Normative Run-Time Reasoning for Institutionally-Situated BDI Agents

Tina Balke
Information Systems Management
University of Bayreuth, Germany
tina.balke@uni-bayreuth.de

Marina De Vos, Julian Padget and Dimitris Traskas
Department of Computer Science
University of Bath, UK
{*mdv,jap*}*@cs.bath.ac.uk, d.traskas@bath.ac.uk*

***Abstract*—Institutions offer the promise of a means to govern open systems, in particular open multi-agent systems. Research in logics, and subsequently tools, supports the specification, verification and enactment of institutions. Most effort to date has focussed on the design-time properties of institutions (either on the normative or the system level), such as whether a particular state of affairs is reachable or not from a given set of initial conditions. Such models are useful in forcing the designer to state their intentions precisely, and for testing (design) properties. However, we identify two problems in the direct utilization of event-based design-time models in the governance of live (running) systems: (i) over-specification of constraints on agent autonomy and (ii) generation of design-time model artefacts. In this paper we present a methodology to tackle these two problems and extract the run-time model from the design-time one. We demonstrate how to derive an event-based run-time model of institutions that can be incorporated into the reasoning processes of autonomous BDI agents to realize practical norm-governed multi-agent systems.**

I. Introduction

The motivation for this work derives from two issues: the first is the general goal of run-time governance of open distributed systems and the second is a specific case study of such a system: future mobile phone networks (called wireless grids), where institutions are key to governing the interaction of participants. The problem for the designer is how to balance the tension between institutions and agents: the latter are (supposed to be) autonomous, while institutions constrain autonomy. Often, in norm-governed MAS, this problem is "solved" by regimenting agents and their actions and thus not allowing any norm-deviation. In contrast, we use a more social institution, where agents can query its (run-time) properties in order to examine how current situations were achieved and to evaluate possible futures.

Traditionally when trying to analyze normative effects the real world is formalized as two *separate* models: a system model and a normative/institutional model of which only the design-time properties are analyzed. While useful, this can be problematic when wanting to analyze the actual interplay between agents and institutions. Furthermore it poses the problem of how to account for any run-time effects between and in the two models. Thus, in contrast to the separate design-time analysis of the two models, we are interested in an integration and coherent analysis of *both* models as well as their interaction. To reach that goal, we approach institutional and system modelling in two phases: We start with a *design-time model:* where both normative and system perspectives are expressed as a logic program under answer set semantics (ASP) [1]. For example, we build an institutional model of the wireless grid concept to evaluate whether it makes sense to pursue the idea. This model hard-codes simplifications of the environment in which the agents interact, but it can be used for validation purposes and helps to expose requirements issues. Secondly we build a *run-time model:* which is extracted from the design-time model, by removing all but the normative information and domain facts. While normative information is still the subject of ASP reasoning, the exogenous events that trigger normative change are created by a MAS simulation. The run-time model provides the (BDI) agents in the simulation with a kind of oracle, that can respond to queries both about the current state and the normative consequences of actions.

The experience gained during the development and execution of these two phases has lead to the main contribution of this paper: a *methodology* for developing design-time and run-time institutional models, that is, models that play a key part in *developing and running* either an application or, as in our case, a simulation, and expressing the rules of governance for open systems. In that respect, the simulation and its results are tangential to the present focus, which is how to make normative models accessible to agents.

II. Case Study: Wireless Grids

The process and implications of modelling normative systems for agent reasoning can usefully be illustrated by a case study. The case study is situated against the background of the next generation of mobile phones, where *wireless grids* have been proposed to address the energy issues inherent in these devices [2]. Batteries have a fixed capacity that puts limits on the operational time for a device. The increasing sophistication of mobile phones has had a significant impact on power consumption, leading to shorter stand-by times, as well as higher battery temperature unless there is active cooling [2]. The idea of wireless grids is, that in contrast to distributing digital content exclusively via an expensive (in terms of power and money) connection to a structured network, mobile phones cooperate and share content via a

cheap(er) ad-hoc connection as well.

Energy gains aside, the scheme has the intrinsic weakness of distributed cooperative architectures: it relies on cooperation to succeed. Cooperation in this context implies that participants volunteer to share the data they obtain via the structure network. However, a cost is involved as sharing uses battery consumption. As a consequence, (bounded) rational users prefer to receive resources without any commitment of their own, which jeopardises the whole grid. So incentives for cooperation are essential.

In this paper we show that a normative system can be used to prototype and verify a cooperation mechanism—the design-time model—and subsequently govern the mechanism in a running system, using the run-time model. This two-phase approach demonstrates that we can build a norm-governed system that is: (i) *flexible:* by changing the institutional model, it is possible to influence agent behaviour, without modifying individuals—assuming a suitable goal-driven agent—and (ii) *realistic:* in this scenario, as in those foreseen for multi-agent systems, we cannot predict or control with total certainty the behaviour of agents, but it is hoped that institutions can provide functions similar to those found in the physical world.

III. Norm Governed Systems

For our formal model, we adopt the one proposed by Cliffe *et al* [3]. Its event-driven model and mathematical foundation with computational support makes it ideal for use in an agent-based simulation. A normative model describes which actions are permitted by agents given the current state of affairs. This implies these actions have to be observed by the normative model and be interpreted within the current normative context. We refer to these observed events as exogenous ($\mathcal{E}_{ex}$), and use conventional generation, inspired by the concept of 'counts-as' [4], to generate normative events from them. Thus, we can determine the effect of $\mathcal{E}_{ex}$ on the normative framework, given its current state. The state of the framework is represented as a set of fluents consisting of brute facts [5] and normative fluents defining powers, permissions, obligations and violations. Events trigger the initiation and termination of fluents, as specified by the consequence relation. Given an $\mathcal{E}_{ex}$ and the current normative state, the new state is obtained from the application of the transitive closure of the generation and the consequence relations to determine the initiation and termination of (institutional) fluents. The normative semantics is defined over a sequence, called a trace, of $\mathcal{E}_{ex}$.

The formalization of the framework is realized as a computational model through Answer Set Programming (ASP) [1], [3]. The benefits of using ASP for modelling normative framworks are given in [3]. Cliffe *et al.* also put forward a domain-specific action language, Inst*AL* , that translates to $AnsProlog$. An Inst*AL* program consists of two parts: the normative specification and a domain file. The specification consists of two logical parts: a template part describing the institution and its components and the initiation part. The template provides the name of the institution, the events and fluents it uses, generic rules and a disjoint set of monomorphic types. The actual values of each type are specified in the domain file. Extracts from the wireless grid scenario are given in Fig. 1 and 2. We explain the main syntax elements of Inst*AL* as follows: Events are defined by *typeOfEvent* `event` *nameOfEvent*`;` with type being one of `exogenous`, `create`, `inst` or `violation`, while fluents are defined by `fluent` *nameOfFluent*`(`*ParameterType*`, ...);`. Generation of normative events from exogenous events is specified using the `generates` statement, while `initiates` and `terminates` define the two parts of the consequence relation. Conditions on the state are expressed using `if`. The `initially` statement specifies the fluents in the initial state. For ease of specification, Inst*AL* also introduces non-inertial fluents. While conventional fluents, once initiated, remain true until terminated, non-inertial ones' truth value is evaluated in every state on the basis of the specified conditions. It requires `noninertial fluent` *nameOfNonInertialFluent*`(`*ParameterType*`, ...);` for the specification and *nameOfNonInertialFluent* `when` *condition;* for the truth-condition.

IV. Modelling the Wireless Grid Scenario

Having explained the main concepts of normative systems, in this section we present the specification of the case study in the form of a design-time and a run-time model. We underline the intimate relationship of the design-time and run-time models by marking the latter in bold within the specification of the former in Fig. 1 and 2.

We demonstrate the applicability using a simplified scenario where a number of handsets are allocated a set of chunks (parts) of a file that they need to download from the base station and then share with other handsets. The system enforces the narm that handsets must share in order to receive. To avoid details that would unnecessarily complicate the specification, we impose the simplification that each file chunk is assigned to exactly one handset and that each handset is assigned the same number of chunks. We are not concerned with the process that brings this allocation about.

The Design-Time Model: The allocation of chunks for download by each handset is given in the initial state of the model (see Fig. 2, lines 123–126) where the `downloadChunk` fluents indicate which handsets are tasked with downloading which chunks from the base-station. The handsets are also given the necessary permissions (lines 118–122). In the download phase, each handset downloads its assigned chunks from the base-station. The full specification of this phase is given in Figure 1. Each handset can only obtain one chunk at a time from the base station, and each channel can only be used to download a single chunk. This is modelled using the non-inertial fluents `busyBReceiving`

```
1   institution grid;
3   type Handset;
4   type Chunk;
5   type Time;
6   type Channel;
9   exogenous event clock;
10  exogenous event download(Handset,Chunk,Channel);
11  exogenous event send(Handset,Chunk);
14  create event creategrid;
17  inst event intDownload(Handset,Chunk,Channel);
18  inst event intSend(Handset);
19  inst event intReceive(Handset,Chunk);
20  inst event transition;
23  violation event misuse(Handset);
26  fluent downloadChunk(Handset,Chunk);
27  fluent hasChunk(Handset,Chunk);
28  fluent areceive(Handset,Time);
29  fluent asend(Handset,Time);
30  fluent creceive(Handset,Time);
31  fluent csend(Handset,Time);
32  fluent transmit(Channel,Time);
33  fluent previous(Time,Time);
36  noninertial fluent busyHSending(Handset);
37  noninertial fluent busyHReceiving(Handset);
38  noninertial fluent busyBReceiving(Handset);
39  noninertial fluent busyChannel(Channel);
40
47  download(A,X,C) generates intDownload(A,X,C)
48       if not busyChannel(C), not busyBReceiving(A), not busyHSending(A);
50  download(A,X,C) generates transition;
51  clock generates transition;
53  intDownload(A,X,C) initiates hasChunk(A,X);
54  intDownload(A,X,C) initiates creceive(A,4), transmit(C,4);
56  transition initiates transmit(C,T2) if transmit(C,T1), previous(T1,T2);
57  transition initiates creceive(A,T2) if creceive(A,T1), previous(T1,T2);
58  transition initiates pow(intDownload(A,X,C)) if creceive(A,1);
59  transition terminates csend(A,Time);
60  transition terminates creceive(A,Time);
61  transition terminates transmit(C,Time);
63  intDownload(A,X,C) terminates pow(intDownload(A,X1,C1));
64  intDownload(A,X,C) terminates pow(intDownload(B,X1,C));
65  intDownload(A,X,C) terminates downloadChunk(A,X);
66  intDownload(A,X,C) terminates perm(download(A,X,C1));
68  busyChannel(C) when transmit(C,T2);
69  busyBReceiving(A) when creceive(A,T2);
```

Figure 1. Model Declaration and Generation and consequence relations for downloading

and `busyChannel` which are implied on the basis of the handset downloading and the base-station transmitting (lines 68–69). The first Inst*AL* rule (lines 47–48) indicates that a request to download a chunk is granted whenever there is an available channel and the handset is not currently receiving from the base-station and is not busy sending another chunk. When a chunk is downloaded, the handset and the channel are busy for a fixed amount of time–4 time steps (line 54). From the first instant of the handset interacting with the base-station, it is deemed to have downloaded the chunk, so parts can be shared (line 53). As soon as a channel and a handset are engaged, the framework (i) removes the power from the handset and from the channel to engage in any other interactions (lines 63–64), (ii) stops the handset from needing the chunk (line 65) and (iii) cancels the permission to download the chunk again (line 66).

In the design-time case, we need a mechanism to mark the passage of time. For this purpose, each exogenous event generates a transition event (lines 50–51), while the `clock` event indicates that there was no interaction with the institution. The `transition` event counts down the interaction time between the channel and handset (line 56–57). When the the interaction finishes, `transition` restores the power for a handset to download via the channel and for the handset to download more chunks (line 58). The event also terminates any unnecessary busy fluents (lines 59–61).

In the sharing phase each handset sends chunks to or

```
77  send(A,X) generates intSend(A) if hasChunk(A,X),
78          not busyHSending(A), not busyHReceiving(A), not busyBReceiving(A);
80  send(A,X) generates intReceive(B,X)
81       if not hasChunk(B,X), not busyHSending(B), not busyHReceiving(B),
82          hasChunk(A,X), not busyHSending(A), not busyHReceiving(A),
83          not busyBReceiving(A);
85  send(A,X) generates transition;
86  clock generates transition;
88  viol(intReceive(A,X)) generates misuse(A);
89  misuse(A) terminates pow(intReceive(A,X));
91  intReceive(A,X) initiates hasChunk(A,X);
92  intSend(B) initiates perm(intReceive(B,X));
93  intReceive(A,X) initiates areceive(A,2);
94  intSend(B) initiates asend(B,2);
103 intReceive(A,X) terminates perm(intReceive(A,X));
104 intReceive(A,X) terminates pow(intReceive(A,X));
105 intSend(A) terminates pow(intSend(A));
106 intReceive(A,X) terminates perm(intReceive(A,Y));
108 busyHReceiving(A) when areceive(A,T2);
109 busyHSending(A) when asend(A,T2);
110
115 initially pow(transition), perm(transition),
116      perm(clock),
117      pow(intDownload(A,B,C)),
118      perm(intDownload(A,B,C)),
119      perm(download(alice,x1,C)),
120      perm(download(alice,x3,C)),
121      perm(download(bob,x2,C)),
122      perm(download(bob,x4,C)),
123      downloadChunk(alice,x1),
124      downloadChunk(alice,x3),
125      downloadChunk(bob,x2),
126      downloadChunk(bob,x4);
132 initially pow(transition), perm(transition),
133      perm(clock),
134      pow(intReceive(Handset,Chunk)),
135      pow(intSend(Handset)),
136      perm(send(Handset,Chunk)),
137      perm(intReceive(Handset,Chunk)),
138      perm(intSend(Handset));
```

Figure 2. Generation and consequence relations for sharing and the initial state of the model, post negotiation

receives chunks from another handset, with the goal that at the end of the process, each handset has a complete set of the chunks. The full specification is given in Figure 2. The idea behind the model is similar to the downloading phase, but with two critical differences. First, the sending of a chunk by one handset automatically triggers the reception of the chunk by the partners (line 80), thus the design-time model assumes no network failures, etc. Furthermore, we incorporate a very basic mechanism to encourage handsets to share their chunks with others rather than just downloading them: when a chunk is received by sharing, the receiving handset loses permission to receive another chunk until it has sent a chunk (lines 106 and 92 respectively). Continuous receiving without sending (detection of unpermitted `intReceiving`) results in a violation event named `misuse` (line 88). The simple penalty applied here is that the violating handset permanently loses the power to `intReceiving` (line 89), which means the handset is expelled from the group. The traces generated by the design-time model verify that when agents follow the norms, the entire community benefits—except if norms are breached at the end of the trace, as the penalty has no effect. While this might not cause problems if participants never meet again, penalties can always be applied at the next encounter. This information gives us sufficient confidence to implement the protocol in our energy-saving simulation, where handsets might engage in several sharing contracts over a period of time and historical information can be used against them and propagated through the network.

The Run-Time Model: For a given normative system, both the design-time and run-time model should have the

same normative intentions, making the design-time model a sensible starting point for the development of the run-time one. A first step is to remove rules and conditions that deal with simulating a running system. The run-time model only needs to monitor normative behaviour. Thus, it only monitors the external events resulting from agents' actions. As a consequence, we no longer need to model system data, such as whether a channel is being used at a given moment or that a particular handset is incapable, from a technical perspective, of sending or receiving chunks. Removing the rules involving the respective events and corresponding fluents from the remaining rules, we almost achieve the specification printed in bold.

In the design-time model, we penalized misbehavior by taking away the power of a handset to receive chunks. While this may be a reasonable simplification in a design-time model for verification purposes, it cannot be enforced in a running system unless one expects agents to penalize themselves. Instead, the system notes the violation and agents may use this information in future interactions. Thus, we remove the violation event `misuse` (line 23), its generate rule (line 88) and any rules that terminate the power of agents. In the run-time model, the assignment of chunks to agents (i.e. the initial configuration of the agent/chunk combinations indicated by the `initially` identifier in the Inst*AL* specification) is determined at run-time by agents, which meet, decide to cooperate and negotiate which agent is to download and share which chunk.

V. BDI Agents and Institutions

For the implementation of the run-time reasoning we use the Jason platform [6], a Java-based interpreter for an extended version of AgentSpeak. We link it to the institutional model and answer set solver using system calls. For maintaining the normative state in our running system we introduce a special type of agent or entity: the Governor [7]. When created it is given the template part of the normative specification. When agents agree to collaborate they create a contract comprising their agent IDs, the chunks involved, the channels they will use and their allocation of which chunks to download from the base station. This information is expressed as a custom domain file and initial institutional state. Each contract is represented as a new instantiation of the institution. Whenever an action takes place that affects a contract, the Governor is informed of the agent IDs and the action and computes the next normative state for that contract using the current state (for the `initially` part) and the associated domain file. Having the information for the initial contract, as well as tracking the normative state of each contract by analyzing the respective exogenous events, the Governor can act as an institutional query processor for the agents. Contracting agents can query the current state and establish consequences of potential actions. This is done whenever the current step of the agent's reasoning cycle requires perceptions and as a result, an update of the agent's belief base takes place; i.e. the agent stores the percepts in its belief base and can use them for reasoning from that point onward. Based on its internal reasoning, an agent will perform actions in the MAS. These actions are registered in the environment and result in exogenous events, about which the Governor is informed.

VI. Discussion

In this paper we demonstrate how institutions can be incorporated into a multi-agent simulation, and consequently, in live MAS. To achieve this, the design-time model, used for verifying design-time properties of the system, can be reduced to a run-time model containing just the normative information and the relevant domain fluents.

To use the run-time model in a live system, it needs to be encapsulated in a monitor object—which we call a Governor—whose sole purpose is to manage normative states and to answer queries from norm-aware agents.

In our simulation, one Governor object manages several instantiations of the same normative framework. We observe that often, more than one normative framework is active within an application. Furthermore, some of these may interact with one another. In [3], the authors present the concept of multi-institutions where events in one institution cause events in another or change another institution's state. Extension of the Governor to accommodate multi-institutional reasoning is an important part of future work, along with the issue of using conventional distributed systems techniques, such as replication, as a means to avoid the Governor becoming a bottleneck/single point of failure.

References

[1] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, vol. 9, no. 3-4, pp. 365–386, 1991.

[2] F. H. P. Fitzek and M. D. Katz, "Cellular controlled peer to peer communications: Overview and potentials," in *Cognitive Wireless Networks*. Springer, 2007, pp. 31–59.

[3] O. Cliffe, M. De Vos, and J. Padget, "Specifying and reasoning about multiple institutions," in *COIN II*, ser. LNAI, vol. 4386. Springer Berlin / Heidelberg, 2007, pp. 67–85.

[4] A. J. Jones and M. Sergot, "A Formal Characterisation of Institutionalised Power," *ACM Computing Surveys*, vol. 28, no. 4, p. 121, 1996.

[5] John R. Searle, *The Construction of Social Reality*. Allen Lane, The Penguin Press, 1995.

[6] R. H. Bordini, M. Wooldridge, and J. F. Hübner, *Programming Multi-Agent Systems in AgentSpeak using Jason*, ser. Wiley Series in Agent Technology. John Wiley & Sons, 2007.

[7] P. Noriega, *Agent mediated auctions: The Fishmarket Metaphor*, PhD Thesis, Universitat Autònoma de Barcelona, 1997.