Formal specification and analysis of trusted communities

Jan-Philipp Steghöfer, Florian Nafz, Wolfgang Reif, Yvonne Bernard, Lukas Klejnowski, Jörg Hähner, Christian Müller-Schloer

Angaben zur Veröffentlichung / Publication details:

Steghöfer, Jan-Philipp, Florian Nafz, Wolfgang Reif, Yvonne Bernard, Lukas Klejnowski, Jörg Hähner, and Christian Müller-Schloer. 2010. "Formal specification and analysis of trusted communities." In *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop, 27-28 September 2010, Budapest, Hungary*, edited by Scott Alexander, Jonathan Smith, and Robert Laddaga, 190–95. Piscataway, NJ: IEEE. https://doi.org/10.1109/sasow.2010.39.

Formal Specification and Analysis of Trusted Communities

Jan-Philipp Steghöfer, Florian Nafz, Wolfgang Reif
Institute for Software & Systems Engineering
Augsburg University
Augsburg, Germany
Email: {steghoefer, nafz, reif}
@informatik.uni-augsburg.de

Yvonne Bernard, Lukas Klejnowski,
Jörg Hähner, Christian Müller-Schloer
Institut für Systems Engineering – SRA
Leibniz Universität Hannover,
Hannover, Germany
Email: {bernard, klejnowski, haehner, cms}
@sra.uni-hannover.de

***Abstract*—Trusted Communities are a way to improve the performance of self-organising agent systems by limiting the interactions to trustworthy partners and excluding agents that behaved uncooperatively. We describe the mechanism in an abstract way and identify three central decisions each agent has to make if it supports Trusted Communities. Based on a formal specification of the agent behaviour in an instantiation of the mechanism for Desktop Grid Systems, we identify one of the system goals and show by formal verification that this goal is reached. Additionally, we provide certain requirements for the decision procedures that become evident during the analysis process.**

***Keywords*-formal methods; agents; trust**

I. Introduction

In open self-organising software systems, autonomous agents cooperate to achieve a common goal. However, it can not be assumed that all agents behave in a benevolent fashion. Instead, such systems require measures to identify and punish selfish or outright malevolent agents to protect the system from them. Research in Organic Computing deals with open self-organising systems and is now starting to investigate the implications that trust considerations have on such systems [1].

One of the mechanisms that deal with the untrustworthiness of some agents in an open environment is Trusted Communities [2] (TCs). A TC is an association of agents that mutually trust each other due to previous experiences from direct interactions and knowledge about the other agents' reputation. If an agent behaved in a non-desired fashion in the past, the TC is able to detect the adverse effect this has on the system and excludes the agent from further interactions. This is achieved solely by the use of local rules and without an explicit notion of the TC.

The crucial elements for the success and the correct behaviour of the mechanism are the functions that evaluate with which other agent to interact, if an interaction with another agent should be pursued, and, in case another agent rejected an interaction, if this rejection influences the other agent's trustworthiness. This paper, starting from an abstract description of Trusted Communities and an instantiation for Desktop Grid Systems, will show how a formal specification and analysis of TCs informs the design of these functions in a concrete system. Formal specification requires a concrete and detailed specification which reveals problems and hidden properties that would be concealed in a system that is only implemented. The ultimate goal will be to identify which factors have to be regarded in order to achieve the desired system goals with the help of these low level decision procedures. The formal model and the proof sketch given in the following are a first step in this direction.

The paper is organised as follows: Section II introduces the notion of Trusted Communities and outlines their function in an abstract and informal fashion. Section III outlines the case study of Desktop Grid Systems to which Trusted Communities have been applied successfully. The formal model of the agents and their behaviour is described in Section IV while the system's goals are outlined in Section V. The analysis and its results are given in Section VI before the paper concludes with an outlook to future work.

II. Implicit Trusted Communities

An Implicit Trusted Community is a dynamic agent organisation that is build in a bottom-up fashion by agents that have developed mutual trust relations during a line of interactions. Agents that do not cooperate, i.e. with whom interactions are unsatisfactory, are excluded from the Trusted Community. It is implicit because there is no explicit notion of the community in any of the agents.

The way cooperation is defined is, however, highly dependent on the application the agent system is tailored to. This becomes evident if a general definition of the term is applied: in a general sense, cooperation means that two or more agents work towards *a common goal*. The common goal has to be defined for each application or even for different agent groups in an open competitive system.

If an agent only interacts with agents it trusts, the results of the interactions are more likely to be beneficial as if it would interact with untrusted agents as well. An additional benefit of Trusted Communities is that with growing trust, certain security precautions, like, e.g. additional protocol steps, can be omitted or more information could be revealed to ease cooperation when trusting agents interact. Therefore,

individual agents profit from being part of the community and the entire system profits as redundant and over-cautious interactions decrease. The formation of Trusted Communities can even have a normative benefit: as uncooperative agents will see a decline in their interactions' benefits, the mechanism promotes cooperation.

While a reputation mechanism is a prerequisite that has to be taken care of by the system, the individual agent's behaviour is an interesting facet and open to analysis. How does an agent have to implement its decisions in order to achieve a desirable outcome? After introducing a concrete implementation of Trusted Communities for Desktop Grid Systems in the next section, this paper will show the possibilities formal methods provide to deal with this question.

III. Trusted Communities in Desktop Grid Systems

In this paper, the system under consideration is a Desktop Grid System in which computing resources are shared to facilitate and accelerate the computation of algorithmically complex tasks. Agents can act as *workers* that offer their computational resources to other agents and accept *work units* that contain a self-contained task. The task is computed and returned to the *submitter*. A submitter splits a *job* that has been generated by a user application into smaller work units and distributes those work units to workers according to a specific strategy. It then waits for the workers to return the work units and combines the results to achieve the overall result for the entire job. An agent can act both as a worker and as a submitter at the same time.

A. Benefits and Adapted Definition

Trusted Communities are beneficial in Desktop Grid Systems as they allow a submitter to identify those agents that reliably return completed work units and use them for future computations. Agents that refuse to accept work units or do not deliver completed work units in time are recognized and will eventually be excluded from the community. A reputation mechanism ensures that other reliable agents can benefit from the knowledge individual submitters gathered about other workers. Overall, this mechanism forms an ensemble of agents that trust each other and that favour each other when work units need to be distributed while excluding untrustworthy agents. This yields performance benefits and decreases overall system load as work units do not need to be resubmitted or processed multiple times in order to make sure that jobs will be completed.

In the Desktop Grid scenario, the members of the community are those agents that have accepted and actually processed work units when requested by submitters and rejected requests only under certain conditions. Uncooperative agents, on the other hand, are those that repeatedly rejected requests without just cause or accepted work units but never returned the processed data.

B. Important Agent Decisions

The description above already hints at some of the choices an agent has to make if it is part of a system in which Trusted Communities play a role. To make the requirements more concrete, an agent has to make the following decisions:

- **Decision 1**: Which peer to give a work unit? This decision depends on the trustworthiness of the peer, as well as on other factors such as its current load. Different strategies that incorporate different parameters in the decision are possible. [2] outlines two strategies that use different weightings for three parameters: trustworthiness, work queue length, and resource specification.
- **Decision 2**: Should I accept a work unit? There are several factors that can play a role in the decision of a worker to accept a work unit or not. The trustworthiness of the submitter, the current work load, resource restrictions or even reasoning about the development of one's own trust level can come into play. Different types of agents will react differently to a request.
- **Decision 3**: Was a rejection justified? When a worker rejected the processing of a work unit, it might have had a good reason to do so. If, e.g. the worker is currently used to capacity, it might be beneficial for the submitter that the worker actually rejected the processing because the work unit can now be given to a less busy worker.

These decisions are also the ones which will be the focus of analysis with regard to Trusted Communities. In the following, first steps towards this direction are made. To set the stage, however, a formal model of the agents and their environment will be given first.

IV. Formal Model of Agents in Desktop Grid Systems

The model of the agents, their communication and their environment is formalized in ITL$^+$ [3], a variant of Interval Time Logic (ITL) that allows to explicitly include steps performed by the environment. ITL is in turn based on Linear Time Logic. A run (trace) of a system is a sequence of states. We distinguish system transitions and environment transitions. For the latter, an additional primed intermediate state is introduced. A run is thus an alternating sequence of unprimed and primed states, respectively system and environment transition. The formula $x' = x + 1$ denotes that the value of x in the primed state is the current value of x plus one. As a convention for this paper, only variables that appear in a formula are changed.

Besides the standard predicate logic operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (equivalence) and quantifiers $\forall, \exists$ which are evaluated over a single state, the logic contains the temporal operators $\Box\ \varphi$ (φ holds now and *always* in the future), $\diamond\ \varphi$ (φ holds now or *eventually* in the future) and $\circ\ \varphi$ (φ holds in the next state).

The formulas that describe the agents also refer to data types and variables, either global or local to the agents. Each

agent ag has a queue Q_{ag} that stores incoming messages. A message msg is a triple (snd, per, dat) consisting of the sender, a performative, i.e. an identifier of the message's purpose, and a data payload, e.g. a work unit. The elements of the triple are accessed by msg_{snd}, msg_{per}, and msg_{dat} respectively. Further, an agent has operations enq and deq to enqueue and dequeue a message to or from a queue.

A submitter relies on the presence of some data structures to store and retrieve information. The hashes e_g and e_b hold the number of either good or bad experiences any of the agents has had with any other agent. Another hash AWU stores the number of work units that any agent currently accepted to work on. The hash value for a particular agent ag as a key is selected by $e_g[ag]$. The Timeout Table TT : WorkUnit × Agent × Int is stored locally by each submitting agent and holds the agent a work unit was given to as well as a timer for this work unit.

Next, trustworthiness as used in the system will be defined. Then, work unit submission will be regarded, before the behaviour of different worker "characters" is modeled.

A. Trustworthiness

The trustworthiness of an agent ag is expressed by its *aggregated Trust Level* ($aggTL$). To calculate this value, a local fitness function $f(ag)$ is used. The fitness value is calculated by a weighted summation of the numbers of good (e_g) and bad (e_b) experiences the agents in the system have had with the worker ag.

$$f(ag) = \alpha \cdot e_g[ag] - (1 - \alpha) \cdot e_b[ag] \tag{1}$$

Usually, the weighting factor α reflects that bad experiences are perceived as having a higher impact than good ones. Based on this fitness function, the agent discretises the aggregated Trust Level of ag as follows:

$$aggTL(ag) = \begin{cases} 1, & f(ag) > 0 \\ 0, & f(ag) = 0 \\ -1, & f(ag) < 0 \end{cases} \tag{2}$$

While in the following, we mainly rely on $f(ag)$ (and the two functions are interchangeable for our purposes), $aggTL$ is a basis for future extensions, e.g. if direct experience and reputation are to be regarded separately or other sources of trust information are introduced.

B. Work Unit Submission

Whenever a work unit wu is available for dispatch (expressed by a predicate $wuAv$) the submitter snd will send the work unit to a worker to by putting it into to's message queue. The performative used in this case is req (request).

$$\begin{aligned} wuSubmit :\Leftrightarrow\ & wuAv \rightarrow \\ & to' = selto() \wedge \circ enq((snd, \texttt{req}, wu), Q_{to'}) \end{aligned} \tag{3}$$

The crucial part here is the selection of the agent to, constituting Decision 1 as defined in Section III-B. to will receive the work unit and will be asked to process it. Therefore, a function $selto : Agents \rightarrow Agent$ specifying the selection strategy needs to be in place.

If the worker acknowledges the receipt of the work unit and that it will start working on it by sending an accept message (performative: acc), the submitter increases the number of work units the worker is currently processing and puts the work unit and the worker into the Timeout Table TT. ε denotes a predefined value for the initial timer.

$$\begin{aligned} wuAcc(msg) :\Leftrightarrow\ & msg_{per} = \texttt{acc} \rightarrow \\ & (AWU'[msg_{snd}] = AWU[msg_{snd}] + 1 \\ & \wedge\ TT'[msg_{dat}] = (msg_{snd}, \varepsilon)) \end{aligned} \tag{4}$$

If a work unit has been rejected, the rejection needs to be evaluated. Agents can reject a work unit for a variety of reasons, some of which might be in the interest of the system while others are purely in the interest of the rejecting agent (Decision 3). The VALID predicate evaluates the rejection and becomes true when the rejection was justified, i.e. if it indeed was in the interest of the system. Otherwise, a bad experience is filed and the work unit is removed from the Timeout Table.

$$\begin{aligned} wuRej(msg) :\Leftrightarrow\ & msg_{per} = \texttt{rej} \wedge \neg VALID \rightarrow \\ & ((e'_b[msg_{snd}] = e_b[msg_{snd}] + 1) \\ & \wedge TT' = rem(msg_{dat}, TT)) \end{aligned} \tag{5}$$

After an agent accepted a work unit for processing, the submitter has to wait until the agent either returns the completed work unit or a timeout occurs. In the latter case, a bad experience is filed for the sender. In all cases, the number of accepted work units has to be decreased.

In order to detect a timeout, the submitter will have to evaluate the Timeout Table in every time step. If the timer becomes 0, a timeout has occurred, a bad experience has to be filed and the number of accepted work units for the agent has to be reduced. The operation rem removes the entry from the table.

$$\begin{aligned} evTT :\Leftrightarrow\ & \forall (wu, ag, timer) \in TT : timer = 0 \rightarrow \\ & (\ TT' = rem(wu, TT) \wedge e'_b[ag] = e_b[ag] + 1 \\ & \wedge AWU'[ag] = AWU[ag] - 1) \end{aligned} \tag{6}$$

Only if the agent completed the work unit in a timely fashion, a good experience for the agent is added. At the same time, the number of accepted work units is decreased and the timeout for the work unit is removed:

$$\begin{aligned} wuDone(msg) :\Leftrightarrow\ & \\ & msg_{per} = \texttt{done} \rightarrow (e'_g[msg_{snd}] = e_g[msg_{snd}] + 1 \\ & \wedge AWU'[msg_{snd}] = AWU[msg_{snd}] - 1 \\ & \wedge TT' = rem(msg_{dat}, TT)) \end{aligned} \tag{7}$$

In every time step, the submitter updates the entries in its Timeout Table by decreasing the respective values by one.

$$upTT :\Leftrightarrow \forall(wu, ag, timer) \in TT : \quad TT'[wu] = (ag, timer - 1) \tag{8}$$

The submitter is a combination of the above formulas. In each time step, it checks for new work units that are available for dispatch, updates the Timeout Table and evaluates it. If there is no message to process and the queue is not empty, it dequeues the message and stores it in msg for processing in the next step. If, however, there is a message, it checks the message type and reacts accordingly. Also, the message variable is emptied by assigning $\perp$ to it to indicate that it has been processed.

$$\begin{aligned} Submitter :\Leftrightarrow\ & wuSubmit \wedge upTT \wedge evTT \wedge \\ & (msg = \perp \wedge Q_{Sub} \neq \emptyset \rightarrow msg' = deq() \\ & \vee msg \neq \perp \rightarrow msg' = \perp \wedge wuRej(msg) \\ & \wedge wuAcc(msg) \wedge wuDone(msg)) \end{aligned} \tag{9}$$

C. Worker Behaviour

All workers base their decision to accept a work unit on the trustworthiness of the submitter. If a submitter's aggregated trust level is negative, any work unit submitted is rejected. Other than that, different agent characters behave differently when they are asked to process work units.

Similar to the submitter, a worker can either dequeue a message in each time step or process it. When a message was processed, the variable holding the message is emptied. Additionally, a worker can also do nothing (indicated by $idle$) which is used to model delays in the system or in the message delivery system. In this case, none of the variables of the worker or its environment is changed. Whenever a submitter puts a work unit into a worker's queue, the agent will dequeue the message and react to it. The reaction will be based on Decision 2, the choice whether or not to accept a work unit. Depending on the agent character, different reactions are possible: the agent can either reject the work unit which is performed by enqueueing a message with the `rej` performative in the submitter's message queue or accept it, indicated by an `acc` message. Some agents require additional deliberation, e.g. about their current work load or the sender of the work unit. If the work unit was accepted, the agent will either return it eventually, as indicated by a `done` message which contains the processed work unit as its payload, or it will not get processed and thus will never be returned to the submitter (i.e. the worker performs a `null` action). This will cause a timeout on the submitter's side.

We distinguish four agent characters, where the first three implement Decision 2 without special considerations:

An **altruistic agent** will process all work units that are submitted by a trustworthy agent. If a work unit was accepted, it will eventually be processed and returned to the submitter.

$$\begin{aligned} AL :\Leftrightarrow\ & idle \\ & \vee msg = \perp \wedge Q_{AL} \neq \emptyset \rightarrow msg' = deq() \\ & \vee (msg \neq \perp \wedge msg_{per} = \texttt{req} \rightarrow \\ & \quad ((aggTL(msg_{snd}) \geq 0 \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{acc}, msg_{dat}) \\ & \quad\quad \wedge (\Diamond enq(msg_{snd}, \texttt{done}, msg_{dat}))) \\ & \quad \vee ((aggTL(msg_{snd}) < 0 \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{rej}, msg_{dat})) \wedge msg' = \perp)) \end{aligned} \tag{10}$$

A **free rider** rejects all work unit processing requests, regardless of the submitter's trust level or other properties.

$$\begin{aligned} FR :\Leftrightarrow\ & idle \\ & \vee msg = \perp \wedge Q_{FR} \neq \emptyset \rightarrow msg' = deq() \\ & \vee msg \neq \perp \wedge msg_{per} = \texttt{req} \rightarrow \\ & \quad enq(msg_{snd}, \texttt{rej}, msg_{dat}) \wedge msg' = \perp \end{aligned} \tag{11}$$

An **egoistic agent** accepts work unit requests but might cancel some of the work units. In the Desktop Grid scenario, egoistic behaviour represents the activation of user-defined constraints on resource usage (e.g. the user needs the system resources for other purposes). Also, we use egoism to model malevolent behaviour.

$$\begin{aligned} EG :\Leftrightarrow\ & idle \\ & \vee msg = \perp \wedge Q_{EG} \neq \emptyset \rightarrow msg' = deq() \\ & \vee (msg \neq \perp \wedge msg_{per} = \texttt{req} \rightarrow \\ & \quad (aggTL(msg_{snd}) \geq 0 \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{acc}, msg_{dat}) \\ & \quad\quad \wedge (\Diamond enq(msg_{snd}, \texttt{done}, msg_{dat}) \vee \texttt{null})) \\ & \quad \vee ((aggTL(msg_{snd}) < 0 \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{rej}, msg_{dat})) \wedge msg' = \perp)) \end{aligned} \tag{12}$$

A **rational agent** accepts work units from submitters that are trustworthy, as long as certain conditions are met. These can be of different nature, e.g. how many work units it already processed in comparison to the number of work units it submitted or how much computing resources are currently available. This more complicated variant of Decision 2 is included in the formula with the abstract predicate BUSY.

$$\begin{aligned} RA :\Leftrightarrow\ & idle \\ & \vee msg = \perp \wedge Q_{RA} \neq \emptyset \rightarrow msg' = deq() \\ & \vee (msg \neq \perp \wedge msg_{per} = \texttt{req} \rightarrow \\ & \quad (aggTL(msg_{snd}) \geq 0 \wedge \neg BUSY \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{acc}, msg_{dat}) \\ & \quad\quad \wedge (\Diamond enq(msg_{snd}, (\texttt{done}, msg_{dat}))) \\ & \quad \vee aggTL(msg_{snd}) < 0 \vee BUSY \rightarrow \\ & \quad\quad enq(msg_{snd}, \texttt{rej}, msg_{dat}) \wedge msg' = \perp) \end{aligned} \tag{13}$$

In addition, there are predicates AL(ag), FR(ag), EG(ag), and RA(ag) for each character that evaluate to `true` if the agent in question adheres to the respective specification.

V. System Goals

The key beneficial property of Trusted Communities is that they exclude selfish or free riding agents. This can be expressed in terms of the fitness function f (1).

$$FR(ag) \rightarrow \Box(f'(ag) \leq f(ag) \wedge \Diamond f'(ag) < f(ag)) \quad (14)$$

It is valid to assume that a submitter will always pick an agent with a high fitness value to submit work units to. After all, a high fitness value shows that there have been beneficial interactions in the past. (14) means that the fitness value of a free rider decreases monotonically, i.e. only bad experiences are registered for it. Whenever an interaction with it occurred, the fitness will decrease and will eventually be low enough so it does not get picked as a worker any more. If we assume a system in which not all agents are free riders, this behaviour will effectively lead to an exclusion of free riders. As long as there are other agents available to process the work units, their fitness will be higher and they will be chosen for processing.

In principle, the same argument is valid for egoistic agents. However, as an egoist sometimes delivers results, its trust value is not monotonically decreasing. In some cases, when cooperative agents are rare, free riders will still be asked to process work units. This allows them to change their behaviour and adopt a more altruistic stance in order to return to the Trusted Community.

VI. Analysis

Sections IV and V provided the formal foundation for different kinds of analyses. In [4], different properties, both local to an agent and global to a system, have been shown by logical deduction in Temporal Belief Logic. Others are happy with a specification which in itself yields benefits with regard to eliminating ambiguity and ensuring conciseness. Z is often used in such cases and complex agent architectures like BDI can be regarded [5]. The language is also popular when systems have to be engineered and formal analysis is used to determine how a global behaviour has to be broken down into local agent behaviours [6].

An approach inspired by [6] is pursued in the following. The aim is to show that the system reaches its goal to exclude free riders. In order to do this, however, assumptions about the parts that have been left open in the descriptions above (like the $selto$-function and the VALID predicate) have to be made explicit. This does not necessarily mean that a concrete instantiation has to be given but that certain properties will be required in the proof which are equivalent to the requirements the parts have to adhere to.

In the following, a very short sketch for the proof of (14) will be given, to show how this formal model can be utilized for formal verification and to give a feeling for what such a proof looks like. The property to prove is that the specification of a Free Rider as given in (11) and the specification of an environment consisting of other agents and agents that act as submitters will lead to a monotonically decreasing fitness of free riders. $Init$ denotes the initial system in which all queues, TT, and AWU are empty.

$$\texttt{FR} \wedge Env \wedge Init \Rightarrow (14) \quad (15)$$

The environment Env in this case consists of many different agents, each of which can be of type FR, AL, EG, or RA. The only relevant element of the environment, however, is a submitter that creates new work units ($wuAv$) and submits them to the free rider under consideration. The proof obligation can therefore be rewritten as

$$\texttt{FR} \wedge Submitter \wedge \Box\Diamond wuAv \wedge Init \Rightarrow \quad (16)$$
$$\Box(f'(FR_{id}) \leq f(FR_{id}) \wedge \Diamond f'(FR_{id}) < f(FR_{id}))$$

A. Proof Sketch

The proof strategy uses symbolic execution to perform a stepwise evaluation of a proof obligation. As for infinite traces executing all steps may not be possible, we further use an induction technique for parallel programs. In principle, this technique constitutes noetherian induction over the number of steps necessary to reach the final state and is similar to Dynamic Logic induction over the number of cycles of a loop. For more technical details about the calculus, see [7].

The symbolic execution of the Free Rider results in one case distinction for every disjunction during the system step and the subsequent environment step. In the following, space restrictions limit us to an informal description of the most interesting cases generated by symbolic execution. Nevertheless, we hope to clarify the structure of the proof.

The execution of the first step results in two proof obligations. If $wuAv$ was true, the submitter sent a request to the Free Rider (case A1)[1]. Here we assume that $selto$ at least selects the Free Rider from time to time. If $wuAv$ was false (case A2), no work unit was available and the submitter just updated TT. In both cases (14) still holds as no update to the fitness was done.

Advancing the next step for case A2 results in the same proof obligations as in the beginning and premises can be closed by induction or contracted to the first case. Case A1, however, yields four new premises.

- B1 Free Rider dequeued the first message and $wuAV = false$. Unchanged fitness value.
- B2 Free Rider dequeued the first message and $wuAV = true$. Unchanged fitness value.
- B3 Free Rider idles and $wuAV = false$. Therefore, the first package is still the only one in the queue. Submitter leaves the fitness value unchanged.

[1]We omit the case that the request was sent to another agent as it has no impact on the fitness. Of course, this case has to be proven as well.

B4 Free Rider idles, $wuAV = true$, and submitter has sent another package. There are now two messages in the queue. Unchanged fitness value.

We only consider the most important cases B1 and B3. The others can be shown with an analogous argumentation.

Premise B1: Executing the next step leads to the same case distinction as in case A2 with the addition of a case in which the Free Rider enqueued a reject message in the submitter's queue. In this case, the submitter picks it up in the same time step. The VALID predicate in (5) then evaluates to `false`, indicating that the Free Rider should not have rejected the work unit. The submitter thus decreases the fitness of the free rider. With an inductive argument and using $\Box\Diamond wuAV$ – stating that always eventually a work unit is available again – the latter part of the system goal ($\Diamond f'(FR_{id}) < f(FR_{id})$) can be proven.

Premise B3: In case B3, the system is further executed and the Free Rider idles until the submitter decreases $TT[FR_{id}]$ to 0. The submitter will then register a bad experience with the agent in $evTo$, constituting a true decrease of the Free Riders' fitness. The same arguments as in premise B1 can then be used to close this open premise.

All other premises which are opened during the proof can be closed the same way or contracted to the other cases. □

B. Results and further considerations

During the proof, when steps were advanced, an interesting condition became evident: after a timeout caused the submitter to register a bad experience, the free rider could still process the message and send a reject to the submitter. Without additional effort, this leads to the submitter registering another bad experience for the free rider for the same interaction. While the system goal does not explicitly forbid this (after all, the fitness value of the agent is still decreasing monotonically), this behaviour is not desired. Each interaction should influence the experiences exactly once. This is a good example of how formal analysis reveals properties of the system that go unnoticed in simulations and how additional system goals arise from such analysis. Especially problems that occur in concurrent execution of the different agents (interleaving) are very hard to grasp intuitively and are much better revealed with formal methods.

A good way to prevent the submitter to register two bad experiences for such an interaction is to have VALID check for such a condition. If a timeout already occurred, a subsequent reject would be justified and VALID would evaluate to `true`. The bad experience would not be counted.

One of the assumptions that has to be in place for $selto$ is that it will always select the Free Rider again. As long as it keeps sending a work unit to the free rider from time to time, the verified property is valid. If the system goal is the only one we want to have, every implementation of $selto$ with respect to that property is valid. The AWU table, e.g., is a possibility to distinguish agents in $selto$ and VALID.

VII. Conclusion

In this paper, we gave an abstract description of Trusted Communities, a mechanism to enhance system performance by encouraging cooperative behaviour and identifying and excluding uncooperative agents. The description was then instantiated for Desktop Grid Systems and a formal specification for Trusted Communities in these systems was given. Based on that and a formalisation of the main system goal, an analysis was performed that showed that the system reaches the goal and also revealed a previously unnoted flaw in the specification.

Future work will include analysis of other agent types and which goal the system is trying to achieve with regard to their behaviour. Also, the implementation of an actual free rider can be checked against the specification presented here. This will enable us to verify that the implementation also exhibits the property we showed above. Another interesting aspect that can be analysed is the relationship between the predicates BUSY and VALID. A rational agent should not be punished for a rejection if it actually was busy and if it was beneficial for the system that it rejected the work unit.

Acknowledgment

This research is partly sponsored by the research unit "OC-Trust" (FOR 1085) of the German research foundation (DFG).

References

[1] J.-P. Steghöfer, R. Kiefhaber, K. Leichtenstern, Y. Bernard, L. Klejnowski, W. Reif, T. Ungerer, J. Hähner, and C. Müller-Schloer, "Trustworthy Organic Computing Systems – Challenges and Perspectives," in *Autonomic and Trusted Computing*. Springer, 2010.

[2] Y. Bernard, L. Klejnowski, J. Hähner, and C. Müller-Schloer, "Towards trust in desktop grid systems," in *Cluster Computing and the Grid, IEEE International Symposium on*. Los Alamitos, CA: IEEE Computer Society, 2010, pp. 637–642.

[3] S. Bäumler, M. Balser, F. Nafz, W. Reif, and G. Schellhorn, "Interactive verification of concurrent systems using symbolic execution," *European Journal on Artificial Interlligence (AI Communication)*, vol. 23, no. 2-3, pp. 285–307, 2010.

[4] M. Fisher and M. Wooldridge, "On the formal specification and verification of multi-agent systems," *Int. Journal of Cooperative Information Systems*, vol. 6, no. 1, pp. 37–66, 1997.

[5] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, "A formal specification of dMARS," *Intelligent Agents IV Agent Theories, Architectures, and Languages*, pp. 155–176, 1998.

[6] G. Smith and J. Sanders, "Formal development of self-organising systems," in *Autonomic and Trusted Computing*. Springer, 2009, pp. 90–104.

[7] M. Balser, "Verifying concurrent system with symbolic execution – temporal reasoning is symbolic execution with a little induction," Ph.D. dissertation, University of Augsburg, Augsburg, Germany, 2005.