A Programming Model for Hybrid Collaborative Adaptive Systems

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

A Programming Model for Hybrid Collaborative Adaptive Systems

Ognjen Scekic*, Tommaso Schiavinotto†, Svetoslav Videnov*,
Michael Rovatsos‡, Hong-Linh Truong*, Daniele Miorandi†, Schahram Dustdar*
* Distributed Systems Group, TU Wien, Vienna, Austria
Email: oscekic | truong | svidenov | dustdar @dsg.tuwien.ac.at
† U-Hopper, Trento, Italy
Email: tommaso.schiavinotto | daniele.miorandi @u-hopper.com
‡ Centre for Intelligent Systems and their Applications, University of Edinburgh, UK
Email: mrovatso@inf.ed.ac.uk

Abstract—Hybrid Diversity-aware Collective Adaptive Systems (HDA-CAS) are a new generation of socio-technical systems where both human and machine peers collectively participate in complex cognitive and physical tasks. These systems are characterized by the fundamental properties of hybridity and collectiveness, hiding from users the complexities associated with managing the collaboration and coordination of human-machine teams. The SmartSociety platform is a set of integrated software components that jointly provide a number of advanced HDA-CAS functionalities. As part of the CAS initiative, we have developed a programming model and Java APIs that make the use of those functionalities easy and accessible to application developers. In this paper we present the SmartSociety programming model elements, including the principal contributions – Collectives and Collective-based Tasks. We describe and discuss their functionality, implementation and runtime environment. Finally, we qualitatively evaluate the programming model and the language constructs with respect to the desired HDA-CAS properties.

Index Terms—socio-technical systems, collective adaptive systems, crowdsourcing, social computing, programming model

1 Introduction

We have recently witnessed the evolution of conventional social computing and the appearance of new classes of socio-technical systems, which attempt leveraging human expertise for carrying out intellectually challenging tasks [1, 2, 3, 4, 5, 6]. This type of systems is opening up the possibility for novel forms of interaction, collaboration and organization of labor, building upon the complementary strengths of humans and computers. The state-of-the-art, however, is limited to systems using computers to support and orchestrate purely human collaborations, usually based on patterns of work that can be predictably modeled before the execution (Section 6). The innovative approach considered in this paper implies blurring the line between human and machine computing elements, and considering them under a generic term of *peers* – entities that provide different functionalities under different contexts; participating in *collectives* – persistent or short-lived teams of peers, representing the principal entity performing the computation (task).

Peers and collectives embody the two fundamental properties of the novel approach: *hybridity* and *collectiveness*, offered as inherent features of the system. Systems supporting these properties perform tasks and computations transparently to the user by assembling or provisioning appropriate collectives of peers that will perform the task in a collaborative fashion. We call the whole class of these emerging socio-technical systems Hybrid Diversity-Aware Collective Adaptive Systems (HDA-CAS). Engineering and managing such systems is a challenging task, as they present coordination and communication problems that go well beyond what state-of-the-art solutions can tackle. This is particularly apparent when we consider participating humans not merely as computational nodes providing a service at request, but put them on an equal footing and allow them to actively drive computations.

In this paper we present the programming framework and the API for accessing and using the *SmartSociety*[1] *Platform*[2], a novel HDA-CAS supporting a wide spectrum of collaboration scenarios – from simple, independent crowdsourcing tasks to fully human-driven collaborations involving non-trivial execution plan composition with constraint matching and human negotiations (e.g., ride-sharing, collaborative software development).

The paper describes how the presented programming framework design tackles the fundamental HDA-CAS novelty requirements and showcases how the introduced language constructs can be used to encode and execute hybrid collaborations on the SmartSociety platform. In the previous version of this paper [7] we presented the main concepts of a programming model for HDA-CASs. The major additions with respect to the previous work are: (i) the description of a complete implementation of the presented model into an actual, working framework; (ii) the contrivance and implementation of a set of demonstrative examples highlighting the main functionalities and how they can be used to design and

Extended version of: Scekic O. et al., "Programming Model Elements for Hybrid Collaborative Adaptive Systems", IEEE CIC'15, Hangzhou, China. http://dx.doi.org/10.1109/CIC.2015.17

[1] EU FP7 research project (www.smart-society-project.eu)
[2] www.smartcollectives.com

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

manage HDA-CASs; and (iii) an evaluation of the framework's ability to effectively support application developers in programming collective tasks.

The paper is organized as follows: In Section 2, we present the fundamental concepts of HDA-CAS systems and the concrete executable context of the programming framework – the SmartSociety HDA-CAS. In this section we also define the fundamental design requirements that we later use in the evaluation section. In Section 3, the principal programming model elements are introduced and their functionality is described. Section 4 presents the associated programming API. The evaluation of the programming model and the API is presented in Section 5. Related work is described in Section 6 and contrasted to our approach. Finally, Section 7 concludes the paper and points out future directions.

2 Hybrid Collective Adaptive Systems

2.1 Concepts

The focus of our work is on technical support for performing collective activities (tasks) using augmented hybrid collectives. After analyzing the way the current state-of-the-art agent-based systems, workflow systems and socio-technical/crowdsourcing platforms combine human and machine/software elements (peers) for performing complex collaborative activities we were able to observe two general approaches: modeling machine peers to resemble human peers, and modeling humans to be able to cooperate with software peers. The former approach is typical of agent-based systems, where complex software peers are modeled to imitate human peers in an effort to simulate or delegate human behavior/functionality. However, such approaches need to make a large number of assumptions regarding the communication, representations, computational and coordination mechanisms in order to bootstrap the collaborative activity. We end up with elaborate computational agents, which are, however, restricted exclusively to the foreseen, application-specific collaboration scenarios.

The latter approach models the humans as machine elements in an attempt to include humans into existing workflow/orchestration platforms. Through this simplification and reduction of human peers to an API, we willingly renounce the extraordinary cognitive and creative capabilities that a human can provide in order to include it in an existing system. Such approaches are able to support complex collaborative activities if the execution plan (execution steps, execution order and conditions, executing roles) is known at design-time. Both approaches seem to lack the versatility required when attempting to manage collective collaborations spanning both human and software elements in the physical world, where the environment, peers and the workflow itself are volatile. Therefore, the challenge of SmartSociety was to design an HDA-CAS that would be appropriate for such environments.

The following are therefore the principal defining properties of the SmartSociety HDA-CAS which also needed to be reflected in the programming model as its *design requirements*:

a) **Collectiveness** – Individual peer is secondary to the group/team. The collective is the first-class entity managed by the platform. *b)* **Hybridity** – The platform supports a mixture of different types of peers (humans, software services, devices) working in concert within the same collective. *c)* **Diversity** – The platform is able to manage heterogeneous peers towards a common collective goal, by composing or aligning their individual diverging characteristics, abilities and goals. *d)* **Adaptivity** – The platform is able to dynamically compose and execute runtime

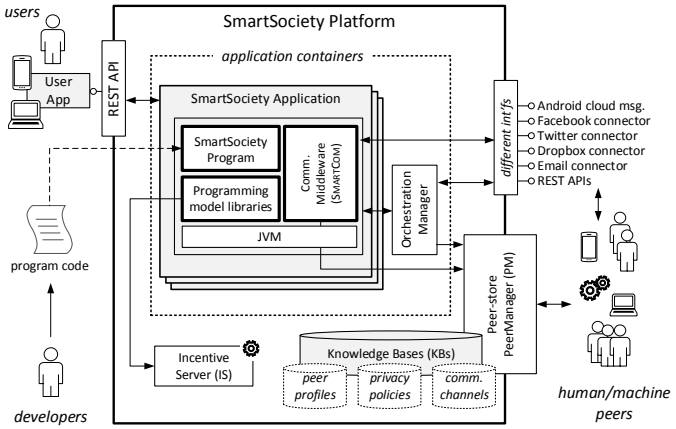

Fig. 1: SmartSociety platform users and architecture. Programming framework elements outlined.

(ad-hoc) workflows, as opposed to executing static, predefined workflows. *e)* **Human orchestration** – Closely related to the previous point, the human orchestration refers to possibility of human peers determining and actively influencing (adapting) the workflow at runtime. This is especially important for solving cognitive, creative, artistic problems. *f)* **Privacy compliance** – The platform discloses as little private information about the peers as required to take part in the collective activity. The peer's preferences and terms of participation are transparent and respected by the platform.

Furthermore, the SmartSociety Platform was designed to handle complex collaborative tasks, where the complexity is not reflected in the numerosity of the participants, but the complexity of the tasks. Unlike the embarrassingly-parallel tasks typical of conventional crowdsourcing, we put focus on *neighborhood-scale* collectives with up to $\mathcal{O}(10^3)$ participants.

2.2 Implementation – the SmartSociety Platform

The SmartSociety platform, shown in Figure 1, is a software framework intended for use by the following user roles:

1) *Users* – external human clients or applications who need a complex collaborative human-machine task performed.
2) *Peers* – human or machine entities participating in task executions managed by a platform application.
3) *Developers* – external individuals providing the business logic in form of programming code that is compiled and executed on the platform as a platform application.

The platform acts as intermediary between users and peers, providing a collaborative task execution environment and workforce management functionality. Note that the same physical person can act at the same time both as a user and a peer. The platform offers a variety of commonly used coordination, orchestration, communication and incentivization mechanisms as ready-made concepts exposed through the programming API.

Interested human peers register their profiles with the platform and enlist for performing different professional activities. The platform uses this data for locating and engaging peers into different collaborative efforts. In case of human peers, the platform asks for an explicit approval, enabling the peer engagement under a short-term contractual relationship. In case of a software peer, the services are contracted under conventional service-level

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

agreements (SLAs). Registered users are the basis from which appropriate peers are selected to take part in *collectives*, which then execute collaborative tasks. A collective is composed of a team of peers along with a *collaborative environment* assembled for performing a specific task. The environment consists of a set of software communication and coordination tools. For example, as described in [8], the platform is able to set up a predefined virtual communication infrastructure for the collective members and provide access to a shared data repository (e.g., Dropbox folder). The complete collective lifecycle is managed by the platform in the context of a SmartSociety *platform application* (Fig. 1). A platform application consists of different modules, one of which is a *SmartSociety program* – a compiled module containing the externally provided code that: *a)* implements the desired business logic of the user; *b)* manages the communication with the corresponding user applications; and *c)* relies on libraries implementing the programming model to utilize the full functionality of the platform. Through a *user application*, users can submit *task requests* to the platform. The user application communicates with the corresponding platform application.

Platform Architecture & Functionality

A simplified, high-level view of the SmartSociety platform architecture is presented in Fig. 1. The rectangular boxes represent the key platform components. The principal component-interoperability channels are denoted with double-headed arrows. Communication with peers is supported via popular commercial services (e.g., Twitter, Dropbox, Android cloud messages). User applications contact the platform through the REST API component. All incoming user requests are served by this module that dispatches them to the appropriate SmartSociety program, which will be processing and responding to them. The program is a Java application making use of SmartSociety platform's programming model libraries, which in turn expose the functionality of different platform components:

PeerManager (PM): Central peer data-store (*peer-store*) of the platform. Manages all peer and application information and allows privacy-aware access and sharing of the peer/collective data among platform components and applications. Details on how the PeerManager is implemented can be found in [9].

Orchestration Manager (OM): Component in charge of orchestrating collaborative activities among human peers. OM's core functionalities [10] (also reflected in the programming model) are: a) Composition – generating possible execution plans to meet user-set constraints and optimize wanted parameters; and b) Negotiation – coordinating the negotiation process among human peers leading to the changes in the execution plan and the overall agreement and ultimate acceptance of the plan.

Incentive Server (IS): An independent component that monitors and motivates peer participation through controlled interventions, using machine learning methods to adapt to various contexts [11]. It supports two modes of operation: a) Sustained incentivization over a longer period of time where the IS algorithms monitor participation and through machine learning determine optimal intervention times and types of incentive messages that are sent to peers; and b) Per-task incentivization, where actual SmartSociety applications prompt incentivization via IS and determine the target group and possibly intervention times.

Communication and Virtualization Middleware: The middleware named SMARTCOM is used as the primary means of communication between the platform and the peers and among the peers. It supports the typical messaging middleware functionalities (delivery, routing, transformation) thus virtualizing peers by homogenizing the communication with both human and software-based peers to the remainder of the platform [8]. SMARTCOM is tightly integrated into the programming model, where it is used to allow collective communication, peer negotiations and setting up of collaborative environments.

3 PROGRAMMING FRAMEWORK

3.1 Runtime Environment & Execution Model

The developer who wishes his/her application deployed on the SmartSociety platform provides a set of classes (`Application`, `TaskRequestDefinition`, `TaskRunner`, `SmartSocietyApplicationContext`) in a Java jar file. The provided classes specify: a) actual business logic of the new application; b) the runtime context of the application (e.g., initialization and configuration parameters for handlers (Sec. 3.2, loading of predefined collective kind definitions); c) the logic for interpreting and (un)marshalling the requests and responses. Upon submission, a set of Docker containers are created. The main one will be running the application. The submitted jar is injected into this container. Remaining containers are created for deploying other platform components. A new application ID is generated and assigned to the application. The application is then registered as a user with special access privileges with other platform components. Finally, the runtime initializes the context, the application-specific URL endpoint, and starts the application.

Whenever a request is received via the URL endpoint it is interpreted according to the `TaskRequestDefinition` to produce a new `TaskRequest` which represents the main input for our programming model. Figure 2 illustrates this process. For each request a thread is started, executing the application's arbitrary business logic from the `TaskRunner` class. When this logic requires some collaborative processing the developer uses the programming model library constructs to create and concurrently execute a *Collective-based Task (CBT)* – an object encapsulating all the necessary logic for managing complex collective-related operations on the SmartSociety platform: team provisioning and

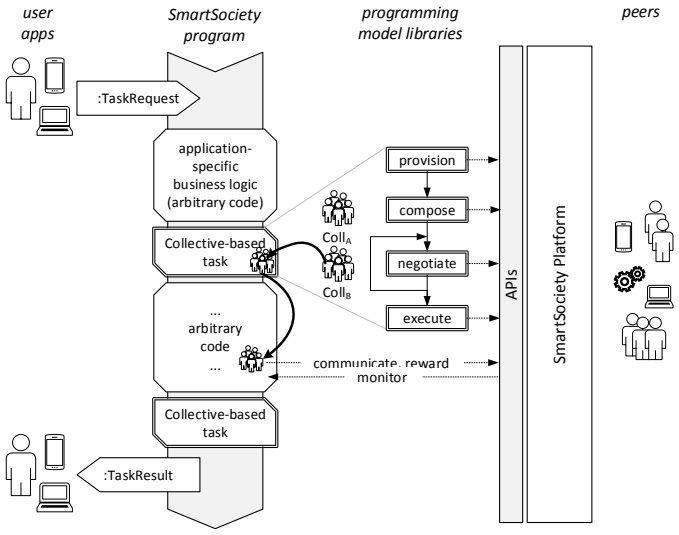

Fig. 2: Using the SmartSociety programming framework.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

assembly, execution plan composition, human participation negotiations, and finally the execution itself. During the lifetime of a CBT, various *Collectives* related to the CBT are created and exposed to the developer for further (arbitrary) use in the remainder of the code outside of the context of the originating CBT or its lifespan. Developer can communicate with the collectives, incentivize them, persist them, and combine them to produce new collectives to pass as inputs to other CBTs.

3.2 Task Management

A collective-based task (CBT) is the element of the programming model keeping the state and managing the lifecycle of a collective task. A CBT instance is always associated with a `TaskRequest` containing input data and possibly a `TaskResult` containing the outcome of the task. Both are very generic interfaces meant to hide from the programming model the application-specific format of the input and output data, respectively. In fact, the programming model is designed to be *task-agnostic*. This is in line with the general HDA-CAS principle that unconstrained collaboration should be supported and preferred when possible. This design choice was made to allow subsequent support of different task models which will be interpretable by the application-specific Orchestration Manager, or by human peers directly. This is arguably at the same time a weakness of our approach, making it vulnerable to possible accidental or purposeful misinterpretations. However, we believe that it is a necessary sacrifice to make in order to achieve an effective inclusion of humans into hybrid collaborations.

A CBT can be processed in one of the two collaboration models – (*on demand* and *open call*); or a combination of the two, as specified by the developer. Table 1 lists the allowed combinations and describes them in more detail.

on_demand = true ∧ *open_call* = true
A collective of possible peers is first provisioned, then a set of possible execution plans is generated. The peers are then asked to negotiate on them, ultimately accepting one or failing (and possibly re-trying). The set of peers to execute the plan is a subset of the provisioned collective but established only at runtime.
on_demand = true ∧ *open_call* = false
The expectedly optimal collective peers is provisioned, and given the task to execute. The task execution plan is implicitly assumed, or known before runtime. Therefore no composition is performed. Negotiation is trivial: accepting or rejecting the task.
on_demand = false ∧ *open_call* = true
"Continuous orchestration". No platform-driven provisioning takes place. The entire orchestration is fully peer-driven (by arbitrarily distributed arrivals of peer/user requests). The platform only manages and coordinates this process. Therefore, neither the composition of the collective, nor the execution plan can be known in advance, and vary in time, until either the final (binding) agreement is made, or the orchestration permanently fails due to non-fulfillment of some critical constraint (e.g., timeout). Note that in this case the repetition of the process makes no sense, as the process lasts until either success or ultimate canceling/failure.
on_demand = false ∧ *open_call* = false
Not allowed/applicable.

TABLE 1: CBT collaboration models and selection flags

At CBT's core is a state machine (Fig. 3) consisting of states representing the eponymous phases of the task's lifecycle: `provisioning`, `composition`, `negotiation` and `execution`. An additional state, `continuous_orchestration`, is used to represent a process combining composition and negotiation under specific conditions, as explained in Table 1. The collaboration model selection flags are used in state transition guards to skip certain states. Each state consumes and produces input/output collectives during its execution. All these collectives get exposed to the developer through appropriate language constructs (Listing 5) and are subsequently usable in general program logic.

Each state is associated with a set of *handlers* with predefined APIs that needs to be executed upon entering the state in a specific order. The handlers registered for a specific application are assumed to know how to interpret and produce correct formats of input and output data, and wrap them into `TaskRequest` and `TaskResult` objects. By registering different handler instances for the states the developer can obtain different overall execution of the CBT. For example, one of the handlers associated with the `execution` state is the 'QoR' (quality of result) handler. By switching between different handler instances, we can produce different outcomes of the execution phase. Similarly, by registering a different handler, an OM instance with different parameters can be used. The programming model libraries provide a set of default handlers exposing the ground platform functionalities, such as orchestration and negotiation algorithms provided by the Orchestration Manager or provisioning algorithms (e.g., [12]). Concrete handlers are registered at initialization for each CBT type used in the application.

Provisioning state: The input to the state is the CBT input collective specified at CBT instantiation (often a predefined collective representing all the peers accessible to the application). In our case, the process of provisioning refers to finding a set of human or machine peers that can support the computation, while being optimized on e.g., highest aggregate set of skills, or lowest aggregate price. See [12] for examples of possible provisioning algorithms. Provisioning is crucial in supporting hybridity in the programming model, because it shifts the responsibility of explicitly specifying peer types or individual peers at design time from the developer onto the provisioning algorithms executed at runtime, thus making both human and machine-based peers eligible depending on the current availability of the peers and the developer-specified constraints. The bootstrapping aspect of provisioning refers to finding and starting a software service, or inviting a human expert to sign up for the participation in the upcoming computation; and setting up the communication topology among them via SMARTCOM (cf. Sec. 3.5). The output of the state is the 'provisioned' collective, that gets passed on to the next state during the execution.

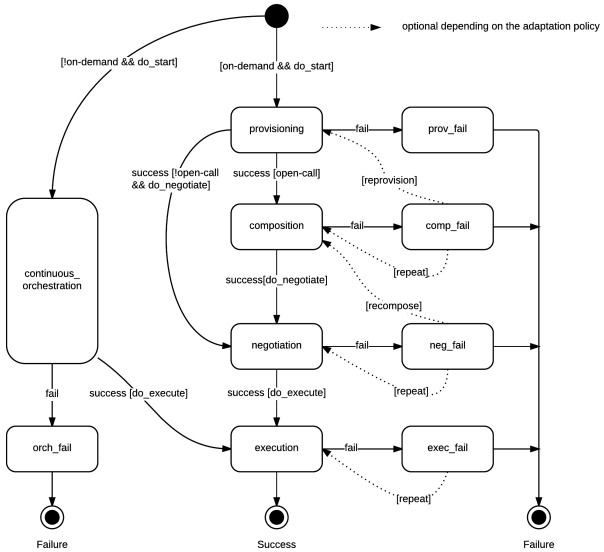

Fig. 3: CBT state diagram.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

Composition state: The composition process calculates feasible task execution plans, consisting of ordered activities (steps) required to process the given task and associated performer peers. Generation of execution plans is usually a task-specific, non-trivial problem involving advanced planning and constraint satisfaction algorithms, going well beyond the scope of this paper; the description of the currently offered composition algorithms can be found in [10]. If there is no requirement of optimality, then a human peer can be used to compose an ad-hoc, non-optimal plan. From the programming model's perspective, however, it suffices to know the required inputs and outputs of this state: the input is the 'provisioned' collective from the previous state, while the output is a list of collectives 'negotiables', associated with composed execution plans, which get passed on to the following state.

Negotiation state: Involves selecting one or more execution plans passed as inputs from the `composition` state and enacting a negotiation process on them. If the state is entered directly from the `provisioning` state, the execution plan is implied, and assumed to be implicitly understood by participating peers. The negotiation is a complex collaborative process involving human peers, members of the collective associated with the plan, expressing their participation conditions and (potential) participation acceptance. How exactly a negotiating process unfolds is guided by the *negotiating pattern* specified by the developer. For example, the pattern may stipulate that at a given time only one plan can be actively negotiated, and that the participation in this plan must be reached through the consensus of all peers belonging to the associated collective. An alternative pattern may allow negotiation of multiple plans in parallel, and termination of the negotiation process as soon as one plan is accepted by a simple majority. The output of the negotiation process is the single 'agreed' collective and the associated execution plan.

Continuous orchestration state: Continuous orchestration (cf. Table 1) does not separate composition and negotiation, but rather allows continuous switching between (re-)composing and negotiating. Each new task request submitted by user re-triggers composition, allowing the peers to temporarily accept plans and later withdraw, until the plan is ultimately considered accepted and thus becomes ready for execution, or ultimately fails. Note that repetition of this state is not applicable, because repetition is generally done in case of remediable failures, but in this case the orchestration lasts until the execution starts (a non-revocable success) or a non-revocable failure is detected (e.g., a ride to work makes no sense after working hours have already begun). As continuous orchestration is completely human-driven, the developer is expected to specify only the reference collective while the planning and negotiations are handled by the peers. The output is the 'agreed' collective (a subset of the input one) and the associated execution plan.

As an example of real-world continuous orchestration, assume a ride sharing scenario: users submit driving offers, peers submit passenger offers. An execution plan in this case is the description of the possible route along with information on vehicle, driver and passengers for each route section. If enough requests are submitted, a number of plans matching hard (time/destination) constraints are generated. However, a number of soft constraints influence the human negotiations: drivers prefer different passengers (due to personal preferences or price); passengers prefer different rides depending on the vehicle, fellow-passengers, cost, duration and the number of transfers. All potential driver/passenger peers are allowed to participate in negotiations for multiple plans in parallel, and to accepting or withdraw from multiple plans while they are valid. As soon as all required peers accept it, the plan is considered agreed. However, the plan can exist in agreed state, but still revert to non-agreed if some peers change their mind before the actual execution takes place. Furthermore, this affects other plans: if a passenger commits to participating in ride A, then ride B may become non-agreed if his presence was a required condition for executing the ride B. When the actual plan (ride) finally starts executing, or its scheduled time is reached, the plan is non-revocable; if it is in addition in agreed state, it can get executed. Otherwise, the `orch_fail` state is entered. More details on the supported orchestration algorithms are provided in [10].

adaptation policy	*description*
`ABORT`	Default. Do nothing, and let the fail state lead to total failure.
`REPEAT`	Repeats the corresponding active state, with (optionally) new handler(s).
`REPROVISION`	Transition into `provisioning` state, with (optionally) a new provisioning handler.
`RECOMPOSE`	Repeat the composition, with (optionally) a new handler.

TABLE 2: CBT adaptation policies.

Execution state: The execution state handles the actual processing of the agreed execution plan by the 'agreed' collective. In line with the general HDA-CAS guidelines, this process is willingly made highly independent of the developer and the programming model and let be driven autonomously by the collective's member peers. Since peers can be either human or software agents, the execution may be either loosely orchestrated by human peer member(s), or executed as a traditional workflow, depending on what the state's handlers stipulate. For example, in the simplified collaborative software development scenario shown in Listing 5 both CBTs are executed by purely human-composed collectives. However, the `testTask` CBT could have been initialized with a different type, implying an execution handler using a software peer to execute a test suite on the software artifact previously produced by the `progTask` CBT. Whether the developer will choose software or human-driven execution CBTs depends primarily on the nature of the task, but also on the expected execution duration, quality and reliability. In either case, the developer is limited to declaratively specifying the CBT's type (handlers), the required the termination criterion and the Quality of Results (QoR) expectations. The state is exited when the termination criterion evaluates to true. The outcome is 'success' or 'failure' based on the value of QoR metric. In either case, the developer can fetch the `TaskResult` object, containing the outcome, and the evaluation of the acceptability of the task's quality.

Fail states: Each of the principal states has a dedicated failure state. Different failure states are introduced so that certain states can be re-entered, depending on what the selected *adaptation policy* specifies. Some failure states react only to specific adaptation policies; some to none.

Adaptation policies

An adaptation policy is used to enable re-doing of a particular subset of CBT's general workflow with different functionality and parameters, by changing/re-attaching different/new handlers to the CBT's states, and enabling transitions from the failure states back to active states. The policies are triggered upon entering failure states, as shown in Figure 3. The possible transitions are marked with dotted lines in the state diagram, as certain policies make sense only in certain fail states. Adaptation policies

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

allow for completely changing the way a state is executed. For example, by registering a new handler for the `provisioning` state a different provisioning algorithm can be used. Similarly, a new handler installed by the adaptation policy can in a repeated `negotiation` attempt use the "majority vote" pattern for reaching a decision, instead of the previous "consensus" pattern. Natively supported predefined policies are described in Table 2. Only a single adaptation policy is applicable in a single failure state at a given time. If no policy is specified by the developer, the `ABORT` policy is assumed (shown as full-line transition in CBT state machine diagram).

3.3 Collective Management

The notion of "collective" in HDA-CAS terminology sometimes denotes a stable group or category of peers based on the common properties, but not necessarily with any personal/professional relationships (e.g., 'Java developers', 'students', 'Vienna residents'); in other cases, the term refers to a team – a group of people gathered around a concrete task. The former type of collectives is more durable, whereas the latter one is short-lived. Therefore, we make following distinction in the programming model:

Resident Collective (RC): is an entity defined by a persistent peer-store identifier, existing across multiple application executions, and possibly different applications. Resident collectives can also be created, altered and destroyed fully out of scope of the code managed by the programming model. The control of who can access and read a resident collective is enforced solely by the 'peer-store' (in our case the PeerManager component). For those resident collectives accessible from the given application, a developer can read/access individual collective members as well as all accessible attributes defined in the collective's profile. When accessing or creating a RC, the programming model either passes to the peer store a query and constructs the corresponding object from returned peers, or passes an ID to get an existing peer-store collective. In either case, in the background, the programming model will pass to the peer-store its credentials. The peer store then decides based on the privacy rules which peers to expose (return). For example, for the requested collective with ID 'ViennaResidents' we may get all Vienna residents who are willing to participate in a new (our) application, but not necessarily all Vienna residents from the peer-store's DB. By default, the newly-created RC remains visible to future runs of the application that created it, but not to other applications. The peer-store can make them visible to other applications as well. At least one RC must exist in the application, namely the collective representing all peers visible to the application.

Application-Based Collective (ABC): Differently than a resident collective, an ABC's life cycle is managed exclusively by the SmartSociety application. Therefore, it is not possible (and is meaningless) to access an ABC outside of the application's execution context. The ABCs are instantiated: *a*) implicitly – by the programming model libraries as intermediate products of different states of CBT execution (e.g., 'provisioned', 'agreed'); or *b*) explicitly – by using dedicated collective manipulation operators to clone a resident collective or as the result of a set operation over existing Collectives. Also differently than resident collectives, ABCs are atomic and immutable entities for the developer, meaning that individual peers cannot be explicitly known or accessed/modified from an ABC instance. The ABCs embody the principle of collectiveness, making the collective an atomic, first-class citizen in our programming model, and encouraging the developer to express problem solutions in terms of collectives and collective-based tasks, rather than single activities and associated individuals. Furthermore, as collective members and execution plans are not known at design time, this enhances the general transparency and fairness of the virtual working environment, eliminating subjective bias.

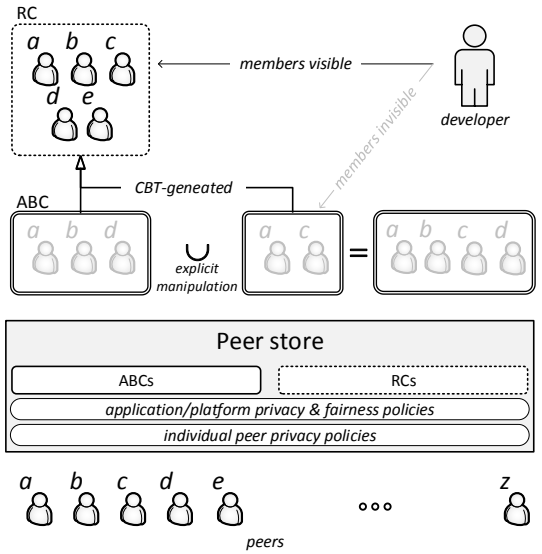

Fig. 4: Differences between RCs and ABCs. ABCs are atomic to the Developer, although the Developer is able to manipulate and create new descending ABCs.

One of the reasons for introducing the concept of collectives with the described properties is to prevent the User/Developer from using individual human peers as mere computing/processing nodes being assigned activities to perform, instead favoring a more ethical (teamwork) approach. Furthermore, the distinction and existence of both RC and ABC `Collective` models (Fig. 4) allows a trade-off between hand-picking the team members and the flexibility offered between a platform-managed collective provisioned based on user's requirements. The rationale in the latter case is similar to cloud computing – the user specifies the infrastructural requirements and constraints and the platform takes care to provision this infrastructure, without letting the user care about which particular VM instances are used and changed. Different use-cases, privacy and fairness policies may dictate or favor the choice of one `Collective` type over the other. For example, when assembling an input collective of experts for a CBT, the User may require to use as source the RC representing the peers with whom the User had positive previous experiences with. Although this seems like a reasonable request, over time the peer community might start exhibiting the characteristics of a scale-free network due to the preferential attachment method of choosing the collective members [13]. This, in turn, may lead to discouragement of less prominent peers, and in overall, increase the attrition rate. To prevent this, the fairness policy of the application/platform enforced at the peer store may prevent handpicking of peers, and impose the use of ABCs provisioned transparently to the Developer/User in accordance with the fairness policy (e.g., round-robin or random peer assignment with reputation threshold). This is important for establishing attractive and competitive virtual crowd marketplaces [14].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

3.4 Peer Management

Personal Data Management and Privacy Compliance

In order to fulfill the HDA-CAS design requirement of privacy compliance, the programming framework inherits the privacy features of the PeerManagr component described in [9]. It allows the programming framework to fetch peer profiles from the peer-store through a privacy-enforcing PM API containing only the data (possibly semantically obfuscated) previously approved by the user himself (upon signing up and creating the profile) for use by that particular application. By performing the queries through the PM intermediary, the programming framework is not allowed to access nor can it perform queries over non-approved peer/collective attributes. The most of the privacy-assurance responsibility is thus delegated to the PM.

The attribute definitions are provided within the *collective kind*. It is a set of collective attribute descriptors supplied at the application registration time and subject to approval by the platform operator. If approved, the collective kinds used by the application are referenced in the `SmartSocietyApplicationContext` class that is used to initialize the application. Thereafter, the developer can access those attributes directly from the application's code through the `ABC.get/setAttribute()` methods. Note that the use of these methods cannot override the original privacy settings – originally obfuscated/hidden data is still not accessible, and setting a user attribute cannot override an original attribute.

For the purposes of testing and usage in non-privacy-critical environments, the programming framework additionally implements a local MongoDB-based peer-store implementing the same PM API but without privacy enforcement. The local peer-store can also be used to cache the PM peer profiles for faster access.

3.5 Communication Management

As mentioned in Section 2.2, the programming framework now directly incorporates a fully-fledged communication middleware SMARTCOM, allowing it to perform advanced collective communication in the background transparently to the developer. While a broader description of the offered functionalities goes beyond the scope of this paper (provided as [8] for reference), we would like to briefly discuss the capabilities that set apart the SMARTCOM and, consequently, the programming framework from the existing state-of-the-art.

By supporting a single communication endpoint model, but different delivery channels, the framework is able to communicate with heterogeneous peers during the various CBT lifecycle phases (e.g., inviting peers at provisioning, task acceptance during negotiations, termination determination during execution). Each individual peer can use personally stipulated communication protocols and modes to interact with the framework/application, e.g., a human peer can communicate via email and Twitter interchangeably, receive task descriptions and track progress through a web application, and communicate with other peers within the collective through a dedicated mobile app. Human peers can make use of software peers in the collective, serving as collaborative and utility tools. For example, a software service like Dropbox can be used as a common repository for sharing artifacts.

This rich communication model is regulated through the use of *privacy* and *delivery policies*. Each peer can individually stipulate a set of privacy policies which regulate: a) times at which the platform may communicate with the peer (e.g., working days 09-17h); and b) blacklisted message senders (to avoid becoming a member of collective with unwanted peers). On the other hand, the developer or the programming framework can stipulate different delivery policies for peers and collectives. The policy determines what constitutes a successful delivery: in case of peers – with respect to different delivery channels; in case of collectives – with respect to collective members. The programming framework supports out of the box a number of basic policies, such as best effort (`TO_ANY`) or exhaustive (`TO_ALL`). In practice, the delivery policies allow us to fine-tune the sensitivity and reliability of the communication with collectives. For example, we can consider that a whole collective has been notified if a single member (perhaps the leader) has been notified of an event; we can request delivery reports, or repeat delivery attempts. Taken together, the privacy and delivery policies are a versatile mechanism to combine and match the preferences of both peers and the developer. They are both storable both in the local as well as in the PM peer-store.

3.6 Incentive management

The programming model uses the Incentive Server's *per-task incentivization mode* (Sec. 2.2) to provide the incentivization functionality through the programming model. In this mode, the IS relies on external components for providing the incentivization logic and application conditions, and limits itself to providing the delivery of motivational messages in a privacy and ethically-compliant way. This allows the IS to be used in arbitrary scenarios/applications (unlike the *sustained incentivization* mode, when IS functions as a fully independent, but application-specific component that needs to be manually set up).

```
1 { "target_collective_ID": 4853,
2   "timestamps": [T1, T2, ... Tn],  # optional
3   "inc_type": ["someID", {incentive-specific params}] }
```

Listing 1: Setting up an incentive intervention through IS.

Due to the high scenario-specificity of incentives, the programming framework currently does not itself provide any incentive logic and management capabilities. Instead, in order to remain task agnostic, the responsibility for specifying the type of incentives, the target peers, and the intervention timing is left to the developer using the programming API, who can explicitly trigger the incentive campaign over a certain collective when deemed appropriate:

- `cbt.incentivize(incType [, times])`

The currently active collective of the CBT upon invocation of incentivization is considered as the target collective (e.g., the initial input collective if the CBT is in the provisioning state). If no timings are specified (as a sequence of timestamps), a single immediate intervention is triggered. In the future, a finer grained and automated timing specification will be implemented, allowing to declaratively specify incentive strategies to coincide with the execution of specific CBT handlers, as incentives often need to be focused to a specific phase and collective of the CBT's life cycle (e.g., at repeated execution of provisioning, or only during the negotiation/execution). Alternatively, the developer can invoke the incentivization directly on a collective (see Listing 6):

- `targetCollective.incentivize(incType [, times])`

In order to invoke the IS incentivization intervention (Listing 1) the PM is assumed as intermediary, not only to uniquely identify the collective's members via the collective ID, but also to prevent unsolicited incentivization messages to the peers that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

have opted out. Internally, the IS also uses a private SMARTCOM instance to handle communication with the peers.

Here are two incentivization examples to illustrate the current incentivization capabilities:
1) CBT is executing and we are at 80% of the allowed execution time, but a certain subset of the agreed peers has not performed their part of the job. Isolate only those as the target collective and try to motivate them to contribute.
2) CBT has finished executing successfully, and the developer wants to foster future relationship with the peers that took part by instructing the IS to issue reminders about how appreciated their contribution was 3x within next 6 months.

4 PROGRAMMING API

The functionality of the programming model is exposed through various associated language constructs constituting the SmarSociety *Programming API*. Due to space constraints, in this section we do not describe the full API, which is rather provided as a separate document[3]. Instead, we describe the supported groups of constructs and their functionality, and some representative individual methods. The examples in Section 5.2 showcase the use of these constructs.

CBT instantiation: This construct allows instantiating CBTs of a given type, specifying the collaboration model, inputs (task request and input collective) as well as configuring or setting the non-default handlers. In order to offer a human-friendly and comprehensible syntax in conditions where many parameters need to be passed at once, we make use of the nested builder pattern to create a "fluent interface".

CBT lifecycle operations: These constructs allow testing for the state of execution, and controlling how and when CBT state transitions can happen. Apart from getters/setters for individual CBT selection (state) flags, the API provides a convenience method that will set at once all flags to true/false:

- `setAllTransitionsTo(boolean tf)`

Since from the initial state we can transition into more than one state, for that we use the method:

- `void start()` – allows entering into `provisioning` or `continuous_orchestration` state (depending which of them is the first state). Non-blocking call.

Furthermore, CBT implements the Java 7 `Future` interface[4] and preserves its semantics. This offers a convenient and familiar syntax to the developer, and allows easier integration of CBTs with legacy code. The `Future` API allows the developer to control and cancel the execution, and to block on CBT waiting for the result:

- `TaskResult get()` – waits if necessary for the computation to complete (until `isDone() == true`), and then retrieves its result. Blocking call.
- `TaskResult get(long timeout, TimeUnit unit)` – same as above, but throwing appropriate exception if timeout expired before the result was obtained.
- `boolean cancel(boolean mayInterruptIfRunning)` – attempts to abort the overall execution in any state and transition directly to the final fail-state. The original Java 7 semantics of the method is preserved.
- `boolean isCancelled()` – Returns true if CBT was canceled before it completed. The original Java 7 semantics of the method is preserved.

Listing 6 (:3-5, 7, 16, 21, 28) shows the usage of some of the constructs.

CBT collective-fetching operations: As explained in Sec. 3.3, during the CBT's lifecycle multiple ABCs get created ('input', 'provisioned', 'negotiables', 'agreed'). These constructs serve as getters for those collectives. At the beginning of CBT's lifecycle, the return values of these methods are null. During the execution, the executing thread updates them with current values. Listing 5 (:20-21) shows examples of these constructs.

Collective manipulation constructs: These constructs allow instantiations of RCs by running the queries on the peer-store (PeerManager), or by creating local representations of already existing peer-store collectives with a well-known ID. We assume that the peer-store checks whether we are allowed to access the requested collective, and filters out only those peers whose privacy settings allow them to be visible to our application's queries.

- `ResidentCollective createFromQuery(PeerMgrQuery q, string to_kind)` – Creates a collective by running a query on the PeerManager.
- `ResidentCollective createFromID(string ID, string to_kind)` – Creates a local representation of an already existing collective on the PeerManager, with a pre-existing ID.

This group also contains methods for explicitly instantiating ABCs. Due to specific properties of ABCs (Sec. 3.3), they can only be created through cloning or set operations from already existing collectives (both RCs and ABCs). These operations are performed in a way that preserves atomicity and immutability. Finally, a method for persisting the collectives at the peer-store is also provided.

- `ABC copy(Collective from, [string to_kind)]` – Creates an ABC instance of kind `to_kind`. Peers from collective `from` are copied to the returned ABC instance. If `to_kind` is omitted, the kind from collective `from` is assumed.
- `ABC join(Collective master, Collective slave, [string to_kind)])` – Creates an ABC instance, containing the union of peers from Collectives `master` and `slave`. The resulting collective must be transformable into `to_kind`. The last argument can be omitted if both `master` and `slave` have the same kind.
- `ABC complement(Collective master, Collective slave, [string to_kind)])` – Creates an ABC instance, containing the peers from Collective master after removing the peers present both in master and in slave. The resulting collective must be transformable into `to_kind`. The last argument can be omitted if both `master` and `slave` have the same kind.
- `void persist()` – Persist the collective on peer-store. RCs are already persisted, so in this case the operation defaults to renaming.

Listing 5 (:1-2, 19-22) shows examples of these constructs.

5 EVALUATION

The implementation of the programming framework refers to the two principal elements – the *programming API* and the *programming model libraries* (including the various subcomponents required to integrate the functionality of other SmartSociety platform components). In this section we present a *qualitative evaluation* of the two elements, and an initial performance evaluation. This is a common evaluation methodology during the development and prototyping phase [15, 16] when the immaturity of the implemented prototype would affect the validity of a full-scale quantitative evaluation. For evaluating the programming model, we analyze the fulfillment of the the HDA-CAS design requirements formulated in Section 2.1 by presenting and discussing an example suite covering the said properties. For evaluating the programming API, we perform an analysis of language characteristics with respect to functionality and usability. Finally, we assess the gained performance and conciseness improvements when using the presented programming model. Comparative analysis was not applicable in this case, due to nonexistence of similarly expressive

[3] http://tinyurl.com/smartsoc-prog-api
[4] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

models (Section 6). The source code of the programming framework, as well as of the presented examples is provided via our repository[5].

5.1 Programming Model Evaluation

Table 3 shows the coverage of HDA-CAS design requirements by the evaluation examples that are presented in the continuation.

	Ex. 1	Ex. 2	Ex. 3	Ex. 4
Hybridity	✓	✓		
Diversity			✓	✓
Adaptivity		✓		
Collectiveness			✓	✓
Privacy-compliance	✓	✓	✓	
Human orchestration		✓	✓	✓

TABLE 3: Coverage of HDA-CAS design requirements.

Example 1

A user posts a question to the platform about restaurants with specific eating options in the vicinity. The question is processed by the platform application representing our example and returned to the user. The processing collective is hybrid, as it involves software services (Google/Yelp) that are queried for restaurants as well as human peers queried via email or Twitter in accordance with their personal delivery and privacy policies. The orchestration (plan composition) is software-managed – the workflow is predetermined, and there is no need for negotiation. The workflow prescribes how the queries are dispatched and how the replies are aggregated (e.g., preference for human replies if high reputation or multiple matching answers from more than two peers; otherwise, preference for the software peer reply).

The relevant application code for performing such a task is short and straightforward (Listing 2). Note that the code for collective provisioning and execution (aggregation) is already packaged within the CBT handlers, which form part of the programming library, meaning that the developer only needs to provide the parameters to use them. While the collective itself is hybrid, there is no direct collaboration (collectiveness) between the peers. This scenario therefore simulates the conventional crowdsourcing scenarios, where a task is solved through microtask parallelization with a predefined workflow without human orchestration.

```
/* Initialize and parameterize provided library handlers: */
pH1 = new ...pf.cbthandlers.provisioning.AskSmartSocietyProvHandler(N)
eH1 = new ...pf.cbthandlers.execution.AskSmartSocietyExecHandler()
cH1 = new ...pf.cbthandlers.composition.AskSmartSocietyExecHandler(5, T)

TaskRequest taskReq = ... ;
Collective nearbyPeers =
    ABC.createFromQuery(new CollectiveQuery(requestor.location, ...));

TaskFlowDefinition tfd =
    TaskFlowDefinition.onDemandWithOpenCall(pH1,cH1,null,eH1)
                      .withInputCollective(nearbyPeers);

CollectiveBasedTask cbt =
    CBTBuilder.from(tfd).withTaskRequest(taskReq).build();

cbt.start();

return cbt.get(10, TimeUnit.MINUTES);
```

Listing 2: Hybrid, crowdsourced answering with SmartSociety.

Example 2

In this example, we extend the previous example to include adaptivity and human orchestration. In particular, we want to allow local experts to select the top-pick restaurant among the list of proposed ones. However, this scenario presents an issue, in that the local expert may be unavailable or slow in responding, which could result in a failure of the whole task. In such cases, we need a failover strategy. The programming model offers 2 possibilities in this case: a) detect the delay/failure through the use of the CBT API and do an arbitrary corrective action, e.g., create a new CBT with different parameters, perhaps along with an incentivization intervention to increase the likelihood of success, or simply apologize to the user for failing; or b) define an adaptation policy to take an automated corrective action (in our case failover to the scenario from Example 1). Listing 3 shows the relevant code snippet variant b). We provide the source code for both variants in our repository.

```
/* Initialize library handlers -- pH2, cH2, eH1, eH2.
 * eH2 uses human orchestration. Failover handler eH1 (from Ex. 1). */

TaskRequest taskReq = ... ;
Collective nearbyPeers = ...;

pH2 = pH2.onFail(AdaptationPolicy.REPEAT, Ph1);

TaskFlowDefinition tfd =
    TaskFlowDefinition.onDemandWithOpenCall(pH2, cH2, null, eH2)
                      .withInputCollective(nearbyPeers);

CollectiveBasedTask cbt =
    CBTBuilder.from(tfd).withTaskRequest(taskReq).build();

cbt.start();

/*...*/
```

Listing 3: Human orchestration and adaptation.

Example 3

This example simulates a simplified ride-sharing application. Each interested party submits ride requests to the application, either as a driver offering a ride, or as a passenger seeking to take a ride and share the costs. A ride request contains the following information: [driver/passenger, origin, destination, departure time, arrival time, price/contribution, seats offered/requested]. Upon each submission of a ride request, the platform re-runs the composition algorithm that creates new feasible ride plans in terms of timing/seats/route constraints being met. Viable plans are then put on negotiation between potential peer participants. After each individual ride request, acceptation, non-responding or refusal new plans can be generated or existing ones invalidated/undecided. The platform, therefore, needs to continuously adjust the state. As described in Section 3.2, in the course of the SmartSociety project a fully-fledged set of algorithms were developed for handling particular this class of problems [10]. They are made readily available to any developer through the programming API, i.e., through the CBT's *continuous orchestration* mode, making the necessary developer-side coding extremely simple (Listing 4).

```
/* Initialize the library handler for continuous orchestration coH3
 * coH3 delegates orchestration to the platform component OM */

TaskRequest rideRequest = new RideRequest.parseJSON(input);

CollectiveBasedTask cbt =
  CBTBuilder.from(TaskFlowDefinition.usingContinuousOrchestration(coH, null))
        .withTaskRequest(rideRequest)
        .build();

cbt.start();
if (cbt.get() != null) {...}
```

Listing 4: Ride-sharing scenario.

The developer initializes the appropriate CBT type, by specifying a library-provided handler for continuous orchestration. Upon each new ride request, a new instance of this CBT is

[5]https://gitlab.com/smartsociety/programming-framework

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2702578, IEEE Transactions on Emerging Topics in Computing

instantiated. In the background, the parameters of the requested ride are forwarded to the OM, triggering anew the ride matching, plan generations and possible new negotiation. Ultimately, some rides and the corresponding CBTs will succeed (with 'agreed' collective representing the passengers), while others will fail. It is up to the developer to decide what will be the arbitrary application logic in both cases (e.g., updating the user reputation, charging for the ride).

Compared to the previous two examples, this one is not characterized by hybridity. Namely, only 2 peer types are present and both are human peers. On the other hand, the peers are characterized by a great diversity in goals (destination, times, acceptable costs). These types of optimization problems are inherently difficult for humans to handle, as they involve too many variables to consider. For this reason, the platform is taking over this complex computational burden and leaving it to the developer to provide the remainder of the application's business logic suited to his needs. On the other hand, the platform is not actively searching for humans and engaging them, but merely reacting to human requests. In this respect, the execution is truly human-orchestrated at runtime.

Example 4

Example 4 (Listing 5) illustrates[6] another important class of tasks – intellectually-challenging (engineering, creative) collective tasks. A programming (software engineering) task is the prime example of such a problem. The simplified scenario used in this example assumes submitting a natural language description of a Java software artifact that the platform application needs to produce for an external user. The software artifact is produced by a simplified 2-stage methodology – the artifact is first coded by collective members submitting to a joint repository, then tested against externally provided unit tests. Since they involve primarily human experts with diverse skills, such tasks are not characterized by much hybridity but by a high diversity. In order to solve such a task successfully, the team first needs to be carefully assembled to contain compatible expert roles. This means that the provisioning phase needs to offer advanced matching algorithms. In our case, we make use of the in-house developed algorithm [12] for fuzzy skill matching inside the provisioning handler, hiding the collective formation complexity from the developer. The `TaskRequest swImplTaskReq` needs to contain the requested number of experts and natural-language (fuzzy) descriptions of the required skills (e.g., `<"Java EE developer", "very good">`).

Unlike the previous examples, here we deal with a purely on-demand task – a problem at hand that needs to be solved by peers actively located and engaged by the platform. This implies that peers cannot self-initiatively apply for participation, nor is there a plan composition phase. Instead, the plan is determined collectively by the provisioned peers through unmanaged (direct) communication based on the understanding of the task as provided in the task request, implying that the overall orchestration is human-driven. In order to support the communication requirement, the programming framework puts at peers' disposal the collective communication capabilities described in Section 3.5.

```
/* Initialize library handlers pH4, nH4, eH4dev, eH4test */

// parameterize handlers
nH4 = nH4.withArguments(NegotiationPattern.AGREEMENT_THRESHOLD, 0.5);

Collective javaDevs =
  ResidentCollective.createFromQuery(myQuery("JAVA_DEV");

CollectiveBasedTask progTask =
  CBTBuilder.from(TaskFlowDefinition.onDemandWithoutOpenCall(pH4, nH4, eH4dev)
                                    .withInputCollective(javaDevs))
            .withTaskRequest(swImplTaskReq).build();

progTask.start();

/*  ... assume negotiation on progTask done  ... */

Collective testTeam; //will be ABC
if (progTask.isAfter(CollectiveBasedTask.State.NEGOTIATION)) {
  // out of provisioned devs, use other half for testing
  testTeam = Collective.complement( progTask.getCollectiveProvisioned(),
                                    progTask.getCollectiveAgreed() );
}

while (!progTask.isDone()) { /* do stuff or block */ }

nH4 = nH4.withArguments(NegotiationPattern.AGREEMENT_THRESHOLD, 1.0);

CollectiveBasedTask testTask =
  CBTBuilder.from(TaskFlowDefinition.onDemandWithoutOpenCall(null,nH4, eH4test)
                                    .withInputCollective(testTeam))
            .withTaskRequest(new TaskRequest(progTask.get()))
            .build();

if (testTask.get().QoR() < 0.7) return TaskResponse.FAIL;
```

Listing 5: Software engineering scenario.

5.2 Programming API Evaluation

Collective API

Consider the software engineering scenario introduced in Sec. 5.1-Ex. 4, partially depicted in Listing 5. First, the developer creates a RC `javaDevs` containing all accessible Java developers from the peer-store. This collective is used as the input of the `progTask` CBT. `progTask` is instantiated as an on-demand collective task, meaning that the `composition` state is omitted. The output of the `provisioning` state is the 'provisioned' collective, a CBT-built ABC collective, containing the selected programmers. Since it is atomic and immutable, the exact programmers which are members of the team are not known to the application developer. The negotiation used in the example requires from the peers simply to agree or reject participating in the task, and will form an 'agreed' collective out of the first 50% peers who give acceptance. After the `progTask`'s execution this ABC becomes exposed to the developer, who uses it to construct another collective `testTeam`by set operations (:21-22), containing Java developers from the 'provisioned' collective that were not selected into the 'agreed' one. This collective is then used to perform the second CBT `testTask`, which takes as input the output of the first CBT (:32).

Collective-Based Task API

Listing 6 shows some examples of interaction with a CBT. An on-demand CBT named `cbt` is initially instantiated. For illustration purposes we make sure that all transition flags are enables (true by default), then manually set `do_negotiate` to false, to force `cbt` to block before entering the `negotiation` state, and start the CBT (:3-5). While CBT is executing, arbitrary business logic can be performed in parallel (:7-10). At some point, the CBT is ready to start negotiations. At that moment, for the sake of demonstration, we dispatch the motivating messages to the members of the collective manually (:12-14) instead through the incentivization functionality, and let the negotiation process begin. Finally, we block the main thread of the application waiting on the `cbt` to finish or the specified timeout to elapse (:20-21), in which case we explicitly cancel the execution (:28).

```
CollectiveBasedTask cbt =  /*... assume on_demand = true ... */

cbt.setAllTransitionsTo(true); //optional
```

[6]To save space, the accompanying code snippet is partial and shared with the `Collective`'s API evaluation example from the Section 5.2, shown in Listing 5. Full implementation is provided in the repository.

```
cbt.setDoNegotiate(false);
cbt.start();

while (cbt.isRunning() && !cbt.isWaitingForNegotiation()) {
    //do stuff...
}

for (ABC negotiatingCol : cbt.getNegotiables() {
    negotiatingCol.send(
        new SmartCom.Message("Please accept this task"));
}
cbt.setDoNegotiate(true);

TaskResult result = null;
try {
    //blocks until done, but max 5 hours:
    result = cbt.get(5, TimeUnit.HOURS);
    /* ... do something with result ... */
}catch(TimeoutException ex) {
    if (cbt.getCollectiveAgreed() != null){
        cbt.getCollectiveAgreed().incentivize(
          new IncentiveServer.IncStrategy.SimpleThanks("Thanks anyway"));
    }
    cbt.cancel(true);
}
//...
```

Listing 6: Controlling CBT's lifecycle.

5.3 Performance & Usability Evaluation

An approximate quantitative insight into the productivity improvements can be given by considering the lines of code (LOC) metric for the two scenarios that were previously implemented by the project partners in plain Java, using the same platform components, but without the use of the programming framework (PF), and comparing them with the functionally equivalent Examples 2 and 3 presented in Section 5.1. As shown in Table 4, the use of the programming API drastically reduces the amount of newly-written code, which is usually concurrent and error-prone, often boilerplate and repetitive. As a consequence, the principal business logic can be encoded in a concise and easily understandable, human-readable manner, further simplifying debugging and subsequent changes, as well as integration with legacy code.

Scenario	*LOC without PF*	*LOC with PF*	*improvement*
Example 2	~ 3.5K	< 0.5K	$> 7\times$
Example 3	~ 40K	< 1K	$> 40\times$

TABLE 4: Overview of conciseness and productivity improvements when using the Programming Framework (PF).

The number and diversity of the platform's components and supported scenarios prevented us from establishing a single comprehensive scalability and performance assessment benchmark for the entire platform. Instead, the two most critical subcomponents in terms of scalability (for messaging and continuous orchestrations) were individually evaluated in [10] and [8], respectively, on "neighborhood-scale" $\mathcal{O}(10^3)$ collectives. While the messaging throughputs proved scalable and satisfactory for the targeted collectives ($3200 - 5000 msg/sec$), the existing continuous orchestration algorithms have proven feasible for collective sizes up to 100 participants due to long execution times ($\sim 30 min$), consequence of their high computational complexity. However, our in-field test pilots in Israel and Italy have shown that in practice most collectives fall in this category. Furthermore, most real-life complex tasks involving humans have execution times in the order of hours and a drastically lower concurrency, making our orchestration algorithms fully applicable. Due to space restrictions, further details are presented in the cited papers.

6 Related Work

Here we present an overview of relevant classes of socio-technical systems, their typical representatives, and compare their principal features with the SmartSociety programming model. Based on the way the workflow is abstracted and encoded the existing approaches can be categorized into three groups [5]: *a*) programming-level approaches; *b*) parallel-computing approaches; and *c*) process modeling approaches.

Programming level approaches focus on developing a set of libraries and language constructs allowing general-purpose application developers to instantiate and manage tasks to be performed on socio-technical platforms. Unlike SmartSociety, the existing systems do not include the design of the crowd management platform itself, and therefore have to rely on external (commercial) platforms. The functionality of such systems is effectively limited by the design of the underlying platform. Typical examples of such systems are CrowdDB [17] and AutoMan [2]. CrowdDB outsources parts of SQL queries as Amazon Mechanical Turk microtasks. Concretely, the authors extend traditional SQL with a set of "crowd operators", allowing subjective ordering or comparisons of datasets by crowdsourcing these tasks through conventional micro-task platforms. From the programming model's perspective, this approach is limited to a predefined set of functionalities which are performed in a highly-parallelizable and well-know manner. AutoMan integrates the functionality of crowdsourced multiple-choice question answering into the Scala programming language. The authors focus on automated management of answering quality. The answering follows a hard-coded workflow. Synchronization and aggregation are centrally handled by the AutoMan library. The solution is of limited scope, targeting only the designated labor type. Neither of the described systems allows explicit collective formation, or hybrid collective composition.

Parallel computing approaches rely on the divide-and-conquer strategy that divides complex tasks into a set of subtasks solvable either by machines or humans. Typical examples include Turkomatic [18] and Jabberwocky. For example, Jabberwocky's [1] *ManReduce* collaboration model requires users to break down the task into appropriate map and reduce steps which can then be performed by a machine or by a set of humans workers. Hybridity is supported at the overall workflow level, but individual activities are still performed by homogeneous teams. In addition, the efficacy of these systems is restricted to a suitable (e.g., MapReduce-like) class of parallelizable problems. In practice they rely on existing crowdsourcing platforms and do not manage the workforce independently, thereby inheriting all the underlying platform's limitations.

The process modeling approaches focus on integrating human-provided services into workflow systems, allowing modeling and enactment of workflows comprising both machine and human-based activities. They are usually designed as extensions to existing workflow systems, and therefore can perform certain peer management. The currently most advanced systems are CrowdLang [3], CrowdSearcher [4] and CrowdComputer [5]. CrowdLang brings in a number of novelties in comparison with the previously described systems, primarily with respect to the collaboration synthesis and synchronization. It enables users to (visually) specify a hybrid machine-human workflow, by combining a number of generic (simple) collaborative patterns (e.g., iterative, contest, collection, divide-and-conquer), and to generate a number of similar workflows by differently recombining the constituent patterns, in order to generate a more efficient workflow at runtime. The use of human workflows also enables indirect encoding of inter-task dependencies. The user can influence which workers will be chosen for performing a task by specifying a predicate for each subtask that need to be fulfilled. The predicates

are also used for specifying a limited number of constraints based on social relationships, e.g., to consider only Facebook friends. The PPLib [6] similarly uses the principle of process recombination, but supports automated recombination and composition of subprocesses (operators) in search of an optimal process for a given task. In a similar vein, CrowdSearcher presents a task model composed of a number of elementary crowdsourcable operations (e.g., label, like, sort, classify, group), associated with individual human workers. Such tasks are composable into arbitrary workflows, through application of a set of common collaborative patterns which are provided. This allows a very expressive model but on a very narrow set of crowdsourcing-specific scenarios. This is in full contrast with the more general task-agnostic approach taken by the SmartSociety programming model. The provisioning is limited to the simple mapping "1 microtask $\leftrightarrow$ 1 peer". No notion of collective or team is not explicitly supported, nor is human-driven orchestration/negotiation. Finally, CrowdComputer is a platform allowing the users to submit general tasks to be performed by a hybrid crowd of both web services and human peers. The tasks are executed following a workflow encoded in a BPMN-like notation called BPMN4Crowd, and enacted by the platform. Tasks are assigned to individual workers through different 'tactics' (e.g., marketplace, auction, mailing list).

7 Conclusions & Future work

In this paper we presented a novel framework for effectively programming hybrid diversity-aware collective adaptive systems (HDA-CASs). The framework reflects the defining HDA-CAS properties and exposes to the developer the platform's functionality though an intuitive API. The platform is able to host user-provided applications and to manage collaborative computations on their behalf. Even if related systems allow a certain level of runtime workflow adaptability, they are limited to patterns that need to be foreseen at design-time. Our approach differs from these systems by extending the support for collaborations spanning from processes known at design-time to fully human-driven, ad-hoc runtime workflows. The spectrum of supported collaboration models and runtime workflow adaptability are exposed through the newly introduced "CBT" and "Collective" constructs. The CBT is task-agnostic, delegating the responsibility of providing a mutually-interpretable task description to the developer, which allows the construct to be generally applicable for the entire class of collaborative activities supported by the platform. Under the hood of CBT, the programming framework offers advanced composition of execution plans, coordination of the negotiation process and virtualization of peers. The Collective construct, coming in two flavors (RC and ABC), highlights the collective aspect of the task execution and prevents assigning individuals to workflow activities. At the same time, it allows the platform to enforce desired privacy and fairness policies, and prevents exploiting human peers as individual processing nodes. The core platform components are currently being evaluated in-field, by providing a ride-sharing application for the commuters between the city of Milan and the Cremona province in northern Italy. The future work will include the integration of more advanced incentive management models into the programming model.

Acknowledgment

Supported by EU FP7 SmartSociety project, grant 600854.

References

[1] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar, "The jabberwocky programming environment for structured social computing," in *UIST '11*. ACM, 2011, pp. 53–64.

[2] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "Automan: A platform for integrating human-based and digital computation," *SIGPLAN Not.*, vol. 47, no. 10, pp. 639–654, Oct. 2012.

[3] P. Minder and A. Bernstein, "Crowdlang: A programming language for the systematic exploration of human computation systems," in *Social Informatics*, ser. LNCS. Springer, 2012, vol. 7710, pp. 124–137.

[4] A. Bozzon, M. Brambilla, S. Ceri, A. Mauri, and R. Volonterio, "Pattern-based specification of crowdsourcing applications," in *Proc. 14th Intl. Conf. on Web Engineering (ICWE) 2014*, 2014, pp. 218–235.

[5] S. Tranquillini, F. Daniel, P. Kucherbaev, and F. Casati, "Modeling, enacting, and integrating custom crowdsourcing processes," *ACM Trans. Web*, vol. 9, no. 2, pp. 7:1–7:43, May 2015.

[6] P. M. D. Boer and A. Bernstein, "Pplib: Toward the automated generation of crowd computing programs using process recombination and auto-experimentation," *ACM Trans. Intell. Syst. Technol.*, vol. 7, no. 4, 2016.

[7] O. Scekic, T. Schiavinotto, D. Diochnos, M. Rovatsos, H.-L. Truong, I. Carreras, and S. Dustdar, "Programming model elements for hybrid collaborative adaptive systems," in *Proc. 1st IEEE Int. Conf. on Collab. and Internet Comp. (CIC'15)*, Hangzhou, China, Oct 2015.

[8] P. Zeppezauer, O. Scekic, and H.-L. Truong, "Technical report - smartcom design," 2014. [Online]. Available: https://github.com/tuwiendsg/SmartCom/wiki#technical-report

[9] M. Hartswood, M. Jirotka, R. Chenu-Abente, A. Hume, F. Giunchiglia, L. A. Martucci, and S. Fischer-Hübner, "Privacy for peer profiling in collective adaptive systems," in *Privacy and Identity Mgmt. for Future Internet (IFIP'14)*, 9 2014, pp. 237–252.

[10] M. Rovatsos, D. I. Diochnos, Z. Wen, S. Ceppi, and P. Andreadis, "Smartorch: An adaptive orchestration system for human-machine collectives," in *32nd ACM Symposium on Applied Computing (SAC'17)*, Apr 2017.

[11] A. Segal, Y. K. Gal, E. Kamar, E. Horvitz, A. Bowyer, and G. Miller, "Intervention strategies for increasing engagement in crowdsourcing: Platform, predictions, and experiments," in *IJCAI'16, New York, USA*.

[12] M. Z. C. Candra, H.-L. Truong, and S. Dustdar, "Provisioning quality-aware social compute units in the cloud," in *ICSOC'13*. Springer, 2013.

[13] J. Kleinberg, "The convergence of social and technological networks," *Comm. ACM*, vol. 51, no. 11, pp. 66–72, Nov. 2008.

[14] A. Kittur, J. V. Nickerson, M. Bernstein, E. Gerber, A. Shaw, J. Zimmerman, M. Lease, and J. Horton, "The future of crowd work," in *Proc. CSCW '13*, ser. CSCW '13. ACM, 2013, pp. 1301–1318.

[15] P. Mohagheghi and Ø. Haugen, "Evaluating domain-specific modelling solutions," in *Advances in Conceptual Modeling*, ser. LNCS, J. Trujillo *et al.*, Eds. Springer, 2010, vol. 6413, pp. 212–221.

[16] A. Seffah, M. Donyaee, R. B. Kline, and H. K. Padda, "Usability measurement and metrics: A consolidated model," *Software Quality Control*, vol. 14, no. 2, pp. 159–178, Jun. 2006.

[17] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "Crowddb: Answering queries with crowdsourcing," in *ACM SIGMOD Intl. Conf. on Mgmt. of Data*, ser. SIGMOD '11. ACM, 2011, pp. 61–72.

[18] A. P. Kulkarni, M. Can, and B. Hartmann, "Turkomatic: Automatic recursive task and workflow design for mechanical turk," in *CHI EA '11*. ACM, 2011, pp. 2053–2058.

Ognjen Scekic is postdoctoral researcher and lecturer at the Distributed Systems Group, TU Wien.

Tommaso Schiavinotto is Senior Product Manager and Software Architect at U-Hopper, Trento, Italy.

Svetoslav Videnov is scientific/research developer at the Distributed Systems Group, TU Wien.

Michael Rovatsos is Senior Lecturer at the School of Informatics at The University of Edinburgh and Director of the Centre For Intelligent Systems and their Applications.

Hong-Linh Truong is Assistant Professor at the Distributed Systems Group, TU Wien.

Daniele Miorandi is VP Executive of R&D at U-Hopper, Trento, Italy. Daniele holds a PhD and MSc in Communications Engineering.

Schahram Dustdar is Full Professor of Computer Science and Head of the Distributed Systems Group at the TU Wien. He is an IEEE Fellow, ACM Distinguished Scientist and IBM Faculty Award recipient.