Initial Steps Towards Run-Time Support for Norm-Governed Systems

Visara Urovi[1], Stefano Bromuri[1], Kostas Stathis[1], and Alexander Artikis[2]

[1] Department of Computer Science,
Royal Holloway, University of London, UK
{visara,stefano,kostas}@cs.rhul.ac.uk
[2] Institute of Informatics & Telecommunications,
NCSR "Demokritos", Greece
a.artikis@iit.demokritos.gr

Abstract. We present a knowledge representation framework with an associated run-time support infrastructure that is able to compute, for the benefit of the members of a norm-governed multi-agent system, the physically possible and permitted actions at each time, as well as sanctions that should be applied to violations of prohibitions. To offer the envisioned run-time support we use an Event Calculus dialect for efficient temporal reasoning. Both the knowledge representation framework and its associated infrastructure are highly configurable in the sense that they can be appropriately distributed in order to support real-time responses to agent requests. To exemplify the ideas, we apply the infrastructure on a benchmark scenario for multi-agent systems. Through experimental evaluation we also show how distributing our infrastructure can provide run-time support to large-scale multi-agent systems regulated by norms.

Keywords: social interaction, run-time service, GOLEM, event calculus

1 Introduction

An open multi-agent system [33], such as an electronic market, is often characterized as a computing system where software agents developed by different parties are deployed within an application domain to achieve specific objectives. An important characteristic of this class of applications is that the various parties developing the agents may have competing goals and, as a result, agent developers for a specific party will have every interest to hide their agent's internal state from the rest of the agents in the system. Although openness of this kind may encourage many agents to participate in an application, interactions in the system must be regulated so that to convince skeptical agents that the overall specification of the application domain is respected.

Norm-governed multi-agent systems [21], [2] are open multi-agent systems that are regulated according to the normative relations that may exist between member agents, such as permission, obligation, and institutional power [22], including sanctioning mechanisms dealing with violations of prohibitions and non-compliance with obligations. Although

knowledge representation frameworks for specifying such relations exist, these frameworks often focus on the expressive power of the formalism proposed and often abstract away from the computational aspects and experimental evaluation. The existing works for representing executable specifications normally do not provide experimental evaluations of multi-agent system deployment over distributed networks. The computational behaviour of many representation frameworks for norm-governed systems is often studied theoretically only, sometimes under simplifying, unrealistic assumptions.

The aim of this paper is to use a specific knowledge representation framework to develop an infrastructure for computing at run-time the physically possible actions, permissions, and sanctions, and eventually the obligations, and institutional powers of the members of a norm-governed system. The need for such an infrastructure is motivated by the observation that agents cannot be expected to be capable of computing these normative relations on their own. Practical reasons for this include (a) computational constraints agents may have (e.g. due to lack of CPU cycles, memory, or battery), and (b) incomplete knowledge agents may have about the application state (e.g. due to a partial view of the environment).

Our run-time infrastructure integrates selected versions of the Event Calculus [25] for describing an open multi-agent system as two concurrent and interconnected composite structures that evolve over time: one representing the physical environment of the open multi-agent system and the other representing the social environment. The focus of our knowledge representation framework and its associated run-time infrastructure is to provide real-time responses to agent requests. The novelty of our approach relies on the ability of our framework to provide a distributed implementation of the Event Calculus for norm governed systems. The advantage here is that by distributing a norm-governed application we can efficiently compute the distributed social and the physical states of the system.

The paper is organised as follows. In Section 2 we introduce a scenario of a norm-governed multi-agent system. We then use this scenario to describe our run-time infrastructure in Section 3, the knowledge representation framework and extensions of this framework to support a social state with norms. In Section 4 we show an experimental evaluation of the approach, followed by a comparison with related work in Section 5. Finally, in Section 6, we summarize our approach and outline plans for future work.

2 The Open Packet World

To exemplify the framework and experiment with the proposed infrastructure we will use the *Packet World* [36]. As seen in Fig. 1(a)(i), a set of agents are situated in a rectangular grid (8×8 here) consisting of a number of colored packets (squares) and destination points (circles). Agents (`a1`, `a2`, `a3`, and `a4` in Fig. 1(a)(i))) move around the grid to pick colored packets which they must deliver in destinations that match a packet's

color. As agents can see only part of the grid at any one time (the square around agent a2 represents the perception range of this agent), they often need to collaborate with each other. Collaboration results in agents forming teams to deliver packets and placing flags in locations for letting other agents know that a particular area has been explored and has no packets left. Also, each agent is powered by a battery that discharges as the agent moves in the grid. The battery can be recharged using a battery charger (situated in location (7,8) of Fig. 1(a)(i))). This charger emits a gradient whose value is larger if the agent is far away from the charger and smaller if the agent is closer to the charger.

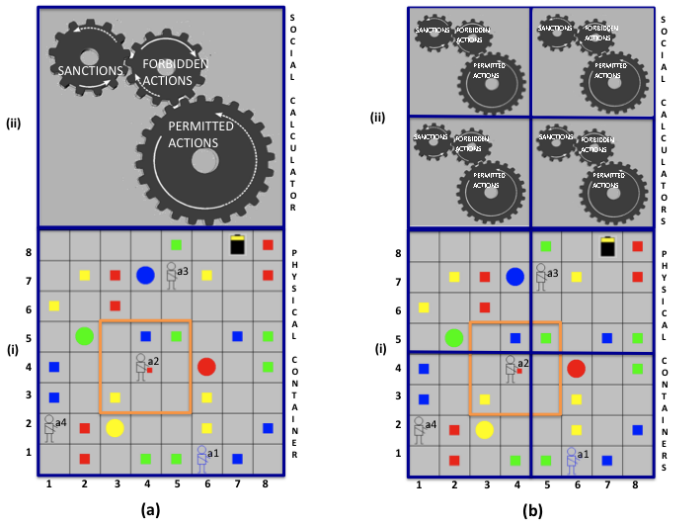

Fig. 1. Open Packet-World as a Norm-Governed System

We are interested here in a variation of Packet World, which we will refer to as *Open Packet World* (OPW). This variation differs from the original version as follows. We make the scenario competitive by giving points to agents if they deliver packets to appropriate destinations. Agents are now antagonistic and may be developed by different parties. For instance an agent may try to deceive other agents by placing a flag in an area that has packets. As a result of these extensions we introduce norms. Violation of norms results in sanctions. One type of sanction, in this example, is the reduction of points of the violating agent. In this paper we focus on permissions and sanctions.
OPW exhibits a number of features of that make it very appealing for a norm-governed systems test-bed:

– Unlike other practical applications, e.g. electronic markets, it does not require speech acts only but also physical actions, which in turn necessitate the representation of physical possibility in the system. Physical possibility requires the representation of a physical environment whose state should be distinct from the state of the social environment.
– OPW is also convenient from the point of view of experimentation in that we can make the experimental conditions harder by increasing the size of the grid, the number of agents and the number of packets/destinations. Moreover, it is natural to distribute OPW and thus test reasoning frameworks that operate on distributed knowledge bases.
– Because of the intuitive nature of actions taking place in it, OPW can be easily visualized.

3 Run-time Infrastructure

To experiment with our scenario we use the GOLEM agent platform[3]. GOLEM supports the deployment of *agents* - cognitive entities that can reason about sensory input received from the environment and act upon it, *objects* - resources that lack cognitive ability, and *containers* - virtual spaces containing agents and objects, capturing their ongoing interactions in terms of an *event-based* approach.
A GOLEM container represents a portion of the distributed agent environment and it works as a mediator for the interaction taking place between agents and objects. In general, events in a container can be caused by agents, objects and processes of the agent environment as discussed in [3], [4]. In this paper we focus on the social part of the agent environment which involves events produced mainly by agents. Agents decide what actions to perform according to their goals and to the last environmental state observed. They can modify and observe the state of the environment by using the interface provided by the container to attempt actions in the environment. The containers mediate these actions by performing the necessary updates in the state of the environment.
In this paper we are not concerned with the implementation of agents. Instead, we are concerned with an implementation of a software framework informing the decision-making of agents by computing, on their behalf, the normative positions as they evolve in time. Whether an agent complies with these positions depends entirely on the implementation of the agent. In general, there is a clear separation between the agent code and the code of our software framework. The code presented in the paper belongs entirely to the proposed software framework. A part of that code — the specification of norms and physical possibility — are application-dependent.

3.1 The Open Packet World in GOLEM

The simplest way to model OPW in GOLEM is shown in Fig. 1(a), where we deploy one container representing the world (see Fig. 1(a)(i))

[3] http://golem.cs.rhul.ac.uk

and extend it in a way that contains a social state representing the normative aspects of the system (see Fig. 1(a)(ii)).
Although a single container specification for the original Packet World has been implemented in [3], this container did not have a social state. Here we extend a container with a social state managed by an active object which we call *Social Calculator*. This object computes the agents' permissions and sanctions and publicises this information upon request. An alternative way to model OPW is to split the physical state of a single container into smaller states that we distribute in different containers. A possible distribution is shown in Fig. 1(b), where we use four 4 x 4 adjacent containers for OPW (see Fig. 1(b)(i)) together with their corresponding Social Calculators (see Fig. 1(b)(ii)). Issues such as distributing the perception range of an agent in different containers (as it is the case with ag2) and moving between containers are already described in [4]. Here we show how containers can use a social state to support a norm-governed system.

3.2 The Physical State of Containers

To represent the state of a GOLEM container we use the object-based notation of C-logic, a formalism that describes objects as complex terms that have a straightforward translation to first-order logic [5]. The complex term below, for example, represents the state of a 2 x 2 packet world with one agent, one packet, one destination and one battery:

```
packet_world:c1[
 address ⇒ "container://one@134.219.7.1:13000",
 type ⇒ open,
 grid ⇒ {square:sq1, square:sq2, square:sq3, square:sq4}
 entities ⇒ {picker:ag1, packet:p1, dest:d1, battery:b1}
]
```

Object instances of this kind belong to classes (e.g. packet_world), are characterized by unique identifiers (e.g. c1), and have attributes (e.g. address). The representation of the 8 x 8 grid of Fig. 1 is similar but larger, i.e. there are more agents, packets, destinations, and squares.
In GOLEM complex instances of objects evolve as a result of events happening in the state of a container. An event happens as a result of entities, such as agents, attempting to act in the environment. For example the assertions:

```
attempt(e14, 100).
do:e14 [actor ⇒ ag1, act ⇒ move, location⇒ sq3].
```

describe an attempt e14 at time 100, containing a physical action made by agent ag1 wishing to move to location sq3. In GOLEM, an attempt becomes an event that happens if the attempt is possible:

```
happens(Event, T) ← attempt(Event, T), possible(Event, T).
```

Happenings of events cause the state of a container C to evolve over time. To query the value Val of an attribute Attr for an entity Id of container C at a specific time T, we will use the definition:

```
solve_at(C, Id, Class, Attr, Val, T) ←
    holds_at(C, container, entity_of, Id, T),
    holds_at(Id, Class, Attr, Val, T).
```

holds_at/5 is defined by the top-level clauses of the *Object Event Calculus* (OEC) [24] and specified as:

```
holds_at(Id, Class, Attr, Val, T)←
  happens(E, Ti), Ti ≤ T,
  initiates(E, Id, Class, Attr, Val),
  not broken(Id, Class, Attr, Val, Ti, T).

broken(Id, Class, Attr, Val, Ti, Tn)←
  happens(E, Tj), Ti < Tj ≤Tn,
  terminates(E, Id, Class, Attr, Val).
```

The above definitions utilise a logic programming approach based on negation as failure [8]. They describe how the value Val of an attribute Attr for specific Class instance identified by Id holds at a particular time T, as in the usual Event Calculus [25]. Given an event E, the initiates/5 predicate assigns to the attributes Attr of an object identified by the Id and of class Class a value Val. The terminates/5 predicate has a similar meaning, with the only difference that the event E terminates the value Val of the attribute of an object. The remaining OEC clauses (see [24] for more details) describe how events create instances of C-logic like objects, assign these instances to classes, represent basic hierarchical inheritance where sub-classes inherit attributes from super-classes, destroy complex terms, and terminate single value and multi-valued attributes.
The possible/2 are application dependent rules that specify physical possibility. Below, we show an example of how we use the OEC to express a possible/2 rule in OPW:

```
possible(E, T)←
  do:E [actor ⇒ A, act ⇒ move, location⇒ SqB],
  solve_at(this, A, picker, position, SqA, T),
  adjacent(SqA, SqB),
  not occupied(SqB, T).
```

The above rule states that it is possible for an agent to move to a location SqB if the agent is currently in location SqA, SqA is adjacent to SqB, and SqB is not occupied. The keyword **this** is used here to refer to the identifier of the current container.

3.3 Containers with Social State

We extend GOLEM containers with a social state, formalized as a C-logic structure that has a reference to the physical state, and extends this physical state with social attributes to hold information about (a) any current sanctions imposed on any of the agents and (b) how many points agents have collected so far. An example snapshot of a social state for OPW is shown below:

```
packet_world_social_state: s1 [
    physical_state⇒ packet_world:c1,
    sanctions⇒ {sanction:s1 [agent ⇒ a2, ticket ⇒ 5]},
    records⇒ {record:r1[agent ⇒ a1, points ⇒ 35],
              record:r2[agent ⇒ a2, points ⇒ 25]}
]
```

The term above states that agent a2 has been sanctioned with 5 points. We show the records of two agents only to save space. Agent a1 has collected 35 points, while a2 has collected 25 after the sanction is applied. Sanctions change the points of the agent, and they are stored in the social state to keep a history of all sanctions occurred to the agents during the execution. A social state does not contain explicitly the permitted actions. These are defined implicitly in terms of rules. We write:

```
permitted(Event, T)← not forbidden(Event, T).
```

to state that actions specified in events are permitted only if they are not forbidden. Forbidden actions and the evolution of the social state due to these actions are specified in an application dependent manner. A forbidden/2 rule in OPW can be expressed as follows:

```
forbidden(E, T) ←
   do:E[actor ⇒ A, act⇒drop, object⇒flag, location⇒SqA],
   solve_at(this, Id, packet, position, SqB, T),
   adjacent(SqA, SqB).
```

states that it is forbidden for an agent A to drop a flag in location SqA if there are packets nearby.
When a forbidden act has taken place, the Social Calculator raises a violation, which results in a sanction.

```
initiates(E, R, record, points, Points)←
   happens(E,T),
   violation:E[sanction:S [ticket⇒ SanctionPs, agent ⇒ A]],
   solve_at(this, R, record, agent, A, T),
   solve_at(this, R, record, points, OldPoints, T),
   Points = OldPoints - SanctionPs.
```

initiates/5 updates the points of agent A as a consequence of receiving a sanction S at time T. This simple example shows how events happening in the physical environment (e.g. dropping a flag in a location of the grid) affect the social state of the application (e.g. through the initiation of a sanction on the agent that dropped the flag). More complex permissions and sanctions are formalized similarly.
Similarly to prohibition and permission, we can also represent a basic form of empowerment. For example, we can express the fact that an agent in a leader role within a team of collaborating agents (here identified by the class instance team) is empowered to request a second agent to collect a packet if the second agent is also a member of the same collaboration team. We express this rule as follows:

empowered(E,T)←
do:E[actor ⇒ A, act⇒collect, agent ⇒B, square ⇒ Sq],
neighbouring_instance_of(this, [], _, Max, TID, team,T),
solve_at(**this**, TID, team, leader, A, T),
solve_at(**this**, TID, team, member, B, T).

The above clause states that an agent A, who is a leader in the collaboration team identified as TID, is empowered at time T to request from another agent B of the same team to collect a packet in a square Sq. For a general discussion on empowerment see [22].
Our framework also supports obligations using rules of the form:

obliged(E, T)←
neighbouring_instance_of(this, [], _, Max, TID, team, T),
solve_at(**this**, TID, team, leader, A, T),
do:E[actor ⇒ B, act⇒pick, obj⇒ObjId, square ⇒ Sq],
request:E_j[actor ⇒ A, act ⇒ E],
happens(E_j, T_j),
$T_j < T$,
not fulfilled(E, T_j, T),
solve_at(**this**, ObjId, packet, position, Sq, T).

The clause above states that an agent B is obliged at time T to pick a packet in the location Sq if the leader of the team A has requested it before and this has not been done yet, and the packet is still in the location Sq. For our scenario, such rules provide a very basic form of obligations that implicitly persist until they are fulfilled. More complex application scenarios will require more sophisticated treatment of obligations. However, this discussion is beyond the scope of this work.

3.4 Distributing a Norm-Governed Application

One important feature of our knowledge representation framework is that we can distribute the state of a norm-governed application into multiple

containers in order to support the parallel evaluation of physical and social states. Distributing a system among multiple containers is not a novel architectural idea; however, the proposed architecture — distributed implementation of the Event Calculus supporting norm-governed systems — is, to the best of our knowledge, novel.
GOLEM supports this feature with the *Ambient Event Calculus* (AEC) [4]. The AEC uses the OEC, described earlier, to query C-logic like objects and their attributes that may be situated in distributed containers. For example, in OPW, we can distribute the grid representing the agent environment into four containers as shown in Fig. 1(b). Every container manages a part of the grid and is defined as a neighbour to the other containers. The neighborhood defines the relationship between the distributed portions of the agent environment and it is used in AEC to perform distributed queries. GOLEM supports also hierarchical representation of the agent environment (where one container contains other sub-containers) for which the interested reader is referred to [4].
The rules bellow specify how we can query the properties of objects in the agent environment:

```
neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
 Max >= 0,
 locally_at(C, Path, Path*, Id, Cls, Attr, Val, T).

neighbouring_at(C, Path, Path*, Max, Id, Cls, Attr, Val, T)←
 holds_at(C, container, neighbour, N, T),
 not member(N, Path),
 Max* is Max - 1,
 append(Path, [C], New),
 neighbouring_at(N, New, Path*, Max*, Id, Cls, Attr, Val, T).
```

Using the above specification we can query whether an object identified as Id, with class Cls, has an attribute Attr, whose value is Val at time T. In the clause above, Max represents the maximum number (decided at design time) of adjacent neighbors that the distributed query has to consider, Path represents the neighbors visited so far, while Path* represents the resulting path to the neighbor where the query has succeeded. In particular, the first clause checks whether the object is in the local state of a container. locally_at/8 checks with holds_at/5 to find the object in the container's state, including sub-containers, if any (See [4] for a full definition of locally_at/8). The second clause looks for neighbors. If a new neighbor N is found, this neighbor is asked the query but in the context of a New path and a new Max*.
We are now in a position to customize our representation for distributing the physical and social state by redefining the solve_at/6. The definition below has the effect of changing all the physical and social rules to work with distributed containers:

```
solve_at(C, Id, Class, Attr, Val, T) ←
    neighbouring_at(C, [], _, 1, Id, Class, Attr, Val, T).
```

The [] list above states that the initial path is empty, the underscore '_', that we are not interested in the resulting path, and the number 1 indicates that we should look at all neighbors whose distance is one step from the current container. In this way, we can query all the neighbors of a container in the OPW of Fig. 1(b).

3.5 Implementation Issues

The AEC is implemented on top of OEC [30] which is an object-oriented optimised version of EC. EC has been implemented in many different ways. Mueller [29], for example, has developed an implementation using satisfiability solvers, whereas Farrell et al [13] have developed a Java implementation of EC. The vast majority of EC implementations, however, are in the context of logic programming. We also adopted the logic programming due to the formal and declarative semantics — see [31], for instance, for the benefits of a logic programming EC implementation. The top-level description of OEC is specified below:

```
holds_at(Obj,Attr,Val,T):-
    object(Obj,Attr,Val,start(E)),
    time(E,T1), T1 =< T,
    not (object(Obj,Attr,Val,end(Evstar)),
        time(Evstar,T2), T2>T1, T2 <T).
```

The main difference between this OEC version and the one discussed earlier is that now we add all new properties that are initiated/terminated as object/4 assertions whenever a new event description is added to the container's state. Time intervals are used to store how the properties of objects change their value in time, which is similar to the approach followed in METATEM [14]. Additionally, the object/4 assertions store the state of the environment distinguishing between objects in the container by their identification Obj. This means that we have a double indexing on the properties of the state, the first one is time and the second one is the identification of the objects that define the state of the container, while in METATEM [14] the indexing is done only in terms of time. We denote the time periods by using start(e1) and end(e2) terms. For example, in OPW the assertions below:

```
time(e1, 2).
time(e2, 7).
object(ag1, position, [3,4], start(e1)).
object(ag1, position, [3,4], end(e2)).
object(ag1, position, [4,4], start(e2)).
```

describe how agent a1 moved to position [3,4] at time 2 and changed it to [4,4] at time 7. We know that the periods in the state of a container are either closed or open intervals which persist into the future. A new

event such as e2 either starts a new period of time (i.e. start(e2)) for a conclusion or ends a period of time which was started by another event (i.e. end(e2)). The optimization is obtained now because the new event is either related to the attributes of objects or the class membership, so we do not need to check all the events that have happened, as with the previous OEC version. Our implementation also uses indexing on the arguments of object/4 assertions, so that if the first three arguments are specified, the time to retrieve the term is $O(1)$ (which is typically the case with GOLEM queries).
When we distribute the system in many containers we may have a synchronisation problem due to the different timing in different containers. This issue was already addressed in [4] by applying a precise time protocol between sub and super containers. In this paper we assume that a network of distributed containers, however it is structured, it has always one root container that deals with the synchronisation of the containers.
Another important component of our implementation is that queries to the social and physical environment are executed in parallel. An example of the multi-threaded implementation is shown below for how we implement attempts of agents:

```
attempt(E, T):-
    par([exec(possible(E, T), true), exec(permitted(E,T), R)]),
    add(E, R, T).
```

The above program will be called by an agent that wishes to perform an action specified as an event E. The event provides input to two parallel threads, one executing possible(E,T) (which must succeed i.e. return true) and the other executing permitted(E,T) (which must have result R i.e. return true or false). If the event is concluded possible by the first thread, it will be added in the state of the container using add/3; otherwise, the attempt will fail. If the event is concluded possible by the first thread but not permitted (R=false) by the second thread, then the Social Calculator will be triggered by add/3 to produce a sanction in the social state.

4 Experimental Evaluation

Using OPW, we conducted a number of experiments to evaluate the performance of the system with different configurations. In particular, we measured the performance with a distributed versus centralised deployment of the system to show how the number of entities, the size of the environment and the distribution affect performances. In all experiments, we measured the time to compute whether an action is physically possible and whether an action is permitted. More specifically, we measured the time taken for possible/2 and forbidden/2 rules against an action performed by an agent in the environment. Then we related this time with the number of events produced.
In the first series of tests, we tested OPW in a centralized GOLEM container deployed in an Intel Centrino Core 2 Duo 2.66GHz with 4GB of

RAM. The environment was represented by a 40x40 grid and 100 packets were collected by the agents and released into one of the 8 destinations in the grid. We run the first test with 10 agents, the second test with 30 agents and the third test with 50 agents. In all of the runs, the agent "minds" (reasoning components) were deployed in a separate machine and were remotely connected with their "bodies" (action execution components) deployed in the GOLEM container.

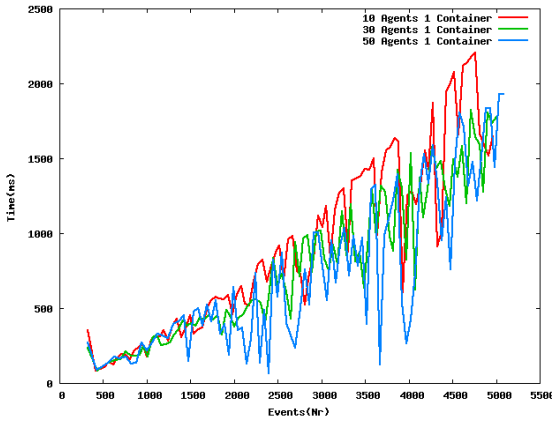

Fig. 2. Time to query the physical and the social state of one container

Fig. 2 shows three linear curves representing the average time to compute a query in a single GOLEM container with respectively 10, 30 and 50 agents. Since the evaluation of the two states is done concurrently, the curves represent the worst case between the social and the physical state.

All the three curves follow a linear behaviour suggesting that the time to query a GOLEM container grows linearly with the number of events produced in the container. The fluctuations in the curves are explained as follows. The high peaks show the worst case where the attempted action was either impossible or not permitted or both. As we check possible and permitted actions in parallel and we wait for both threads to finish the execution, the time shown is the one that took longer between the two. Alternatively, the lowest peaks show the best case where the attempted action was either possible or permitted or both. As before, the one shown is the one that took longer.

We can represent the time T_c to compute the social and physical state for a centralized container with the following equation:

$$T_c = a * E + t0 \text{ with } a \sim Ne/Na$$

where Ne is the number of entities in the system, Na is the number of active entities performing events, E is the number of events in the system and $t0$ is initial time to register the entities in the container. As the number of agents increases, then Na increases, which means that a decreases, which results in better performance. This is due to the fact that OEC is optimized to deal with events indexed by the identifiers of entities in the agent environment. For example, if we have 10 agents, 5000 events, and assuming that all agents perform the same number of events, each time that we call a solve_at/6 predicate (e.g solve_at(c1, ag1, picker, position, [3,4], 100)), the search for the value of an agent attribute will evaluate a maximum of 500 entries (5000/10), while when we have 50 agents and the remaining conditions are the same, the search will evaluate a maximum of 100 entries (5000/50). Of course, if we consider an increasing number of agents, this also means that they produce more events in less time, but it also means that given the same number of changes applied to the environment, the environment responds better with an increasing number of agents. Thus, the environment as supported by GOLEM scales up better in situations when there are many agents rather than few.

In the second series of experiments we distributed the OPW grid (40x40) first into two containers (20x40) and then into four (20x20) different containers. For the distribution of the containers we used an Intel Centrino Core 2 Duo 2.66GHz with 4GB of RAM and an Intel Centrino Core Duo 1.66Ghz with 1GB of RAM. The agents were deployed between the distributed containers and could move from one machine to another by means of the mobility capabilities offered by GOLEM [4].

Fig. 3 shows what happens when we distribute the environment in multiple containers and use AEC to link these containers.

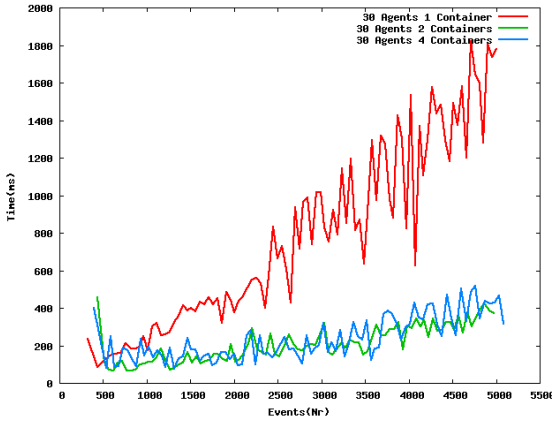

Fig. 3. The effects of distribution

As shown in Fig. 3, with a growing number of events if we increase the number of containers, we improve considerably the performance. In Fig. 3 we show that in a system with a small number of events (0-500), it is better to compute the physical and social state in one container. With a bigger number of events, the experiment shows that we can achieve a big improvement in performance if we distribute in two or four containers instead of one container.
In the distributed version the size of the grid managed by a single container becomes smaller and less complex terms (agents, packets and destinations) are registered in a single container. Between 500 to 3500 events, in average, having four or two containers does not make much difference. However, after 3500 events the performance of the application with two containers is better from the performance of the application deployed in four containers. This is due to the fact that with less packets on the grid (most of them after 3500 events have been delivered to the destinations), the agents moving on the grid are more likely to change containers in search for packets. The smaller the grid, the bigger the number of times agents try to move from a container to another. This introduces a distribution cost related to the cost of interactions between containers. For this reason, in the presented experiments there is no improvement when we change from two to four containers.
In general, the time to compute the physical and social state distributed over many containers is defined by the equation:

$$T_d = \frac{T_c}{d} + i \times c$$

where T_c is the time to compute the same experiment with a centralised container, d is number of containers used in the decentralized version, i is the number of interactions between containers and c is the cost of container interaction. In other words, when we distribute the agent environment in multiple containers, the time to compute the physical and the social state is inversely proportional to the number of containers, thus improving the performance. However, there is an additional delay to compute the physical and social state which is due to the interactions between the containers.

5 Related Work

There exist several approaches in the literature for executable specifications of norm-governed systems. Consider, for instance, the 'Law-Governed Interaction' (LGI) [28, 27] framework that has been used to regulate distributed systems. The Moses software mechanism [26] is an implementation of LGI that employs regimentation devices monitoring the behaviour of agents, blocking the performance of forbidden actions and enforcing compliance with obligations. Laws in Moses are written in pure Prolog or Java.
The Electronic Institution (EI) approach [9–11] and the AMELI framework [12], use organisational concepts [18] to model the interaction of

the agents. AMELI is an agent-based middleware for executing a set of normative rules, expressions which impose obligations or prohibitions on communicative actions such as the computation of the permissions and obligations of the agents at the current state. AMELI deploys governor agents for every external interacting agent and scene manager agents in charge of maintaining the state of the interactions between agents. One of the differences between our approach and AMELI is that we handle agent interaction via containers and we do not use mediating agents, such as AMELI governors. An explicit feature of our approach is that the state of the interaction in the agent environment is easily inspectable, while in EIs agents need to communicate to build a coherent state. Moreover the EI approach allows only for permitted actions to happen. It has been argued that regimentation is not always desirable or practical [22]. Therefore, in here we opted for sanctioning mechanisms as opposed to developing regimentation devices.

In [19] Garcia-Camino et al propose AMELI+, an extended version of the AMELI framework [12], with a mechanism to handle distribution of norms and possible conflicts [17] that may arise due to normative positions generated from the actions of agents. The AMELI architecture is extended with additional normative manager agents which together with governor agents and scene manager agents are in charge of the system. AMELI+ is based on a hierarchical structure of agents that deal with the enforcement of norms. These internal agents decide how to update and propagate the changes made in the state of the system to other internal agents interested in these changes. In contrast with our model, AMELI+ defines regimentation mechanisms, while we include a sanction mechanism for prohibited actions. Additionally, the AMELI+ approach requires many internal agents being involved into propagating messages for changes they perform locally. Instead of using internal agents we deploy containers which update the state locally and then use the AEC predicates to perform distributed queries to the containers of the agent environment. Thus we do not need to have additional propagation mechanisms and the same AEC mechanism can be used by the agents to query what is happening in the state of the environment. Moreover, AMELI+ does not support constraints as part of the norm language nor norms include reasoning with properties of the state changing in time.

Several action languages and corresponding software tools have also been employed for specifying and executing norm-governed systems. Fox et al. [16], for example, utilised an automated reasoning tool to execute 'organisational rules' formalised in the Situation Calculus [32]. Farrell et al [13] propose a formalisation of the Event Calculus in XML and apply it to the representation of contracts to facilitate the automated tracking of the contract state. Commitment protocols [7, 15] have been formalised in, among others, the action language $C+$ [20] and various dialects of the Event Calculus. Moreover, the Causal Calculator implementation of $C+$, and the Discrete Event Calculus reasoner [29] have been employed to execute commitment protocols.

Recently, norm-governed systems specifications have been formalised in semantic web languages [34, 23]; furthermore, various automated reasoning tools have been utilised for executing the specifications.

Our logic programming implementation of the Event Calculus has the following benefits. First, it exhibits a declarative semantics whose advantages, compared to procedural semantics, have been well-documented. Second, the Event Calculus offers a formal representation of the agents' actions and their effects. This is in contrast to semantic web languages that offer limited temporal representation and reasoning. Third, the availability of the full power of logic programming, which is one of the main attractions of employing the Event Calculus as the temporal formalism, allows for the development of very expressive social and physical laws. Fourth, we do not have to know from the outset the domain of each variable. Fifth, the OEC and the AEC versions used here provide an efficient and scalable reasoning mechanism, offering the kind of run-time support that is required for norm-governed multi-agent systems. The last point differentiates our work from approaches offering computational support for norm-governed systems. The last three points differentiate our work from other action language implementations.

6 Conclusions and Future Work

We presented a knowledge representation framework with an associated run-time infrastructure that is able to compute, for the benefit of the members of a norm-governed multi-agent system, physically possible and permitted actions at each time, as well as sanctions that should be applied to violations of prohibitions. The presented infrastructure is highly configurable in the sense that it can be appropriately distributed to offer run-time support for large-scale norm-governed systems.

We evaluated the platform based on the OPW scenario and showed that the distributed infrastructure is feasible. The tests showed that distribution can considerably improve the performances of the MAS system and that the distributed topology of the environment depends on the representation of the environment, the number of entities populating it and the number of events they generate.

We specify norms separately from the physical rules governing the environment, to be able to define how agents should interact at a social level. This allows us to have agents in the system that are implemented by different designers and with different strategies. The different agents can query the state of the environment and decide what actions to take exclusively based on their own internal strategy. We found out that the run-time infrastructure simplifies the implementation of the agent reasoning because agents do not need to manage the state of the interactions. Agents can reason about what acts to perform and query the infrastructure if such act is conformant with the social rules defined in the environment.

In addition, for a large norm governed application, there is a significant design decision to be taken for using our approach. This has to do with how we distribute the containers. Namely, how to organise the containers in such a way so bottle necks in evaluating the norms are avoided. In OPW we have found useful to limit the frequency of the distributed queries. In other applications, similar heuristics may be used, however

this discussion is out of the scope of the paper as it requires further investigations.
There are various directions of further work. One is to experiment with various techniques, such as those proposed in [6], [1], in order to further improve the temporal reasoning. Two, we aim to perform experiments with larger multi-agent systems in order to determine the extent to which our infrastructure can be used for run-time support. Finally, we aim to extend the game based approach presented in [35] to coordinate the distributed social state of an application in terms of complex games. Different games are governed by different normative relations and complex games use coordination mechanisms to combine normative relations by activating/deactivating one or more games. In this way we will be able to show how to define complex agent interactions and maintain a parallel evaluation between social and physical consequences of the actions performed by agents in the system.

References

1. A. Artikis, M. Sergot, and J. Pitt. A logic programming approach to activity recognition. In *Proceedings of ACM Workshop on Events in Multimedia*, 2010.
2. A. Artikis, M. J. Sergot, and J. V. Pitt. Specifying norm-governed computational societies. *ACM Transactions in Computational Logic*, 10(1), 2009.
3. S. Bromuri and K. Stathis. Situating Cognitive Agents in GOLEM. In *Engineering Environment-Mediated Multi-Agent Systems, EEMMAS 2007*, volume 5049/2008 of *Lecture Notes in Computer Science*, pages 115–134. Springer, 2007.
4. S. Bromuri and K. Stathis. Distributed Agent Environments in the Ambient Event Calculus. In *DEBS '09: Proceedings of the third international conference on Distributed event-based systems*, New York, NY, USA, 2009. ACM.
5. W. Chen and D.S. Warren. C-logic of complex objects. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 369–378, New York, NY, USA, 1989. ACM.
6. L. Chittaro and A. Montanari. Efficient temporal reasoning in the cached event calculus. *Computational Intelligence*, 12:359–382, 1996.
7. A. Chopra and M. Singh. Contextualizing commitment protocols. In *Proceedings of Conference on Autonous Agents and Multi-Agent Systems (AAMAS)*, pages 1345–1352. ACM, 2006.
8. Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.
9. M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In C. Castelfranchi and L. Johnson, editors, *Proceedings of the First International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1045–1052. ACM Press, 2002.

10. M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. In J.-J. Meyer and M. Tambe, editors, *Intelligent Agents VIII*, LNAI2333, pages 348–366. Springer-Verlag, 2002.
11. M. Esteva, J. Rodríguez-Aguilar, C. Sierra, and W. Vasconcelos. Verifying norm consistency in electronic institutions. In *Proceedings of the AAAI-04 Workshop on Agent Organizations: Theory and Practice*, pages 8–14, 2004.
12. M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. Ll. Arcos. Ameli: An agent-based middleware for electronic institutions. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 236–243, Washington, DC, USA, 2004. IEEE Computer Society.
13. A. D. H. Farrell, M. J. Sergot, M. Sall, and C. Bartolini. Using the event calculus for tracking the normative state of contracts. *International Journal of Cooperative Information Systems*, 14:99–129, 2005.
14. M. Fisher and R. Owens. From the past to the future: Executing temporal logic programs. In *In Proceedings of Logic Programming and Automated Reasoning (LPAR*, pages 369–380. Springer Verlag, 1992.
15. N. Fornara and M. Colombetti. *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, chapter Formal specification of artificial institutions using the event calculus. IGI Global, 2009.
16. M. Fox, M. Barbuceanu, M. Grüninger, and J. Lin. An organizational ontology for enterprise modeling. In M. Prietula, K. Carley, and L. Gasser, editors, *Simulating Organizations: Computational Models for Institutions and Groups*, pages 131–152. AAAI Press/The MIT Press, 1998.
17. D. Gaertner, A. Garcia-Camino, P. Noriega, J. A. Rodriguez-Aguilar, and W. Vasconcelos. Distributed norm management in regulated multiagent systems. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.
18. A. García-Camino, P. Noriega, and J. Rodríguez-Aguilar. Implementing norms in electronic institutions. In *Proceedings of the Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 667–673. ACM Press, 2005.
19. A. García-Camino, J. A. Rodríguez-Aguilar, and W. Vasconcelos. A distributed architecture for norm management in multi-agent systems. In *COIN'07: Proceedings of the 2007 international conference on Coordination, organizations, institutions, and norms in agent systems III*, pages 275–286, Berlin, Heidelberg, 2008. Springer-Verlag.
20. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.
21. A. Jones and M. Sergot. On the characterisation of law and computer systems: the normative systems perspective. In *Deontic Logic in Computer Science: Normative System Specification*, pages 275–307. J. Wiley and Sons, 1993.
22. A. Jones and M. Sergot. A formal characterisation of institutionalised power. *Journal of the IGPL*, 4(3):429–445, 1996.

23. L. Kagal and T. Finin. Modeling communicative behavior using permissions and obligations. *Journal of Autonomous Agents and Multi-Agent Systems*, 14(2):187–206, 2006.
24. F. N. Kesim and M. Sergot. A Logic Programming Framework for Modeling Temporal Objects. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):724–741, 1996.
25. R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.
26. N. Minsky. *Law-Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction and a Reference Manual)*, 2005. Retrieved October 24, 2008, from `http://www.moses.rutgers.edu/documentation/manual.pdf`.
27. N. Minsky. Decentralised regulation of distributed systems: Beyond access control, 2008. Submitted for publication. Retrieved October 24, 2008, from `http://www.cs.rutgers.edu/~minsky/papers/IC.pdf`.
28. N. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):273–305, 2000.
29. E. Mueller. *Commonsense Reasoning*. Morgan Kaufmann, 2006.
30. K. Nihan and S. Marek. Implementing an object-oriented deductive database using temporal reasoning. *J. Database Manage.*, 7(4):21–34, 1996.
31. A. Paschke and M. Bichler. Knowledge representation concepts for automated SLA management. *Decision Support Systems*, 46(1):187–205, 2008.
32. J. Pinto and R. Reiter. Temporal reasoning in logic programming: a case for the situation calculus. In D. Warren, editor, *Proceedings of Conference on Logic Programming*, pages 203–221. MIT Press, 1993.
33. J. Pitt, A. Mamdani, and P. Charlton. The open agent society and its enemies: a position statement and research programme. *Telematics and Informatics*, 18(1):67–87, 2001.
34. G. Tonti, J. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, rei and ponder. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *Proceedings of Second International Semantic Web Conference*, volume LNCS 2870, pages 419–437. Springer, 2003.
35. V. Urovi and K. Stathis. Playing with agent coordination patterns in mage. In Julian A. Padget, Alexander Artikis, Wamberto Weber Vasconcelos, Kostas Stathis, Viviane Torres da Silva, Eric T. Matson, and Axel Polleres, editors, *Coordination, Organizations, Institutions and Norms in Agent Systems V, COIN 2009 International Workshops. COIN@AAMAS 2009, Budapest, Hungary, May 2009, COIN@IJCAI 2009, Pasadena, USA, July 2009, COIN@MALLOW 2009, Turin, Italy, September 2009. Revised Selected Papers*, volume 6069 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2009.

36. D. Weyns, A. Helleboogh, and T. Holvoet. The packet-world: A testbed for investigating situated multiagent systems. In *Software Agent-Based Applications, Platforms, and Development Kits*, pages 383–408. Birkhauser Verlag, Basel - Boston - Berlin, 2005.