26e
            ↪ " name="a" type="CurrentMgr">
           <filterFlowTo>
             <innerFilterFlow Id="b3f4d715-4277-4fab-b8b3-660ab3ae6a5c
↪ " isComplemented="false">
               <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/Candidates/p2" />
             </innerFilterFlow>
           </filterFlowTo>
         </innerFilterInstance>
       </containerHasInnerFilterInstances>
       <containerHasInnerFilterInstances Id="1d9aa599-3245-45f9-9f89-6
          ↪ f57a0d4cad0">
         <innerFilterInstance Id="2680d058-9f71-4a70-bc15-e8d50329ec2a
            ↪ " name="b" type="BestTeamWrk">
           <filterFlowTo>
             <innerFilterFlow Id="01848dd9-75bd-47df-ab4a-a38821cdd7ea
↪ " isComplemented="false">
               <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/Candidates/p2" />
             </innerFilterFlow>
           </filterFlowTo>
         </innerFilterInstance>
       </containerHasInnerFilterInstances>
       <containerHasInnerFilterInstances Id="99817452-f697-442e-a97e-
          ↪ ce2f624d568e">
         <innerFilterInstance Id="0648fcb1-0249-4da8-af16-11f21c99b05a
            ↪ " name="p2" position="Final" type="Passthru" />
       </containerHasInnerFilterInstances>
```

```
    </innerFilterInstances>
    <pRINGLParameter>
      <elementHasParameters Id="e973c616-86be-49cf-9f76-6aaeafc29f62"
↪ >
        <pRINGLParameter Id="c66bf554-5806-41f0-ae13-5c173d99d435"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ a.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="974c97fd-38b7-4c2c-8094-739a3533aacc"
          ↪ >
        <pRINGLParameter Id="638e9c4d-6b53-405e-b1d3-23fe5c6ca9d4"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ b.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="fccef187-1a82-4040-ac96-339e867dcf85"
          ↪ >
        <pRINGLParameter Id="6c80a610-3e36-4a7b-bb6e-e5f28654f961"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </compositeFilter>
 </incentiveStrategyHasIncentiveElements>
 <incentiveStrategyHasIncentiveElements Id="c0bb7fc0-c146-4305-aaea-
    ↪ d6c070789030">
  <compositeFilter Id="298b0628-0a6b-401f-b623-0cded195f898" name="
      ↪ BestTeamWrk">
    <innerFilterInstances>
      <containerHasInnerFilterInstances Id="d0ea6466-bc26-4151-be3e
↪ -54093ac40a7f">
        <innerFilterInstance Id="a607cb0d-9906-47d7-b3b0-8558b7a50f74
            ↪ " name="a" position="Initial" type="GetTeam">
          <filterFlowTo>
            <innerFilterFlow Id="37562012-eed7-4671-b680-c191c6bf47f2
↪ " isComplemented="false">
              <innerFilterInstanceMoniker name="/2dcbc39f-22ae-42b2-
                  ↪ ae48-9fdcf45fdd58/BestTeamWrk/b" />
            </innerFilterFlow>
          </filterFlowTo>
        </innerFilterInstance>
      </containerHasInnerFilterInstances>
      <containerHasInnerFilterInstances Id="0e58f588-dc52-4d0a-aaa4-9
          ↪ c9fffb62d5e">
        <innerFilterInstance Id="864546c7-dd66-4edd-b586-7b788ee1474c
            ↪ " name="b" position="Final" type="GetBest" />
      </containerHasInnerFilterInstances>
    </innerFilterInstances>
    <pRINGLParameter>
      <elementHasParameters Id="11565e37-16b3-4aae-8fde-43d6907a25a9"
↪ >
        <pRINGLParameter Id="a64c19c7-65e3-4fca-993c-72ded133a6b0"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ a.teamID" />
      </elementHasParameters>
```

```
      <elementHasParameters Id="3da948e0-4016-4171-b04e-90f1a5e888ae"
          ↪ >
        <pRINGLParameter Id="4eea26ae-79af-4ab7-a5e8-ca6ec3725f8b"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ b.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="bc24fe41-7d16-4635-bd65-f74b35632577"
          ↪ >
        <pRINGLParameter Id="346cfc73-e728-4ee1-8f1b-66e9bcccdad7"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </compositeFilter>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="4f677662-5d1b-4d0f-99f9-
    ↪ f5abf87fd2bd">
  <filter Id="73afe080-aeec-423e-bcbc-35bb4fa237bf" name="GetTeam"
      ↪ time_restr="null" temp_spec="default" auxiliary="null"
      ↪ predicate="IsTeamMember">
    <pRINGLParameter>
      <elementHasParameters Id="3f52ee34-32b8-4f86-8088-ba654b839185"
↪ >
        <pRINGLParameter Id="429a78f1-1344-47d9-ab1a-3517216c875b"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
      <elementHasParameters Id="01d096b2-184c-42a6-9b79-d75e455ea1c9"
          ↪ >
        <pRINGLParameter Id="e5beefcd-da27-45f2-b88e-0e8a19faa9e6"
            ↪ type="int" initialValue="teamID" propagation="-" name="
            ↪ predicate.teamID" />
      </elementHasParameters>
    </pRINGLParameter>
    <pRINGLInstance>
      <definitionReferencesInstance Id="f2d17ae3-2203-4f0f-a4bd-7041
↪ be0ff739">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
            ↪ fdcf45fdd58/IsTeamMember" />
      </definitionReferencesInstance>
    </pRINGLInstance>
  </filter>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="c5e618f5-f4e0-4863-aa8c
    ↪ -47152a022afb">
  <filter Id="3c3c4003-a213-4c8a-befe-2c69d815e741" name="GetBest"
      ↪ time_restr="null" temp_spec="default" auxiliary="
      ↪ GetWrkBestMetric" predicate="IsBest">
    <pRINGLParameter>
      <elementHasParameters Id="75febffa-0ea7-4028-bf43-9b64e8f1d742"
↪ >
        <pRINGLParameter Id="810dc801-60b9-4c4f-bcab-88a15d58e83e"
            ↪ type="string" initialValue="&amp;quot;effort&amp;quot;"
            ↪ propagation="-" name="auxiliary.metricName" />
      </elementHasParameters>
```

```
    </pRINGLParameter>
    <pRINGLInstance>
      <definitionReferencesInstance Id="662a210c-4346-4d81-8947-691
↪ d8969f524">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/GetWrkBestMetric" />
      </definitionReferencesInstance>
      <definitionReferencesInstance Id="a4be4efb-bc20-48ee-ade8-15
         ↪ e49332710f">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/IsBest" />
      </definitionReferencesInstance>
    </pRINGLInstance>
  </filter>
 </incentiveStrategyHasIncentiveElements>
 <incentiveStrategyHasIncentiveElements Id="7e575c2d-7826-4abe-9860-54
    ↪ b869b0fac0">
  <filter Id="6dc1fb7c-6cd5-4a62-9345-d761a8df9dfb" name="GetManager"
     ↪  time_restr="null" temp_spec="default" auxiliary="
     ↪ GetMgrByRelations" predicate="IsManager">
    <pRINGLParameter>
      <elementHasParameters Id="8b511ab6-eb23-4f11-98aa-07eefa25489e"
↪ >
        <pRINGLParameter Id="7e82611b-5b14-45d2-9380-4e35871314c0"
           ↪ type="int" initialValue="teamID" propagation="-" name="
           ↪ auxiliary.teamID" />
      </elementHasParameters>
      <elementHasParameters Id="409cd665-5768-4377-bfbe-0e704500b650"
         ↪ >
        <pRINGLParameter Id="119925bf-62d4-4c5b-b2b3-2cb355fc8fb2"
           ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
      <elementHasParameters Id="6af42219-c448-4ede-a2a4-d63c931a821e"
         ↪ >
        <pRINGLParameter Id="65ced596-3bb8-46bd-bdc2-464da13bf843"
           ↪ type="int" initialValue="0" propagation="-" name="mgrID
           ↪ " />
      </elementHasParameters>
    </pRINGLParameter>
    <pRINGLInstance>
      <definitionReferencesInstance Id="7a19218d-c0c6-4129-a998-5
↪ e97f51dba39">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/GetMgrByRelations" />
      </definitionReferencesInstance>
      <definitionReferencesInstance Id="c904e0a4-faff-45a6-897a-37844
         ↪ f65b35a">
        <pRINGLInstanceMoniker name="/2dcbc39f-22ae-42b2-ae48-9
           ↪ fdcf45fdd58/IsManager" />
      </definitionReferencesInstance>
    </pRINGLInstance>
  </filter>
 </incentiveStrategyHasIncentiveElements>
```

```
<incentiveStrategyHasIncentiveElements Id="23f63e91-dfdd-4309-8bf4-
    ↪ df89101b1ff9">
  <pRINGLInstance Id="7c28c07b-c7c5-46b0-ac3c-7ddad7f5f221" name="
      ↪ IsTeamMember" metatype="Logic" subMetatype="PredicateLogic"
      ↪ />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="8fc3c69e-11fc-4a28-a80a-
    ↪ d97d8fd0541e">
  <incLogic Id="346089b1-9499-440f-80f7-64dc70ee1713" name="
      ↪ SetManager" subMetatype="ActionLogic" outputType="IEnumerable
      ↪ &lt;Worker&gt;" code="/* BEGIN SetManager */ Collection&lt;
      ↪ Worker&gt; affected = sm(teamID,_ws.First().ID); return
      ↪ affected; /* END SetManager */">
    <pRINGLParameter>
      <elementHasParameters Id="4e128e41-6678-4430-bd94-f222faccaaa7"
↪ >
        <pRINGLParameter Id="057c7ef8-a21f-4b18-bc21-1210196c23e4"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="07c35bbe-413a-43ed-8a9d
    ↪ -285219f11409">
  <pRINGLInstance Id="86c4f99e-0b73-4c30-972f-01254690f6a7" name="
      ↪ GetWrkBestMetric" metatype="Logic" subMetatype="FilterLogic"
      ↪ />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="9637523b-fd13-4122-9edb-
    ↪ da399e514369">
  <pRINGLInstance Id="f8975814-86be-4251-8aff-6376cb60dc8c" name="
      ↪ IsBest" metatype="Logic" subMetatype="PredicateLogic" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="b63b07b4-ca65-44c3-bb65
    ↪ -8354a6e45f2b">
  <pRINGLInstance Id="fd265e8a-20dc-4772-a7e9-9be5e17b70f8" name="
      ↪ GetMgrByRelations" metatype="Logic" subMetatype="FilterLogic"
      ↪  />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="79357124-eb71-47b4-8db5-
    ↪ ee3785388aba">
  <pRINGLInstance Id="fcbf0d07-c211-45c3-bd5e-7edca123cef3" name="
      ↪ IsManager" metatype="Logic" subMetatype="PredicateLogic" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="1d63ab43-0c3c
    ↪ -4220-9179-5379097c0bdd">
  <incLogic Id="c2667156-ec0b-47f7-9883-b9432e9b1a59" name="
      ↪ IsTeamMember" subMetatype="PredicateLogic" outputType="bool"
      ↪ code="/*  BEGIN IsTeamMember */ var teams = (DICT)_w.GetData
      ↪ (&quot;teams&quot;, COMPOSITE); return teams.ContainsKey(
      ↪ teamID.ToString()); /* END IsTeamMember */">
    <pRINGLParameter>
      <elementHasParameters Id="6e3ae621-935a-4ac0-8572-e2c753e17783"
```

```
↪ >
        <pRINGLParameter Id="503ad2c3-e898-4d1f-a832-c2418df0765a"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="2febb1f2-da83-44d4-bf70-4
    ↪ f9bea9b7d60">
  <incLogic Id="0266b097-18c3-4850-87cd-7dc6ed2f21ca" name="
      ↪ SET_MANAGER" subMetatype="StructureLogic" outputType="void"
      ↪ code="/*␣␣BEGIN␣SET_MANAGER*/␣/*GRGEN:SET_MANAGER*/␣/*␣END␣
      ↪ SET_MANAGER*/">
    <pRINGLParameter>
      <elementHasParameters Id="87d1c383-7c98-4640-a71e-3175bdf0fb58"
↪ >
        <pRINGLParameter Id="bd24bfd0-96c7-4fe4-89c0-c4fa648a48c7"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
      <elementHasParameters Id="d06eb973-b8b2-4fb7-937c-31ba03e403ff"
          ↪ >
        <pRINGLParameter Id="b825f36b-ccc8-4012-9872-7bd7b120b252"
            ↪ type="int" propagation="+" name="newMgrID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="2590ae5a-c409-47f0-a453-
    ↪ a03cb730b326">
  <incLogic Id="8e08b4f2-1c5c-4cf6-a4d4-d250f0789f7f" name="
      ↪ GetMgrByRelations" subMetatype="FilterLogic" outputType="void
      ↪ " code="␣/*␣␣BEGIN␣GetMgrByRelations␣*/␣if␣((int)_parent.
      ↪ getParam(&amp;quot;mgrID&amp;quot;)␣!=␣0)␣return;␣foreach␣(Worker␣w␣
      ↪ in␣_ws)␣{w.mark(teamID);}␣_parent.setParam(&amp;quot;mgrID&amp;quot;,
      ↪ ␣gm(teamID));␣/*␣END␣GetMgrByRelations*/">
    <pRINGLParameter>
      <elementHasParameters Id="a77104ac-ce01-4383-9612-dfa13a546177"
↪ >
        <pRINGLParameter Id="d3cce9ae-2c18-4c51-bbd0-93b1a2f72ade"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="9efbd047-fbf3-4da5-ae77-1
    ↪ d8282687a08">
  <incLogic Id="d79671fb-bddb-4f59-bd10-f0a21d75d05c" name="NotSame"
      ↪ subMetatype="PredicateLogic" outputType="bool" code="/*␣
      ↪ NotSame␣BEGIN␣*/␣if␣(_ws.Count()␣&amp;gt;␣1)␣␣␣return␣true;␣␣␣␣
      ↪ else␣␣return␣false;␣/*␣NotSame␣END␣*/" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="797a6426-71ad-4280-8eee-03
    ↪ ad6f6797fe">
```

```
  <incLogic Id="90d85ba5-9854-45e6-bb80-bf85a15fb2ba" name="
      ↪ GetWrkBestMetric" subMetatype="FilterLogic" outputType="void"
      ↪  code="/*␣␣BEGIN␣GetWrkBestMetric␣*/␣static␣Worker␣theBest;␣
      ↪ double␣bestResult␣=␣_ws.Max(x␣=&gt;␣(double)␣x.GetData(
      ↪ metricName,␣DOUBLE));␣theBest␣=␣_ws.First(x␣=&gt;␣(double)␣x.
      ↪ GetData(metricName,␣DOUBLE)␣==␣bestResult);␣/*␣
      ↪ GetWrkBestMetric␣END␣*/">
    <pRINGLParameter>
      <elementHasParameters Id="0c708e67-962e-461c-8645-93148ea01c57"
↪ >
        <pRINGLParameter Id="ec9e8318-4c28-4b39-9405-dbd2d5f66034"
            ↪ type="string" propagation="+" name="metricName" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="ef4b8508-9777-4fb9-949e-
    ↪ ed0bf61b468c">
  <incLogic Id="064c0f5b-a33a-4b95-8b70-2ec2ee9f5ac1" name="IsManager
      ↪ " subMetatype="PredicateLogic" outputType="bool" code="␣/*␣␣
      ↪ BEGIN␣IsManager␣*/␣return␣((int)_parent.getParam(&quot;mgrID&
      ↪ quot;)␣==␣_w.ID);␣/*␣END␣IsManager*/">
    <pRINGLParameter>
      <elementHasParameters Id="a9a2d242-baa7-4a89-9ced-3320b4bba15b"
↪ >
        <pRINGLParameter Id="e5ab2f6c-ebcf-402a-9a5b-f055f7d8b450"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="b2f54279-3a10-4145-a334
    ↪ -7275df020b9b">
  <incLogic Id="a73d33bb-5c99-4fc3-a755-728f6cefc0a1" name="IsBest"
      ↪ subMetatype="PredicateLogic" outputType="bool" code="/*␣␣
      ↪ BEGIN␣IsBest␣*/␣return␣(_w␣==␣GetWrkBestMetric.theBest);␣/*␣
      ↪ END␣IsBest␣*/" />
</incentiveStrategyHasIncentiveElements>
<incentiveStrategyHasIncentiveElements Id="5497bb44-02fb-4fd8-9456-
    ↪ bde402720800">
  <incLogic Id="a7663160-3053-42fe-bb83-0fbff6c908c2" name="
      ↪ GET_MANAGER" subMetatype="StructureLogic" outputType="int"
      ↪ code="/*␣␣BEGIN␣GET_MANAGER*/␣/*GRGEN:GET_MANAGER*/␣/*␣END␣
      ↪ GET_MANAGER*/">
    <pRINGLParameter>
      <elementHasParameters Id="687af626-809a-40ae-a728-42729bbaae94"
↪ >
        <pRINGLParameter Id="4bdf927b-b506-4760-bcd0-65ed7df435ba"
            ↪ type="int" propagation="+" name="teamID" />
      </elementHasParameters>
    </pRINGLParameter>
  </incLogic>
</incentiveStrategyHasIncentiveElements>
```

```
  </incentiveElements>
</incentiveStrategy>
```

Glossary

authority The entity engaging the workers for productive purposes, administering incentives upon them.

dysfunctional behavior Worker behavior targeted by incentive mechanisms, often occurring as a reaction to the application of other incentive mechanisms.

gamification is the use of game elements (e.g., point, competitions, rules and other game mechanics) in non-gaming environments and activities, with the aim of attracting and motivating users to perform the activity in the given environment, which could otherwise prove non-interesting.

incentive Any activity or scheme employed by the authority to stimulate (motivate) increased level of certain work-related activities (e.g., productivity, speed, quality of work, number of participants) or to discourage certain activities (e.g., drop-out rate), before the actual execution of those activities.

incentive element An atomic component (construct) in terms of which incentive mechanisms can be expressed.

incentive mechanism A concrete rule for assigning/applying the rewards targeting a specific (group of) workers, based on certain logical, temporal and spatial criteria; A concrete implementation of an incentive for a given application context.

incentive scheme Combined global effect of the application of a set of incentive mechanisms..

principal See Authority.

reward Any kind of recompense for worthy services rendered or retribution for wrongdoing exerted upon workers during the execution of the activity or after its completion. A reward can be made equivalent of an economic value (money or physical goods), or a social status like prestige, rank, or expertise.

Acronyms

DSL Domain-Specific Language.

MMod Mapping Model. See Section 5.2.2..

PRINGL PRogrammable INcentive Graphical Language.

RMod Rewarding Model. See Section 4.2..

SCU Social Compute Unit.

SDT Self-Determination Theory.