Dealing with incomplete normative states

Juan Manuel Serrano
University Rey Juan Carlos
Madrid, Spain
juanmanuel.serrano@urjc.es

Sergio Saugar
University Rey Juan Carlos
Madrid, Spain
sergio.saugar@urjc.es

***Abstract*—This paper puts forward a normative framework for computational societies which enables the handling of incomplete knowledge about normative relations. In particular, attempts to perform a social action are evaluated as permitted, prohibited (i.e. not permitted) or *pending* for execution (i.e. neither permitted nor prohibited). This latter category of attempts can eventually be resolved as permitted or prohibited attempts using the speech acts *allow* and *forbid*. We make use of the support for incompleteness of action language K in the formalisation of the framework. The proposal will be illustrated with some scenarios drawn from the management of university courses.**

I. Introduction

Empowerments and permissions are two common normative devices in the design of computational societies [1], [2], [3], [4], [5]. The former notion allows us to model the institutional capabilities ascribed to agents of the society; the latter one serves to represent those desirable institutional states or courses of action which do not lend themselves to violation. The relations between both notions are commonly considered application-dependent, i.e. in some domains permission may be a necessary condition for empowered agents to act, whereas in others empowerment alone may be a sufficient condition. Concerning permissions, a difference is also made between regimentation and enforcement mechanisms in the implementation of normative systems [6]: regimented infrastructures (e.g. AMELI [7]) effectively prevent agents from executing some action if the corresponding permission does not hold; on the contrary, systems based upon enforcement rely on a subsidiary normative corpus of checking and sanctioning rules to bias the behaviour of agents towards the desired courses of actions. Finally, a common assumption in the literature is to consider that both empowerments and permissions are necessarily either true or false.

This paper challenges this last assumption for the case of permissions. In particular, it considers those situations in which the designers of the computational society do not have enough knowledge so as to generate a complete set of permission rules for certain classes of actions. For instance, let us consider a computational society designed to support the management of university courses. As part of the resulting specification, empowerment and permission rules are defined which partially regulate the social processes of the application domain, namely assignments, examinations, tutoring, lecturing, and so forth. In particular, the following norms concerning the creation of assignment groups will be considered. Firstly, empowerment to *set up* a working group for some assignment is granted to any student of the course who has not yet passed that assignment. The attempt of setting up an assignment group will be permitted if, and only if, the assignment has been published, the specified submission deadline has not yet passed and the student is not participating in any other working group for that assignment. Alternatively, instead of setting up her own working group, a student may *join* a working group set up by another colleague in order to collaborate with him. Empowerment conditions for joining assignment groups coincide with those identified for the setting up of new groups. Some of the permission conditions for setting up working groups are also relevant, although only as necessary conditions. In particular, if some student attempts to join some working group and the corresponding submission deadline passed or she is already participating in another working group, then the attempt will be prohibited (i.e. not permitted). If none of these conditions hold, then there are no grounds for prohibiting the attempt. However, this does not mean that there are grounds for permitting the execution of the social action, since the ultimate decision on the permission or prohibition of the attempt lies with the initiator of the group (i.e. the student who set up the group). Thus, sufficient conditions for permitting or prohibiting the joining action can not be specified in advance by the designer of the society. In these scenarios, it would be very convenient to give the initiator of the group the possibility of either *allowing* or *forbidding* the corresponding social action so that the attempt is eventually permitted or prohibited.

This paper puts forward a formal model of permission which enables the representation of incomplete information about the normative status of social action attempts, such as the one commented above. Moreover, it formalises the meaning of the speech acts *allow* and *forbid* in the context of the previous model. In order to proceed with this formalization, we build on the notions of empowerment and permissions reported in [5]. That work introduces an operational semantics of computational societies using the action language C+ [8]. In this paper, we opt for the alternative action language K [9] due to its support for incompleteness.

The rest of the paper is structured as follows. Firstly, the most salient features of action language K for the purpose of this paper will be reviewed. Then, the general framework for social action processing will be introduced, describing the major features of the action description in language K. Next, the speech acts of allowing and forbidding will be formalised. Last, the major differences with previous work will be discussed and current work briefly described.

II. Review of action language K

Action languages are formal techniques for representing and reasoning about the performance of actions in dynamic domains. The semantics of action languages is given in terms of transition systems, namely graphs whose states and arcs represent, respectively, the possible configurations of the domain and its evolution due to the concurrent execution of a set of actions. Commonly, action languages such as C+ describe transitions between states of the *physical* world, i.e. states which represent complete configurations of the domain, where each fluent is necessarily either true or false. In contrast, action language K [9][10] allows us to describe transitions between states of *knowledge*, where the truth values of some fluents may be neither true nor false, but unknown. The motivation behind action language K was thus to support agents with an incomplete view of the world in their planning processes.

An action description in language K is composed of a set of fluent and action declarations, a set of causation rules and a set of executability conditions:

- A fluent or action p is declared using an expression of the form:

  $$\mathrm{p(X_1,\ldots,X_n)}\ \texttt{requires}\ \mathrm{t_1,\ldots,t_n}$$

  where X_i are variables and t_i are positive literals (i.e., true atoms) which specify the types of the corresponding variables[1].
- Causation rules are expressions of the form:

  $$\texttt{caused}\ \mathrm{f}\ \texttt{if}\ \mathrm{B}\ \texttt{after}\ \mathrm{A}$$

  If the subexpression f is a fluent literal, the causation rule expresses that f is known to be true in the current state if B holds in the current state and A also holds in the preceding state. The subexpressions B and A are actually sequences of literals, possibly prefixed with the default (or weak) negation operator not. The expression not f holds if f is not known to be true, whereas the expression not $-$f holds if f is not known to be false. If both expressions hold then the truth value of f is *unknown*.
- The subexpression B can only refer to type or fluent predicates, whereas A can also refer to action predicates. If sequences B and A are empty, the corresponding `if` and `after` parts can be dropped from the expression. If the `after` part is empty the rule is called *static*, otherwise the causation rule is *dynamic*. Moreover, if f is the atom false, the causation rule represents an static (resp. dynamic) integrity constraint which allows us to filter out from the transition system ill-formed states (resp. transitions). Dynamic rules can be used to represent the non-executability conditions and effects of actions. In particular, the following macro rule is a shorthand of a dynamic constraint to represent that condition B blocks the execution of action a [10, sec. 2]:

  $$\texttt{nonexecutable}\ \mathrm{a}\ \texttt{if}\ \mathrm{B}$$

- Executability statements are primitive (i.e. not macro) expressions of the form

  $$\texttt{executable}\ \mathrm{a}\ \texttt{if}\ \mathrm{B}$$

  This kind of declaration expresses that action a is *eligible* for execution in any state of knowledge in which B holds. If we want the execution of action a to be not only possible but also mandatory, then a dynamic constraint can be declared. Since this a common requirement, this paper introduces the following macro rule which allows us to declare B as a sufficient condition for executing a:

  $$\texttt{executed}\ \mathrm{a}\ \texttt{if}\ \mathrm{B} \Leftrightarrow \begin{array}{l}\texttt{executable}\ \mathrm{a}\ \texttt{if}\ \mathrm{B}\\ \texttt{caused}\ \text{false}\ \texttt{after}\ \text{not a, B}\end{array}$$

III. Social action processing

Departing from its original motivation, action language K will be used in this paper for describing transitions between states of *institutional* worlds, rather than states of *knowledge* of some planning agent. In particular, the technical apparatus of language K will be exploited to represent institutional states where some normative fluents (e.g., permissions) may have an inherent, non-epistemic indeterminacy. In software engineering terms, the dynamic domain to be modeled is thus the social middleware infrastructure in charge of the management of the institutional state of the computational society, rather than the software components participating as agents in the society. The corresponding action description is partitioned in several sub-specifications corresponding to the different types of social entities of the computational society, namely *social interactions*, *agents* and *social actions*. A complete account of this specification, however, is beyond the scope of this paper[2]. Instead, the focus here will be on the major features concerning the processing of social actions.

A. Social interactions

The institutional state of computational societies is hierarchically structured in terms of a tree of *social interactions*. The root of this tree, or top-level interaction, represents the social context within which the whole agent activity takes place; the other sub-interactions represent the social contexts for particular joint activities (i.e. social processes). For instance, assignment groups are represented by social interactions which are sub-interactions of courses, another type of social interaction. Social interactions may be *initiated* within the context of some other interaction, and eventually *finished* by the social middleware. The conditions which cause the execution of these actions are, in general, dependent on the type of interaction. Thus, university courses are automatically initiated when the new academic year begins, and assignment groups are initiated when some student successfully declares its initiation through the performance of the *set up* social action – in accordance with the empowerment and permission rules of the society. In this latter case, the *initiator* of the social interaction can be defined as the performer of the *set up* action.

[1] This is actually a slightly simplified version of this construction. See [10] for the full version and the meaning of *type* predicates.

[2] But see the C+ version [5] of the specification (which is not able to handle incomplete states), and the full K implementation available from http://zenon.etsii.urjc.es/~jserrano/speech/k-impl.tgz

```
fluents:                                                    1
   state_i(I,S) requires                                    2
      interaction(I), interaction_state(S).                 3
   context_i(I1,I2) requires                                4
      interaction(I1), interaction (I2).                    5
   initiator(I,A) requires                                  6
      interaction(I), agent(A).                             7
   ...                                                      8
actions:                                                    9
   initiate(I1,I2) requires                                 10
      interaction(I1), interaction(I2).                     11
   ...                                                      12
always:                                                     13
   nonexecutable initiate(I1,I2) if                         14
      state_i(I1,open).                                     15
   caused state_i(I1,open) after                            16
      initiate(I1,I2).                                      17
   caused context_i(I1,I2) after                            18
      initiate(I1,I2).                                      19
   ...                                                      20
```

Listing 1. K-specification of social interactions

```
fluents:                                                    1
   state_a(A,S) requires                                    2
      agent(A), agent_state(S).                             3
   context_a(A,I) requires                                  4
      agent(A), interaction (I).                            5
   player(A1,A2) requires                                   6
      agent(A1), agent(A2)                                  7
   ...                                                      8
actions:                                                    9
   play(A,I) requires                                       10
      interaction(I), agent(A).                             11
   play_for(A1,A2,I) requires                               12
      agent(A1), agent(A2), interaction(I).                 13
   ...                                                      14
always:                                                     15
   nonexecutable play(A,I) if                               16
      state_a(A,playing).                                   17
   caused state_a(A,playing) after                          18
      play(A,I).                                            19
   caused context_a(A,I) after                              20
      play(A,I).                                            21
   executed play(A1,I) if                                   22
      play_for(A1,A2,I).                                    23
   caused player(A1,A2) after                               24
      play_for(A1,A2,I).                                    25
   ...                                                      26
```

Listing 2. K-specification of agents

Listing 1 shows some relevant features of the specification in language K of social interactions. In particular, it shows the declaration of fluents state_i, context_i and initiator, which represent, respectively, the run-time state of social interactions (either open or closed, values of the interaction_state predicate defined elsewhere), its interaction context and the possible initiator. Also shown is the declaration of the initiate action together with its non-executability conditions and effects (lines 14–19). These rules are declared within the scope of the `always` section, since they apply to every possible institutional state.

B. Agents

Agents are software components which interact through the social middleware as members of a given social interaction context, with the purpose of achieving some goal. In order to do so, they are empowered to perform social actions such as *setting up* new interactions, *joining* existing interactions, and so forth. For instance, the purpose of students is to pass the course to which they belong as members. In order to achieve this goal, students are empowered to set up working groups or to join existing ones in order to carry out some mandatory assignment. In case that the purpose of agents is too complex, its whole activity may be arranged in terms of a role-playing hierarchy of further agents. Thus, the activity of students within the context of working groups is represented by a new kind of agent role played by the course student. Agent roles are *played* and *abandoned* by the social middleware according to certain conditions. For instance, a course student role is automatically abandoned as soon as the agent passes the subject; a working group student is automatically created for the initiator of the working group, and for any student who successfully manage to *join* a pre-exiting working group.

Listing 2 partially shows the K-specification of the agent type, which includes the declaration of general fluents and actions shared by any kind of agent. Particularly, it shows the declaration of the fluents state_a, context_a and player, which represent the run-time state of agents (playing or abandoned), the social interaction context to which the agent belongs and its player agent, respectively. Also, it shows the declaration of the actions play and play_for. The former one causes some agent to be created within some social interaction context. The general specification only includes its non-executability condition and effect (lines 16–21). The action play_for causes some agent to be played (line 22) for a particular player agent (line 24).

C. Social actions

The activity of agent components within a multiagent society manifest itself through the performance of *attempts*. This external action allows an agent component to perform a given *social action*, namely to *say* something, *manipulate* the environment or *observe* the current state of some social entity. Due to lack of space, this paper will refer only to speech acts and, particularly, to declarations such as *set up* and *join*. Nevertheless, the processing of attempts by the social middleware is independent of the kind of social action. This process takes into account the *empowerments* and *permission* rules of the society. In particular, empowerments shall represent the institutional capabilities of agents, i.e. which social actions a given agent is capable of performing; permissions shall denote the circumstances under which these institutional capabilities can be exercised. Attempts by agent components are processed according to the following procedure:

- If the agent is empowered to perform the specified social action, then the attempt will be taken into account; otherwise, i.e. either if it is known for certain that the agent is not empowered, or it can not be concluded that it is empowered, the external action will be dismissed. In this latter case, the institutional state of the multiagent society will not be altered at all.
- If the agent is empowered to perform the action, but it is known that the specified performer is not permitted to perform it (i.e. it is prohibited), then the process is finished with a *prohibited* attempt status. On the contrary, if the agent is both empowered and permitted, then the social action is performed by the middleware. The effects caused through this execution depend on the kind of social action being performed.
- If the agent is empowered to perform that action, but it is neither known that the action is permitted nor prohibited, then the social action is kept in a *pending* state. This state will be eventually resolved into a *performed* or *prohibited* state as soon as it is known whether the action is permitted or prohibited.

This procedure is formalised as part of the *social action* type specification, whose major features are shown in listing 3. The signature of this specification includes the action declaration attempt(Act,A), which represents the attempt made by some agent component A to perform the social action Act. This action is exogenous, i.e. its cause is to be found outside the system being modeled; correspondingly, it is unconditionally declared as `executable` (cf. line 20).

The different scenarios described above concerning the processing of attempts are modeled through different groups of static and dynamic rules. Firstly, if the intended agent is empowered to perform the social action then it will be declared as its performer (line 22), irrespective of the permission status. Empowerments, permissions and the performers of social actions, are represented by the fluents empowered, permitted and performer, respectively. If the agent attempting to perform the social action is empowered then the social action will be brought about in the next state to one of three execution states (represented by the fluent state_sa): pending, prohibited or performed (the possible values of the type predicate social_action_state).

- Firstly, if it is known that the action is not permitted (i.e. −permitted(Act,A)) then the resulting state will be prohibited (line 24).
- Secondly, if it is permitted then the action will be performed (line 27). Execution of actions is represented by the action perform, whose only effect at this level of abstraction is the change in the run-time execution state (line 39).
- Last, if the social action is neither known to be permitted nor prohibited, then the social action is kept in a pending execution state in the resulting system state (line 30). Eventually, the circumstances may change in such a way that the social action is known to be permitted or prohibited. In those cases, the social action will be resolved to the execution or the prohibition states by the corresponding rules (lines 34–37). Otherwise, the social action will persist until the performer agent is abandoned (i.e. its run-time state is changed to abandoned) or the interaction context is closed.

```
fluents:                                                          1
   state_sa(Act,S) requires                                       2
      social_action(Act),                                         3
      social_action_state(S).                                     4
   context_sa(Act,I) requires                                     5
      social_action(Act), interaction (I).                        6
   performer(Act,A) requires                                      7
      social_action(Act), agent(A).                               8
   empowered(Act,A) requires                                      9
      social_action(Act), agent(A).                               10
   permitted(Act)                                                 11
      requires social_action(Act).                                12
    ...                                                           13
actions:                                                          14
   attempt(Act,A) requires                                        15
      social_action(Act), agent(A).                               16
   perform(Act) requires                                          17
      social_action(Act).                                         18
always:                                                           19
   executable attempt(Act,A).                                     20
   ...                                                            21
   caused performer(Act,A) after                                  22
      attempt(Act,A), empowered(Act,A).                           23
   caused state_sa(Act,prohibited) after                          24
      attempt(Act,A), empowered(Act,A),                           25
      −permitted(Act).                                            26
   executed perform(Act) if                                       27
      attempt(Act,A), empowered(Act,A),                           28
      permitted(Act).                                             29
   caused state_sa(Act,pending) after                             30
      attempt(Act,A), empowered(Act,A),                           31
      not permitted(Act), not −permitted(Act).                    32
   ...                                                            33
   caused state_sa(Act,prohibited) after                          34
      state_sa(Act,pending), −permitted(Act).                     35
   executed perform(Act) if                                       36
      state_sa(Act,pending), permitted(Act) .                     37
   ...                                                            38
   caused state_sa(Act,performed) after                           39
      perform(Act).                                               40
```

Listing 3. K-specification of social actions

IV. Forbidding and allowing social actions

Those social actions pending for execution will be resolved as prohibited or permitted attempts as soon as the rules of the society enables a definite conclusion on its permission status. As a complementary mechanism, particularly useful in the absence of general rules, run-time agents may also change the permission status through the speech acts *allow* and *forbid*[3].

[3]Of course, since *allow* and *forbid* are speech acts, their performance is also governed by the corresponding empowerment and permission rules. For instance, initiators of assignment groups are unconditionally empowered and permitted to allow other students to join their groups. In other application domains, however, it may happen, for instance, that some agent is required to allow other agent to allow some other agent to do something.

```
fluents:
   new_role(Join,A) requires
      join(Join), agent(A).
   ...
always:
   executed play_for(A1,A2,I) if
      join(Join), perform(Join),
      context_sa(Join,I),
      performer(Join,A2), new_role(Join,A1).
   ...
```

Listing 4. K-specification of the *join* social action

```
fluents:
   action(Allow,Act) requires
      allow(Allow), social_action(Act).
   ...
always:
    nonexecutable attempt(Allow,A) if
       allow(Allow), action(Allow,Act),
       not state_sa(Act,pending).
    caused permitted(Act) after
       allow(Allow), perform(Allow),
       action(Allow,Act).
    ...
```

Listing 5. K-specification of the *allow* social action

This section provides a formal account of the meaning of these speech acts and illustrates the formalisation of the assignment group scenario with the execution of a planning query. To account for a complete example, besides the *allow* and *forbid* speech acts, the *join* declaration will also be formalised.

A. Formalizing social actions

The specification of a new type of social action t_1 proceeds, firstly, by declaring a rule `social_action(x) :- t_1(x)`. This rule establishes that any entity of the new type shall be regarded as a social action, so that the rules which define the general structure and dynamics of social actions (cf. listing 3) are applicable for entities of that type. Secondly, new fluents representing the additional arguments of the new social action type must be declared. Last, new rules for representing the post-conditions of the performance of the new type of action, as well as their additional non-executability conditions, etc., have to be declared as well.

For instance, listing 4 shows the formalisation of the *join* declaration. By performing this speech act, the speaker declares that a new role is played within some interaction by it. The interaction and the speaker are represented by generic fluents of social actions, namely context_sa and performer. The new role to be played is declared as a new fluent, new_role, pertaining to this kind of speech act. The rest of the specification includes the particular effects associated to the execution of this kind of declaration, which are indirectly achieved through the internal action play_for (cf. listing 3).

Figure 5 shows the partial specification of the *allow* speech act. In this case, the generic social action specification is extended with the new fluent action_a, which represents the social action targeted by the *allow* speech act. The specification includes a condition which establishes that the social action to be allowed must be pending for execution (line 6). The effect of performing the *allow* action is to explicitly cause that the social action is permitted (9). The specification of the *forbid* speech act is similar to the one shown in listing 5. The only major difference pertains to its post-condition, which in this case resorts to the strong negation operator, i.e.

```
caused -permitted(Act) after
   forbid(Forbid), perform(Forbid),
   action_f(Forbid,Act)
```

```
initially:
   -has_state_sa(join1). -has_state_a(s21).
   -has_state_sa(allow1).
always:
   state_a(s1,playing). context_a(s1,top).
   state_a(s2,playing). context_a(s2,top).
   state_i(wg1,open). initiator(wg1,s1).
   state_a(s11,playing). context_a(s11,wg1).
   player(s11,s1). empowered(s2,join1).
   action_a(allow1,join1).
   empowered(allow1,s11). permitted(allow1).
goal:
   member(wg1,A), player(A,s2)? (3)
```

Listing 6. Working group scenario

B. Planning query

This section illustrates the semantics of the previous speech acts through a simplified implementation of the working group scenario. This implementation, shown in listing 6, features a consistent situation where the top-level interaction represents the university course to which two student agents, s_1 and s_2, belong as members. The university course has a single working group wg_1, previously set up by student s_1. The activity of this student within the working group is represented by the role s_{11}. The query posed to the $\texttt{DLV}^k$ planner (an implementation of action language K as a front end to the DLV answer set programming framework [9], [11]) asks for the possible ways in which the student s_2 may play a role within the working group wg_1, in exactly three planning steps.

The output of the $\texttt{DLV}^k$ planner is shown below. As expected, the first action that needs to be performed is an attempt by agent s_2 to join the working group. Two additional objects have to be declared in the scenario in order for this action to be performed: a *join* social action, $join_1$, and the agent to be played within the working group, s_{21}. These objects initially belong to the pool of objects which are available for the planning process[4]. Since the student is empowered to perform the join action but no permission rules are declared, the attempt to perform it results in a *pending* status. The next state features an attempt by agent s_{11} to allow the performance of the join action, namely to perform action $allow_1$. Since this

[4]Formally, these are objects which have no state, e.g. agents which are being neither played nor have been abandoned.

agent is both empowered and permitted to perform that social action, the permissions to execute the $join_1$ action are in effect in the next state. This, in turn, causes the performance of the join action and the consequent playing of the student agent within the working group.

```
STATE 0: state_a(s2,playing) state_i(wg1,open)
         empowered(s2,join1) new_role(join1,s21) ...
ACTIONS: attempt(join1,s2)
STATE 1: state_sa(join1,pending) performer(join1,s2)
         empowered(s11,allow1) permitted(allow1)
         action_a(allow1,join1) ...
ACTIONS: attempt(allow1,s11) perform(allow1)
STATE 2: state_sa(allow1,performed)
         state_sa(join1,pending) performer(join1,a1)
         permitted(join1) ...
ACTIONS: perform(join1) play_for(s21,s2,wg1) play(s21,wg1)
STATE 3: state_a(s21,playing) player(s21,s2)
         context_a(s21,wg1) ...
```

V. Discussion

The model of empowerment and permission put forward in this paper contrasts with other approaches based on ASP [3], [12], the event calculus [4] or action languages [2] in several respects. Firstly, the subjects of empowerments and/or permissions in these approaches are *events* which represent the observable or institutional actions to be evaluated. Moreover, normative fluents are boolean so that these events are evaluated in a single transition step either as permitted or prohibited. In contrast, permissions are applied in our framework to a particular kind of social *entity*, viz. social actions, which can be assigned a permitted and prohibited status, but also an unknown one. Thus, our framework does not force the designer of the computational society to add a complete set of permission rules.

Secondly, the strong negation operator allows the designer to explicitly declare prohibition rules, whereas other approaches have to resort to the default "everything which is not permitted, is prohibited", which may not always be adequate. In particular, explicit prohibitions are very convenient in order to represent necessary conditions of permission rules. For instance, the following rule states that a necessary condition to join some working group is that the deadline for submitting the corresponding assignment has not passed yet:

```
caused -permitted(Join) if
    context_sa(Join,W), working_group(W),
    assignment(W,A), deadline(W,D),
    current_time(T), D<T.
```

The assignment group scenario also served to illustrate a situation where empowerments and permission rules concerning a single type of social action are, respectively, complete and incomplete. This represents a good case in favour of the distinction between empowerments and permissions, which some approaches neglect (dispensing with one of the two notions).

Lastly, two normative social actions, *allow* and *forbid*, are smoothly introduced within the normative framework in order to handle those situations of incomplete normative knowledge. The semantics proposed for these actions is aimed at particular cases that can not be solved using the general normative knowledge of the society. This ad-hoc character tallies well with the natural language meaning of the corresponding English speech act verbs [13].

The normative framework reported in this paper is part of a larger research project aimed at the specification of a language for programming social applications [5], viz. software systems designed to support human interaction in arbitrary social contexts. This broad class of target applications include common online communities, but also other software systems deployed in more specialised settings such as business process management. This general goal partly explains some of the features of the proposed normative framework, such as its bias towards regimentation. Current work focuses on extensions to support commitments, an essential construct for many social application domains.

Acknowledgment

The authors would like to thank the anonymous reviewers for their detailed comments. Research sponsored by the Spanish MICINN, project TIN2006-15455-C03-03.

References

[1] A. J. I. Jones and M. J. Sergot, "A formal characterisation of institutionalised power," *Logic Journal of the IGPL*, vol. 4, no. 3, pp. 427–443, 1996.

[2] A. Artikis, M. Sergot, and J. Pitt, "Specifying norm-governed computational societies," *ACM Transactions on Computational Logic*, vol. 10, no. 1, 2009.

[3] O. Cliffe, M. D. Vos, and J. A. Padget, "Answer set programming for representing and reasoning about virtual institutions," in *CLIMA VII*, ser. Lecture Notes in Computer Science, K. Inoue, K. Satoh, and F. Toni, Eds., vol. 4371. Springer, 2006, pp. 60–79.

[4] N. Fornara and M. Colombetti, "Specifying artificial institutions in the event calculus," in *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, V. Dignum, Ed. IGI Global, 2009, ch. 14, pp. 335–366.

[5] J. M. Serrano and S. Saugar, "Run-time semantics of a language for programming social processes," in *9th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA IX)*, ser. Lecture Notes in Artificial Intelligence, M. Fisher, F. Sadri, and M. Thielscher, Eds., vol. 5405. Springer, 2009, pp. 37–56.

[6] D. Grossi, *Designing Invisible Handcuffs*. SIKS Dissertation Series No. 2007-16, 2007.

[7] M. Esteva, B. Rosell, J. A. Rodríguez-Aguilar, and J. L. Arcos, "AMELI: An agent-based middleware for electronic institutions," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, vol. 1, 2004, pp. 236–243.

[8] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner, "Nonmonotonic causal theories," *Artif. Intell.*, vol. 153, no. 1-2, pp. 49–104, 2004.

[9] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, "A logic programming approach to knowledge-state planning, ii: The dlv$^{\mathrm{k}}$ system," *Artif. Intell.*, vol. 144, no. 1-2, pp. 157–211, 2003.

[10] ——, "A logic programming approach to knowledge-state planning: Semantics and complexity," INFSYS Research Report, Tech. Rep. 1843-01-11, Oct. 2002.

[11] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Trans. Comput. Log.*, vol. 7, no. 3, pp. 499–562, 2006.

[12] M. Gelfond and J. Lobo, "Authorization and obligation policies in dynamic systems," in *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, ser. Lecture Notes in Computer Science, M. G. de la Banda and E. Pontelli, Eds., vol. 5366. Springer, 2008, pp. 22–36.

[13] A. Wierzbicka, *English speech act verbs. A semantic dictionary*. Australia: Academic Press, 1987.