Programming social middlewares through social interaction types

Juan Manuel Serrano
University Rey Juan Carlos
C/Tulipan S/N
Madrid, Spain
juanmanuel.serrano@urjc.es

Sergio Saugar
University Rey Juan Carlos
C/Tulipan S/N
Madrid, Spain
sergio.saugar@urjc.es

***Abstract*—This paper describes a type-oriented approach to the programming of social middlewares. It defines a collection of metamodeling features which allow programmers to declare the social interaction and agent types which make up the program of a multiagent society for some application domain. These features are identified and formalised taking into account a specification of social middlewares as programmable, abstract machines. Thus, the proposed approach results in the type system of an interaction-oriented programming language. The paper uses the C+ action language and the CCALC tool as formal devices, so that metamodeling features are given formal semantics as new social law abbreviations which complement the causal law abbreviations of C+. This programming language approach contrasts with the common modeling approach endorsed by organizational methodologies, and promotes higher levels of formality and reusability in the specification of multiagent societies.**

I. INTRODUCTION

Social middlewares are the responsible software infrastructures for the run-time management of software component interactions in computational societies. Unlike traditional object-oriented, service-based or messaging middleware approaches, social middlewares (e.g. AMELI [5], S-MOISE+ [11], MadKit [9], INGENIAS toolkit [8], etc.) provide software components with high-level *communicative* and *organizational* interaction mechanisms, which build to different extents on *normative* concepts such as empowerments, permissions, obligations, commitments, etc. It is claimed that the increased flexibility and expressiveness of these interaction mechanisms will result in a better management of component interactions in large-scale, multi-organizational, open distributed systems.

However, current social middlewares must overcome a number of shortcomings in order to achieve their full potential. Firstly, the set of generic interaction mechanisms which they are designed to support is not *extendable*, so that programmers are constrained to use the pre-defined abstractions (scenes, teams, groups, etc.) provided by the organizational metamodel of choice (e.g . ISLANDER [4], AGR [6], Moise+ [10], INGENIAS [13]). Secondly, applications can not be developed from generic, *reusable* modules which are specialised in the target application domain. Last, the run-time semantics of the organizational metamodels are not formally specified, which limits the *understandability* of the language constructs and the *portability* of the social middleware.

To address these limitations, this paper puts forward an approach to the programming of social middlewares which can be characterised along the following premises. Firstly, we build on a primitive and flexible notion of *social interaction* which attempts to provide the basic building blocks for the specification of any kind of communicative or organizational mechanism. Secondly, the social middleware is regarded as a *programmable machine* which is formally specified in technologically neutral terms as an abstract machine. Last, the behaviour of the social middleware is programmed through social interaction *types* which declare the characteristic structure and rules that govern the social interactions of the target application domain. The first two premises have been addressed in previous work, namely [15] and, respectively, [16]. The goal of this paper is to elaborate on the third premise. In particular, a set of metamodeling features for declaring social interaction types will be identified and their semantics formalised in terms of the underlying abstract machine. The chosen technique for formalising the overall approach is the action language C+ [7] and its accompanying tool CCALC [1]. In accordance with this election, metamodeling features will be represented as social law abbreviations which complement the standard set of causal laws provided by C+.

The rest of the paper is structured as follows. The next section briefly recalls the results shown in [16], namely it introduces social middlewares as programmable abstract machines. This section also describes, albeit briefly, the major features of the C+ formalisation and the conference management example used throughout the paper. Then, section 3 describes the partial set of metamodeling features of social interaction types, leaving their proper formalisation to the appendix of the paper. Finally, the paper concludes with a discussion of the major results with respect to competing paradigms, and a summary of current and future work.

II. SOCIAL MIDDLEWARE AS AN ABSTRACT MACHINE

The social middleware is in charge of managing the interactions between software components participating in a multiagent society. For instance, let's consider the development of an application to support the management of scientific conferences. In this setting, the social middleware is in charge of maintaining and driving the social processes which make up

the management of conferences (submissions, assignment and reviewing of papers, etc.). It does not deal with the actual job of software components that will engage in those processes at run-time[1], but with the management of processes themselves. This management is performed in accordance with the rules of conference management as specified by the types of social interactions that implement them. Before describing in the next section how to declare these types of interactions, this one summarizes the general structure of multiagent societies and the generic behaviour of the social middleware.

A. Social middleware structure

The interaction space of the middleware is shaped in terms of a tree of nested social interactions, so that the root of this tree represents the computational society being managed by the middleware. Besides social interactions, the computational society is made up of three major kinds of social entities: *agents*, *resources* and *social actions*. Agents and resources represent the two kinds of *roles* that can be played by software components attached to the social middleware within a particular social interaction [15]. These software components may be intelligent BDI components programmed with high-level languages (e.g. 2APL, Jason, etc.), or plain user interfaces (e.g. a web browser)[2]. Resources represent non-autonomous software components which store information and/or provide computational services to the society. On the other hand, agents represent autonomous components which purport to achieve some goal. The activity of agents within the overall society can be decomposed into a role-playing tree of further agents deployed in arbitrary social interaction contexts.

In order to achieve its purpose, the activity of some agent eventually resolves itself into the performance of social actions, namely saying something to other agents (i.e. *communicative actions*– CAs), calling the services of computational resources (i.e. *invocations*) and seeing the state of social entities (i.e. *observations*). CAs, invocations and observations are *atomic* interaction mechanisms which take place within the context of social interactions. This paper exclusively focuses on CAs, particularly on those pre-defined CAs which make up the *standard* CA library of the language. This library includes, amongst others, the declarations *SetUp*, *Close*, *Join* and *Leave*. The *SetUp* CA allows agents to declare that a new interaction of certain type be initiated by the middleware; *Close* is used by agents to force the middleware to finish some ongoing interaction; last, *Join* and *Leave* allow agents to become members of some interaction and, respectively, abandon a role currently being played by them.

Figure 1 represents a run-time snapshot of a social middleware for conference management. The following graphical conventions are followed: social interactions instances are

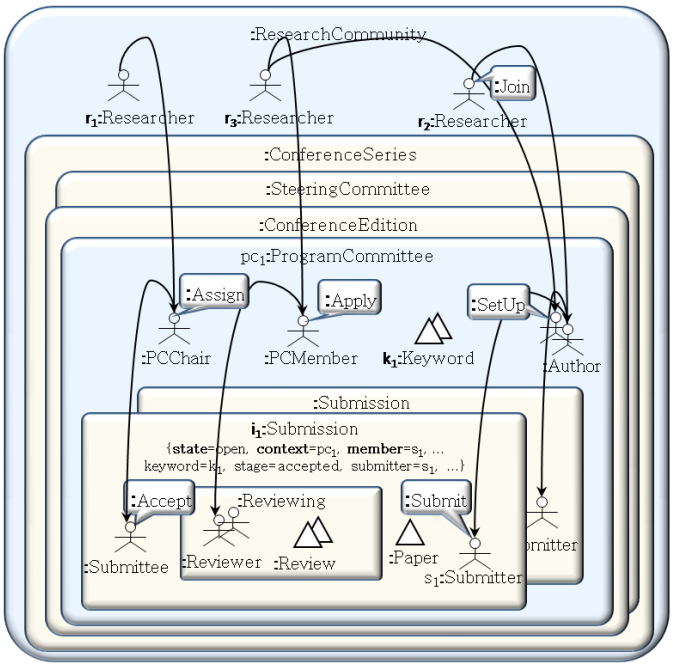

Fig. 1. Run-time snapshot of a multiagent society for conference management

represented by round corner rectangles, agents by stick figures and resources by triangles; last, speech bubbles represent the performance of communicative actions. A brief explanation of the different types of social entities involved follows in the next paragraphs:

- The root of the social interaction hierarchy is the *research community*, which is the context within which conferences actually take place. A given *conference series* is managed by its *steering committee*, which is responsible for the major issues concerning the different *conference editions*. The activity within a given edition is structured around the *program committee* (PC), *local committee*, etc. Within the PC, different *submissions* (one for each submitted paper) will take place, which in turn provide the context for the *reviewing* team in charge of evaluating that particular submission.
- The environments of the different interactions is made up of resources such as: the *paper* of submissions; the *reviews* uploaded within the reviewing team; the *calendar* and *keywords* created within the PC; and so forth.
- Three *researcher* agent roles and their corresponding role-playing hierarchies are shown in figure 1. The first one, r_1, plays the role of *PC Chair* within PC pc_1, which in turn behaves as *submittee* within the submissions of the PC. The second researcher, r_2, participates in the PC as author, who in turn plays the *submitter* role in two different submissions. The third researcher, r_3, participates in the PC both as *PC member* and *author*. As PC member it plays the role of *reviewer* in several submissions (typically, no more than three). Thus, agents may play several roles of different types within the same

[1]In this application, software components attached to the social middleware will typically consist of plain user interfaces (e.g. a web browser).

[2]Thus, the use of *agent programming languages* for the implementation of *agent components*, i.e. components playing some *agent role*, is not mandatory. The reader is warned about the particular notion of *agents as roles* (vs. *agent as components*) endorsed by this paper [15, section 2].

kind of context, and roles of the same kind in different contexts of the same type.

- Figure 1 also illustrates the performance of several communicative actions (in particular, declarations). For instance, researchers become authors of PCs by *joining* those interactions to play a role of that kind. Once they are authors, they may *set up* a submission in order to *submit* a given paper. PC members may *apply* for certain papers in order to express their reviewing preferences. Papers will be definitely *assigned* for reviewing by the PC chairs. Eventually, the PC chair, as submittee of a submission, may *accept* the submitted paper, i.e. declare the paper as part of the conference program.

B. Social middleware dynamics

As far as the *dynamics* is concerned, two kinds of major forces which influence the evolution of the society can be considered: *external actions*, performed by software components over the middleware; and *internal triggers*, mainly related to the life-cycle of social entities. With respect to the latter, the social middleware is responsible for checking the conditions which signal that some interaction must be automatically *initiated* or *finished*; that some agent must be *played* or *abandoned*; etc. For instance, once a conference edition is initiated, a program committee is automatically initiated by the middleware. Similarly, a new reviewer role within a particular submission is automatically created for a given PC member when the PC chair assigns this agent the corresponding paper to review. As these examples illustrate, the life-cycle management of social entities mostly depends upon rules declared by its particular types, as the next section will show.

External actions are the means whereby software components may *enter* the society as an agent to participate in some interaction; *exit* the society as the player of some agent, thus abandoning any further role; or *attempt* one of its agents to perform a given social action. This paper exclusively focuses on this latter kind of external action. The processing of attempts by the social middleware is driven by *empowerments* and *permissions* rules. Empowerment rules establish which agents are institutionally capable of performing some social action. Permissions, on the other hand, establish the circumstances in which these powers may be exercised. For instance, any researcher is empowered to join a PC as author, but this action is only permitted within the submitting stage of the PC. If some agent is not empowered to do some action, the corresponding attempt causes no change at all in the institutional state; if some agent is empowered but not permitted to do the action, the forbidden attempt is registered accordingly; last, if the agent is both empowered and permitted, the action is executed by the middleware. For instance, the attempt of an author to *join* a PC in its submission stage causes the internal action *play* to execute, which in turn causes the creation of the corresponding agent role.

C. Formalisation in the action language C+

The specification of the social middleware has been formalised using the action language C+. The reader is referred to [16] for a detailed explanation of the following discussion. The action description which defines the abstract social middleware infrastructure is structured around a collection of generic, application-independent sorts, which encapsulate the common structure and dynamics of social interactions, agents, resources and social actions. Thus, the generic sort $\mathcal{I}$, whose specification is partially shown in figure 2, declares the fluents and action constants which characterise the state and dynamics of any kind of social interaction. These *standard* or pre-defined state parameters include the following fluents: *state*, which represents the execution state of the interaction (*none*-existent, *open* or *closed*); the boolean fluents *member*, *env* and *sub*, which represent the member agents, environmental resources and sub-interactions of a given interaction; and the statically determined fluents *context* and *initiator* which represent the interaction context of the interaction and the agent who set up the interaction (if it was not automatically initiated by the middleware). Figure 1 shows the values of some of these attributes for the submission i_1. In particular, this interaction is open, its context is the PC pc_1 and has as member the submitter agent s_1. Moreover, the figure also shows the values of other non-standard attributes which are characteristic of submission interactions: the *keywords* of the submission, its *stage* (accepted, in this case) and the *submitter* agent.

Figure 2 also shows the declaration of the action constants *initiate* and *finish*, together with the laws that define the preconditions and effects of the latter action. Thus, according to law 1, the action *finish* causes an interaction i to be closed; law 2 establishes that this kind of action can not be executed if the specified interaction i is not open; and law 3 declares that this action is not executed by default, leaving the specification of particular sufficient causes to application-dependent types (as will be described in the next section).

:- sorts
$\mathcal{I}$; $\mathcal{S}_{\mathcal{I}}$.
:- objects
$open$, $closed$ :: $\mathcal{S}_{\mathcal{I}}$.
:- constants
state($\mathcal{I}$) :: **inertialFluent**($\mathcal{S}_{\mathcal{I}}$+$none$);
member($\mathcal{I}, \mathcal{A}$), env($\mathcal{I}, \mathcal{R}$), sub($\mathcal{I}, \mathcal{I}$) :: **inertialFluent**;
context($\mathcal{I}$) :: **sdFluent**($\mathcal{I}$+$none$);
initiator($\mathcal{I}$) :: **sdFluent**($\mathcal{A}$+$none$);
...
initiate($\mathcal{I}, \mathcal{I}$), finish($\mathcal{I}$) :: **action**.
:- variables
i, i_c,... :: $\mathcal{I}$.
/* laws */
...
finish(i) **causes** state(i) = $closed$. (1)
nonexecutable finish(i) **if** state(i)≠$open$. (2)
default ¬finish(i). (3)

Fig. 2. Partial specification of the generic social interaction type $\mathcal{I}$

III. Programming the social middleware

Programming a multiagent society consists of specifying the social interaction types which model the relevant social processes of the target application domain. The specification of social interaction types involves in turn the specification of their member agent types and environmental resource types, as well as their characteristic types of CAs. Thus, the implementation of a multiagent society for conference management is made up of the types of social entities identified in figure 1: the social interaction types *ConferenceEdition*, *ProgramCommittee*, *Submission*, etc., and their accompanied environmental resource types (*Paper*, *Review*, etc.), member agent types (*Researcher*, *Author*, *Submitter*, etc.) and characteristic communicative action types (*Apply*, *Assign*, *Submit*, etc.).

In order to identify the metamodeling features which allow to declare the different types of social entities, it will be convenient to recall the three major ways in which the social middleware can be programmed. Firstly, the programmer may extend the set of standard attributes of social entities to account for the particular characteristics of the application domain. Secondly, the programmer may specify the particular conditions under which the middleware must create and destroy social entities (i.e. initiate and finish interactions, play and abandon agents, etc.). Last, the programmer may declare the empowerment and permission rules which drive the processing of social action attempts. Consequently, three classes of metamodeling features will be considered in the declaration of social entity types: *structural*, *life-cycle* and *attempt processing* features.

Formally, social entity types are defined using the subsort mechanism provided by the input language of CCalc [1, section 3] and the generic C+ sorts which implement the structure and behaviour of the social middleware [16]. Thus, the definition of a social interaction type proceeds, firstly, by declaring a new subsort of the generic interaction sort $\mathcal{I}$ (figure 2); then, new fluent constants are declared which extend the definition of the generic sort; and, finally, new causal laws are provided which specify the structure and behaviour of the social middleware with respect to the new kind of social interaction. In particular, two kinds of causal laws can be used: those corresponding to the standard set of causal law abbreviations of the C+ language [7, appendix B]; and those defined by a new catalogue of *social law* abbreviations, which formalise the different metamodeling features of social entity types (partially listed and formalised in the appendix of this paper). Due to space limitations, the following subsections only introduce some of the devised social law abbreviations for the definition of social interaction and agent types.

A. Social interaction types

The metamodeling features of social interaction types will be illustrated with the specification of the *submission* interaction type, $\mathcal{S}$, shown in figure 3. To aid readability of the specification, social and causal laws are listed according to the life-cycle of interactions: firstly, those related to their initiation; next, those pertaining to their life-time; and, finally, those concerning their finishing.

:- sorts
$\mathcal{I} \gg \mathcal{S}$; $\mathcal{S}_{\mathcal{STAGE}}$.
:- objects
submitted, *accepted*, *rejected* :: $\mathcal{S}_{\mathcal{STAGE}}$.
:- constants
*/*inputs*/*
keyword($\mathcal{S}, \mathcal{K}$) :: **inertialFluent**;
*/*outputs*/*
crc($\mathcal{S}$) :: **inertialFluent**($\mathcal{P}$+*none*);
*/*local attributes*/*
stage($\mathcal{S}, \mathcal{S}_{\mathcal{STAGE}}$) :: **inertialFluent**;
paper($\mathcal{S}$) :: **inertialFluent**($\mathcal{P}$+*none*);
*/*aliases*/*
pc($\mathcal{S}$) :: **sdFluent**($\mathcal{PC}$+*none*);
submitter($\mathcal{S}$) :: **sdFluent**($\mathcal{SUBMITTER}$+*none*);
submittee($\mathcal{S}$) :: **sdFluent**($\mathcal{SUBMITTEE}$+*none*);
reviewingTeam($\mathcal{S}$) :: **sdFluent**($\mathcal{REV}$+*none*).
:- variables
s:: $\mathcal{S}$.
/* laws */
*/*Initiation laws*/*
input keyword(s, k). (4)
context s **is** $\mathcal{PC}$ **alias** pc. (5)
empowered a **to** setUp(s, pc) **if** member(a, pc). (6)
permitted a **to** setUp(s, pc)
if paperRegistration(calendar(pc))=t &
$\mathrm{t_{now}}$($clock$)<t. (7)

*/*Life-time laws*/*
member s **is** $\mathcal{SUBMITTER}$ **alias** submitter. (8)
member s **is** $\mathcal{SUBMITTEE}$ **alias** submittee. (9)
environment s **is** $\mathcal{P}$ **alias** paper. (10)
subinteraction s **is** $\mathcal{REV}$ **alias** reviewingTeam. (11)
empowered *submitter* **to** submit. (12)
permitted *submitter* **to** submit
if paperSubmission(calendar(pc))=t &
$\mathrm{t_{now}}$($clock$)<t. (13)
...
*/*Finishing laws*/*
empowered chair(pc(s)) **to** close(s). (14)
permitted a **to** close(s). (15)
finish s **if** stage(s, *rejected*) ++
state(submitter(s))=*abandoned*. (16)
output crc(s). (17)

Fig. 3. Submission interaction type $\mathcal{S}$

a) Structural features: The fluents declared for a new social entity type (i.e. not only social interactions) can be classified into four groups: *aliases*, *input*, *output* and *local* state parameters. Informally, the first group stands for those fluents which are introduced as aliases of standard fluents (e.g. *member*) to allow for more readable specifications. Input attributes are state parameters which must be set when the social entity is created. On the other hand, the meaning of output fluents directly refers to the destruction conditions of social entities: for instance, an interaction is automatically finished by the middleware when its output attributes are set. As for local fluents, these are normal C+ fluents whose meaning is established through common causal laws and

allow to simplify and improve the readability of other laws. The social law abbreviations `input` and `output` allow to specify newly declared fluent as input and output attributes, respectively; the `context`, `member`, `environment` and `subinteraction` abbreviations stand for the declaration of aliases pertaining to social interaction types. The formal definition of these abbreviations can be found in the appendix of this paper.

Thus, according to figure 3, a submission is modeled as a kind of process which receives as input a set of *keywords* (cf. law 4, which refers to variables s and k of the submission and keyword types, respectively), and has as goal the generation of the *CRC* (Camera Ready Copy) of a research paper (an output attribute, as declared by law 17). Thus, a submission can not be initiated without providing one keyword at least, and, as soon as the CRC of the paper is set the corresponding submission process will be automatically finished. On the other hand, the *stage* of the submission process is a local state parameter which holds the values *submitted*, *rejected* or *accepted*, as declared by the auxiliary $\mathcal{S}_{\mathcal{STAGE}}$ type. As for aliases, the member agents of a submission can be identified using the *submitter* and *submittee* fluents (besides the standard *member* fluent – see figure 2); similarly, the *paper* and *reviewingTeam* fluents stand for the environmental resources and subinteractions of the corresponding types; last, the *pc* fluent stands for the program committee context to which the submission belongs.

b) Life-cycle features: The specification of the generic social interaction type $\mathcal{I}$ establishes that the execution of the *initiate* and *finish* actions are disabled by default (cf. law 3). Moreover, sufficient condition for the execution of these actions are absent from the specification. Therefore, programmers have to provide these conditions when defining a particular type of interaction $\mathcal{T}$, unless type $\mathcal{T}$ is intended as an *abstract* type to be reused later in the definition of more specific types. Life-cycle conditions can be specified in two non-exclusive ways: directly, by defining new sufficient conditions for the *initiate* and *finish* actions; and indirectly, through the empowerment and permissions rules of the *SetUp* and *Close* standard CAs. The former approach allows the middleware to automatically initiate and finish interactions. The later provides agents with the possibility to force the execution of these internal actions. The social law abbreviations `initiate` and `finish`, defined in the appendix of this paper, allow to declare the social laws which allow the middleware to automatically govern the life-cycle of interactions.

For instance, submissions are only initiated if some author *sets up* an interaction of this kind within the program committee. Thus, these types of interactions are not automatically initiated by the middleware. On the contrary, a reviewing interaction is not set up by any agent but automatically initiated by the middleware when the stage of the submission is changed to *submitted*. Concerning finishing conditions, the submission process of some paper is automatically finished when its stage is set to *rejected* or the submitter agent is abandoned (law 16). A submission is also automatically finished when the CRC of the paper is set by the submitter, as described above (law 17). Besides these "normal" ways of finishing a submission, the PC chair is also given extraordinary power to prematurely *close* a submission.

c) Attempt processing features: The definition of a new type of social entity may encompass the definition of new types of CAs which somehow aim at modifying the overall state of its instances. For example, the stage of submissions is set through the execution of particular CAs executed by the author and PC Chair agents, namely *Submit*, *Accept* and *Reject*. Besides these domain-dependent CAs, the *SetUp* and *Close* CAs also affect the execution state of social interactions. The attempts to perform any of these actions is subject to the their empowerments and permissions rules, which are absent from the generic specification. Therefore, the programmer is provided with two new social law abbreviations, `empowered` and `permitted`, which allow to govern the middleware behaviour with respect to the processing of CAs targeted at social entities of the new type.

For instance, social laws 6 and 7 declare the empowerment and permission rules for setting up a new submission process according to the requirements established above. Similarly social laws 14 and 15 declare the corresponding rules for prematurely closing a submission. Concerning the *Submit* CA, social laws 12 and 13 establishes (1) that the submitter is the only agent empowered to submit the paper of the submission; and (2) that permission to submit the paper is granted if the deadline for paper submission did not pass.

B. Agent types

Metamodeling features for defining agent types will be illustrated with the specification of the *submitter* agent type, partially represented in figure 4. Due to space limitations, the formal specification of the new abbreviations for social laws introduced in this section are skipped in the appendix. Similarly, the discussion of structural features is omitted.

d) Life-cycle features: The `play` and `abandon` social law abbreviations allow programmers to declare the particular rules which govern the automatic playing and abandonment of agents of the defined type. For example, the definition of the submitter agent type exploit the former abbreviation in social law 18, which establishes that a submitter agent is automatically created for a given author if that author is the initiator of the submission process and no submitter has already being created; the purpose of this new agent, as specified by law 19, is to set the CRC of the submission (i.e. to publish the submitted paper through the conference program). Concerning automatic abandonment conditions, the submitter agent type does not introduce any specific rule besides the ones declared by the generic agent type $\mathcal{A}$ [16, section 5].

e) Attempt processing: The specification of empowerment and permission rules for new agent types employ the same abbreviations described in the last subsection for interaction types. The only difference lies in the kind of social actions pertaining to the specification: in this case the CAs *Join* and *Leave*. In the case of the submitter role, the creation of these

:- sorts
$\mathcal{A} \gg \mathcal{SUBMITTER}$.
:- constants
*/*aliases*/*
$\text{submission}(\mathcal{SUBMITTER})$ :: **sdFluent**($\mathcal{S}$+$none$);
$\text{author}(\mathcal{SUBMITTER})$ :: **sdFluent**($\mathcal{A}$+$none$);
...
:- variables
$submitter$:: $\mathcal{SUBMITTER}$.
/* laws */
*/*Playing laws*/*
play $submitter$ **for** a **within** s
if $\text{state}(s)$=$open$ & $\text{initiator}(s)$=a &
$\neg[\bigvee_{submitter} |\text{submitter}(s)$=$submitter]$. (18)
purpose $submitter$
is $[\bigvee_p |\text{crc}(\text{submission}(submitter))$=$p]$. (19)
...
*/*Abandonment laws*/*
empowered a **to** $\text{leave}(submitter)$. (20)
permitted a **to** $\text{leave}(submitter)$
if $\neg\text{stage}(s, accepted)$. (21)

Fig. 4. Submitter agent type $\mathcal{SUBMITTER}$

agents rely on the rules declared for automatic agent playing described above. Concerning its abandonment, author agents may prematurely leave one of its submitter roles, thereby causing the abandonment of the role and the cancellation of the submission (according to law 20). This power, however, may only be exercised if the paper has not already being accepted, as the permission law 21 specifies.

IV. Discussion

This paper has put forward a *type*-oriented approach to the programming of social middlewares. Essentially, this approach is characterised by using types (of social interactions, agents, resources and CAs) as modules which encapsulate those structural and behavioural rules of the multiagent society which pertain to social entities of a certain kind. Moreover, the identification and formalization of the metamodeling features used in the declaration of social types strongly builds upon the specification of the social middleware as an abstract, programmable machine. The overall approach can be thus characterised as a *programming language* approach. We opted to call the resulting language SPEECH, given the relevance of CAs in the overall architecture of the language.

This *interaction*-oriented language contrasts with and complements common *component*-oriented languages such as 2APL, AgentSpeak, etc., aimed at the development of intelligent BDI agent components. Conversely, SPEECH is closely aligned with the attempt at designing a programming language for organizational artifacts reported in [17], [3]. In contrast with this work, however, we place agent components outside the realm of the social middleware, which helps to ensure their full autonomy and heterogeneity. Another significant difference is related to the nature of roles. In particular, the SPEECH specification of agent role types is devoid of any kind of computational feature, so that agent role instances just represent the *public interface* of agent components within the multiagent society. On the contrary, positions in [17] (i.e. agent role instances) can execute plans to perform tasks delegated to them by their player agents. In fact, the specification of roles (i.e. agent role types) resorts to the typical constructs of BDI agent component languages. In our opinion, this blurs the distinction between agent components and agent roles, and undermines the separation of concerns between interaction and computation which lies at the heart of organizational programming languages. Last, besides agent roles the SPEECH language places a strong emphasis on social interaction types as a modularisation mechanism.

The SPEECH language is also closely related in spirit to common organizational metamodels for the specification of multiagent organizations such as ISLANDER [4], MOISE+ [12] and AGR [6]. Several methodological and conceptual differences, however, can be highlighted. Firstly, the programming language approach of SPEECH favours a higher degree of formality in the specification of the metamodel. Thus, in contrast to the common informal meanings of metamodeling constructs, the metamodeling features presented in this paper are grounded in the social middleware abstract machine presented in [16], and given formal semantics using the C+ action language. Thus, the proposal of this paper can be characterised as a first step towards the *type system* of a social interaction language, rather than as an organizational metamodel. Secondly, the SPEECH language places a strong emphasis on *specialisation* as a reusability mechanisms. Although some of the above-mentioned metamodels also support an inheritance relationship, this metamodeling features is at the core of the SPEECH specification. In fact, the operational semantics of the language is formalised through generic social types which are specialised in the specification of application-dependent types. In particular, the subsort mechanism of the C+ action language allows the programmer to override default laws of the super-sort, extend its signature with new fluents and actions, and/or refine the specification with new constraints. This strongly promotes the development of libraries of social types. For instance, an application-independent *submission* interaction may be defined, which could then be specialised for the particular case of paper submission. Similarly, other social interactions such as *invitations*, *discussion groups*, etc., may also be part of a generic library of social interactions, readily available for developers of arbitrary social process applications.

Current work deals with several extensions to the language to deal with obligations and sanctions (e.g. [3]), event processing, full-fledged communicative actions (e.g. [2]) and computational resources. Moreover, we aim to exploit the flexibility of the social interaction mechanism to define common metamodeling features of organizational metamodels (e.g. cardinalities and compatibility relations in MOISE+, performative structures in ISLANDER, etc.). In a more practical vein, current work also focuses on the implementation of a web-based social middleware for the language [14]. In this regard, the suitability of C+/CCalc for real deployments of multiagent societies is debatable. However, as a specification tool, they are of foremost importance to test in a techno-

logically neutral framework the features of the language. Moreover, they also provide invaluable help in the development of simple application prototypes. In particular, the conference management specification can be downloaded from http://zenon.etsii.urjc.es/~jserrano/speech/apps/c+apps.tgz.

REFERENCES

[1] Varol Akman, Selim T. Erdogan, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Representing the zoo world and the traffic world in the language of the causal calculator. *Artif. Intell*, 153(1-2):105–140, 2004.

[2] Guido Boella, Rossana Damiano, Joris Hulstijn, and Leendert van der Torre. A common ontology of agent communication languages: Modeling mental attitudes and social commitments using roles. *Applied Ontology*, 2(3-4):217–265, 2007.

[3] Mehdi Dastani, Nick Tinnemeier, and John-Jules Meyer. A programming language for normative multi-agent systems. In Virginia Dignum, editor, *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, chapter 16. IGI Global, 2008.

[4] Marc Esteva, David de la Cruz, and Carles Sierra. ISLANDER: an electronic institutions editor. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1045–1052. ACM Press, July 2002.

[5] Marc Esteva, Bruno Rosell, Juan A. Rodríguez-Aguilar, and Josep Ll. Arcos. AMELI: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, volume 1, pages 236–243, 2004.

[6] Jacques Ferber, Olivier Gutknecht, and Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. In *AOSE*, pages 214–230, 2003.

[7] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artif. Intell.*, 153(1-2):49–104, 2004.

[8] Jorge J. Gómez-Sanz, Rubén Fuentes-Fernández, Juan Pavón, and Iván García-Magariño. Ingenias development kit: a visual multi-agent system development environment. pages 1675–1676, 2008. May 12-16, 2008, Estoril Portugal.

[9] Olivier Gutknecht and Jacques Ferber. The MADKIT agent platform architecture. *Lecture Notes in Computer Science*, 1887:48–55, 2001.

[10] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Moise+: towards a structural, functional, and deontic model for mas organization. In *The First International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2002, July 15-19, 2002, Bologna, Italy, Proceedings*, pages 501–502. ACM, 2002.

[11] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. S-moise+: A middleware for developing organised multi-agent systems. In Olivier Boissier, Virginia Dignum, Eric Matson, and Jaime Simo Sichman, editors, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, volume 3913 of *LNCS*, pages 64–78. Springer, 2006.

[12] Jomi Fred Hübner, Jaime Simão Sichman, and Olivier Boissier. Developing organised multi-agent systems using the moise+ model: Programming issues at the system and agent levels. *IJAOSE*, 1(3/4):370–395, 2007.

[13] Juan Pavón and Jorge Gómez-Sanz. Agent oriented software engineering with ingenias. In V. Marik, J. Muller, and M. Pechoucek, editors, *Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems*. Springer Verlag, 2003.

[14] Sergio Saugar and Juan Manuel Serrano. A web-based virtual machine for developing computational societies. In Matthias Klusch, Michal Pechoucek, and Axel Polleres, editors, *Cooperative Information Agents XII, 12th International Workshop, CIA 2008, Prague, Czech Republic, September 10-12. Proceedings*, volume 5180 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2008.

[15] Juan Manuel Serrano and Sergio Saugar. Operational semantics of multiagent interactions. In Edmund H. Durfee, Makoto Yokoo, Michael N. Huhns, and Onn Shehory, editors, *6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), Honolulu, Hawaii, USA, May 14-18*, pages 889–896. IFAAMAS, 2007.

[16] Juan Manuel Serrano and Sergio Saugar. Run-time semantics of a language for programming social processes. In Michael Fisher, Fariba Sadri, and Michael Thielscher, editors, *9th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA IX)*, volume 5405 of *Lecture Notes in Artificial Intelligence*, pages 37–56. Springer, 2009.

[17] Nick Tinnemeier, Mehdi Dastani, and John-Jules Meyer. Roles and norms for programming agent organizations. In Decker, Sichman, Sierra, and Castelfranchi, editors, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 121–128, 2009.

APPENDIX

Abbreviation 1. An expression of the form "**input** $c(i, v)$ / $c(i)$.", where c is a boolean (non-boolean) fluent constant, whose first argument i is a variable of an interaction sort I_d and whose second argument (result) is of sort V, stands for the following causal law, where i_c and v are variables of the interaction sort $\mathcal{I}$ and sort V, respectively.

$$\textbf{nonexecutable } \mathrm{initiate}(i, i_c) \textbf{ if } \neg[\bigvee_v |c(i, v) \ / \ c(i) = v].$$

Thus, the resulting effect of declaring a new input parameter is the addition of a domain-dependent precondition to the *initiate* internal action (see figure 2).

Abbreviation 2. A set of expressions of the form "**output** $c_j(i)$.", where $j \in \{1 \dots n\}$, i is a variable of an interaction sort I_d, and c_j are non-boolean, optional constants of sort S_j, stands for the following action dynamic law, where v_j are variables of sorts S_j

$$\textbf{caused } \mathrm{finish(i)} \textbf{ if } \mathrm{state(i)}=\mathit{open} \ \wedge \ \bigwedge_{j\in\{1\dots n\}} [\bigvee_{v_j} |c_j(i) = v_j].$$

Thus, a set of expressions "**output** c_j." implicitly establishes sufficient conditions for the execution of the *finish* standard action.

Abbreviation 3. A set of expressions of the form "**member** i **is** A_j **alias** $\mathrm{f_j}$.", where $j \in \{1 \dots n\}$, i is an interaction variable of sort I_d, A_j a collection of agent sorts, and f_j a set of boolean (non-boolean) binary (unary) fluent constants whose first argument is of sort I_d and its second argument (*optional*) result sort is A_j, stand for the following set of causal laws, where a_j are variables of sort A_j and a is an agent variable of sort $\mathcal{A}$

$$\begin{array}{l}\textbf{constraint } \mathrm{member}(i, a) \rightarrow \bigvee_{j\in\{1\dots n\}}[\bigvee_{a_j} |a = a_j].\\ \textbf{caused } f_j(i, a_j) \ / \ f_j(i){=}a_j \textbf{ if } \mathrm{member}(i, a_j).\\ \textbf{caused } \neg\ f_j(i, a_j) \ / \ f_j(i){=}\mathrm{none}\\ \quad \textbf{if } \neg\ \mathrm{member}(i, a_j) \ / \ \neg\ [\bigvee_{a_j} |\mathrm{member}(i, a_j)].\end{array}$$

Thus, the introduction of a new alias has also the intended meaning of constraining the types of agents that can be members of the interaction. The meaning of `context`, `environment` and `subinteraction` aliases can be similarly formalised.

Abbreviation 4. Let i be a variable of an interaction sort $\mathcal{I}_d$. The expression "**finish** i **if** F." stands for the action dynamic law:

$$\textbf{caused } \mathrm{finish}(i) \textbf{ if } \mathrm{F}.$$

Thus, the expression "**finish** i **if** F." is simply a wrapper of the corresponding action dynamic law which enacts the execution of the *finish* internal action. A similar abbreviation may be defined for declaring the automatic initiation of interactions.

Abbreviation 5. Let a and α be agent and institutional action variables. The expression "**empowered/permitted** a **to** α **if** F." stands for the static law

$$\textbf{caused } \mathrm{empowered/permitted}(a, \alpha) \textbf{ if } \mathrm{F}.$$

Thus, this social law abbreviation is just a wrapper of the static law which defines the predefined *empowered/permitted* fluent for the corresponding social action and agent sorts. The subexpression "if F" in the proposed abbreviations may be dropped if F is true.