Programming Model Elements for Hybrid Collaborative Adaptive Systems

Ognjen Scekic*, Tommaso Schiavinotto†, Dimitrios I. Diochnos‡,
Michael Rovatsos‡, Hong-Linh Truong*, Iacopo Carreras†, Schahram Dustdar*
* Distributed Systems Group, Vienna University of Technology, Austria
Email: oscekic | truong | dustdar @dsg.tuwien.ac.at
† U-Hopper, Trento, Italy
Email: tommaso.schiavinotto | iacopo.carreras @u-hopper.com
‡ Centre for Intelligent Systems and their Applications, University of Edinburgh, UK
Email: d.diochnos | mrovatso @inf.ed.ac.uk

***Abstract*—Hybrid Diversity-aware Collective Adaptive Systems (HDA-CAS) is a new generation of socio-technical systems where both humans and machine peers complement each other and operate collectively to achieve their goals. 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 hybrid human/-machine teams. In this paper we present the key programming elements of the SmartSociety HDA-CAS platform. We first describe the overall platform's architecture and functionality and then present concrete programming model elements – Collective-based Tasks (CBTs) and Collectives, describe their properties and show how they meet the hybridity and collectiveness requirements. We also describe the associated Java language constructs, and show how concrete use-cases can be encoded with the introduced constructs.**

I. Introduction

We have recently witnessed the evolution of conventional social computing and appearance of novel types of socio-technical systems, attempting to leverage human experts for more intellectually challenging tasks [1, 2, 3, 4, 5]. These types of systems are opening up the possibilities for novel forms of interaction, collaboration and organization of labor where humans and computers complement each other. However, even the cited systems limit themselves to using computers to support and orchestrate purely human collaborations, usually based on patterns of work that can be predictably modeled before the execution (Section VI). 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 HDA-CAS[1]. However, building such systems is a challenging task, requiring solutions that go well beyond traditional coordination and communication problems; especially so, when participating humans are not merely considered as computational nodes providing a service at request, but are put on an equal footing and allowed to actively drive computations.

In this paper we present the programming model and associated language constructs for the *SmartSociety Platform*[2], a novel HDA-CAS supporting a wide spectrum of collaboration scenarios. This paper can be considered a follow-up to the complementary paper [6], which presents the functionality of particular platform components, the overall architecture, and the performance evaluation. The paper describes how the presented programming model design tackles the fundamental HDA-CAS novelty requirements of hybridity and collectiveness and showcases how the introduced language constructs can be used to encode and execute hybrid collaborations on the SmartSociety platform.

The paper is organized as follows: In Section II we present the necessary background and the intended usage context or the programming model – the SmartSociety platform. In Section III the principal programming model elements are introduced and their functionality described. Section IV presents the associated programming language constructs that are validated in Section V. Related work is described in Section VI and contrasted to our approach. Finally, Section VII concludes the paper and points out directions for future activities.

II. Background – the SmartSociety platform

The SmartSociety platform (*platform*) [6], shown in Figure 1, is a software framework intended for use by:

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.

[1]Hybrid Diversity-Aware Collective Adaptive Systems, http://focas.eu/

[2]The platform is being developed in the context of the EU FP7 research project 'SmartSociety' URL: http://www.smart-society-project.eu/

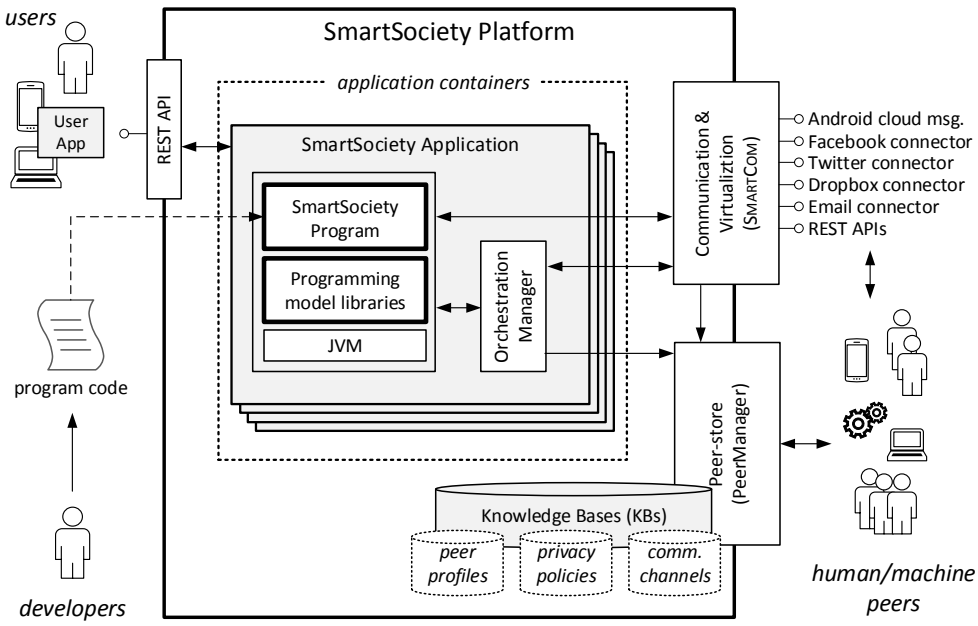

Fig. 1: SmartSociety platform users and architecture. Paper contributions outlined.

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. The platform is not limited to a particular class of tasks. Supported task complexity ranges: from simple, independent crowdsourcing tasks (e.g., translation); over inter-dependent complex tasks (e.g., collaborative question answering and refinement); over team-based tasks (e.g., predictive maintenance [7]); to the fully human-driven collaborations involving non-trivial execution plans with constraint matching and human negotiations (e.g., ride-sharing). However, implementing the desired collaborative effort specification is entirely left to the developers in the context of a particular SmartSociety platform application. The platform facilitates this process by offering a variety of commonly used coordination, orchestration, communication and adaptation mechanisms as ready-made concepts exposed through the programming API.

A. Usage Context & Key Notions

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 agreements (SLAs). Registered users are the basis from which appropriate peers are selected into *collectives* participating in executions of collaborative tasks. A collective is composed of a team of peers along with a *collaborative environment* assembled for performing a specific task. The collaborative environment consists of a set of software communication and coordination tools. For example, as described in [7], 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 corresponding *user application* users submits *task requests* to be executed to the platform. The user application communicates with the corresponding platform application.

B. Platform Architecture & Functionality

A simplified, high-level view of the SmartSociety platform architecture is presented in Fig. 1. The rectangle boxes represent the key platform components. The principal component-interoperability channels are denoted with double-headed arrows in the figure. Communication with peers is supported via popular commercial protocols to allow a broader integration with existing communication software and allow easier inclusion of peers into the platform. User applications contact the platform through the REST API component. All incoming user requests are served by this module that verifies their correctness and 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, exposing to the developer the functionality of different platform components.

In the remainder of the section, we briefly describe the

principal platform components and their functionality, necessary for understanding the subsequently presented design of the programming model. Full details on platform's architecture and functionality are provided in the complementary paper [6].

PeerManager (PM): This is the central peer data-store (*peer-store*) of the platform. It manages all peer and application information, and allows privacy-aware access and sharing of the peer/collective data among platform components and applications. More details provided here[3].

Orchestration Manager (OM): Each platform application features a dedicated OM component[5]. The OM is the component in charge of preparing and orchestrating collaborative activities among peers. Concretely, this includes the following functionalities, reflected in the programming model and the library language constructs (Section III):

- *Discovery*— Provisioning or locating existing human and machine peers appropriate for the given task and forming collectives.
- *Composition*— Generating possible *execution plans* to meet user-set constraints and optimize wanted parameters.
- *Negotiation*— Coordinating the negotiation process among human peers leading to the overall agreement and acceptance of the execution plan.
- *Execution*— Monitoring the execution of the selected execution plan during the runtime.

The OM module implements various algorithms for the above-described functionalities. Discovery can be either performed by actively picking members [8], or by coordinating the process of self-formation of the collective as integral part of the composition and negotiation phases. In the latter case, the OM uses a static decision tree for scheduling the messages of subscription, agreement and withdrawal to the proposed plans originating from human peers [9]. At the moment, during composition the OM generates all possible execution plans that include the participants who satisfy the required constraints. Hence, even if the current approach is not computationally efficient, it suffices as a fully-functional, prof-of-concept implementation. More details on the OM performance are provided in [6].

Communication and Virtualization Middleware: The middleware named SMARTCOM is used as the primary means of communication between the platform and the peers, but also among the peers. It supports routing and exchange of messages over different protocols, performing automated message transformations depending on the recipient's type (human/machine) and supported formats. The virtualization functionality of SMARTCOM assumes rendering uniform the representation and communication with both human and software-based peers to the remainder of the platform. In addition, it can also be used to provide an ad-hoc communication environment for the members of a particular collective. The developer makes use of SMARTCOM indirectly through the provided programming API to communicate with collectives. Internally, the OM also uses SMARTCOM when enacting negotiation protocols.

[3]http://www.smart-society-project.eu/publications/deliverables/D_4_2/

III. PROGRAMMING MODEL

Figure 2 illustrates the intended usage of the programming model. The developer writes a SmartSociety program performing arbitrary business logic and handling the interaction with user applications. When a task requiring collaborative hybrid processing is needed, 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: team provisioning and assembly, execution plan composition, human participation negotiations, and finally the execution itself. These operations are provided by various SmartSociety platform components, which expose a set of APIs used by the programming model libraries. 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, even outside of the context of the originating CBT or its lifespan. This allows the developer to communicate directly with the collective members, monitor and incentivize them, but also to use existing collectives to produce new ones, persist them, and pass them as inputs to other CBTs at a later point. In the remainder of the section, we will look in more detail into the design and functionality offered by CBT and Collective constructs.

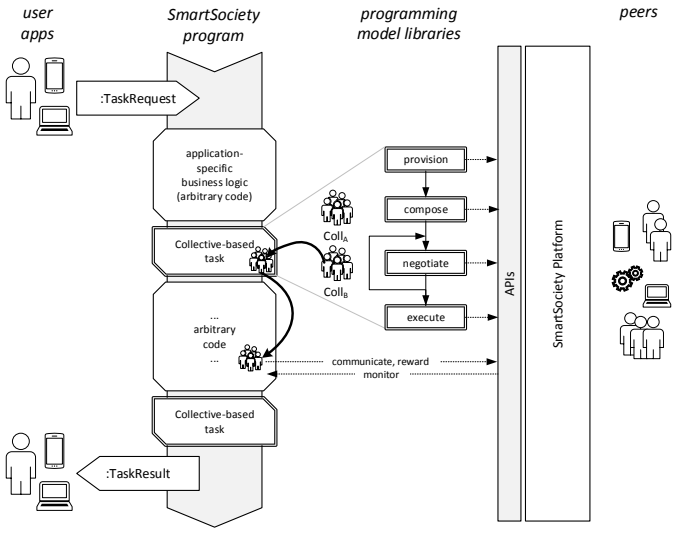

Fig. 2: Using the SmartSociety programming model.

A. Collective-Based Tasks (CBT)

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 (cf. Fig.2). 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.

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

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 I: CBT collaboration models and selection flags

At CBT's core is a state machine (Fig. 3) driven by an indepent execution thread managing transitions between states representing the eponymous phases of the task's lifecycle: `provisioning`, `composition`, `negotiation` and `execution`. An additional state, named `continuous_orchestration`, is used to represent a process combining composition and negotiation under specific conditions, as explained in Table I. 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 2) and are subsequently usable in general program logic.

Each state is associated with a set of *handlers*[6] 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. This feature is used to implement adaptation policies (Sec. III-B). The programming model provides default dummy handlers. In addition, the aim is to provide to the developer a library of useful, precompiled handlers exploiting the full functionality of the various components of the SmartSociety platform, such as orchestration and negotiation algorithms provided by the Orchestration Manager, or external provisioning algorithms (e.g., [8]). Concrete handlers are pre-registered for each CBT type exposed to the developer.

Provisioning state: The input to the state is the CBT input collective specified at CBT instantiation (most commonly 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 [8] 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 (e.g., a shared Dropbox folder) and communication policies among them. Details of how this is achieved are provided in [7]. The output of the state is the 'provisioned' collective, that gets passed on to the next state during the execution.

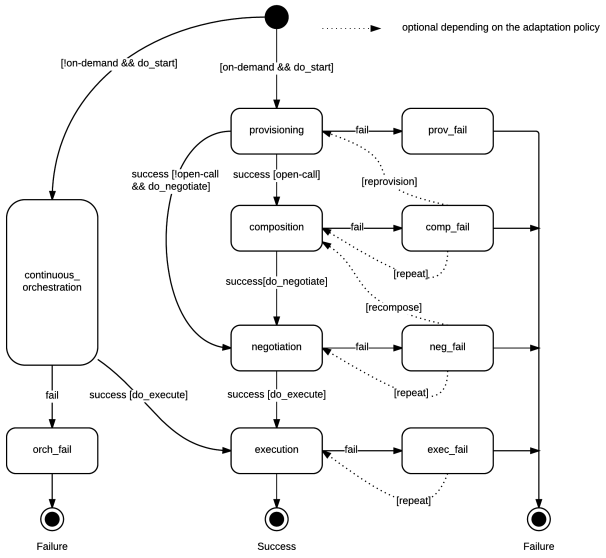

Fig. 3: CBT state diagram.

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 [9] and here[4]. 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

[4]http://www.smart-society-project.eu/publications/deliverables/D_6_1/

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 negotiation patterns currently offered by the platform and through the programming model libraries are described here[5]. The output of the negotiation process is the single 'agreed' collective and the associated execution plan.

Continuous orchestration state: Continuous orchestration (cf. Table I) 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 provide only the input 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 of the ride along with information which section is driven by which vehicle/driver and with which passengers. 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 the price they offer); passengers prefer different routes depending on the vehicles, fellow-passengers, ride cost/duration and the number of transfers. All potential driver/passenger peers are allowed to participate in negotiations for multiple plans in parallel, and accepting and withdrawing 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 peer changes his 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-revokable; if it is in addition in agreed state, it can get executed. Otherwise, the `orch_fail` state is entered. More details provided here[5].

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 2 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 (Sec. III-B) specifies. Some failure states react only to specific adaptation policies; some to none.

B. 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 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. Since concrete adaptation policies are meant to extend the functionality of the programming model they are usually context-specific. Therefore, the programming model limits itself to offering the mechanism of

[5] http://www.smart-society-project.eu/publications/deliverables/D_6_2/

extending the overall functionality through external policies and itself offers for each failure state only a limited set of simple, generally applicable *predefined policies*. In order to be general, predefined policies assume re-using existing handlers. Natively supported predefined policies are described in Table II. 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).

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). If the developer specifies the new handler we describe the property as 'adaptivity'; if the system automatically determines the new handler, we describe the property as 'elasticity'.
`REPROVISION`	Transition into `provisioning` state, with (optionally) a new provisioning handler.
`RECOMPOSE`	Repeat the composition, with (optionally) a new composition handler.

TABLE II: CBT adaptation policies.

C. Collectives

The notion of "collective" in HDA-CAS community 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 (PeerManager) 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 lifecycle is managed exclusively by the SmartSociety application. Therefore, an ABC cannot be accessed (i.e., is meaningless) 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.

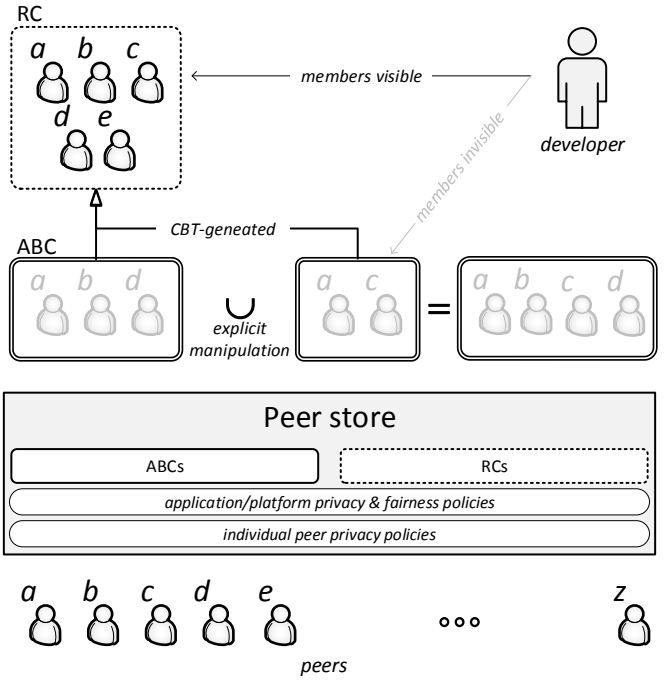

Fig. 4: Differences between RCs and ABCs. ABCs are created from RCs but out of Developer's control. Although the Developer is able to manipulate and create new descending ABCs, they appear atomic to the Developer.

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 [10]. This, in turn, may lead to discouragement of less prominent peers, and in overall, increase the attrition rate [11]. 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 [12].

IV. Language constructs

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[6]. Instead, we describe the supported groups of constructs and their functionality, and some representative individual methods. The examples in Sec.V-A 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"[7], as exemplified in Listing 1.

```
1 CBT cbt = ctx.getCBTBuilder("MyCBTType")
2              .of(CollaborationType.OC) //Enum: OC, OD, OC_OD
3              .forInputCollective(c)
4              .forTaskRequest(t)
5              .withNegotiationArgs(myNegotiationArgs)
6              .build();
7 /* ... */
```

Listing 1: Instantiation of a CBT.

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[8] 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 3 (:3-5, 7, 16, 21, 28) shows the usage of some of the constructs.

CBT collective-fetching operations: As explained in Sec. III-C 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 2 (: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 a 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. III-C), 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`

[6] http://dsg.tuwien.ac.at/research/viecom/SmartSociety/prog-api.pdf. Password: SmartSocietyReviewer

[7] http://www.martinfowler.com/bliki/FluentInterface.html

[8] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

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

Collective-level communication: Programming model fully relies on our messaging and virtualization middleware SMARTCOM [7] developed for supporting the communication with peers and collectives. Programming model allows at the moment only a basic set of communication constructs, namely those for sending a message to a hybrid collective (Listing 3:12-13), and receiving responses from it. Message delivery is in line with individual privacy preferences.

V. EVALUATION

A programming model can be evaluated both qualitatively and quantitatively. Quantitative analysis is usually performed once the associated domain-specific language (constructs) making use of the programming model is considered mature [13], since this type of evaluation includes measuring productivity and subjective satisfaction in an established community of regular users [14]. During the initial development and prototyping phase, the common approach is to use the qualitative evaluation instead [13], which, in general, can include: comparative case studies, analysis of language characteristics and monitoring/interviewing users. Analysis of language characteristics was chosen as the preferred method in our case. Comparative analysis was not applicable in this case, due to nonexistence of similarly expressive models, as shown in Section VI. In order to qualitatively evaluate the overall functionality of the programming model, we are currently integrating the programming model libraries into two existing SmartSociety platform applications tested in-field with human peers: *a*) a ride-sharing application *SmartShare*[9]; and *b*) a hybrid, collective question-answering service *AskSmartSociety!*[10]. As the two applications put focus on continuous orchestration and on-demand collaboration models, respectively, this exercise is a good indicator of the ability of the model to cover the advertised collaboration models. In addition, in order to qualitatively evaluate the API exposed to the developer we encoded a set of examples covering important use-cases derived from the set of real-world scenarios specifically elicited for the purposes of the SmartSociety project, and published here[11]. In the remainder of the section, we present an adapted selection of the encoded examples to illustrate the use of the the fundamental language constructs.

[9]https://gitlab.com/smartsociety/orchestration. Log into GitLab first with: user/pass = `SmartSocietyReviewer` / `sm@rts0c13tyr3v13w3r`

[10]https://gitlab.com/smartsociety/appruntime. Log into GitLab first with: user/pass = `SmartSocietyReviewer` / `sm@rts0c13tyr3v13w3r`

[11]http://www.smart-society-project.eu/publications/deliverables/D_9_1/

A. Examples

Manipulating and reusing collectives: Consider an application that uses SmartSociety platform to assemble ad-hoc, on-demand programming teams to build software artifacts. For this purpose, two CBT types are assumed to be registered: "MyJavaProgrammingTask" and "MyJavaTestingTask". 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 (:4-10). `progTask` is instantiated as an on-demand collective task, meaning that the `composition` state will be skipped, since the execution plan in implied from the task request `myImplementationTReq`. The collective is first processed in the provisioning phase, where a subset of programmers with particular skills are selected and a joint code repository is set for them to use. 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 pattern will select the first 50% of the provisioned developers into the 'agreed' collective that will ultimately execute the programming task. After the `progTask`'s this ABC becomes exposed to the developer, which uses it to construct another collective (:19-22), 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` (:31-37), which takes as input the output of the first CBT.

```
1  Collective javaDevs =
2     ResidentCollective.createFromQuery(myQry,"JAVA_DEVS");
3
4  CBT progTask = ctx.getCBTBuilder("MyJavaProgrammingTask")
5                    .of(CollaborationType.OD)
6                    .forInputCollective(javaDevs)
7                    .forTaskRequest(myImplementationTReq)
8                    .withNegotiationArguments(
9        NegotiationPattern.AGREEMENT_RELATIVE_THRESHOLD, 0.5)
10                   .build();
11
12 progTask.start();
13
14 /*  ... assume negotiation on progTask done  ... */
15
16 Collective testTeam; //will be ABC
17 if (progTask.isAfter(CBTState.NEGOTIATION)) {
18   // out of provisioned devs, use other half for testing
19   testTeam = Collective.complement(
20                progTask.getCollectiveProvisioned(),
21                progTask.getCollectiveAgreed()
22              );
23 }
24
25 while (!progTask.isDone()) { /* do stuff or block */}
26
27 TaskResult progTRes = progTask.get();
28
29 if (! progTRes.isQoRGoodEnough()) return;
30
31 CBT testTask = ctx.getCBTBuilder("MyJavaTestingTask")
32                   .of(CollaborationType.OD)
33                   .forInputCollective(javaDevs)
34                   .forTaskRequest(new TaskRequest(progTRes))
35                   .withNegotiationArguments(
36       NegotiationPattern.AGREEMENT_RELATIVE_THRESHOLD, 1.0)
37                   .build();
38 /*...*/
```

Listing 2: Manipulating and reusing collectives.

Controlling CBT execution: Listing 3 shows some examples of interacting with CBT lifecycle. 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 (or possibly other incentive mechanisms) to the human members of the collective (:12-14), 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).

```
1  CBT cbt =  /*... assume on_demand = true ... */
2
3  cbt.setAllTransitionsTo(true); //optional
4  cbt.setDoNegotiate(false);
5  cbt.start();
6
7  while (cbt.isRunning() && !cbt.isWaitingForNegotiation()) {
8      //do stuff...
9  }
10
11 for (ABC negotiatingCol : cbt.getNegotiables() {
12     negotiatingCol.send(
13          new SmartCom.Message("Please accept this task"));
14     // negotiatingCol.applyIncentive("SOME_INCENTIVE_ID");
15 }
16 cbt.setDoNegotiate(true);
17
18 TaskResult result = null;
19 try {
20     //blocks until done, but max 5 hours:
21     result = cbt.get(5, TimeUnit.HOURS);
22     /* ... do something with result ... */
23 }catch(TimeoutException ex) {
24     if (cbt.getCollectiveAgreed() != null){
25         cbt.getCollectiveAgreed().send(
26 new SmartCom.Message("Thank you anyway, but too late."));
27     }
28     cbt.cancel(true);
29 }
30 //...
```

Listing 3: Controlling CBT's lifecycle.

VI. 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 TurKit [15], CrowdDB [16] and AutoMan [2]. TurKit is a Java library layered on top of Amazon's Mechanical Turk offering an execution model ("crash-and-rerun") which re-offers the same microtasks to the crowd until they are performed satisfactorily. While the deployment of tasks onto the Mechanical Turk platform is automated, the entire synchronization, task splitting and aggregation is left entirely to the programmer. Unlike SmartSociety, the inter-worker synchronization is out of programmer's reach. The only constraint that a programmer can specify is to explicitly prohibit certain workers to participate in the computations. No other high-level language constructs are provided. CrowdDB outsources parts of SQL queries as mTurk 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 hardcoded 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 three 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 [17] 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. Also, in practice they rely on existing crowdsourcing platforms and do not manage the workforce independently, thereby inheriting all underlying platform 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. CrowdSearcher presents a

novel 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. While CrowdComputer assumes splitting of tasks and assignment of single tasks to individual workers through different 'tactics' (e.g., marketplace, auction, mailing list) SmartSociety natively supports actively assembling hybrid collectives to match a task. In addition, by providing a programming abstraction, SmartSociety offers a more versatile way of encoding workflows.

VII. Conclusions & Future work

In this paper we presented the programming model of SmartSociety – an HDA-CAS platform supporting collaborative computations performed by hybrid collectives, composed of software and human-based services. The platform is able to host user-provided applications, managing 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 (VI); SmartSociety 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 two constructs represent the principal contribution of this paper. 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 work activities supported by the platform. Under the hood of CBT, the programming model offers advanced composition of execution plans, coordination of the negotiation process and virtualization of peers. The Collective construct, coming in two flavors (RCs and ABCs) 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. Using the associated API, developers can make use of the two constructs and and leave it to the platform's runtime to provision the collectives, orchestrate the negotiation and agreement between human peers and ultimately perform the task collaboratively. At the moment, a number of simple adaptation strategies are also supported. All these phases are handled transparently to the developer. The API was designed to be comprehensive and familiar and to integrate well with legacy (Java) code.

Currently, the programming model has been qualitatively validated. Future work will see the full implementation and validation of the programming model in real-world experiments, once the full integration of all project-developed components has been performed. Talks are currently under way to run these tests using in municipalities of Northern Italy and Israel.

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 *Proc. 24th Annual ACM Symposium on User Interface Software and Technology*, ser. 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, K. Aberer *et al.*, Eds. 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] O. Scekic *et al.*, "Smartsociety – a platform for collaborative people-machine computation," in *IEEE SOCA'15*, Oct 2015, p. forthcoming.

[7] P. Zeppezauer, O. Scekic, H.-L. Truong, and S. Dustdar, "Virtualizing communication for hybrid and diversity-aware collective adaptive systems," in *WESOA@ICSOC'14*. Springer, 11 2014.

[8] M. Z. C. Candra, H.-L. Truong, and S. Dustdar, "Provisioning quality-aware social compute units in the cloud," in *11th Intl. Conf. on Service Oriented Comp. ICSOC'13*. Springer, 2013.

[9] M. Rovatsos, D. I. Diochnos, and M. Craciun, "Agent protocols for social computation," in *Metaheuristics for Smart Cities*, 2015.

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

[11] O. Scekic, H.-L. Truong, and S. Dustdar, "Incentives and rewarding in social computing," *Commun. ACM*, vol. 56, no. 6, pp. 72–82, Jun. 2013.

[12] 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. of the 2013 Conf. on Computer supported cooperative work*, ser. CSCW '13. ACM, 2013, pp. 1301–1318.

[13] 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.

[14] 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.

[15] G. Little, "Exploring iterative and parallel human computation processes," in *CHI EA '10*. ACM, 2010, pp. 4309–4314.

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

[17] 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.