EAI Endorsed Transactions
on Collaborative Computing
Research Article ICST.ORG

A Collaboration Model for Community-Based Software Development with Social Machines

Dave Murray-Rust[1,*], Ognjen Scekic[2], Petros Papapanagiotou[1], Hong-Linh Truong[2], Dave Roberston[1] and Schahram Dustdar[2]

[1]Centre for Intelligent Systems and Applications,School of Informatics, University of Edinburgh, UK
[2]Distributed Systems Group, Vienna University of Technology, Austria

Abstract

Crowdsourcing is generally used for tasks with minimal coordination, providing limited support for dynamic reconfiguration. Modern systems, exemplified by social machines, are subject to continual flux in both the client and development communities and their needs. To support crowdsourcing of open-ended development, systems must dynamically integrate human creativity with machine support. While workflows can be used to handle structured, predictable processes, they are less suitable for social machine development and its attendant uncertainty. We present models and techniques for coordination of human workers in crowdsourced software development environments. We combine the Social Compute Unit—a model of ad-hoc human worker teams—with versatile coordination protocols expressed in the Lightweight Social Calculus. This allows us to combine coordination and quality constraints with dynamic assessments of end-user desires, dynamically discovering and applying development protocols.

Received on 03 February 2015; accepted on 04 July 2015; published on 17 December 2015
Keywords: auxiliary information, incremental clustering, data growth, collaborative Filtering, NMF

doi:10.4108/eai.17-12-2015.150812

1. Introduction

Most social computing systems today are based around patterns of work that can be predictably modelled before execution, such as translation, bug discovery, image tagging [1, 2]. However, there are many cases where where a traditional workflow approach is too rigid to address the dynamic an unpredictable nature of the tasks at hand, and more flexible crowd working systems must be developed[3].

One example of such dynamic systems is the field of *social machines*—systems where computers carry out the bookkeeping so that humans can concentrate on the creative work[4]. This term covers a diverse class of systems, spanning task-oriented (Wikipedia) to generic (Twitter); scientific or humanitarian (GalaxyZoo, Ushahidi) to social (Instagram) [5, 6].

Despite their diversity, a defining feature of social machines is that interactions between computational intelligence and human creativity are deeply woven into the system, making it difficult to draw a clear line between the human and digital parts—rather they must be analysed synergistically.

Hence, creating a social machine requires understanding of individual and group human behaviour alongside technical expertise, and a view of the system as an interconnected whole containing both human and computational elements.

Furthermore, social machines are subject to population dynamics as the user population changes, and must respond to the exigencies of unfolding situations. In such cases it is important not to over-regulate participating humans, but to let them play an active role in shaping the collaboration during runtime. On the one hand, this points to leveraging human creativity and embracing the uncertainty that comes with it, in order to respond flexibly. On the other hand, it is often necessary to impose certain coordination and quality constraints for these collaborations in order to manage them. The constraints delimit the decision space within which the humans are allowed to self-organize.

Recently, a number of human computation frameworks supporting complex collaboration patterns were proposed (Section 7). They mostly build upon conventional crowdsourcing platforms offering a process management layer capable of enacting complex workflows. While these systems represent important steps on the road to building complex social machines, in cases where unpredictability is inherent to the labour process and we cannot know all of the system requirements in advance, a different approach is needed.

In this paper we present models and techniques for coordination of human workers in crowdsourced software development environments. They support the bootstrapping and adaptation of social machines: using one social machine to generate or alter another one, allowing for flexible, community-driven development, with feedback paths from user populations to development choices. This allows both

*Corresponding author. Email: d.murray-rust@ed.ac.uk

human influence on the execution of a computation and the embedding of computational intelligence.

The introduced concept augments the *Social Compute Unit* (SCU, Section 2.2)—a general framework for management of ad-hoc human worker teams—with versatile coordination protocols encoded in *Lightweight Social Coordination Calculus* (LSC, Section 2.3). This combination allows us to design and model social machines oriented towards crowdsourcing software development. Coordination protocols provide high level organisation of activities around development—including planning and user assessment/feedback—while a set of coordination and quality constraints guide the assignment of workers to tasks. The system as a whole strikes a balance between imposed constraints and creative freedom in the software development cycle. Concretely, this means that the proposed model is able to take into account the feedback from the user population and subsequently alter the process of the development of software artefacts. Although we focus on collaborative software development, the solution we present is generally applicable to a class of similar problems. There are many situations where some form of social machine or community infrastructure is being developed, requiring the developers to react to the changing needs and behaviour of the community; increasingly, another social machine is used to crowdsource the development.

This paper is structured as follows: First we present the motivating scenario of community-influenced, collaborative software development. In Section 2 we analyse how the presented scenario can be modelled in terms of social machines. We then introduce background concepts which we use to build our model: Feature Oriented Software Development (FOSD), the Social Compute Unit (SCU) and the Lightweight Social Calculus (LSC). In Section 3 we present the coordination model for the social machine employing the previously introduced background concepts. In Section 4 a proof-of-concept implementation is presented and evaluated through simulation.

1.1. Motivating Scenario

Developing software for a large user base with diverging interests can be challenging. As an illustrative example, let us consider the problem of developing a forum-like scientific platform—a scholarly social machine—to facilitate multidisciplinary cross-collaboration and sharing of results. This includes functionality such as: paper previews, comments, in-place formulae and data rendering, citation previews and bookmarking. While these are functionalities beneficial to all scientists, preferences for particular formats and services will likely differ among different sub-communities. For example, chemists and mathematicians will have different domain-specific requirements from the platform. Some examples are:

- Computer scientists need code syntax highlighting, LaTeX rendering, embedding of IEEExplore and ACM DL citations. If any of these features is missing, the software is not useful to the community. However, they do not particularly care about chemistry-specific features.
- Chemists often use InChi strings to represent chemical formulae. If the software supports InChi, then chemists would also want support for compound lookup on PubChem, and visualisation with pyMol. Without these features, the platform does not help them particularly. On the other hand, syntax highlighting and IEEExplore integration is not important.
- Individual scientists may be bothered by (lack of) certain features. For example, users may dislike being forced to use a LinkedIn account to log in, due to possibility of a third party accessing unpublished scientific findings.

Some of these features are orthogonal—code syntax highlighting and LaTeX rendering are both useful in their own right—while some are synergistic: a chemist might require both parsing InChi strings and PubChem lookup in order to carry out their particular workflow.

The complexity of developing such software lies in catering to the heterogeneous user needs, requiring numerous trade-offs when deciding which features to implement. Furthermore, different sub-communities tend to change preferences regarding required or newly developed features during the development process which need to be taken into account. Finally, certain members of the scientific community (i.e., targeted users) may decide to take part in the development process themselves.

2. Modelling Community-Based Software Development

The previously presented scenario is representative of many social machines, where a dynamic community forms around a particular (software) artefact. The population is likely to change, and feedback between the human participants and technological infrastructure can lead to changes in the purpose and direction of technological development.

This means that there are two *social machines*: *i*) the *target* social machine which includes the forum software and its community of users and *ii*) the *development* social machine, which is the software developers and their coordination architecture (Figure 1). We use the term *utility* to denote some metric for the benefits which a user (in the *target* social machine) derives from participation. While in principle this metric could include a multitude of components, within this paper we narrow our focus to treat utility as measuring how well the software's feature set matches the user's requirements.

The aim of the development social machine is to increase the overall utility of the user population, by creating features which match community needs and desires. The developers do not know ahead of time the true preferences of individuals,

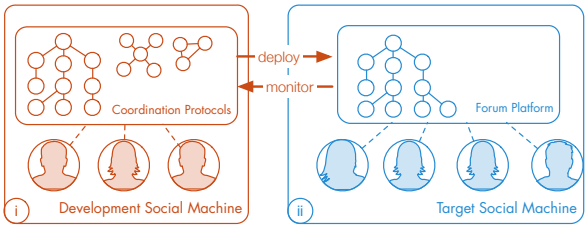

Figure 1. Two connected social machines: i) the *development* social machine, where crowdsourced workers follow coordination protocols to create a software artefact; ii) the *target* social machine, where a community of practice forms around the software artefact created by i).

or the constitution of the community, and hence the effects of software changes on community behaviour are difficult to predict ahead of time.

Since crowd-labour is increasingly used for the development of software, we assume it is necessary to use development methodologies which split work into tasks that are amenable to crowdsourcing. This means that tasks have to be disassembled into simpler subtasks, and mapped to appropriate developers. The latter is itself a complex problem, as it also includes taking care of inter-task implementation dependencies. Hence, the *development* social machine must be able to i) assess user desires and preferences; ii) identify and prioritize features for development; iii) coordinate the development and deployment of these features; iv) organise these tasks over time with respect to a dynamically changing population and limited resources.

These operations and their relation to the user population are outlined in Figure 2

2.1. Feature Trees for Artefact Development

A requirement of our model is the representation of the current state of the artefact under development—the *development artefact* in Figure 2. Since our example is based on software development, we use the *Feature-Oriented Software Development (FOSD)* paradigm, where software artefacts are represented as trees of *features*: "prominent or distinctive user-visible aspect, quality, or characteristic[s] of a software system"[7]. This representation is used so that development can be decomposed into small sets of related tasks that can be handled relatively independently, to aid collaborative creation of software artefacts.

Based on the requirements and possibilities in the scenario outlined in Section 1.1, the feature tree in Figure 4 can be constructed[1]. Here, broad classes of functionality, such as visual embedding of graphical objects are represented as

[1]The tree was created using FeatureIDE. Details of assumed semantics can be found at `http://www.iti.cs.uni-magdeburg.de/iti_db/research/featureide/`

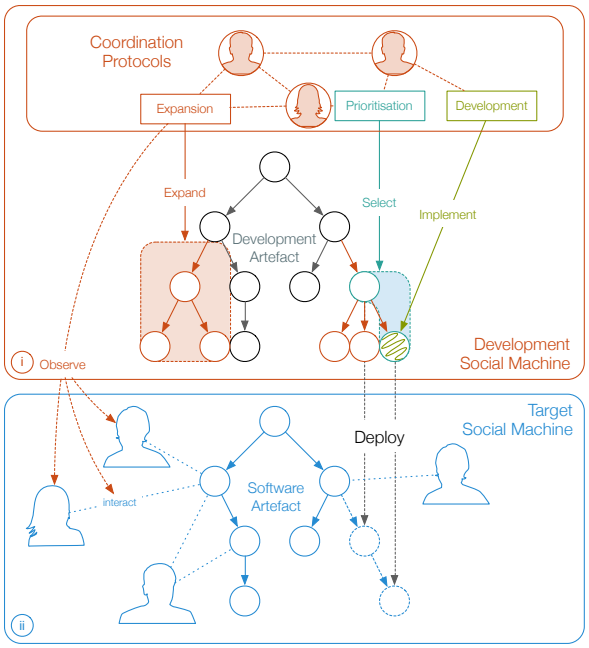

Figure 2. Interaction between the development and target social machines, including development steps, developer interaction, user observation and community interaction.

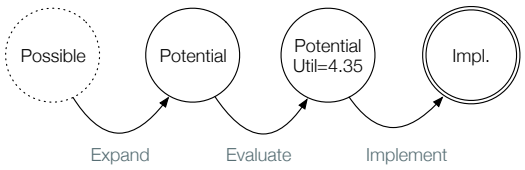

Figure 3. States and operations on a single node in the feature tree. Potential nodes are *expanded* into Possible nodes, which can be *evaluated* against user preferences, before being *implemented*.

branches of the tree, with specific instances such as pyMol viewers forming sub-branches and leaves.

Feature trees can be used to represent the current state of software development; by labelling each node with a state, the team knows whether or not functionality for that feature has been implemented, and whether the conditions for implementing that functionality have been met. This forms a coordination artefact used by the development social machine, to organise construction and monitoring of the software artefact used in the target social machine. As shown in Figure 2, once features in the tree are implemented, the software artefact can be deployed to the user population.

Software development can be modelled as modifications to the feature tree: the re-labelling of nodes as new functionality is conceived of and implemented (before being deployed to

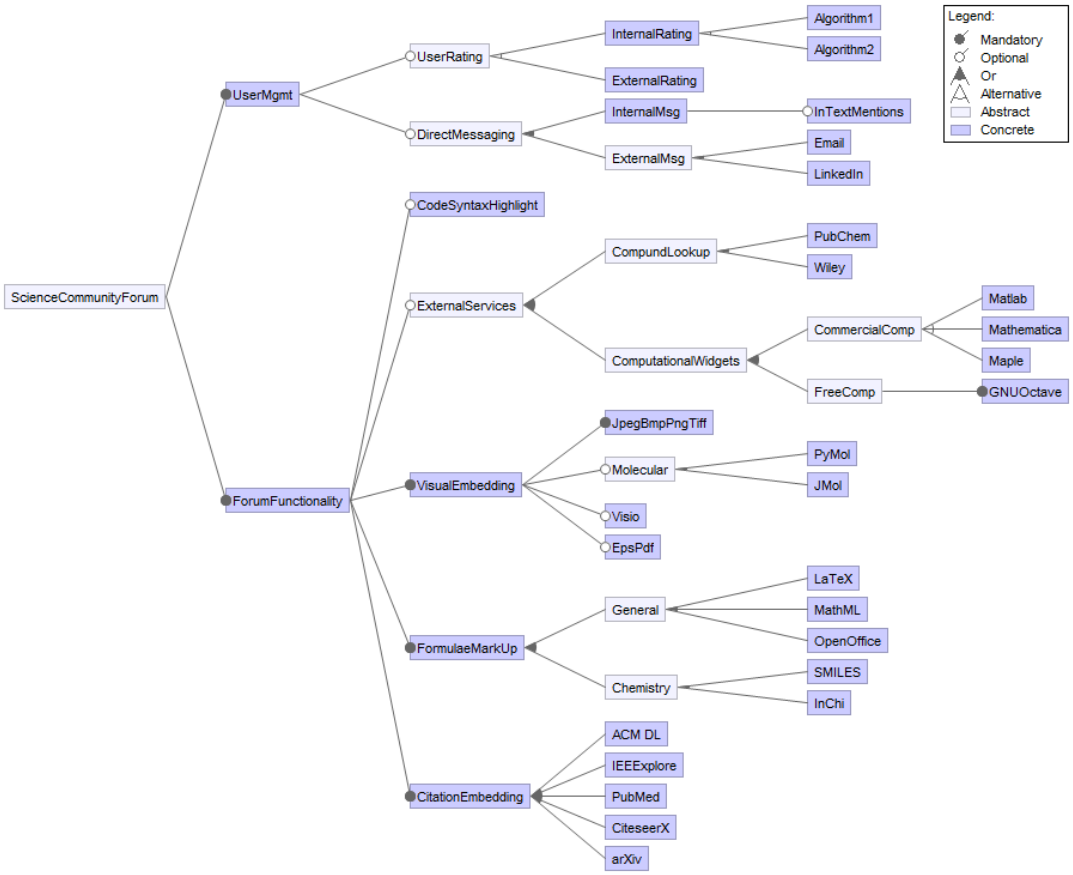

Figure 4. An example feature tree for a scientific forum software system.

the target artefact). In this paper, we use a simple state model (Figure 3), where each node is either:

i) *Implemented*—code has already been created for this feature, and it is available to users; ii) *Potential*—the feature has been conceptualised and designed, but no code exists yet; iii) *Possible*—part of the universe of possible features, but one which is not currently under consideration for implementation[2]

Based on this representation, development can contain the following steps, or *development primitives*, which map to operations of the feature tree:

1. *Expansion* of the tree converts nodes from *possible* to *potential* by finding new features to implement. This might be through expert designers, co-creation or direct user solicitation.

2. *Evaluation* of community needs and their relation to individual features results in labelling nodes with some indication of how well the community will react. There are many ways to do this, including surveying the participants; public consultations; focus groups; monitoring of behaviour; and social media analysis.

3. *Prioritisation* of features to implement, which may be driven by the result of evaluations, voting by the population, investor demands, expert opinion etc. This decision may depend on which type of *costs* the controllers of the artefact would like to optimise (e.g., economic, temporal, social).

4. *Implementation* of the selected features, whether in-house, or crowdsourced, using some particular software design methodology. When implementing features, the constraints contained in the feature tree must be observed (e.g., mandatory features, alternative features).

Within our model, these tasks are carried out by assembling teams of crowd professionals—SCUs (Section 2.2), capable of executing complex workflows. The formation of SCUs and coordination of their actions are carried out through the Coordination Model (CM), introduced in Section 3, which allows flexible workflows that adapt to emerging situations.

[2]The *possible* state is largely a convenience for simulation.

2.2. Social Compute Unit (SCU)

An SCU [8] is a loosely-coupled virtual team of socially-connected experts with skills in the relevant domain. The SCU is created upon request to solve a given task. It uses the crowdsourcing power of its members and their professional connectedness toward addressing the problem which triggered its creation and is dissolved upon problem resolution. The SCU is a programmable entity. This means that its various properties and functionalities (team assembly, task decomposition, runtime collaboration patterns, coordination, task aggregation) can be 'programmed' to support different types of human/machine collaborative efforts. For example, in [9] the authors show how the SCU can support well-defined business-processes, such as workflow patterns for IT incident management. However, SCU can also be used to perform looser collaboration patterns leaving space for human improvisation and creativity [10].

In this paper we use SCUs within the development social machine to execute tasks in the software development cycle, such as implementing a concrete software feature. Concretely, we build upon the particular SCU model presented in [11] and use it in the context of the encompassing social machine's coordination model. Whenever a development primitive (from Section 2.1) needs to be executed, a request with input parameters is sent to the SCU provisioning engine to form a team of developers/experts suitable for that particular primitive (task). The provisioning engine returns the closest-to-optimal matching subset of available developers, representing a SCU for that task.

The full list of available input parameters and descriptions of the team formation algorithms are available in [11]. In this paper, however, we vary only the parameter named *job description set* (J), while assuming default values for the remaining parameters. J contains job descriptions for each subtask: $J = \{j_1, j_2, \cdots, j_k\}$. A job description is a set of tuples $j_i = \{(t_1, q_1), (t_2, q_2), \cdots, (t_m, q_m)\}$, where t_l is a skill type (e.g., 'java developer', 'test engineer') and $q_l = \{`fair', `good', `verygood'\}$ is a fuzzy quality descriptor. The job description (t_l, q_l) specifies which skills a worker needs to possess in order to perform the subtask l successfully.

2.3. Lightweight Social Calculus (LSC)

LSC is an extension of LCC [12], which has been used to represent interaction in many systems [13]. LCC is a declarative, executable specification which can be communicated between agents at runtime; it is designed to give enough structure to manage fully distributed interactions by coordinating message passing and the roles which actors play, while leaving space for the actors to make their own decisions. LSC augments LCC with extensions designed to make it more amenable to mixed human-machine interactions; in practice, this means having language elements which cover user input, external computation or database lookup and storing knowledge and state.

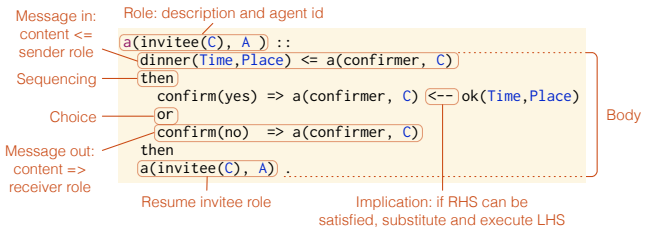

Figure 5. Example LSC clause from the meal organisation interaction model (slightly modified for clarity). An agent playing the role of `invitee` will wait for a message from a `confirmer` specifying the time and place for dinner; the values in the message for `Time` and `Place` are substituted in, and the agent then decides if it `will_attend`, and sends back the appropriate message. It then resumes the role of `invitee` in case of alternate suggestions.

An LSC protocol consists of a set of clauses; the head of each clause is a role specification, and the body a description of what an agent should do when playing that role (see example in Figure 5). The body contains message sending ($M \Rightarrow a(role, ID)$) and receiving ($M \Leftarrow a(role, ID)$), sequencing and choice ($then$ and or), implication ($action \leftarrow condition$), the assumption of new roles ($a(role, ID)$) and any extra computation or conditions necessary.

Each agent's interaction starts with a clause from a protocol, which is then repeatedly re-written in response to incoming events: incoming messages are matched against expected messages, role definitions are replaced with the body of matching clauses, values are substituted for variables and so on. As the interaction progresses, this *state tree* keeps a complete history of the agents actions and communications. This supports the creation of multi-agent institutions [14] where interaction is guided by shared protocols and a substrate which keeps track of state.

LSC is formal enough that it can be computationally manipulated, for example to synthesise new protocols[15]. It shares features with workflow languages—while providing more flexibility—and can be derived from e.g. BPEL4WS to create completely decentralised business workflows [16]. LSC has also been used in the creation of social machines by binding formal interaction models into natural interaction streams [17].

Within our model, LSC is used to model the development social machine, by specifying the interactions among developers, and between developers and the feature tree representing the state of the software artefact. It provides a means to create a formal representation of software development processes, allowing for computational coordination of their enactment, while providing more flexibility than a workflow would allow. LSC provides a bridge between

low level operations—e.g. implementing a particular node—and high level concepts such as "agile methodologies". By formalising the coordination protocols and making them first class objects, it is possible to share, modify, discover and rate individual protocols; by separating the protocol from the domain of application, it is possible to apply the same methodology to new domains. The flexibility of the language allows for sub-protocols to be chosen dynamically, so that development can be adapted in response to changing needs. Over time the system can build up a view of when each protocol is appropriate, and be able to assist with selection of protocols for novel situations.

3. Coordination Model

The *coordination model* represents the artefact regulating the interactions among social machines. It contains the following submodels, regulating different interaction aspects:

- **Data Submodel**: A formal data model used to represent the data that is processed and exchanged by social machines. It serves both as input and output for the social machine. In our example, the data model is represented by the feature tree representing the forum software. For the development social machine it serves to indicate the features to develop and dependencies; but also to track the progress of the development cycle. The resulting tree is then subsequently also used as the input of the target social machine for calculating the overall population utility, as well as to mark elicited features for future development cycles.
- **Quality-of-Service Submodel**: In essence, the development social machine is providing a software-development service to the target social machine. Therefore, we need a set of metrics to express the requested and measure the obtained quality of this service. In this paper, we adopt the metrics already provided by the SCU [11] to formulate requested QoS.
- **Interaction Submodel**: The coordination submodel contains a collection of LSC-encoded protocols managing interactions between social machines and their workers. The coordination submodel contains multiple possible protocols. A *metalevel protocol* is used to make real-time selection and enactment of an appropriate subset of concrete protocols, based on the current state of the coordination model, input from stakeholders or the current behaviour of the community interacting with the artefact. Selection could also include discovery of new protocols to use (e.g., as new development methodologies are introduced) as well as analysis of the historic performance of existing protocols in similar situations. The use of metaprotocols is crucial in order for the development social machine to be responsive to community requirements, and for it to adjust development trajectories accordingly.

Figure 6 illustrates the usage of the coordination model artefact for the scenario introduced in Section 1.1. An iteration in the software development cycle starts by having an active LSC protocol send a request to the SCU Provisioning Engine (Figure 6, ①). The request contains the necessary QoS input parameters (described in Section 2.2) for creation of multiple SCUs. Based on these parameters the SCU Provisioning Engine selects appropriate workers from the crowd of professionals (②), assembles and returns the SCUs. The newly created SCUs are passed the feature tree with nodes selected for implementation (③), finally constituting a functional development social machine. The development social machine starts performing the designated actions on the feature tree (④). The active LSC protocol from the interaction submodel takes care that the actions performed by different SCUs are properly ordered and repeated if necessary (e.g., due to failure, or insufficient quality). After the SCUs finish executing, the resulting feature tree is passed to the target social machine (5).

The modified tree is then evaluated by a function assessing the population utility (6). As explained earlier, this is a measure of target community's satisfaction with the implemented features. Based on the this value, and the new requests from the community, the metaprotocol can decide whether new development iterations are necessary, and if yes, which protocols to use (7). Depending on the new priorities, a different protocol can be chosen to control the development social machine in the new iteration. For example, for the *evaluate* action, new candidate features may be identified by a single SCU of experts, or by having multiple SCUs suggesting new features and then deciding by majority voting. Or, in case of a failure, we may decide to repeat the task with the same SCU, or escalate to a more reliable (and thus a more expensive) one.

In the following sections, we present a proof-of-concept implementation of this coordination model. We evaluate the implemented prototype by simulating a population and running a number of LSC protocols to showcase its functionality.

4. Implementation and Evaluation

4.1. Prototype Implementation

In order to demonstrate the operation of the coordination model, we have implemented a simulation prototype which covers a subset of the conceptual model's possible functionality. In the simulation, a pool of crowd workers participate in improving the scientific forum software introduced in Section 1.1. The (simulated) workers are managed by a system running various LSC protocols, representing different approaches to software development. This includes all of the task selection and implementation activities from Section 2, as well as the team selection work discussed in Section 2.2.

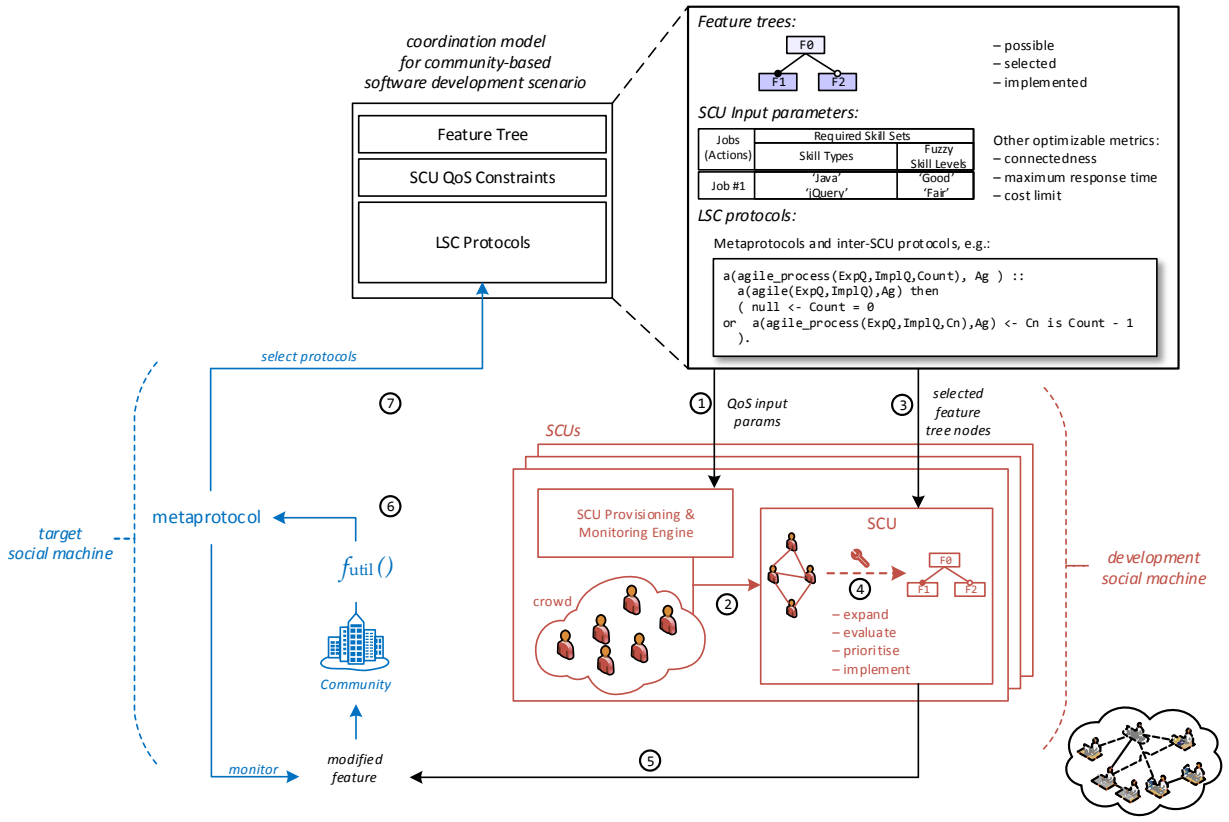

Figure 6. Using the Coordination Model to support the community-based, collaborative software development scenario.

The implementation uses the `scalsc` LSC library, with extensions to model feature trees, labour and team selection, and user populations[3]. Concretely, this comprises:

1. A population of simple software agents representing community members; this is simulated as a heterogeneous group of individuals, each with their own preferences about which features the community software should contain. The preferences are represented as scores for the presence of conjunctions or disjunctions of implemented feature-tree nodes. A typology approach is used, where archetypal users are defined for two classes (chemists and mathematicians), differing in their preferences for functionality. These users are sampled with multiplicative noise ($\mathcal{N}(1, 0.1)$) added to their preference scores to provide limited heterogeneity.

2. A feature tree representing the current state of the software, as defined in Section 3, following the example in Figure 4;

3. A simplified, idealised SCU model, where teams are formed in response to quality constraints, and perform tasks on the feature tree. In this simplified model, we assume that one worker is returned per task, with a skill set that exactly matches the quality constraints [11]. Workers have scores for four skills: *implementation, evaluation, prioritisation, design*, each of which ranges from $0..1$.

4. A labour model, relating worker qualities to the time, cost and quality of carrying out primitive tree operations. This model has been designed to represent the issues at hand in a stylised manner, while having a reduced parameter set to help understand the model's behaviour. Operations are assigned a basic cost C_o, which is then multiplied by the cost of the worker who is carrying it out; worker cost is the sum of the worker's skill levels (S) raised to an exponent $k = 0.5$, so the complete cost of a worker w operation o is: $C(w, o) = C_o \sum s^k$ (for $s \in S$). Details of operator cost, time and implementation specifics are in Table 1[4].

[3]Complete source code and installation instructions can be found at `https://bitbucket.org/mo_seph/social-institutions`

[4]We acknowledge that the simulation will be sensitive to the parameter values chosen (especially k); the results we present are intended only to give

Table 1. Model of primitive tree operations, showing base costs, assumptions made and implementation details. e_{real} is the true population utility for a feature tree, and s_x is the team's skill in x.

Operation	Assumptions	Implementation	Cost, Time
Expand	Only children of *potential* nodes are considered. A better design process will create nodes which better match the population's need	Order nodes by $e_{real}\mathcal{N}(1,(1-s_{design}))$ and select the first.	2, 0.5
Evaluate	A better evaluation process will be closer to the true value population's need	Label nodes with $e_{est} = e_{real}\mathcal{N}(1,(1-s_{eval}))$.	0.5, 0.5
Prioritise	Better prioritisers order nodes more closely to their true evaluation order with respect to population's need	Order nodes by $e_{est}\mathcal{N}(1,(1-s_{prioritise}))$ and label with index.	0.01, 0.01
Implement	Select highest priority node; better implementers have more chance of success.	$P_{impl} = s_{implementation}$	1, 1

In order to effect changes to the feature tree, a set of LSC-based intra-unit- and meta-protocols are used to construct SCU according to quality metrics, and schedule them to carry out operations. Listing 1 shows an example high-level LSC protocol coordinating an agile development process: `form_scu` triggers the SCU formation, based on a set of required skills and the action to enact, while `do_task` controls the execution of the selected actions[5]. These protocols can be written as standard LSC [12], with a small set of extra predicates for forming teams and manipulating trees: `form_scu` and `do_task` mentioned above, as well as `current_tree` and `highest_priority`, which are demonstrated in Listing 1.

4.2. Initial Scenarios

In order to illustrate the operation of the prototype, we run it under two contrasting scenarios, with three different *LSC-based workflows* imposed through appropriate LSC protocols. In the first scenario—*StablePopulation*, a population of 1000 members of the chemistry community (chemists) is simulated throughout the entire simulation runtime. In the second scenario—*DynamicPopulation* the initial population of 1000 chemists is replaced by 200 chemists and 800 mathematicians at timestep 20. This is a crude and stylised approach to representing a shift in user population, where the platform is adopted by a different user community, but it allows us to illustrate the prototype implementation's behaviour.

The coordination models we use are loosely based on current or past practice in software development:

- The *traditional* model starts with a large public consultation, where most of the tree is explored and assessed before any implementation takes place. Actual feature implementation is then carried out by teams (i.e., SCUs) of average-skilled programmers, who have three attempts to implement any given node.
- The *escalation* model begins with the same initial public consultation, but is followed by a development process where initially an average (and cheap) developer attempts to implement each node. If that fails, a high quality, but more expensive, developer is found and brought in to finish the job. This is an example of a simple metaprotocol, allowing alternative development pathways to be chosen at runtime.
- The *agile* model defines a a tight loop of evaluation and implementation, to allow development to respond to a changing set of user requirements.

4.3. Protocol Assessment and Discovery

To illustrate the possibility of evaluating and discovering protocols—due to their status as first-class objects [18]—we: i) implement a collection of performance metrics which might relate to real-world quatitites of interest; ii) assess different parameterisations of the *agile* protocol discussed above. The *agile* protocol was picked due to its potential for use in the *dynamic* protocol described in Section 4.4. The protocol consists of a role which two parameters: $EvalQ$ and $ImplQ$, the skill levels of the evaluating and implementing SCUs respectively, both in the interval $[0,1]$. The values of these two parameters were varied separately so that the effects of prioritising resources towards the implementation or design stages of the process may be apparent.

Several metrics are used based on a combination of implementation pragmatics and applicability to the scenario—these are by no means a recommended or complete set—and the most relevant of these are displayed based on the above factors and an ability to meaningfully distinguish between different protocol settings. As well as four basic

an indication of capabilities, so no formal sensitivity analysis has been carried out.

[5]Further details on the prototype implementation, as well as the source code can be found at `https://bitbucket.org/mo_seph/social-institutions`

Listing 1. Example protocol used to coordinate "agile" development. An SCU is first formed to identify the next best node to implement. Then, another SCU is formed to implement that node. This sequence is then run in a tight loop to carry out responsive development.

```
a(agile(ExpQ,ImplQ),A) ::        %Agent role for doing "agile" development
                                 %Create an SCU to expand the next best node
  form_scu(expand(1), [expansion(ExpQ)], ExpAssign ) then
  current_tree(InTree) then      %Get the current tree
  do_task(ExpAssign, InTree, ExpTree ) then   %Carry out the expansion
  highest_priority(ExpTree,Next) then         %Find the best node to implement
                                 %Form an SCU to implent it
  form_scu(implement, [implementation(ImplQ)], ImplAssign ) then
  do_task(ImplAssign, ExpTree, Result ) .     %Carry out the implementation
```

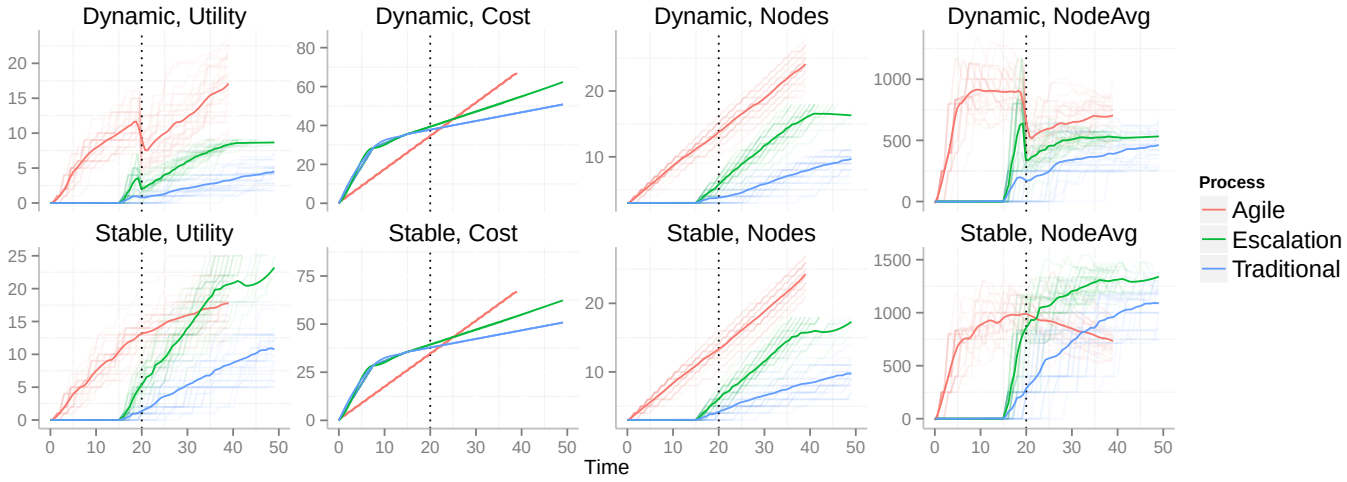

Figure 7. Simulation outcomes for the two scenarios and three workflows. Faint lines represent individual runs ($n = 30$ in each condition), and solid lines are smoothed ensemble averages. *Utility* is the average utility of the feature tree across the population; *Cost* is the financial cost of developing the community-software; *Nodes* is the number of features (nodes) implemented; and *NodeAvg* is the average utility provided by each feature (an indication of the quality of community fit). The dotted vertical lines indicate the time where the *Dynamic* scenario undergoes a step change in the population composition.

properties—average utility, cost, number of nodes and cost per node, five more complex metrics were chosen:

UtilityAvgCost is the cost of creating one unit of utility, that is $Cost/Utility$. This represents the effectiveness of the protocol at creating high value nodes cheaply.

UtilityIncreaseRate-10 is the increase in utility over the last 10 timesteps, showing the protocol's ability to respond to changes and rapidly develop new features.

UtilityPerNode indicates how good the protocol is at targeting crucial nodes, how precise the development is.

IntegratedUtility is aimed at analysing the development of the population. Since we do not model feedbacks between the user population and the state of the artefact, this helps to separate situations where the artifact maintained high utility throughout—and hence is likely to have maintained an active userbase—from situations where development was initially slow, which might have led to users leaving early on.

NodesPerTime measures the average throughput of the protocol in terms of implemented nodes per timestep.

In order to get a better sample, random trees and populations were used for this experiment, according to the following principle:

- a tree was generated, with a given branching factor and maximum depth, with random unique IDs for each node;

- two archetypes were generated for this tree which gained utility for the presence of up to 15 of the nodes in the tree, with the level of utility sampled from $\mathcal{N}(0, 2)$.

- these were then used in a population dynamics model, following the template of the $dynamic$ scenario above, with a step change from purely one type to an equal mix of both types at timestep 20.

Based on the availability of metrics, a simulated protocol discovery mechanism is implemented. This takes the form of a database of protocols and roles, searchable according to individual scores for each metric. These scores were taken from the average across multiple runs of either i) the maximum value or ii) the final value recorded in the simulation as appropriate for the metric. The discovery mechanism uses the calculated scores to select the protocol and role that are expected to either increase, decrease, maximize, or minimize particular metrics.

This discovery mechanism is made available to the agents within the simulation, by means of a special predicate: satisfying $i(discover(\text{“}CostPerNode'', min, Proto, Role))$ would return substitutions such as $Proto \rightarrow agile, Role \rightarrow agile(0.5, 0.8)$, indicating that the way to minimise the cost per node is to enact the role $agile(0.5, 0.8)$ in the *agile* protocol.

4.4. Dynamic Protocols

Another key benefit of protocols is their flexibility. State can be brought in, allowing the protocol to adapt to changing situations. In order to demonstrate this, Listing 2 shows a simple *dynamic* protocol. Here, the agents are provided with an extra predicate—`metric(M,U)`—which retrieves the current value U of a metric M. For example, the predicate `metric("AvgUtility",U)` gives the average utility of the population at this moment in time. Protocols can then be written which react to this. For example, a protocol can use this in order to prioritise economic development when the population is happy, but increase utility at any cost when the population is less happy. There are two steps to this: choosing which goal to prioritise in terms of a metric to minimise or maximise, and then using the discovery mechanism to select the protocol which best fits that goal.

For simulation purposes, a more complex protocol was used which would:

- start with the most rapid development strategy (maximise $NodesPerTime$);

- monitor the $CostPerNode$ and $AvgUtility$;

- if $CostPerNode$ is too high, find a role with a lower expected cost per node

- if the $AvgUtility$ is too low, find a role with a higher $UtilityIncreaseRate - 10$

This protocol was run alongside the previous protocols from Section 4.2: *traditional,escalation* and *agile*, as well as the best performing set of agile parameters (Agile1, with $EvalQ = 0.9, ImplQ = 1.0$) in order to observe the effects of dynamically selecting protocols and parameters. The output of the search over the $agile$ parameter space was used, so that all of the roles which were selected were of the form `agile(E,I)`.

5. Results

Figure 7 shows the simulation outcomes when running the described scenarios and workflows. Under the *Stable* scenario, the *agile* workflow initially performs best, as development starts immediately. Over time, however, the *escalation* workflow achieves higher utility with fewer nodes due to a greater understanding of the complete feature tree. The *traditional* workflow is limited by the speed of its average-skilled developers, but utility does increase gradually. Under the *Dynamic* scenario, the initial behaviour is similar. However, when the population changes at timestep 20, the *agile* workflow adapts more quickly to the change, and creates nodes which better represent the desires of the new population.

To zoom in on the effects of QoS selection, Figure 8 shows the performance of the *agile* protocol with a range of parameterisations for a range of metrics [6]. In most cases, the broad shape of the metric curves is similar. It is clear that the overall system state—few implemented nodes at the beginning versus many towards the end—and the population dynamics have a strong effect on the curves. However, the QoS parameters selected can have a large effect on the ranges of the curves. It is also clear that the metrics are able to differentiate between different parameter settings, with most graphs showing a clear ordering. Different metrics produce different orderings, for example the $Cost$ graph shows that for a value of $EvalQ = 0.8$, the *agile* algorithm costs more in total for $ImpQ = 0.5$ (blue) than for $ImpQ = 0.2$ (cyan), but the $CostPerNode$ metric shows the average cost per implemented node is actually less for $ImpQ = 0.5$ (thus indicating it implements nodes faster).

Figure 9 compares the behaviour of the *Dynamic* protocol (green) with the previous ones over randomly generated trees. Comparing the two simple agile protocols to the dynamically assigned one, it can be seen that the dynamic protocol has the lowest cost in total, per node, and per unit of utility, while maintaining utility and throughput to levels comparable to the ‘optimal’ parameter settings.

6. Discussion

The initial round of simulations demonstrates the ability of the coordination model to represent a flexible process.

[6] As stated previously, these metrics are intended to be plausible, rather than comprehensive—the metrics only need to be good enough to drive the protocol selection described in Section 4.4.

Listing 2. Example dynamic protocol, used to choose roles The $metric$ predicate retrieves the current value of a particular metric. In this particular case, $metric(\text{"}AvgUtility\text{"}, U)$ reports on user satisfaction, which is then fed into a choice of which metric to prioritise

```
a( dynamic_process(Count,T),X ) ::                          % If the average utility is more than a
  ( i(discover("CostPerNode",min,P,Role)) <- metric("AvgUtility",U) ^ U > T
    or                                                      % threshold T, minimise cost per node
    i(discover("UtilityIncreaseRate",max,Proto,Role))       % Otherwise, maximise utility increase
  ) then a(Role,X)                                          % recursion for Count nodes elided
```

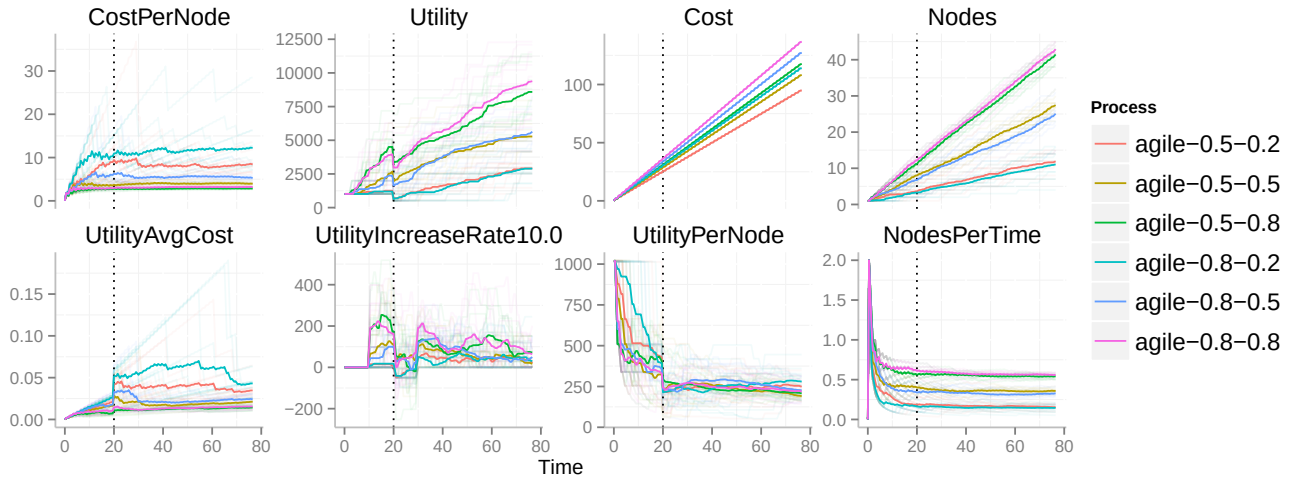

Figure 8. Graph of the performance of the *agile* protocol on several different metrics over a representative sample of parameter space. Parameters $EvalQ$ and $ImplQ$ are sampled at $0.5, 0.8$ and $0.2, 0.5, 0.8$ respectively, giving 6 conditions. The metrics shown are: i) $CostPerNode$, the average cost for each node implemented; ii-iv) $Utility$, $Cost$, and number of $Nodes$; v) $UtilityAvgCost$, the average cost of each unit of utility; vi) $UtilityIncreaseRate10.0$, increase rate in utility over a 10 tick window; vii) $UtilityPerNode$, the total utility divided by the number of implemented nodes; viii) $NodesPerTime$, the average number of implemented nodes per unit of time. All simulations were carried out on randomly generated trees with branching factor 4 and max depth 3, with a population of 1000.

When the user community changes, the coordination model can respond dynamically, by prioritising different software features for implementation. The recovery of the utility curve in the for the *agile* workflow in the *Dynamic* scenario post population-change, indicate its ability to responsively re-plan; by not working with a heavyweight process, it can do what the users need right now.

The difference between the utility curves under the *traditional* and *escalation* workflows demonstrates the effects of bringing QoS constraints into the development protocol. The *traditional* workflow tends to be cheaper per unit time, using only low quality developers. However, the *escalation* workflow creates more nodes per unit cost, by being able to form SCUs with highly skilled workers when necessary to carry out difficult jobs. This also results in more nodes created per unit time, so the population utility rises more rapidly.

In contrast to conventional workflows, the LSC protocols are dynamic, and can be changed or "plugged in" during runtime. For example, the *dynamic* protocol uses the average population utility to influence the choice of protocols. This ability to alter the way that work is coordinated allows for a larger human influence on the execution of complex work processes.

This demonstrates how a dynamic protocol can be responsive to population changes and the needs of system designers at once. In this case, there are two components of the response. Firstly, the protocol is set up to prioritise different goals at different times. This is an opportunity for the system designers to elicit and encode their goals, to use their human knowledge of the system and declare their priorities. Secondly, the protocol can bring in computational machinery to discover protocols which have been previously found to match those goals, and then implement them. This

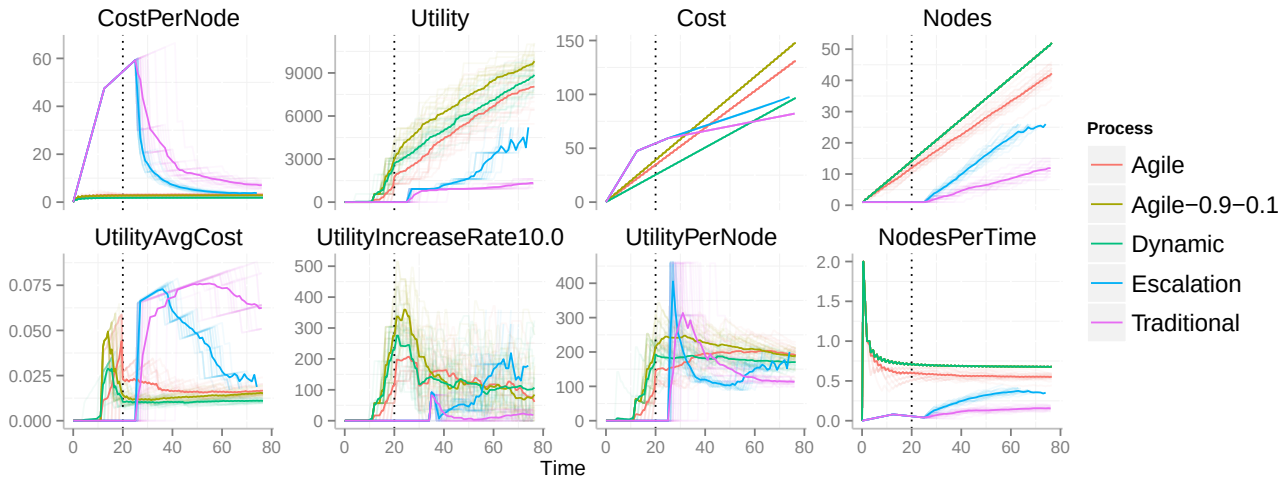

Figure 9. Graph of the performance of the *Dynamic* protocol compared to the more traditional ones. Metrics are the same as in Figure 8. All simulations were carried out on randomly generated trees with branching factor 4 and max depth 3, with a population of 1000.

allows the re-use of community knowledge, and a greater understanding the behaviour of different methodologies in different situtations. Synergistically, these capabilities allow for dynamic, flexible protocols which can draw on a pool of cloud workers to create software artefacts in response to community needs.

6.1. Simulation Modelling

In order to explore a range of parameters and models quickly and flexibly, we have taken a simulation modelling approach. Simulation modelling is used in the computational social sciences to explore theoretical ideas in the context of synthetic populations, particularly where real studies would be impractical [19, 20]. Recently, this has been applied to crowdsourcing, in order to generalise results which otherwise would be tied to a particular situation [21]. However, simulation has the potential to play another role in this area, as developing a computational model of population behaviour can be used to "close the loop" and aid in the design of effective social machines [22].

The simulations presented here represent a highly simplified version of our conceptual model where intelligent computational machinery underpins human creative activity in the development of software artefacts for dynamic communities. The use of flexible process languages such as LSC means that the inter-unit protocols used here could be augmented to embody more refined development methodologies, with complex patterns of coordination where necessary. Similarly, at the metaprotocol level, we have a dynamic adjustment of the protocols and parameters chosen, but there is room for additional intelligence to be brought to bear in order to better balance community and stakeholder demands against time and cost constraints. We have sketched out a mechanism by which the interactions specified in LSC can be rated and discovered, but there is scope for them to be exchanged and modified both computationally or through human intervention. This can help create a better understanding of which methodologies work in which situations. At the intra-unit level, intelligent protocols could be used to more flexibly assign workers to sub-tasks, reacting to developing situations and changing requirements.

7. Related Work

Social machines share common ground with other collective intelligence applications such as human computation and social computing (diagram in [5, p.2]). Many crowdsourcing systems can be seen as social machines. Of the existing commercial platforms, of particular relevance here are Topcoder[7] and ODesk[8], which use different mechanisms to organise diverse participants around software development. As crowdsourcing platforms are becoming widely used as research tools, a number of solutions appeared providing overlay abstractions offering more advanced workflow management and allowing users to perform more complex tasks/computations.

TurKit [23] is a 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. The entire synchronisation, task splitting and aggregation is left entirely to the programmer. The inter-worker synchronisation is out of programmer's reach. The only constraint that a programmer

[7] http://www.topcoder.com/
[8] https://www.odesk.com/

can specify is to explicitly prohibit certain workers to participate in the computations.

Jabberwocky's [24] *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. A number of *map* steps can be performed in sequence, followed by possibly multiple *reduce* steps. Human computations stops the execution until it is performed. While automating the coordination and execution management, Jabberwocky is limited to the MapReduce-like class of problems.

AutoMan [25] integrates the functionality of crowdsourced multiple-choice question answering into Scala programming language. The authors focus on automated management of answering quality. The answering follows a hardcoded workflow. Synchronisation and aggregation are centrally handled by the AutoMan library. The solution is of limited scope, targeting the designated labour type.

CrowdLang [26] brings in a number of novelties in comparison with the other systems, primarily with respect to the collaboration synthesis and synchronisation. It enables users to (visually) specify a hybrid machine-human workflow, by combining a number of generic 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. The use of human workflows also enables indirect encoding of inter-task dependencies.

To the best of out knowledge, at the moment of writing, CrowdLang and SCU are the only two systems offering execution of complex human-machine workflows. However, as explained before, both systems need to know the possible (sub-) workflows in advance. The coordination model presented in this paper complements the functionality offered by systems such as these two, by providing a higher-level coordination management layer.

8. Conclusion

In this paper we introduced a novel coordination model for teams of workers performing creative or engineering tasks in complex collaborations. The coordination model augments the existing Social Compute Unit (SCU) concept with coordination protocols expressed using the Lightweight Social Calculus (LSC). The approach allows combining coordination and quality constraints with dynamic assessments of user-base requirements. In contrast to existing systems, our model does not impose strict workflows, but rather allows for the runtime protocol adaptations, potentially including human interventions. We evaluated our approach by implementing a prototype version of the coordination model for the exemplifying case-study and simulated its behaviour on a heterogeneous population of users, running different scenarios to demonstrate its effectiveness in delivering end-user utility, and illustrated responses to a dynamically changing population.

In summary, we have given a conceptual model for combining process models with crowdsourced teams to create software artefacts in support of dynamic communities. This formalisation paves the way for increased intelligence to be brought into crowdsourced software development, creating a more responsive, community-centred process.

Acknowledgement

This work is partially supported by the EU FP7 SmartSociety project under grant 600854 and the EPSRC SociaM project under grant EP/J017728/1.

References

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

[2] Doan, A., Ramakrishnan, R. and Halevy, A.Y. (2011) Crowdsourcing systems on the World-Wide Web. *Communications of the ACM* **54**(4): 86.

[3] Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M. *et al.* (2013) The future of crowd work. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13 (New York, NY, USA: ACM): 1301–1318.

[4] Berners-Lee, T. and Fischetti, M. (2000) *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor* (Harper Information).

[5] Shadbolt, N.R., Smith, D.A., Simperl, E., Kleek, M.V., Yang, Y. and Hall, W. (2013) Towards a classification framework for social machines. In *SOCM2013: The Theory and Practice of Social Machines* (Association for Computing Machinery).

[6] Smart, P., Simperl, E. and Shadbolt, N. (In Press) A Taxonomic Framework for Social Machines. In Miorandi, D., Maltese, V., Rovatsos, M., Nijholt, A. and Stewart, J. [eds.] *Social Collective Intelligence: Combining the Powers of Humans and Machines to Build a Smarter Society* (Springer Berlin).

[7] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G. and Leich, T. (2014) FeatureIDE : An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* **79**: 70–85.

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

[9] SENGUPTA, B., JAIN, A., BHATTACHARYA, K., TRUONG, H.L. and DUSTDAR, S. (2013) Collective Problem Solving Using Social Compute Units. *Internationl Journal of Cooperative Information Systems* **22**(4).

[10] RIVENI, M., TRUONG, H.L. and DUSTDAR, S. (2014) On the elasticity of social compute units. In *CAiSE*: 364–378.

[11] CANDRA, M.Z.C., TRUONG, H.L. and DUSTDAR, S. (2013) Provisioning Quality-aware Social Compute Units in the Cloud. In *11th International Conference on Service Oriented Computing (ICSOC 2013)* (Berlin, Germany, December 2-5: Springer).

[12] ROBERTSON, D. (2005) A lightweight coordination calculus for agent systems. In LEITE, J., OMICINI, A., TORRONI, P. and YOLUM, P. [eds.] *Declarative Agent Languages and Technologies II* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **3476**, 183–197.

[13] ROBERTSON, D. (2012) Lightweight coordination calculus for agent systems: Retrospective and prospective. In SAKAMA, C., SARDINA, S., VASCONCELOS, W. and WINIKOFF, M. [eds.] *Declarative Agent Languages and Technologies IX* (Springer Berlin Heidelberg), *Lecture Notes in Computer Science* **7169**, 84–89.

[14] D'INVERNO, M., LUCK, M., NORIEGA, P., RODRIGUEZ-AGUILAR, J.A. and SIERRA, C. (2012) Communicating open systems. *Artificial Intelligence* **186**: 38–94.

[15] MCGINNIS, J., ROBERTSON, D. and WALTON, C. (2006) Protocol synthesis with dialogue structure theory. In *Argumentation in Multi-Agent Systems* (Springer), 199–216.

[16] GUO, L., ROBERTSON, D. and CHEN-BURGER, Y. (2008) Using multi-agent platform for pure decentralised business workflows. *Web Intelligence and Agent Systems* **6**(3): 295–311.

[17] MURRAY-RUST, D. and ROBERTSON, D. (2014) LSCitter: Building social machines by augmenting existing social networks with interaction models. In *Proc. World Wide Web Companion*, WWW Companion '14: 875–880.

[18] MILLER, T. and MCGINNIS, J. (2008) Amongst first-class protocols. In *Engineering Societies in the Agents World VIII*, 208–223.

[19] GILBERT, N. and TROITZSCH, K. (2005) *Simulation for the social scientist* (McGraw-Hill International).

[20] MACAL, C.M. and NORTH, M.J. (2010) Tutorial on agent-based modelling and simulation. *Journal of Simulation* **4**(3): 151–162.

[21] BOZZON, A., FRATERNALI, P., GALLI, L. and KARAM, R. (2013) Modeling crowdsourcing scenarios in socially-enabled human computation applications. *Journal on Data Semantics* : 1–20.

[22] KAMAR, E., HACKER, S. and HORVITZ, E. (2012) Combining human and machine intelligence in large-scale crowdsourcing. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1* (International Foundation for Autonomous Agents and Multiagent Systems): 467–474.

[23] LITTLE, G. (2009) Turkit: Tools for iterative tasks on mechanical turk. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*: 252–253.

[24] AHMAD, S., BATTLE, A., MALKANI, Z. and KAMVAR, S. (2011) The jabberwocky programming environment for structured social computing. In *ACM Symposium on User Interface Software and Technology*, UIST '11 (New York, NY, USA: ACM): 53–64.

[25] BAROWY, D.W., CURTSINGER, C., BERGER, E.D. and MCGREGOR, A. (2012) Automan: A platform for integrating human-based and digital computation. *ACM SIGPLAN Not.* **47**(10): 639–654.

[26] MINDER, P. and BERNSTEIN, A. (2012) Crowdlang: A programming language for the systematic exploration of human computation systems. In *Proc. of Social Informatics (SocInfo'12)*: 124–137.