PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

rsta.royalsocietypublishing.org

Research

Cite this article: Robertson D, Giunchiglia F. 2013 Programming the social computer. Phil Trans R Soc A 371: 20120379.
http://dx.doi.org/10.1098/rsta.2012.0379

One contribution of 15 to a Discussion Meeting Issue 'Web science: a new frontier'.

Subject Areas:
human–computer interaction

Keywords:
social machines, networks, knowledge representation

Author for correspondence:
D. Robertson
e-mail: dr@inf.ed.ac.uk

Programming the social computer

David Robertson[1] and Fausto Giunchiglia[2]

[1]School of Informatics, University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK
[2]Department of Computer Science and Information Engineering, University of Trento, 38050 Povo, Trento, Italy

The aim of 'programming the global computer' was identified by Milner and others as one of the grand challenges of computing research. At the time this phrase was coined, it was natural to assume that this objective might be achieved primarily through extending programming and specification languages. The Internet, however, has brought with it a different style of computation that (although harnessing variants of traditional programming languages) operates in a style different to those with which we are familiar. The 'computer' on which we are running these computations is a social computer in the sense that many of the elementary functions of the computations it runs are performed by humans, and successful execution of a program often depends on properties of the human society over which the program operates. These sorts of programs are not programmed in a traditional way and may have to be understood in a way that is different from the traditional view of programming. This shift in perspective raises new challenges for the science of the Web and for computing in general.

1. Introduction

Since 2002, the UK informatics community has maintained a list of grand challenges for computing research. One of the earliest of these challenges was, at the time, paraphrased as 'programming the global computer'. The challenge was stimulated by the realization that computers (in all their many forms) were becoming ubiquitous in much of society, to the extent that 'a sharp distinction can no longer be made between man-made software and natural informatic processes' [1].

Royal Society Publishing
Informing the science of the future

When Milner and others identified this challenge, they took the somewhat provocative stance of identifying computation across the system of global networks as a single global computer. We take a similar view. What has changed since 2002, we claim, is that the embedding of computation into society via personal devices, and sensors is now so deep that the majority of the computation being done in some circumstances is by humans supplying the social frameworks that support the operation of algorithms. This changes the way we must think about computation and about programming, as we shall argue in the remainder of this paper.

2. What is a social computer?

Historically, the term social computing has been used in a broad sense to describe almost any form of problem solving for which people combine efforts on a large scale in order to tackle a joint problem, irrespective of whether or not this activity relies on clearly defined algorithms and the application of programming methods. As we understand more about the ways in which we can embed computing into society and people become attuned to this sort of activity, we are building a class of programs that rely for their operation on algorithms that must run in human society in concert with computer systems.

As an example of the most basic form of social computation, consider the following generalized problem.

> Informal problem specification. *A number of events, identifiable by most individuals in the population, are likely to occur within a given time frame in some geographical area but we do not know where they will occur. It is infeasible to find out by applying technology directly. Nevertheless, we want to find out as quickly as possible when and where the events have occurred.*

A concrete instance of this general specification was the Defense Advanced Research Projects Agency (DARPA) Network Challenge of 2009 in which the aim was to locate in the shortest time possible 10 weather balloons fixed simultaneously at unspecified locations across the USA. In this instance, the events in our informal specification are the fixing of the balloons. No directly applied technology (other than perhaps a massive remote-sensing effort) is likely to find them quickly, but a social computation found all the balloons in under 9 h, which is an impressive achievement on a problem of this geographical scale.

Returning to our generalized problem specification, one way to tackle the problem is as follows.

> Informal solution specification. *Build a system that allows people to register as reporters of the target events and subsequently report the location if they witness an event. If reporters are recruited in sufficient numbers to cover the geographical area, then these act as ad hoc sensors of the events. To help gain this coverage, allow people to recommend their friends as reporters.*

This is a simplified version of the method used to win the DARPA Network Challenge, and we will use this simplified method as an example in the remainder of this section.

The solution given informally above can be expressed more formally (though still with insufficient detail to be a complete specification) as the pair of finite state machines (FSMs) shown in figure 1. The FSM on the left describes the computer system that acts as the coordinator for the computation. It receives inputs of recommendations for new reporters and reports of witnessed events, continuing in its role of coordinator after each comes in. The FSM on the right describes the behaviour required of a participating person in the physical world. Each person starts in the role of an applicant who, having sent as output the necessary registration information, continues in the role of a helper for the computation, which involves (as outputs) recommending friends or reporting witnessed events.

Having specified high-level requirements for the two types of component for this system (the computer system and the individual participant), we now refine these descriptions to specify

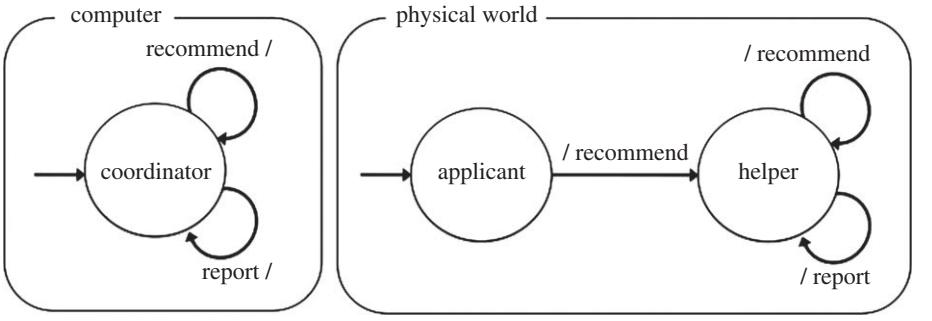

Figure 1. States and transitions for our social computation example.

precisely how messages may be passed between these two components. We use a simplified version of the Lightweight Coordination Calculus ([2,3]; a language used by the authors for peer-to-peer knowledge sharing via shared protocols, as we describe in §4) for this purpose. Our specific choice of language is not important, so we explain below only the basics needed to follow the example. Note also that, although we have indeed used our methods in peer-to-peer environments, our use of the word 'peer' refers to any system (including humans) capable of playing a role in an interaction in whatever form of network infrastructure.

First, we specify the coordinator component that will be computer based. Given an initially empty set of peers, P, and an initially empty set of locations for targets, L, a coordinator, $coordinator(P, L)$, succeeds in an interaction to find all the targets if the list of required target locations is empty, or if a message recommending a new participant, A, is received from person R with coordination continuing; or a report of event, X, is received from person R with coordination continuing. The expression $Z \leftarrow M \Leftarrow R$ denotes that a constraint, Z, must be satisfied upon receipt of a message, M, from R (otherwise, the peer's behaviour is inconsistent with the specification). The operator *or* is committed choice (that is, we may choose A or B to satisfy A *or* B, but we must stick to the choice we have made).

$$
\begin{aligned}
&coordinator(P, L) ::\\
&\quad L = [] \ or\\
&\quad (P = [(A, R_1)|P_1] \leftarrow recommend(A) \Leftarrow R_1 \ then \ coordinator(P_1, L)) \ or\\
&\quad (L = [(X, R_2)|L_1] \leftarrow report(X) \Leftarrow R_2 \ then \ coordinator(P, L_1))
\end{aligned}
$$

Now we specify the required behaviour of a participant operating in the physical world, but interacting with the computer system. The participant begins in the role, $applicant(C)$, which succeeds if the applicant recommends itself to the coordinator, C, and then undertakes the role of $helper(C)$. The role of helper involves sending a recommendation of a friend, A, or a report of a witnessed event, X, and then continuing as a helper. The expression $M \Rightarrow C \leftarrow Z$ denotes that a message, M, can be sent to C, if constraint Z is satisfied.

$$
\begin{aligned}
&applicant(C) :: recommend(self) \Rightarrow C \ then \ helper(C)\\
&helper(C) :: \begin{pmatrix} recommend(A) \Rightarrow C \leftarrow friend(A) \ or \\ report(X) \Rightarrow C \leftarrow witnessed(X) \end{pmatrix} then \ helper(C)
\end{aligned}
$$

The definitions above of $coordinator(P, L)$, $applicant(C)$ and $helper(C)$ together provide a precise specification of a computation. From a mathematical point of view, they appear similar, but in implementation, they are fundamentally different because the execution of the coordinator role depends on the computational system, while the execution of the applicant and helper roles depends on the (human) social system. No useful computation will occur unless the constraints $friend(A)$ and/or $witnessed(X)$ are satisfied, and this can only occur through the

involvement (and in this example, the active efforts) of the humans playing appropriate roles in the computations. Given this, we define a social computation as given below.

(a) Social computation

A computation for which an executable specification exists, but the successful implementation of this specification depends upon computer-mediated social interaction between the human actors in its implementation.

Because social computations, in our sense of the term, are capable of being described mathematically, it may also be possible to specify some of the social properties that a given specification must possess. These properties are not the normal properties we associate with program specifications (such as termination or freedom from deadlock, which also may be desirable), but relate to the drivers for the adoption and spread of the computation through the social group with which it engages. We give below an example of a social property of the specification in our running example.

We define $completed(R)$ to denote that a role, R, has been executed successfully; $all\_found(L)$ to denote that all the events we wish to find are in event report list, L; $reward(N)$ to denote the size, N, of the reward advertised for reporting an event (this supplies an individual with an incentive for participating); and $pay(A, N)$ to denote that person, A, will be paid reward of size N. If the role of coordinator is completed and all the target events are found, then those who located the target events will be paid the advertised reward, and anyone who recommended someone who is paid a reward will be paid half the amount given to the person they recommended.

$$\begin{aligned}&\forall P, L.(completed(coordinator(P.L)) \ \wedge \ all\_found(L)) \rightarrow \\ &\quad (\exists X, R, N.(X, R) \in L \ \wedge \ reward(N) \rightarrow pay(X, N)) \ \wedge \\ &\quad (\exists X_1, N_1, X_2.(pay(X_1, N_1) \ \wedge \ (X_1, X_2) \in P \ \wedge \ \neg(X_1 = X_2) \rightarrow pay(X_2, N_1/2))\end{aligned}$$

The effect of this property, if it is made evident to those who might participate in the computation, is to encourage them to participate (because they may get paid) and also to recommend others (because they might get paid something as a consequence of recommending someone else).

Properties like the one above look similar to the sorts of formal requirements that we often find in the traditional requirements engineering literature. Unlike traditional requirements, which are normally used only as part of the software design process, social properties are an essential part of the execution of the social computation. In our example, each potential participant in the computation must be aware of the property above or he/she will have no incentive to take part in the computation and our computation will not succeed. Furthermore, operations such as $pay(X, N)$ are intended to have a physical consequence in the real world. Social properties, unlike conventional program requirements, are an integral part of the operational program.

(b) Social property

A requirement associated with the specification of social computation that must be maintained, and perhaps communicated, during the execution of the specification in order for the computation to establish the social group needed to run it.

So far, we have discussed individual social computations. To build a social computer, we need an architecture in which we can develop and run a large number of social computations. Part of this architecture is, of course, the underlying infrastructure for networked computation. We know how to run executable specifications like the one we described earlier (for example, using service orchestration systems or even peer-to-peer infrastructures). This, however, provides for only one execution trace in our computer (analogous to running a single thread of computation on a conventional multi-processor machine). On a social computer, we need to run a very large

number of social computations and, crucially, society itself has to decide which computations to run and how they will execute. This involves, as a minimum, three processes.

Initiation of specifications for social computations by individuals or groups; these are the source code for the social computer. The conventional way to initiate these is by programming them, but the complexity of social computation appears to be in the choice of social group (so as to ensure that social properties are preserved) rather than in the difficulty of coding (because the specifications themselves may be comparatively small and stylized). If this is true, then it may be possible to initiate social computations directly (or semi-directly) from the activities of individuals or social groups rather than as a traditional coding activity, for instance, by the sort of example-driven methods proposed in Harel [4].

Adoption of roles in initiated social computations by individuals in the population. To do this, individuals need to be able to identify social computations that they feel are worth joining (thus, these incentives must be evident as part of the computation), and they must be able to align their chosen role with the computation as it runs. On the Open Internet, identification requires discovery mechanisms [5]. Alignment requires individuals to map their personal ontologies to those of the computation [6].

Reinforcement of the social group surrounding the social computation. This relies on the computation being executed in a way that spreads the computation and knits together the social group via further social properties of the computation. For our running example, one such property is that the probability of reward for any participant in the computation increases monotonically with the number of other participants that he/she recommends; this makes it in an individual's self interest to spread the participation (and spread the computation).

The three process above, taken with the earlier definitions, give us a working definition for a social computer.

(c) Social computer

A computer system that allows people to initiate social computations (via executable specifications) and adopt appropriate roles in social computations initiated by others, ensuring while doing so that social properties of viable computations are preserved. A general purpose social computer provides a domain-independent infrastructure for this purpose.

3. Programming the social computer: initiation

In conventional programming, software engineers design and write code. The vast majority of source code for current social computations is also produced this way. Facebook is a startlingly effective example of a traditionally engineered system with social involvement underpinned by a comparatively simple computational model of interaction. It is, however, only one of many social computation systems, with many more continuing to emerge. The difficulty with conventional programming for such systems is that it is impossible for software engineers to envisage the full range of social interactions that may be effective in a large and diverse population. As we have seen, however, social computations can be specified using compact process models in which the roles in the social interaction are modular (though interacting via message passing). These can be generalized as patterns that can be applied to numerous situations. In our running example, a simple generalization is as follows, where α and β correspond to *coordinator* and *applicant* in our example, and other functions and predicate names have been replaced by variables. This now looks similar to the sorts of programming patterns used in the structural synthesis of functional

and logic programs.

$$\alpha(P, L) ::$$
$$L = [] \; or$$
$$(\mathcal{F}_1(A, R_1, P_1) = P \leftarrow G(A) \Leftarrow R_1 \; then \; \alpha(P_1, L)) \; or$$
$$(\mathcal{F}_2(X, R_2, L_1) = L \leftarrow H(X) \Leftarrow R_2 \; then \; \alpha(P, L_1))$$
$$\beta(C) \; :: \; G(A) \Rightarrow C \; then \; \beta_1(C)$$
$$\beta_1(C) \; :: \; \begin{pmatrix} G(A) \Rightarrow C \leftarrow \mathcal{Z}_1(A) \; or \\ H(X) \Rightarrow C \leftarrow \mathcal{Z}_2(X) \end{pmatrix} \; then \; \beta_1(C)$$

Modularity now has an important part to play because it allows us to replace role definitions with other definitions (that may be more effective for different social groups), or even to replace the definitions with sequences of messages derived directly from observation in the physical world. For instance, we could replace the definition of β with any sequence that sends from a given individual a message, $G(A_1)$, followed by any combination of G and H, for example: $[G(A_1) \Rightarrow C, \; G(A_2) \Rightarrow C, \; H(X_1) \Rightarrow C, \ldots]$. This means that, for some social computations, the definitions of the roles may be left to the physical operation of the human/mechanical system playing that role, with the only embodiment of this in the computational world being the message sequence supplied by whatever physical devices are used.

Taking this a step further, we might use inductive (or other) methods to infer specifications like the one for β from the behaviours we observe in message passing (this is similar to ideas such as social signalling [7]) or, following the ideas in Harel [4], we might be able to build devices with the ability to infer some forms of role definition from user requirements expressed via the operation of the device itself.

4. Programming the social computer: adoption

In conventional programming, it is unusual to think of there being critical stages in the execution of a program at which roles in the computation are adopted by relevant people. Social computation, however, relies on adoption: people have to be motivated to join the computation. On the OpenKnowledge project, we invented a novel way of tackling this problem by treating specifications of interactions (like the ones for our running example in §2) as knowledge to be shared. These can then be published and discovered; enacted to provide coordination; and analysed for their effectiveness, with the results of this analysis influencing human choices in discovery. In order to make as few assumptions as possible about the nature of programs being coordinated, we adopted a peer-to-peer architecture (see Kotoulas & Siebes [8] for a view of this in terms of the peer-to-peer architecture used) in which each peer can, potentially, provide the same general functionality in terms of its ability to discover and coordinate interactions. This system of knowledge sharing is, however, applicable to other forms of networked architecture, for example, versions of the OpenKnowledge system were used in differing architectures and application contexts in proteomics [9], emergency response simulation [10], service trading [11] and healthcare protocol enactment [12]. Figure 2 illustrates the basic discovery–enactment–analysis process used in the original OpenKnowledge system.

First, each peer must download the kernel system (a lightweight, 18 MB, Java download from www.openk.org). This provides it with the facilities to perform the necessary coordination functions, plus automatic connection to a default peer-to-peer framework (currently based on Bamboo). Through the kernel, a person can publish executable specifications of the models of interaction required for specific social computations (with associated tags to assist discovery), and can also perform keyword searches for interactions (analogously to conventional Web page search, but in this case, via the peer network). This is assisted by a discovery service (playing a role analogous to a conventional Web search engine, although in this case, using the peer

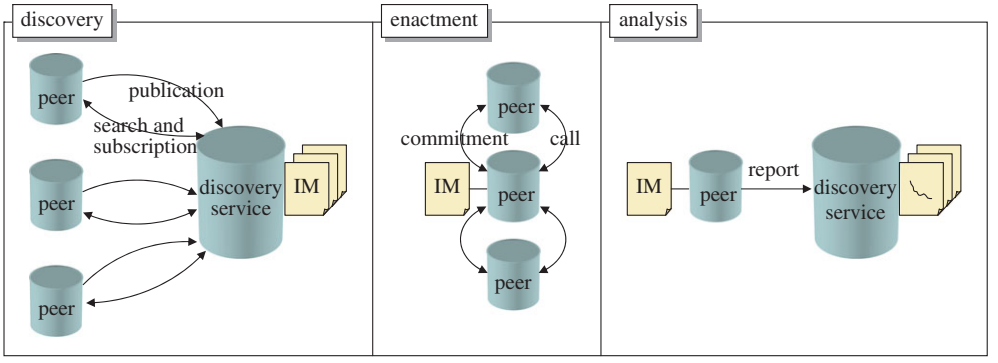

Figure 2. Basic discovery–enactment–analysis phases in the OpenKnowledge system, where 'IM' is an interaction model specifying a social computation. (Online version in colour.)

network). If the person finds an interaction that appears useful, then he or she can subscribe to an appropriate role in the interaction—for example, if the interaction coordinated a negotiation of goods for sale, then it might have a vendor role and a seller role so a person would choose one of those roles depending on his or her objective in the interaction. Once an interaction is fully subscribed (this can occur at any time for multiple interactions over the peer network), it proceeds to an enactment phase in which the subscribed peers commit to that instance of the interaction and coordination of that specific instance is allocated to a peer on the network (the choice of peer is irrelevant because it merely supplies the CPU cycles). The coordinating peer uses the interaction specification to control the sequence of message passing between peers and the structure of message content, via constraints delegated to appropriate peers in the interaction. Upon completion (or failure) of an interaction, the coordinating peer sends a report on the interaction to the discovery service. This can be used for an analysis of the performance of peers in the context of different types of interaction, giving a means of ranking them analogously to the ranking of Web pages by a search engine, as we describe in §5.

5. Programming the social computer: reinforcement

As we explained in §2, individual social computations that are well designed will possess social properties that allow them to reinforce themselves, and thus propagate through society. On a full-scale social computer, however, we have many social computations running, and data shared across these may further reinforce the development of communities of social computation. As an example of this sort of activity, we describe below a simple algorithm that operates in a style similar to the PageRank algorithm [13] used to provide rankings of Web pages. We call this the PeerRank algorithm because it ranks peers based on their interactions with other peers.

Like PageRank, PeerRank works by assigning a ranking to a peer at any given time as a function of its previous ranking modified by the rankings of each peer with which it has interacted (we refer below to such peers as supporting peers). The modification made by each of these peers is spread equally across the set of peers that it supports. In the algorithm definition below, we assume that we have available minimal information about interactions—only knowing for each interaction which peers were involved in it and whether or not it completed successfully. This is less information than we actually can obtain routinely (because the coordinating peer has access to all the structure of a completed interaction), but we make this simplification so as to explore how far we can get with the bare minimum of data. Data storage and lookup requirements are similar to that of established PageRank-based services on the Web (such as Google).

Given

— a dataset of initial rankings for each peer, each of the form $cr(P, T) = R$, and
— a dataset of records of successful or failed interactions, each of the form $im(T, I, P_I, CP)$,

where P is the identifier of a peer; T is either $\oplus$ (denoting the positive ranking) or $\ominus$ (denoting the negative ranking); R is the numerical magnitude of the (positive or negative) ranking; I is the identifier of the interaction model; P_{I} is the initiating peer for an interaction; and CP is the set of peers that subscribe to the interaction initiated by P_{I}.

The algorithm for calculating the current rank of a peer is as follows (where i is the empirically chosen number of iterations used to obtain stable ranking values, often less than 100):

- For i iterations:
  - — For each peer, P:
    - * Calculate $rank(P) = \langle R_p, R_n \rangle$
    - * assign $cr(P, \oplus) = R_p$
    - * assign $cr(P, \ominus) = R_n$

$rank(P)$ calculates the current rank for peer P, where d is an empirically chosen damping value used to tune the ranking system (a frequently used value in page ranking is 0.85),

$$rank(P) = \langle (1-d) + d \times r(s(\oplus, P), \oplus), (1-d) + d \times r(s(\ominus, P), \ominus) \rangle. \tag{5.1}$$

$s(T, P)$ gives the list of peers supporting peer P. If T is $\oplus$, then this is the set of positive support, or if T is $\ominus$, this is the set of negative support. Note that this is a list (which may contain duplicates) rather than a set because we are interested in the number of times each peer is supported,

$$s(T, P) = [Ps | a(P, T, Ps)]. \tag{5.2}$$

$a(P, T, Ps)$ is true when peer P is supported by peer Ps either positively (if T is $\oplus$) or negatively (if T is $\ominus$). Note that this may (intentionally) generate the same instance of Ps more than once. This allows us to count the number of times the same peer is supported,

$$a(P, T, Ps) \leftarrow im(T, I, P_i, CP) \wedge \begin{pmatrix} (P = P_i \ \wedge \ Ps \in CP) \\ \vee \\ (P \in CP) \ \wedge \ Ps = P_i) \end{pmatrix}. \tag{5.3}$$

$r(S, T)$ is the sum of the current ranks for all the peers in peer set, S, each peer's rank being divided by the number of peers it supports (so as to apportion the influence of the rank evenly across those supported peers). If T is $\oplus$, then this is the sum of ranks from positive associations, or if T is $\ominus$, this is the sum of ranks from negative associations,

$$r(S, T) = \sum_{P \in S}^{P} \frac{cr(P, T)}{|(u(T, P)|}. \tag{5.4}$$

$u(T, P)$ gives the list of peers supported by peer P. If T is $\oplus$, then this is the set of positive support, or if T is $\ominus$, this is the set of negative support. Note the symmetry between this and $s(T, P)$,

$$u(T, P) = [Ps | a(Ps, T, P)]. \tag{5.5}$$

If we run the algorithm above on uniform populations of peers, all of which are similarly cooperative in the social computation in which they are involved, then (similar to PageRank for Web pages), we observe a power-law effect in ratings. Dominant peers emerge that acquire high ratings and distance themselves from the majority that retains lower levels of popularity. This encourages strong competition for popularity, which (arguably) may be valuable as a mechanism for stabilizing the social component of the social computer. The graph in figure 3 demonstrates this effect in a population of 32 peers participating many times in succession in the same social computation. Each line on the graph is the positive rating of a peer (measured on the y-axis) as it changes while the number of interactions increases (measured on the x-axis). A single peer achieves dominance with a rapid fall-off to others with lower rank and a 'tail' of low-ranking peers.

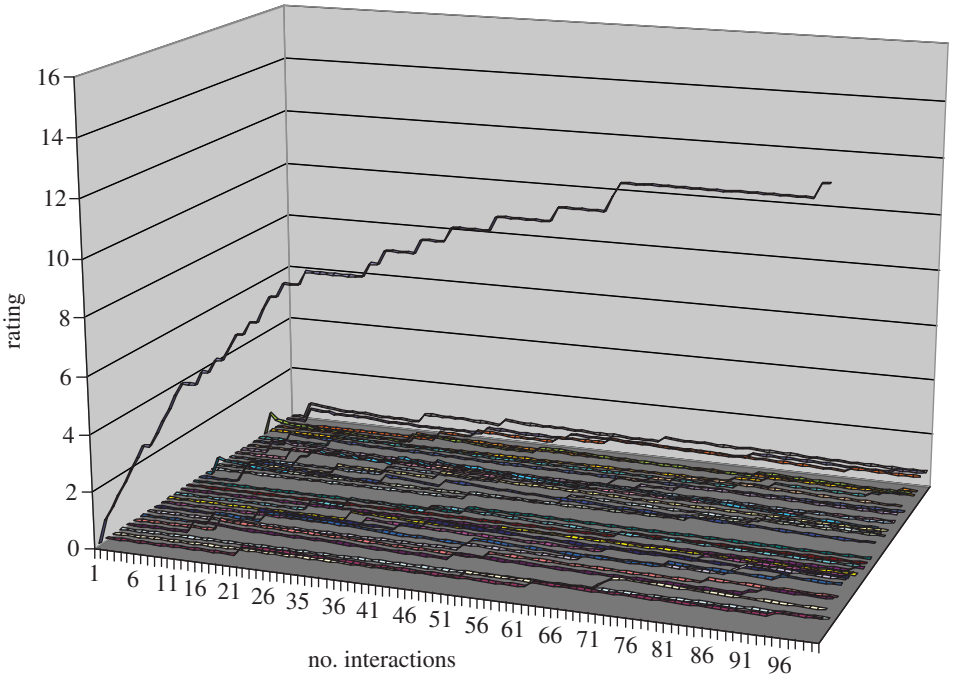

Figure 3. Rating changes with increasing peer group size in a perfectly compliant peer population. (Online version in colour.)

We also want the social computer to be sensitive to changes in peer behaviour, for example, if a peer that initially was compliant becomes less compliant in its behaviour, then we want its rating (eventually) to change. Figure 4 demonstrates this behaviour. The graph shows the change in positive and negative rating for each of four peers. In this simulation, instead of being always compliant, we make peers sometimes non-compliant. We make the likelihood of failure of a peer increase as the peer has more successful interactions, and decrease as the peer has more failed interactions, so that the likelihood of compliance of each peer alternates over time. We see the ratings in figure 4 change in response to this, as the positive and negative rating lines for each peer tend to twine across each other. As a peer is successful, its positive rating exceeds its negative rating, but this success makes it more likely to fail to be compliant (because we made each peer decide to behave less compliantly when it has been successful) so its negative rating then climbs and can exceed its positive rating, making it less likely to be selected by other peers. Overlaid on this twining effect is the basic power-law effect that we demonstrated in figure 3, so peer 4 has a dominant rating (both positive and negative because its popularity attracts many interactions including more that fail as well as more that succeed), followed by peer 3 and then (some way behind) by peers 1 and 2.

We have focused thus far on ranking peers, but the overall pattern of ratings over time can give us evidence of the effectiveness of a social computation in general. Figure 5 shows the ratings for 20 peers running 200 times a social computation that does not work effectively with the peer group. In this particular case, the aim of the social computation is to fix diary dates between peers, and the algorithm used to intersect the diaries makes unrealistic assumptions about the flexibility of individual's diaries. Dark coloured lines in figure 5 represent positive ratings, whereas light coloured lines represent negative ratings. However, the pattern overall is confused, with no dominant peer and negative ratings tending to exceed positive ones.

Figure 6 shows the ratings of the same peers with a social computation that has been designed to work more effectively with the peer group. The chaotic behaviour we saw in figure 5 is corrected. Three of the peers now dominate with positive ratings consistently exceeding their

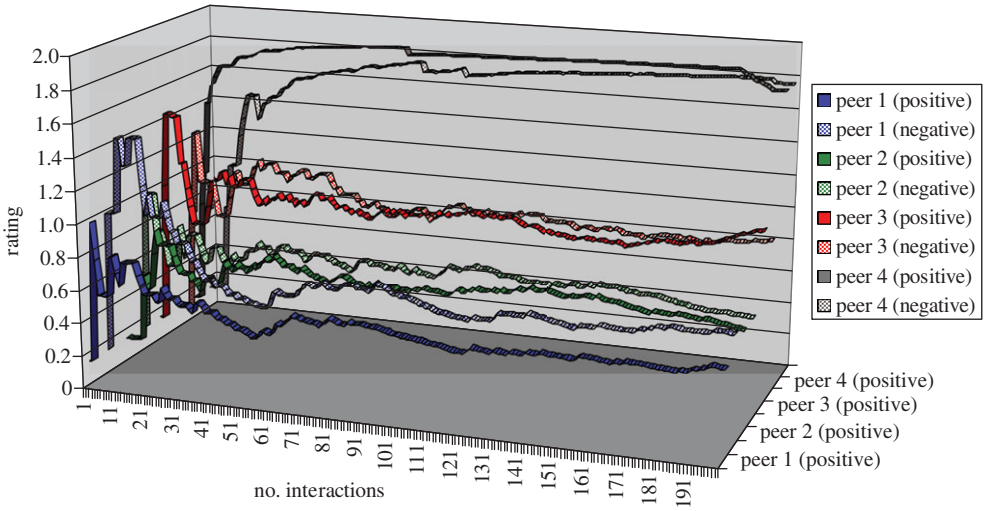

Figure 4. Rating differences with time-variant compliance. (Online version in colour.)

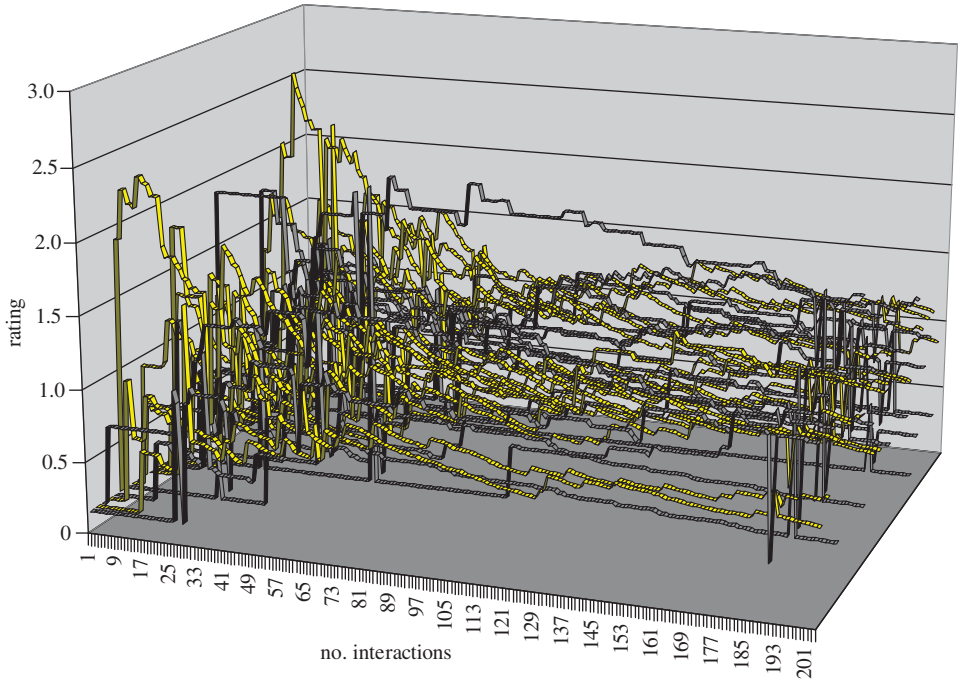

Figure 5. Ratings for a poorly crafted social computation. (Online version in colour.)

negative ratings, and the other peers form an orderly 'tail' of much lower ranked, but stable peer group members. As our example shows, a stable peer rank is dependent on choosing high-quality interactions appropriate to the task. This is (arguably) a beneficial feature because it encourages peers in pursuit of high rankings to be selective in their choice of social computations, which we would expect to force out low-grade computations. Poorly performing computations are easy to detect (they fail frequently), so a discovery service can easily supply this information to peers searching for appropriate interactions.

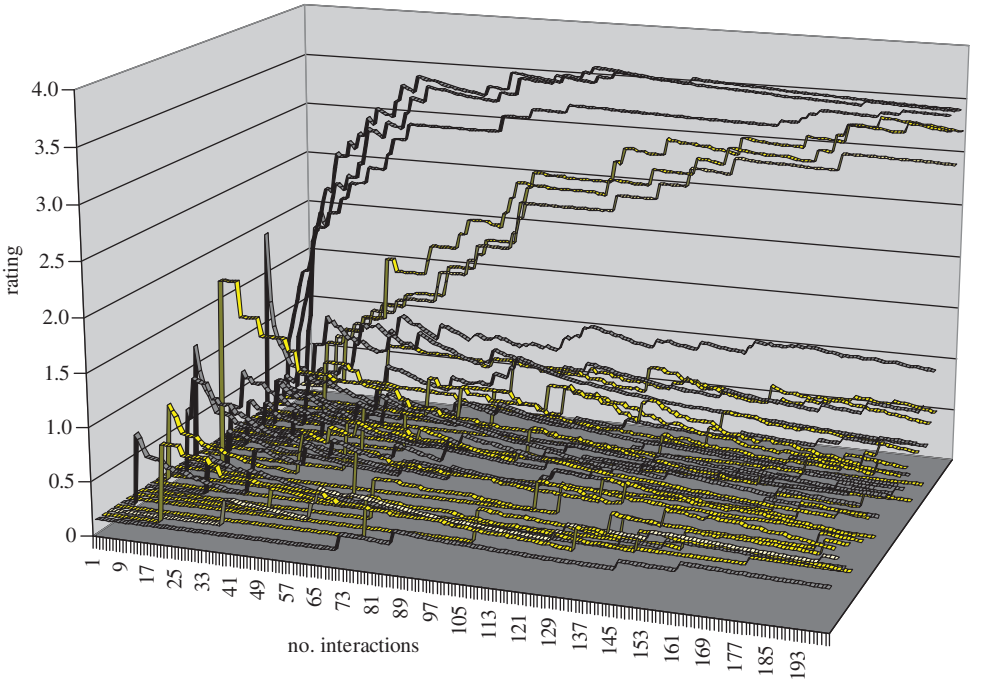

Figure 6. Ratings for an improved version of the social computation of figure 5. (Online version in colour.)

The method described above is simply a rating system that operates independently of the nature of the social interactions involved. It infers statistical information derived from the successes or failures of interactions, but it is insensitive to deeper information about the quality of knowledge shared; the commitments undertaken in sharing knowledge and the trust that builds from that. Trust applied to knowledge sharing differs from narrowly technical notions of dependability in computer networks (such as liveness) because, on top of these, we are concerned with the degree to which the meaning expressed by transmitted knowledge is sufficiently complete and consistent. Normally, it is difficult to assess this simply by scrutiny of the artefact of knowledge sharing—by looking at the things we are told. Instead, we have to consider the process by which knowledge is acquired and be able to assess the reliability of its sources. This is the reason why modelling of reputation has become an important issue in the multi-agent systems community, and why provenance is an important topic in data integration. Even in purely logical, consistent inference systems with perfect reputation and without asynchrony or differences in ontology, it may be impossible to judge our degree of trust in shared knowledge from different reasoners without inspection of the proofs performed by both reasoners [14]. In practice, this is impossible (because most practical reasoners are not purely logical and even those that are may not offer inspectible proofs). Process and provenance therefore are crucial, and for these, practical formal frameworks are emerging— for example, in the process models of Web service interaction languages and in permission/obligation languages for networked software components.

Knowledge sharing in open, peer-to-peer architectures differs from traditional knowledge engineering by dropping many of the assumptions upon which traditional knowledge engineers have relied. No aspect of coordination is centralized. There is no common engineering culture among their designers beyond the minimum necessary for baseline communication. There is no shared ontology (or ontology mapping) other than that established opportunistically through component interaction. No constraints are placed on components allowed to enter an interaction, other than those constraints imposed by the social norms of the interaction itself. It is not assumed that components will guarantee to maintain specific states of knowledge, so if, for example,

monotonicity assumptions are required for some coordinated task, then these must be enforced through the mechanism of coordination because they will not be enforced independently by the components nor by the communication substrate. A consequence of the flexibility of interaction between components is that trust must be assessed dynamically at run time because there will be no predetermined, static boundary within which to perform trust assessment.

In this more open environment, trust comes from numerous sources. One source is in the permissions and obligations of peers, often defined formally as policies governing the choices in behaviour of an individual system [15]. Various styles of description are used to implement this concept: logic-based (e.g. ASL [16], Rei's Prolog implementation, RDL) [17], object-oriented (e.g. Ponder [18], RuleML [19]), markup languages (e.g. Rei's RDF implementation, XACML [20], TPL [21]) and so on. Inference systems also exist for verifying whether peers confirm to protocols, based on theorem proving [22,23] or model checking [24,25]. It is even possible to combine both aspects, so that peers when given the opportunity to engage with new protocols can verify conformance against local policies [26]. Taken together, these approaches (plus the simpler statistically based methods described earlier) give a wide range of engineering options for managing reputation and trust in a social computer.

6. Conclusion

There has been a huge increase in the number of computer systems based on social computations, from highly targeted social efforts such as the DARPA Network Challenge (that we used as a running example in §2) through to mass social networking systems such as Facebook. These systems each have the ability to mobilize and coordinate social groups in different ways, and this is becoming so prevalent that it is legitimate to look for more generic models of computation that could support a wide variety of these sort of algorithms while building on their interaction. We have used the term 'social computer' to refer to this aspiration.

In §2, we described what a social computer would have to do. Three processes, at least, appear fundamental: the initiation of social computations (which we described in §3 as the synthesis of process specifications); the adoption of roles in these computations by individuals (which we described in §4 as sharing and discovery of these specifications); and the reinforcement of adopted specifications (which we examplified in §5 through a scheme of ranking of peers in the social networks created through these computations).

Returning to the 'programming the global computer' theme with which we began this paper, we have argued that the deep social embedding of computation allows us now to take a different view of what a 'program' is in this context. The elementary components of the programs we have described involve the harnessing of human effort and expertise, augmented by automated systems. Engineers of conventional social computation systems have demonstrated the effectiveness of such systems in circumstances where the incentive structure for participation is aligned with the population concerned. The architectures necessary to construct more complex programs incrementally from these sorts of components are now being developed. These use the infrastructures and specification methods familiar from traditional systems engineering (in this paper, we focused on peer-to-peer architectures and process specification), but they are used in unconventional ways (for example, in initiation, adoption and reinforcement) to extend computation across society. This makes us look at formal specification and knowledge representation in a new light.

The authors are grateful to members of the OpenKnowledge (www.openk.org) and Social Computer consortia (www.socialcomputer.eu) for many discussions related to the topics in this paper.

References

1. Kwiatkowska M, Milner R, Sassone V. 2004 Science for global, ubiquitous computing. *Bull. EATCS* **82**, 325–333.

2. Robertson D. 2004 Multi-agent coordination as distributed logic programming. In *Proc. 20th Int. Conf. on Logic Programming, Sant-Malo, France*. Lecture Notes in Computer Science, no. 3132. Berlin, Germany: Springer.
3. Robertson D *et al*. 2008 Models of interaction as a grounding for peer to peer knowledge sharing. In *Advances in web semantics*, vol. 1 (eds E Chang, T. Dillon, R. Meersman, K Sycara). Lecture Notes in Computer Science, no. 4891. Berlin, Germany: Springer.
4. Harel D. 2008 Can programming be liberated, period? *IEEE Comput.* **41**, 28–37. (doi:10.1109/MC.2008.10)
5. Besana P, Patkar D, Barker A, Robertson D, Glasspool D. 2009 Sharing choreographies in openknowledge: a novel approach to interoperability. *J. Softw.* **4**, 833–842. (doi:10.4304/jsw.4.8.833-842)
6. Giunchiglia F, McNeill F, Yatskevich M, Pane J, Besana P, Shvaiko P. 2008 Approximate, structure-preserving semantic matching. In *Proc. 7th Int. Conf. on Ontologies, Databases and Applications of Semantics, Monterrey, Mexico*. Berlin, Germany: Springer-Verlag.
7. Vinciarelli A, Pantic M, Bourlard H, Pentland A. 2008 Social signal processing: state-of-the-art and future perspectives of an emerging domain. In *Proc. 16th ACM Int. Conf. on Multimedia, Vancouver, Canada*, pp. 1061–1070. ACM Press.
8. Kotoulas S, Siebes R. 2007 Adaptive routing in structured peer-to-peer overlays. In *Proc. 3rd Int. IEEE Workshop on Collaborative Service-oriented P2P Information Systems, Paris, France*. IEEE Computer Society.
9. Quan X, Walton C, Gerloff D, Sharman J, Robertson D. 2006 Peer-to-peer experimentation in protein structure prediction: an architecture, experiment and initial results. In *Proc. Int. Workshop on Distributed, High-Performance and Grid Computing in Computational Biology, Eliat, Israel*. Berlin, Germany: Springer-Verlag.
10. Marchese M, Vaccari L, Trecarichi G, Osman N, McNeill F. 2008 Interaction models to support peer coordination in crisis management. In *Proc. 5th Int. Conf. on Information Systems for Crisis Response and Management, Washington, DC*.
11. Guo L, Darlington J, Fuchs B. 2009 Towards an open, self-adaptive and P2P based e-market infrastructure. In *Proc. IEEE Int. Conf. on e-Business Engineering, Macao, China*. IEEE Computer Society.
12. Fox J, Glasspool D, Patkar V, Austin M, Black E, South M, Robertson D, Vincent C. 2010 Delivering clinical decision support services: there is nothing as practical as a good theory. *J. Biomed. Inform.* **43**, 831–843. (doi:10.1016/j.jbi.2010.06.002)
13. Page L, Brin S, Motwani R, Winograd T. 1999 The PageRank citation ranking: bringing order to the Web. Technical Report 422, Stanford University Infolab, Stanford, CA.
14. Correa da Silva FS, Vasconcelos WW, Robertson DS, Melo ACV, Finger M, Agusti J. 2002 On the insufficiency of ontologies: problems in knowledge sharing and alternative solutions. *Knowl. Based Syst.* **15**, 147–167. (doi:10.1016/S0950-7051(01)00152-6)
15. Sloman M. 1994 Policy driven management for distributed systems. *J. Netw. Syst. Manage.* **2**, 333–360. (doi:10.1007/BF02283186)
16. Jajodia S, Samarati P, Subrahmanian VS. 1997 A logical language for expressing authorizations. In *Proc. 1997 IEEE Symp. on Security and Privacy* (*SP '97*), p. 31. Washington, DC: IEEE Computer Society.
17. Kagal L, Finin T, Joshi A. 2003 A policy language for a pervasive computing environment. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. Washington, DC: IEEE Computer Society.
18. Dulay N, Damianou N, Lupu E, Sloman M. 2002 A policy language for the management of distributed agents. In *Revised Papers and Invited Contributions from the Second International Workshop on Agent-Oriented Software Engineering II* (*AOSE '01*), pp. 84–100, London, UK: Springer.
19. Boley H. 2003 Object-oriented ruleml: user-level roles, uri-grounded clauses, and order-sorted terms. In *Proc. 2nd Int. Workshop on Rules and Rule Markup Languages for the Semantic Web, Florida, FL*, pp. 1–16. Lecture Notes in Computer Science, no. 2876. Berlin, Germany: Springer.
20. OASIS. 2003 Extensible Access Control Markup Language (XACML), v. 1.1. See http://www.oasis-open.org/committees/xacml/repository/cs-xacml-specification-1.1.pdf.
21. Herzberg A, Mass Y, Michaeli J, Ravid Y, Naor D. 2000 Access control meets public key infrastructure, or: assigning roles to strangers. In *Proc. 2000 IEEE Symp. on Security and Privacy* (*SP '00*), p. 2. Washington, DC: IEEE Computer Society.

22. Alberti M, Daolio D, Torroni P, Gavanelli M, Lamma E, Mello P. 2004 Specification and verification of agent interaction protocols in a logic-based system. In *Proc. 2004 ACM Symp. on Applied Computing* (*SAC '04*), pp. 72–78. New York, NY: ACM Press.
23. Poutakidis D, Padgham L, Winikoff M. 2002 Debugging multi-agent systems using design artifacts: the case of interaction protocols. In *Proc. 1st Int. Joint Conf. on Autonomous Agents and Multiagent Systems* (*AAMAS '02*), pp. 960–967. Bologna, Italy: ACM Press.
24. Raimondi F, Lomuscio A. 2004 Verification of multiagent systems via ordered binary decision diagrams: an algorithm and its implementation. In *Proc. 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems* (*AAMAS '04*), pp. 630–637. New York, NY: IEEE Computer Society.
25. Kacprzak M, Lomuscio A, Penczek W. 2004 Verification of multiagent systems via unbounded model checking. In *Proc. 3rd Int. Joint Conf. on Autonomous Agents and Multiagent Systems* (*AAMAS '04*), pp. 638–645. New York, NY: IEEE Computer Society.
26. Osman N, Robertson D. 2007 Dynamic verification of trust in distributed open systems. In *Proc. 20th Int. Joint Conf. on Artificial Intelligence, Hyderabad, India*. Morgan Kaufmann.