SDML: A Multi-Agent Language for Organizational Modelling

Scott Moss, Helen Gaylard, Steve Wallis and Bruce Edmonds
Centre for Policy Modelling
Manchester Metropolitan University
{s.moss | h.gaylard | s.wallis | b.edmonds}@mmu.ac.uk

Abstract

A programming language which is optimized for modelling multi-agent interaction within articulated social structures such as organizations is described with several examples of its functionality. The language is SDML, a strictly declarative modelling language which has object-oriented features and corresponds to a fragment of strongly grounded autoepistemic logic. The virtues of SDML include the ease of building complex models and the facility for representing agents flexibly as models of cognition as well as modularity and code reusability. Two representations of cognitive agents within organizational structures are reported and a Soar-to-SDML compiler is described. One of the agent representations is a declarative implementation of a Soar agent taken from the Radar-Soar model of Ye and Carley (1995). The Ye-Carley results are replicated but the declarative SDML implementation is shown to be much less computationally expensive than the more procedural Soar implementation. As a result, it appears that SDML supports more elaborate representations of agent cognition together with more detailed articulation of organizational structure than we have seen in computational organization theory. Moreover, by representing Soar-cognitive agents declaratively within SDML, that implementation of the Ye-Carley specification is necessarily consistent and sound with respect to the formal logic to which SDML corresponds.

Keywords: simulation, organization, computational model, formal logic, cognition

Contact author:

Scott Moss
Director
Centre for Policy Modelling
Manchester Metropolitan University
Aytoun Building
Manchester M1 3GH
UNITED KINGDOM

telephone	+ 44 161 247 3886
fax	+ 44 161 247 6802
email	s.moss@mmu.ac.uk

SDML: A Multi-Agent Language for Organizational Modelling

Scott Moss, Helen Gaylard, Steve Wallis and Bruce Edmonds
Centre for Policy Modelling
Manchester Metropolitan University
{s.moss | h.gaylard | s.wallis | b.edmonds}@mmu.ac.uk

1 Introduction

Implementors of computational models of organizations confront a trade-off between the sophistication of the representation of individual cognition on the one hand and the complexity of the modelled organization on the other. A more elaborate organizational structure is associated in the literature with a simpler representation of individuals. We introduce in this paper a programming language, SDML, which supports fast development of models and, for any representation of cognition, can entail more complex representations of organizations than we have seen previously in the literature.

SDML ("strictly declarative modelling language") has been designed to facilitate flexible multi-agent modelling of organizations. Unlike modelling architectures such as Soar, which incorporates a specific cognitive theory in its agent architecture, SDML is a theory-neutral programming language. However, the requirements underlying its development, and the features whereby these are realised, mean that SDML can easily be used to represent either simple or sophisticated agents and the nature of the social relations which exist among them. Implementing models in SDML does not preclude the use of Soar's particular problem-solving architecture. Because that architecture has proved to be useful in a wide range of computational models, Soar agents are being implemented in SDML.

Soar represents agents as implementations of Newell's (1990) unified theory of cognition and models written in Soar have specified at most two levels in a hierarchical organization (for example, Ye and Carley, 1995). A model in TASCCS (a "synthesis" of Double-AISS and Plural-Soar) by Verhagen and Masuch (1994) was restricted to two agents due to the implementational limitations of Soar. Carley and Svoboda (1996) represent agents by a simulated annealing algorithm or by a stochastic learning process specifically to get round the computational restrictiveness of Soar. So and Durfee (1996) represent organizations as general tree structures comprised of homogeneous agents transmitting message packets to one another.

Tambe and Rosenbloom (1996) suggest that a limitation on the complexity of computational models of agent interaction is the agent architecture. For this reason, they extend the Soar architecture to support the implementation of agents who build models of other agents whose behaviour they must track in real time knowing that they are themselves being tracked and modelled by the other agents. Tambe and Rosenbloom have not reported any models with the sort of hierarchical relationships that are essential to representing agents acting within organizations.

The features of SDML's architecture that support multi-agent modelling are described in section 2. In section 3, we describe an implementation of abstract organizations with extensive multi-agent interaction. We describe in section 4 how full Soar cognition is implemented in SDML and in section 5 we report an SDML implementation of a well known Soar model, Ye and Carley's (1995) Radar-Soar. We demonstrate that the declarative basis of SDML not only relates implementations of cognitive models to formal logic but the features of SDML that establish that correspondence also make the SDML implementation of Soar cognition much less computationally expensive.

2 SDML's Multi-Agent Features

A number of basic requirements underlie SDML's multi-agent features. Individual agents must be able to incorporate rules determining their behaviour, including any or no cognitive theory. For these purposes a declarative representation is appropriate because it enables us to capture the distinction between behaviour and its underlying explanation. This distinction is particularly salient with respect to social phenomena of an inherently emergent nature. The ability to share rules among similar agents is convenient and this is enabled by SDML's object-oriented features. For multi-agent applications, we require the flexibility to represent such structures as organizations with arbitrarily deeply nested levels of agents. Agents must be able not only to communicate with each other but also to maintain privacy in the sense of restricted access to information. We further require the ability to represent organizations as dynamic structures which change over time. Because SDML is *strictly* declarative clauses once asserted to a database may not be retracted. However, change is easily modelled using time levels, an in-built feature of SDML whereby different databases are associated with agents at different periods in time.

2.1 Syntax of Agent Rules

A rule in SDML has *antecedents* and *consequents*. It may be read declaratively as stating that, if the antecedents are true, then the consequents are also true. Both antecedents and consequents consist of *clauses,* which may be conjoined, disjoined, or negated in the case of antecedents, and conjoined in the case of consequents[1]. A clause consists of a *functor* and a number of *arguments*. Typically the arguments in rules are variables which are unified with ground terms and thus instantiated when rules fire. Where the antecedents of a rule contain variables, the rule specifies that the consequents are true for all bindings of the variables such that the antecedents are true.[2]

The object-oriented features of SDML require that terms used as functors and arguments must be defined as objects. Clause definitions specify a functor name and the number of associated arguments and type restrictions for each. SDML's types are discussed below. Clauses are also specified to be *internal*, *private,* or *public.* This is discussed below in the context of agent communication. Finally, clauses are defined to be either forward- or backward- chaining.

Corresponding to forward- and backward-chaining clauses in SDML are forward- and backward-chaining rules which have these clauses in their consequents. Forward-chaining rules are primary, and agents may have a number of forward-chaining rulebases corresponding to different time levels specified by the user. We will return to the topic of time levels below. When forward-chaining rules fire, their consequents are asserted to a database, there being a database corresponding to each such rulebase. Each agent has a single rulebase for each backward-chaining clause definition. Backward-chaining rules are effectively procedures[3] called by the antecedents of forward-chaining rules. Important uses of backward-chaining rules are list-processing procedures similar to those in Prolog and for substituting a mnemonic clause for more cumbersome but related groups of clauses in forward-chaining rules.

SDML's rules are fired using forward chaining by retrieving clauses from databases in order to determine the bindings of variables such that the antecedents are true, substituting those bindings in the consequents, and asserting the resulting clauses to databases. The order in which rules are fired is not specified by the user, but is determined automatically on efficiency grounds. SDML orders the rules according to dependencies among them, dividing the rulebase into partitions whereby rules in a later partition are dependent upon, rather than determiners of, rules in an earlier one.

---

1 Conjunction of consequents applies to forward- but not backward-chaining rules.

2 The semantics are slightly different when the notInferred primitive is used.

3 A procedure in this context is to be understood as a collection of rules or clauses rather than a sequence of actions. Consequently, these features do not make SDML in any sense a procedural language in the computer science sense.

2.2 Object-Oriented Features

SDML supports inheritance through two principal hierarchies.

The type hierarchy supports multiple inheritance. The standard type hierarchy, which is extended by the programmer, is shown in Figure 1. The user adds further subtypes, in particular, subtypes of Object, Agent and SDML's predefined Agent subtypes. The type Agent is distinguished from Object in that it has rulebases associated with it. The number of such rulebases varies with the time levels defined by the user. Time levels are discussed in several contexts below.

Agent is the principal type of interest here. Models are specified in terms of instances of agents but these will not normally be instances of the type Agent but of a user-defined subtype of Agent or one of its predefined subtypes (or of more than one of these). Clause definitions and rules are specified in types and are inherited from them by their instances. In this way, the rules for a number of identical agents can be defined in a shared type. Similarly, agents which are not identical may nevertheless share certain rules by means of a common supertype.

.

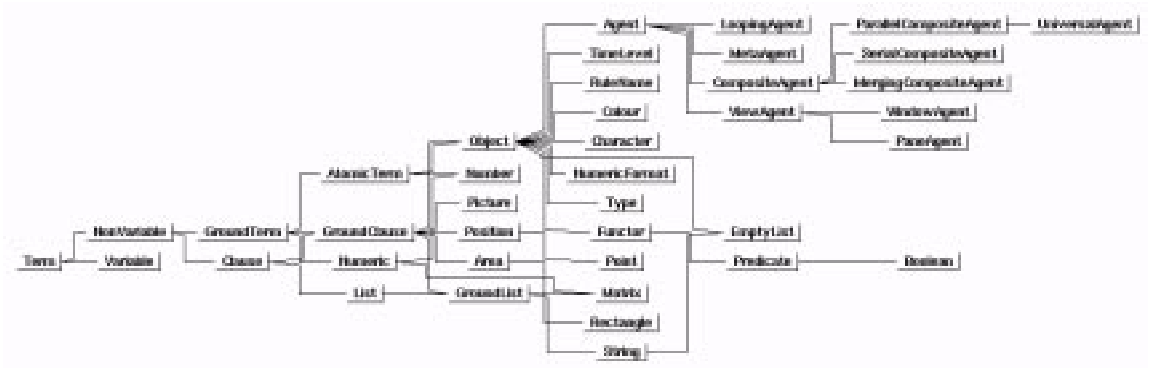


Figure 1: The Basic SDML Type Hierarchy

We will mention the uses of a number of the predefined Agent subtypes here. Composite Agent and its subtypes facilitate the representation of multi-agent structures, as discussed in more detail in the following section. Looping Agents iterate over time and thus any type of agent which contains rulebases for user-defined time levels will have this as one of its supertypes. Meta Agents have the ability to write to rulebases as if they were databases, and thus these provide one means of implementing agents which learn.

2.3 Representing Multi-Agent Structures

The structure of multiple agent models is represented in SDML by the container hierarchy; for instance, persons may be contained within departments contained within firms. The outermost container is always the *universe*. The container hierarchy is related to the type hierarchy principally by Composite Agent and its subtypes. The type Composite Agent allows for the representation of agent hierarchies of arbitrary depth, since any agent within a Composite Agent may itself be a Composite Agent. A type specifies the type of its container and inherits some clause definitions via the container hierarchy[4]. SDML's in-built predicates allow for specification in the rulebases of Composite Agents of those subagents which are active (i.e., for which the rulebases will be fired) and, in the case of Serial Composite Agents, the order in which subagents are activated.

The subtypes of Composite Agent are Serial, Parallel, and Merging Composite Agent. At any time level, the rulebase of the container fires prior to those of its subagents. Serial represents perhaps the most straightforward kind of structure within which subagents are ordered so that each agent acts after, and on the basis of any information made available by, preceding agents. The idea of a Merging Composite Agent is that the rulebases of the agents are merged and the dependencies among the rules calculated as if they were in a single rulebase. This allows extremely flexible and fine-grained interaction since it makes communication among the merged agents effectively simultaneous. In a Parallel Agent, the subagents act at a particular time level with knowledge of the results of others' actions at the previous time but in ignorance of others' current actions. It is in these circumstances that agents will want to generate models of each other. Parallel is the type of agent which is primarily of interest for modelling cognition in a social context.

In a nested parallel structure there will often be a corresponding layer of time levels. For example, corresponding to firms, departments, and persons, there may be years, months, and days. Each agent at each level in the container hierarchy may have rulebases for each time level, but it is likely that there is a bias towards less frequent decisions and actions at the firm level and towards more frequent ones at the level of individual persons. A common feature reflected in the Parallel Composite Agent type is uncertainty about others' actions at the current time, and the duration of the current period of uncertainty

[4] The differences between this and inheritance via the type hierarchy are detailed in the following section.

may be transparently represented in the choice of appropriate time level, corresponding to frequency of interaction. The easy representation of such features is important as they give rise, in part, to the need for cognition in a social setting.

2.4 Communication among Agents

In SDML agents normally communicate by writing the results of their rule firing to their databases or those of a container and reading the results of another agent's rule firing from that agent's database or that of a shared container. Accessibility restrictions permitting, agents can also read from or write to other agents' databases using explicit addressing. The default database to which a clause is asserted when a rule fires depends upon where that clause is defined. If it is defined in the agent's type or a supertype (i.e., inherited via that type hierarchy) then it will be asserted to the agent's own database. However, if the clause definition is inherited via the container hierarchy then it will be asserted to the database of the container where it is defined. It is often convenient for agents to share information via the container as this does not require explicit addressing. This is especially so where the structure of the organization may be changing, as discussed in the section 2.5.

Uncertainty concerning other agents' actions was mentioned above as an important feature which we require models to be able to represent if they are to be adequately expressive with respect to social phenomena. Related to this, SDML allows for different accessibility for agents to other agents' databases. For example, some decisions and actions made by a firm may never be directly accessible to other firms and, similarly, some actions of individuals within it may never be accessible to the firm. Clauses may be defined as *private* to the agent or *public*, or, intermediate between these, *internal* to the defining container, e.g., accessible to all agents within a firm but none outside it. There is also the facility to, for example, make an agent's clauses publicly readable but not writeable.

2.5 Modelling Change

Time levels have already been mentioned, in particular in the context of their function of enabling the user to specify appropriate frequencies of interaction which may vary for different agents within the same model, and, indeed, for the same agent in different roles. Here we will describe time levels in more detail and discuss their role in modelling organizations as dynamic structures.

Time Level is an in-built type in the SDML type hierarchy. The default Time Level is *eternity* which is equivalent to the time it takes for all the forward-chaining rules (including any backward-chaining procedures they call) in the universe to fire. The user is free to specify any number of instances of Time Level and to associate any or none of these with each Agent subtype used in a model. For each of its time levels, a type will have an associated initial and final forward-chaining rulebase as well as the content rulebase which fires iteratively between the initial and final rulebases at the lowest time level. Each forward-chaining rulebase in turn has an associated database on which the results of rule-firing are (by default, in the absence of explicit addressing) stored. Each rulebase need not contain any rules or could contain merely a simple forward-chaining rule devised to call a complex backward-chaining procedure.

We illustrate the order in which rulebases fire with a simple example. Suppose an agent has time levels week and day[5], with 12 weeks and 7 days specified as the duration for each of these time levels. Its initial week rulebase fires at week one, followed by the initial day rulebase on day one, followed by the content rulebase for each remaining six days, additionally the final day rulebase on day seven, followed by the initial day rulebase for day one of week two, and so on, until the final week rulebase fires at the end of week twelve[6]. Where there are nested levels of agents, for any time level the initial rulebase of a container fires before those of its subagents, then, similarly, the content rules of the container are followed by those of subagents, but, however, the final rulebases of subagents precede those of the container.

The distinctive contribution of time levels to organization modelling is in making it straightforward to represent changing structures and relations. Recall that, while a type defines its container type, subagents are only rendered active by clauses on the container's database. Clauses can be asserted to be true either permanently or for any duration in accordance with the time levels defined for a model; for example, all day (equivalent to every hour) or every day (equivalent to all week). This means that, where the organization is regarded as static, each container's subagents can be declared by asserting permanent clauses at the start of a simulation. However, where volatility rather than stability is the

5 The labels attached to time levels are arbitrary and thus need not be naturalistic. For instance, a genetic programming agent reported by Edmonds, Moss and Gaylard (1997) has been programmed using the time levels generation and stage. This co-exists in a model with other agents which use natural time levels corresponding to observations from statistical data series.

6 We deliberately omit mention of eternity here in order to simplify the example!

norm, it is possible to update the structure every day, reasserting the clause that was true on the previous day only where no change is known to have taken place.

3 Modelling Organizations in SDML

In describing SDML's container hierarchy, we used the example of firms containing departments which in turn contain persons. However, because SDML is theory-neutral, it allows us to reject the idea that a firm or department can be, like a person, a cognitive agent. In modelling organizations in SDML, we represent only our lowest-level agents as cognitive agents with rules for making decisions[7]. Hierarchical structure is represented, most importantly from the modelling point of view, not by correspondence with the container hierarchy, but by the different kinds of interactions which take place between different kinds of agents. Similarly, agents are identified by the rules associated with their Agent subtype (e.g., Department Manager) as well as by their position in the container hierarchy.

In order to illustrate the application of these modelling principles, we describe an abstract representation for organizations which has been implemented in a model in SDML. This illustrates how the distinction between the formal vertical hierarchy for the organization and its horizontal structure, constituted by its business processes, may be captured. We assume that, while the formal vertical hierarchy is fixed, the initial, formally imposed horizontal structure comes to be superseded through the evolution of informal business processes. These come to dominate to the extent that they better enable the organization's agents to meet their goals.

3.1 Representing an Organization in SDML

The formal vertical structure of the organization is represented in SDML by the container hierarchy. The user is asked to specify the number of layers in this hierarchy as well as the minimum and maximum span of control (i.e., number of subagents) for each department. Each department is a Parallel Composite Agent containing departments or, in the case of the lowest level in the department hierarchy, business processes as subagents. Business processes are represented as simple Agents, which we term activity agents, and are the only decision-making agents within the organization[8].

[7] It is not always practical, however, for the lowest-level agents to correspond to individual persons.

[8] We assume that a business process at the lowest level may be carried out by a single person. We do not claim that this is a realistic assumption.

The initial horizontal structure of the organization is created randomly but within constraints imposed by the vertical structure. Within each lowest level department pairs of activities are combined to form composite activity agents (represented as simple Agents), which are further combined in this binary fashion until a top-level composite activity for the department is created. The top-level activities for each department sharing a common containing department are then combined by means of the same processes to create the top-level activity for this higher-level department. This process continues up the organizational hierarchy until there is a complex composite activity, or business process, for the organization as a whole.

A measure of the output, or performance, of business processes is determined by a representation of the technological possibilities which uses the features of SDML to ensure that the same basic technology is available to all organizations even though the use of the technology and the efficiency of the activities depending upon that technology will differ according to its use and development by agents within each organization. A detailed description is contained in Appendix 1. For the purposes of the present exposition, the following summary will suffice:

Each composite activity is labelled with a number. This number is a factor used to multiply the sum of the values of the component activities of the composite. This factor will be the same for equivalent composite activities in all organizations. The product of that factor and the sum is the value of the composite activity. In order to value every activity, it remains only to determine the values of each elementary activity. These values are determined by a network search by the activities themselves. The network search is a representation of learning or R&D that is undertaken to improve productivity or reduce costs whether by experience or purposive research[9]. Activity agents incorporate the network search model of R&D reported separately by Moss, Edmonds and Gaylard (1996). Because of the object oriented features of SDML and its modular structure, the module from the R&D module defining the network searching agents could be inserted without change into the organization model reported here.

[9] Activity agents incorporate the network search model of R&D reported separately by Moss, Edmonds and Gaylard (1996). Because of the object oriented features of SDML and its modular structure, the module from the R&D module defining the network searching agents was inserted without change into the organization model reported here.

3.2 Dynamic organizational processes

This model has been devised to simulate the informal development of business processes within the organization. It is assumed that each process looks for a partner with which to combine in order to improve the value of their joint activity relative to the value of existing joint activities. As noted above, the value of the joint activity will be determined by the values of the individual activities and the combining factor. These structures are dynamic because the basic level of productivity associated with each process changes each period as a result of the enhancements generated internally within the atomic business processes and represented here as network search.

The process which drives the dynamic processes is the search by individual agents for alternative activities with which they can combine to produce more value. Agents' search is not constrained by the formal departmental structure of the organization. One effect of formal organizational structure is to impose some order on the combination of activities. Activities undertaken within one department engage in some level of cooperation before cooperating with the activities in another department. But the individuals who undertake the activities might prefer to combine in a different order than that facilitated and supported by the formal structure and procedures. For example, more efficient production might result if design teams were to consult with production teams before bringing in the sales team even if the design and sales teams were within one division of the organization and production in another. Alternatively, goods which have a commercially more successful design might result if the design and sales teams were to collaborate before bringing in the production team. The purpose of the modelling techniques is to support the design of procedures and organizational structures which will enable actual organizations to investigate such alternatives in order to improve on existing arrangements with minimum disruption to the organization's ongoing activities.

The process of informal organizational change captured by this model requires individual activities to find other activities with which to combine. We do not assume that activities know the whole of any abstract activity tree but that they can observe other activities with which combination is feasible and communicate with them. They can ask each other the values of their respective activities but they do not know the factor which multiplies the sum of those values to give the value of the combined activity.

Though we have not implemented this possibility, there is nothing in the structure of the model to prevent agents from lying about the values of their own activities.

Consequently, each activity will have to decide whether a potential collaborator is honest and reliable as well as valuable. This will involve communication among the activities as well as evaluation of experience with collaborators.

3.3 Communication among agents

Because the model, the organizations and the departments are all parallel composite containers, the activities in all departments of every organization are effectively acting in parallel. One agent communicates directly with another by means of an at-clause in the consequent of an instantiated rule. For example, if activity-1 wants to cooperate with activity-2, the at-clause asserted by activity-1 would be

```
at activity-2 (cooperationOfferedBy self)
```

Activity-2 would not be able to read this clause until the following time period because agents acting in parallel cannot perceive the results of any other agents' actions in the same period. In the next time period, activity-2 would find

```
last (cooperationOfferedBy ?anActivity)
```

unified the variable `?anActivity` with activity-1. An agent that always responded to an offer of cooperation by accepting it would have a rule with the above clause as its antecedent and, as a consequent,

```
at ?anActivity (cooperationOfferedBy self)
```

If the offer by activity-2 had remained standing, then in the third time period both activities would know of the offer and the acceptance and could note that the collaboration had been agreed.

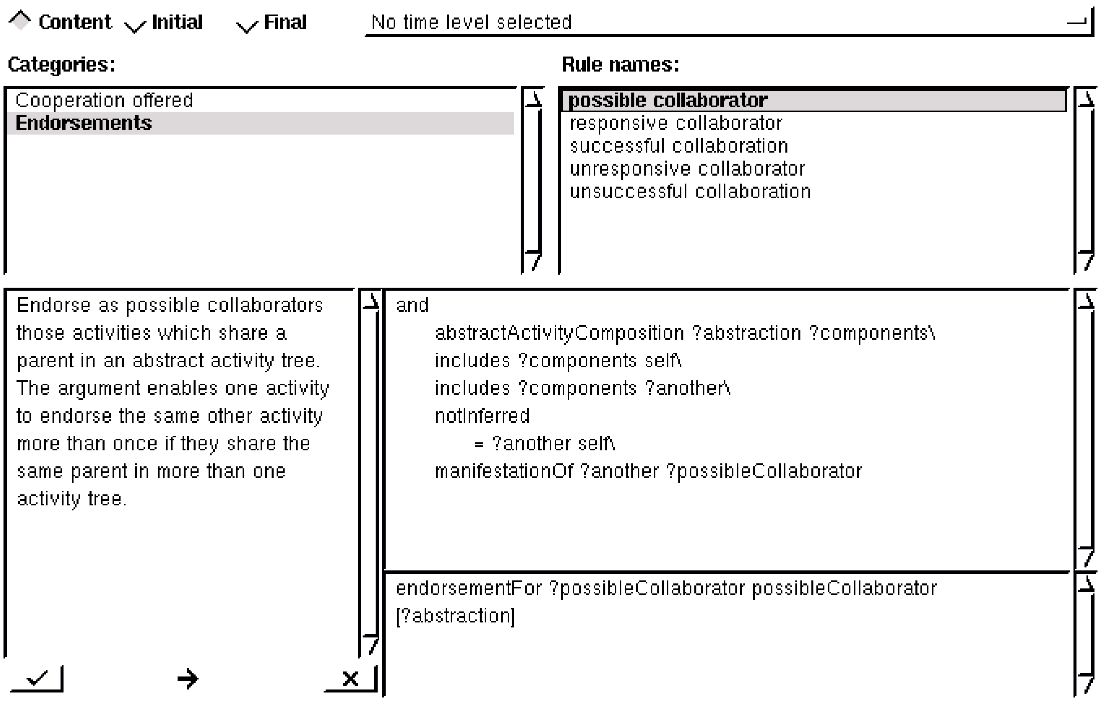
In general, an agreement to collaborate takes at least two and normally three time periods to achieve among agents acting in parallel. In the first period, the first agent offers collaboration to the second. In the second period, the second agent perceives the offer of collaboration and signals acceptance. In the third period, each agent knows that the other wants to collaborate and the collaboration is ratified. If, by chance or design, each of two agents were to signal a desire to collaborate with the other at the same time, then the ratification could take place in the following period.

In order to allow cooperation or collaboration agreements to be made within a single "date" of the model, two time levels were defined: the period and the negotiatingRound. The period is defined by the model agent and, so, applies to all agents contained in the model. The negotiatingRound is defined by the organization agent and applies to itself and all of its subagents which are, in this case, departments and activities. This allows for communication among activities within organizations to take place between the occasions on which the outputs from that organization are notified to the wider environment. In real organizations, of course, communication and, say, production and sales are all going on at the same time. The point of the negotiating rounds is to get more communication relative to other activities than would be the case if the communication and other activities all took place at a single time level. The model was implemented with these time levels so that the activities could form collaborative arrangements and assess them in an environment where the effectiveness of such an arrangement took some time to assess. The period is therefore an accounting period with informal arrangements arising within the accounting period. If, by way of contrast, it would be interesting to assess the effect of production information on the negotiations leading to collaborative arrangements, then the period could be set to take place within the negotiating round by changing the consequent of the time-level-determining rule to timeLevels [negotiatingRound period] from timelevels [period negotiatingRound].

3.4 Representing cognition

Unlike Soar, SDML has no hardwired theory of cognition, so any agent cognition must be represented by the user as sets of rules. The model reported here has a very limited representation of cognition, with agents evaluating other agents as possible collaborators. The basis for evaluation is an extension by Moss (1995) of the endorsements procedure devised by Cohen (1985) as a method of conflict resolution for expert systems. We do not claim either that agents in the model incorporate a wide range of cognitive abilities or that their behaviour reflects an underlying theory of cognition.

Activity-agents have an endorsement scheme which gives values to tokens representing individual endorsements. One agent might endorse another as “reliable”, “successful collaborator" or, indeed, "unreliable", or "unsuccessful collaborator". Endorsements can have negative as well as positive values.[10]



Cooperation offered
Endorsements

possible collaborator
responsive collaborator
successful collaboration
unresponsive collaborator
unsuccessful collaboration

Endorse as possible collaborators those activities which share a parent in an abstract activity tree. The argument enables one activity to endorse the same other activity more than once if they share the same parent in more than one activity tree.

```
and
     abstractActivityComposition ?abstraction ?components\
     includes ?components self\
     includes ?components ?another\
     notInferred
          = ?another self\
     manifestationOf ?another ?possibleCollaborator
```

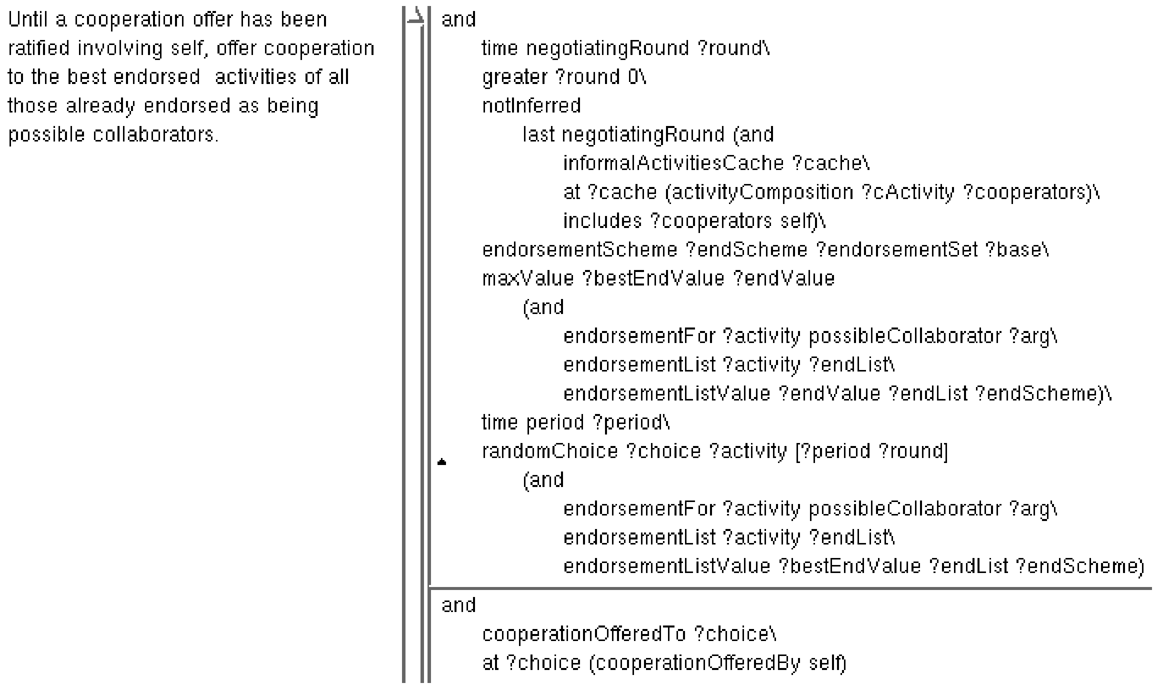
```
endorsementFor ?possibleCollaborator possibleCollaborator
[?abstraction]
```

Figure 2: Rule for endorsing possible collaborators

Agents' endorsements of other agents are the consequences of rule-based evaluation and typically determine either further evaluation or some action. An example, the content rule "possible collaborator" which is defined for individual activities, is given in Figure 2. The first two clauses of the antecedent identify all abstract composite activities which include the rule-calling individual activity. self in any rule is the agent's self-reflexive symbol. The next two clauses ensure that the potential collaboration is not with oneself. The final clause in the antecedents is a backward-chaining clause which, in the case of simple activities identifies their abstractions with themselves or, if the possible collaborator is a composite activity, finds the appropriate clause on the organization's database which associates an abstract with an actual composite activity.

In this case, then, rule-based evaluation results in the recognition that another activity is a possible collaborator and this is manifested in the endorsement of that other agent as possibleCollaborator.

[10] It is also possible to endorse rules as being successfully applied or unsuccessfully applied in the sense that a rule firing was or was not associated with better goal achievement.



Until a cooperation offer has been ratified involving self, offer cooperation to the best endorsed activities of all those already endorsed as being possible collaborators.

```
and
    time negotiatingRound ?round\
    greater ?round 0\
    notInferred
        last negotiatingRound (and
            informalActivitiesCache ?cache\
            at ?cache (activityComposition ?cActivity ?cooperators)\
            includes ?cooperators self)\
    endorsementScheme ?endScheme ?endorsementSet ?base\
    maxValue ?bestEndValue ?endValue
        (and
            endorsementFor ?activity possibleCollaborator ?arg\
            endorsementList ?activity ?endList\
            endorsementListValue ?endValue ?endList ?endScheme)\
    time period ?period\
    randomChoice ?choice ?activity [?period ?round]
        (and
            endorsementFor ?activity possibleCollaborator ?arg\
            endorsementList ?activity ?endList\
            endorsementListValue ?bestEndValue ?endList ?endScheme)
```

```
and
    cooperationOfferedTo ?choice\
    at ?choice (cooperationOfferedBy self)
```

Figure 3: Rule for offering cooperation

In Figure 3, we exhibit the rule editing window for the rule which, if instantiated, entails the offer of cooperation to another agent. The first two clauses ensure that the negotiating round is not the first during the period (for which there is another rule). The notInferred clause[11] ensures that the activity was not already cooperating informally during the previous negotiating round. The next two clauses identify the endorsement scheme of the agent and find the best endorsement value of all other agents endorsed by this agent as possible collaborators. The randomChoice clause uses the period and round to uniquify the choice made from all instantiations of its conjunctive subclause which will identity all other agents who have maximum endorsement value among all possible collaborators.

The consequents of this rule have already been discussed in section 2.4.

After a collaboration has been agreed, other rules continue to add endorsements of the collaborating agent to the activity's database. For example, if the value of the

[11] SDML corresponds to a fragment of strongly grounded autoepistemic logic. There are two levels of negation in such logics: notInferred and not. That is, until clauses have been proved false, they have simply not been proved to be true. Proving full negation is computationally expensive so we allow that, for example, it is not inferred that elephants are green rather than proving the non-existence of green elephants (by, presumably, exhaustive search).

collaboration exceeds the sum of the individual activity values, then the activity endorses the collaborating agent with the token successfulCollaborator. Otherwise, the collaborating agent is endorsed with the token unsuccessfulCollaborator. Successful collaboration increases the strength of the collaboration in the sense that the collaborator has a higher endorsement value and so no other agent is likely to be offered cooperation. Unsuccessful collaboration is doomed because the endorsement value of the collaborator declines each period.

As this example shows, the endorsements can be used both to identify subgroups of possible collaborators and then to rank their desirability. They also allow the representation of a form of learning whereby agents' knowledge of good and poor collaborators is built up over time. As suggested in section 5, the use of endorsements could be an adjunct to Soar cognition which renders it more effective for computational organization theory.

3.5 Development of the model

Currently the model is highly abstract, with the effects of interactions among business processes being simply represented by the numerical values associated with different combinations and the generic representation of output, "productivity". The assumption that each business process must collaborate with just one other in preference to all others is clearly implausible. Furthermore, there is a lack of any meaningful agent cognition, since agents' decisions are made entirely by reference to abstract variables.

Further development will involve modelling the effects of agent cognition by implementing a simple version of "bounded rationality" whereby agents are goal-directed, have limited capacity, and operate under conditions of uncertainty. Related to this change, agents will be explicitly associated with tasks with inputs and outputs. They may have individual goals as well as those (possibly conflicting) imposed in the form of performance targets, and these goals will be met by performing tasks and generating outputs. Agents have a limited capacity and must therefore prioritize. There may be formally prescribed sources for the necessary inputs to tasks, but these may be circumvented where alternatives (or more direct routes to an original source) exist. Other agents will be evaluated on the basis of, for example, reliability or unreliability, timeliness or delay. As well as trying to second-guess the current actions of other agents, agents will be trying to assess the future impact of their own current actions. Thus complex social phenomena may be expected to arise through the interactions of relatively simple cognitive agents.

One aim in introducing a plausible[12] model of agent cognition is enabling representation of the effects of different organizational structures. For instance, given plausible constraints on the capacities of individuals, there will be a greater overall capacity for information exchange in a hierarchical organization than in a relatively flat one. On the other hand, given agents' needs to prioritize, relatively direct information exchange in the flat organization may mean that information is more reliably transmitted. This is just one example of the kind of trade-off between alternative organizational structures which the less abstract, more explicit model of business process interaction will allow us to represent.

We do not suggest that cognition as presently represented in the model's endorsement-based evaluation is at all general and certainly it does not stem from any general cognitive theory. For this reason, incorporating some theory of cognition such as the Soar implementation of Newell's unified theory of cognition is a further natural step to take in the development of SDML. This step will be especially useful because the flexibility of SDML will support modifications to Soar cognition as a means of investigating the extent to which implementation features affect the outcomes without violating the relationships and procedures identified with Newell's theory itself. This will address the difficulties raised by Hunt and Luce (1992) in their point that Newell's description of his unified theory of cognition is not implementation specific. It will also provide a different perspective on the investigation of the implications of different implementations from that of Cooper, Fox, Farringdon and Shallice (1996) precisely because SDML is a general simulation language with strong supporting facilities for modelling agent interaction with structured and hierarchical as well as unstructured environments. A natural development with SDML is to try different implementations of the same theories of cognition that Cooper *et. al.* considered such as Johnson-Laird's (1983) theory of mental models or Barnard's (1985) theory of interacting cognitive subsystems on the behaviour and performance of organizations.

Implementing Soar cognition in SDML is the first step in this line of research.

[12] For this purpose we are concerned primarily with cognition within the particular context of the organizational model, and thus not necessarily with such general approaches to cognition as that embodied in Soar. The latter is a separate project which we outline below.

4 SDML and Soar

Unlike Soar, SDML was developed precisely to capture multi-agent interaction in complex environments, and, unlike SDML, Soar was built around an explicit and sophisticated model of human cognition. Probably the most useful feature of Soar absent from SDML is a hardwired learning mechanism known as *chunking*. SDML agents can learn, but the mechanism must be encoded explicitly into rules. An implementation of Soar cognition in SDML is well advanced and is described below. Although testing is not yet extensive, it appears that SDML can accommodate the Soar theory of cognition within a complex and flexible multi-agent framework.

4.1 Comparison between SDML and Soar

Our intention here is not to provide a comprehensive comparative evaluation of the two languages but, rather, to give an indication of the similarities and differences pertinent to an implementation of Soar agents in SDML.

Rule firing is similar in the two languages. Soar's rules, known as *productions*, fire when their conditions match data in *working memory.* Soar keeps firing productions until all of them with matching conditions have fired. Similarly, in SDML, rules fire whenever their antecedents can be ascertained to be true (by matching the contents of databases), and firing continues until there are no remaining rules with matching antecedents left to fire. In both languages, a rule/production can contain variables, and it may be fired many times with different bindings for the variables.

The primary difference with respect to rule firing is that, whereas the consequents of SDML rules are asserted to databases immediately, Soar productions do not directly affect working memory. Instead, changes are made to *preference memory*, and a decision procedure is used to resolve these *preferences* and determine the changes to be made to working memory. Soar then fires productions again based on the new contents of working memory, resolves preferences again, and continues with this *elaboration cycle* until there are no more changes to be made (which is known as *quiescence*). When quiescence is reached, a new operator is chosen (according to preferences) to be performed next. If a valid and unique operator cannot be determined, then an *impasse* arises which is recorded as a new state in the working memory (to be treated as a subgoal to be solved, in possibly a different problem space, before continuing with the current goal). In either case, Soar goes on to another elaboration cycle, and continues until the problem is solved. This process is known as the *decision* cycle. It is worth mentioning that rules also fire until quiescence in SDML. However, cycles are less common in SDML as the automatic calculation of rule dependencies eliminates, wherever possible, the need to loop. As we shall see in section 5, the SDML rule dependencies mechanism can greatly reduce the computational overheads associated with cognitive representations of agents.

A further difference is that, whereas Soar has retraction, SDML uses time levels to avoid the need for retraction. In Soar, preferences which have *I-support* (instantiation support) are retracted when the productions that created them no longer hold. The effects of retraction can easily be reproduced in SDML since what is true at one time period need not be true at the next (and will not be inferred to still be true unless rules specify that it is). Preferences with O-support (operator support), created in Soar when *operators* are used, are retained whether or not the productions still hold (it is quite likely that they will not continue to hold since operators modify the current state). A similar effect can be achieved in SDML by specifying that the consequents of a rule are true permanently rather than just for the current time period. When the rule fires, the clause is asserted to the agent's permanent database (i.e., corresponding to the time level eternity) rather than the current subdatabase.

4.2 Soar's Problem Solving in SDML

Whereas SDML fires rules, Soar uses its rules to solve problems, as outlined above. What is mainly involved in implementing Soar's problem solving model (NNPSCM, for new problem solving computational model) in SDML is explicitly writing the rules that enable Soar to choose a single value from a number of acceptable ones. First, however, the appropriate time levels corresponding to Soar's problem-solving cycles must be specified.

In Soar, there are two kinds of cycles, the elaboration cycle within the decision cycle. The elaboration cycle is required by Soar as, whenever a change is made in working memory, Soar's productions must be fired again. A time level corresponding to an elaboration would not, however, be required in SDML, as the dependencies among rules are calculated such that a dependent rule will automatically fire again following the firing of a determining rule. What is required is a single time period corresponding to a decision, with looping over this time level giving rise to the decision cycle. Corresponding to each decision time period, a new state or operator will be chosen.

SDML rules can be written corresponding to the different functions of the NNPSCM (listed in the Soar 7 User's Manual) as follows:

- Operator proposal. These rules specify the operators that are acceptable, based on the current state and the goal to be solved.
- Operator comparison. These rules specify how acceptable operators can be compared (representing, for example, the knowledge that one operator is best, or better than another operator).[13]
- Operator selection. These rules specify how to select a single operator. If a single operator cannot be selected, an impasse is detected.
- Operator application. These rules specify how an operator, chosen at the previous time period, can be applied to yield a new state. These rules must ensure that aspects of the state not modified by the operator are retained.
- State elaboration. These rules specify extra information that can be inferred about the state.
- Operator termination. These rules specify the conditions under which operators have been performed successfully.

Some extra rules will also be required for such tasks as copying the old state if no operator has been performed, dealing with impasses, reporting useful information to the user, and setting up the initial state. Such rules will generally be fired at the start or end of the rulebase firing process. The order in which the rules are fired in SDML will be determined automatically based on the dependencies between rules (some rules may be mutually dependent and need to be fired repeatedly). The different NNPSCM functions will not necessarily be performed in a strict ordering[14], but they will be fired roughly in the order shown in Figure 4.

In this way, a problem can be solved by a single NNPSCM agent in SDML. Alternatively, it may be useful, especially for big problems, to use different subagents to solve subproblems. Instead of representing the new state and problem space within the same agent when an impasse occurs, a subagent can be created and its rules fired using a new time level. When the subagent's decision cycle has finished, due to the subproblem being solved or aborted, the original agent continues with its decision cycle. This scheme enables rules applicable to different problem spaces to be defined in the rulebases of different agent types. More general rules, which are applicable to more than one problem

[13] Preferences would be used for this in Soar. Preferences between operators can be stored explicitly on databases, but it may be more convenient when solving some problems in SDML to omit this step and compare operators as part of the selection process.

[14] Although such an ordering can be forced if desired.

space, can be placed in the rulebases of supertypes. Thus, agents' rulebases will only inherit rules applicable to the problem currently being solved.

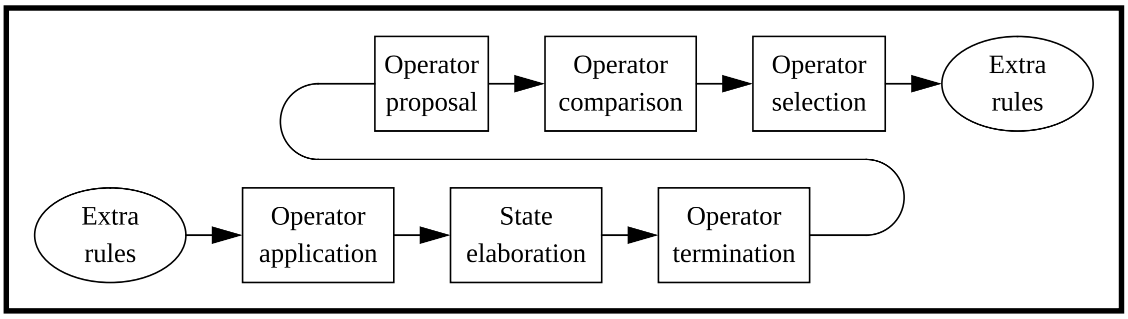


Figure 4: Overview of an NNPSCM agent rulebase in SDML.

4.3 Compiling Soar into SDML

Another useful facility is that for compiling Soar productions into SDML rules. This can be implemented using SDML's meta-agents which can assert rules to other agents' rulebases, as if these were databases. Soar source code is first parsed to yield clauses representing Soar productions, which are then compiled to yield SDML rules.[15]

In order to ensure that the compiled Soar agents have the same behaviour as in Soar itself, a time level should be used corresponding to the elaboration cycle, rather than the decision cycle.[16] Therefore, the compiled Soar rules cannot affect each other within the same time period, but instead retrieve information from the previous subdatabase (representing the contents of working memory) and assert clauses representing preferences.

These preferences can then be resolved by generic SDML rules, asserting clauses representing working memory elements to the current subdatabase, or yielding impasses when the preferences cannot be resolved. However, in practice, most attributes utilise very few kinds of preference. Therefore, preference resolution can be optimised by the compiler generating specific preference resolution rules for each attribute.

15 SDML rules are compiled into Smalltalk before being fired. The resulting Smalltalk code is compiled into bytecodes, which are finally compiled into machine code!

16 Or two time levels could be used, with the elaboration cycle time level within that of the decision cycle. This is the arrangement implemented in section 5.

The preference resolution rules also detect quiescence, in which case an elaboration cycle ends and preferences are resolved for operators. Within each time period, rules in the Soar agent rulebase will be fired roughly as shown in Figure 4.

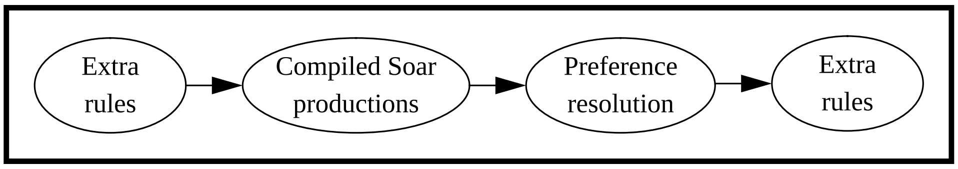


Figure 5: Overview of a Soar agent rulebase in SDML.

Typically, the meta-agent fires its rules at the start of a simulation, to compile Soar source code into SDML rules. The same rules may then be fired at many different time periods during the simulation. If a learning mechanism, such as chunking, is implemented, then further Soar productions (represented as SDML clauses) may be generated during a simulation. When this happens, the meta-agent needs to fire its rules again, in order to generate extra rules to assert to the Soar agent's rulebase.

5 An SDML implementation of Radar-Soar

Because SDML is a general programming language, there is wide scope for choosing a particular form of implementing any model. As indicated above, the procedural nature of Soar can be replicated by ensuring that every elaboration cycle takes place in its own time period and, within elaboration cycles, the specification of problem-space agents ensures that only one problem space applies at a time. In this section, we do not specify problem-space agents. Instead, we implement a a well-known but restricted version of Soar cognition as declaratively as possible. We demonstrate that, as a result, cognitive agents behave as Soar agents but do so in a fraction of the CPU time and with a fraction of the memory requirements. Together with the representation of inter-agent communication as explicit addressing of clauses to other agents' databases and the bulletin-board function of containers, declarative implementation of Soar cognition seems likely to be an important contribution to the increase in the sophistication of representations of cognition combined with increased articulation of modelled organizational structures.

In Radar-Soar (Ye and Carley, 1995), there are two types of cognitive agent: the manager of the radar station and the analysts of the information obtained by the station. The goal of each agent is to decide whether each aircraft they can observe is friendly, neutral or hostile and, in the case of the analysts, to communicate that decision to the manager. The problem-space structure of the manager is given in Figure 6 taken from Ye and Carley (p. 231).

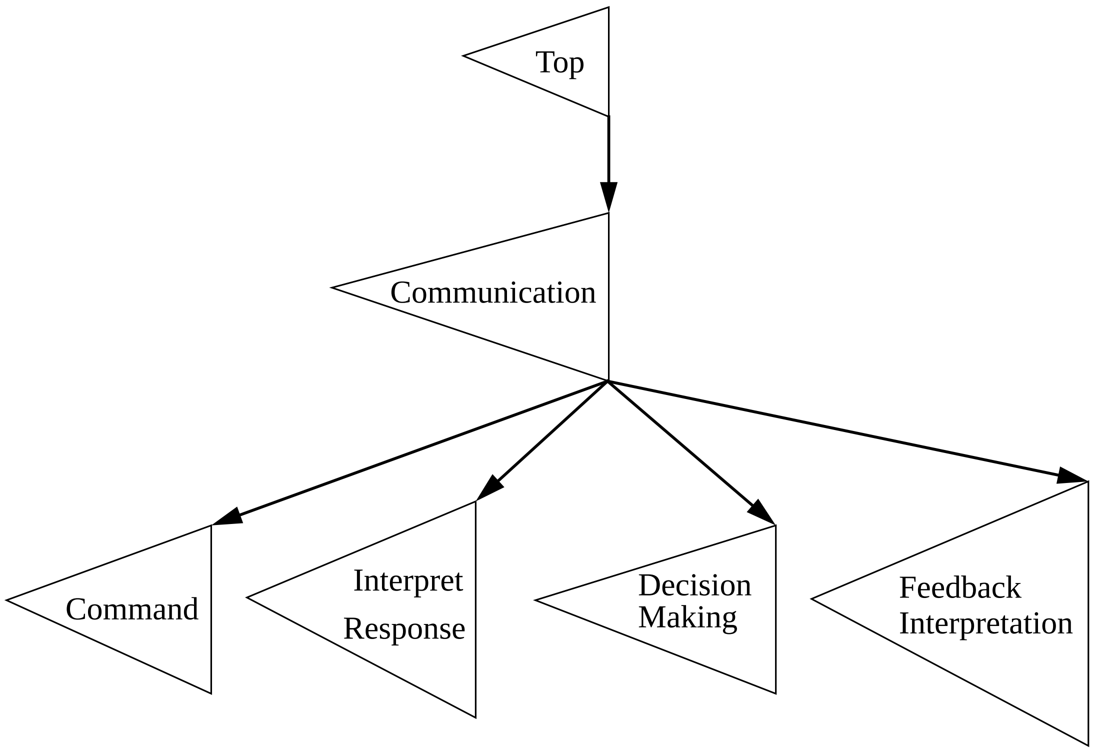


Figure 6: Manager’s problem space structure

The manager starts in the top-level problem space and moves successively to the communication, command, interpret-response, decision-making and feedback-interpretation problem spaces. With each problem space there are a number of tasks to be completed which achieve a subgoal associated with the problem space. If that subgoal is achieved, then the agent pops back up to the problem space above or to the next problem space. If the subgoal cannot be achieved, the agent drops down as from communication to command.

Radar-Soar is a particularly useful model to demonstrate the relationship between Soar and SDML because it has an inherently simple structure due to the absence of chunking from the model.

5.1 The radar model structure in SDML

The information flows in Radar-Soar were straightforward to capture in SDML using the container hierarchy. This is shown in Figure 7. The environment (of type RadarEnvironment) is a serial composite agent. Its subagents are the airspace and the radar station. In ordering the activities of these two composite subagents in this way, the entry of aircraft into the airspace (by generating an instance of aircraft) will be observed by agents in the radar station during the same time frame of the radar environment. The time frame in question is the decision cycle — that time level during which all of the reasoning, learning and deciding necessary to identify an aircraft (rightly or wrongly) is achieved. Within the decision cycle are the elaboration cycles in which the agents in the radar station undertake the actions and make the decisions necessary to achieve the goal or subgoal of the current problem space.

The purpose of the analysis sections (of type RadarAnalysisSection, a subtype of ParallelCompositeAgent) is to filter the information available to individual radar analysts. Each section is given three of the nine aircraft characteristics to observe and each analyst in observes only the characteristics given to its section.

If each problem space caused the agent to move on to a new elaboration cycle, or if each problem space were associated with a subagent, then SDML would be following Soar in representing cognitive acts procedurally. In the SDML implementation, however, every problem space that needed resolving pertained within each elaboration cycle.

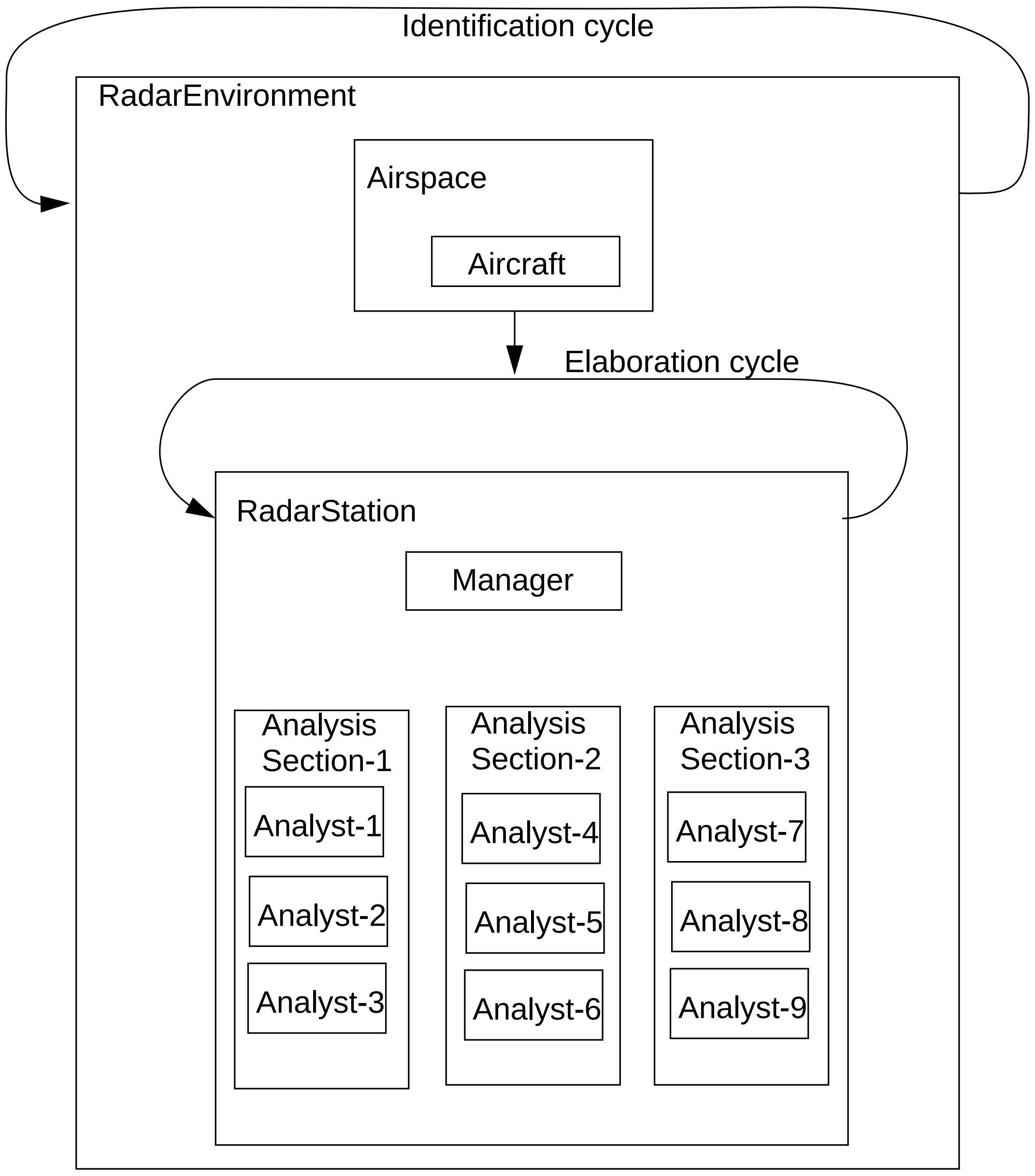


Figure 7: Container hierarchy of Radar-SDML

If we look at the cognitive process represented in the model as logic rather than as procedure, then each agent proves that it is in each problem space on the basis of its current circumstances including the other problem spaces in it has proved itself to be in and the actions it has proved to have been undertaken. Within each elaboration cycle, the agent will prove in succession that it is in the top-level problem state but has not achieved its goal and so should have one or more sub-goals implying corresponding problem states with perhaps further subgoals, and so on. Some of these problem states imply actions to achieve their sub-goals and, if it can, the agent will prove that the actions are implied by other conditions which will themselves possibly be proved to hold by proving first that the conditions can be satisfied in which those further conditions will be proved true.

Three of these declarative elaboration cycles are required in each decision cycle. In the first elaboration cycle, the manager proves that communication is necessary in the form of commands to the analysts to observe the aircraft and report their decision as to its status as friendly, neutral or hostile. Without the reports of the analysts, no further propositions can be proved. The analysts cannot perceive the commands in the same elaboration cycle as they are given because their declarative processes are being undertaken in parallel with those of the manager. In the second elaboration cycle, the manager can get no further than in the first but the analysts can now perceive that, in the previous elaboration cycle, the manager addressed two commands to their respective databases: *observe* and *report*. With these commands in their databases, the analysts have all of the clauses required directly and indirectly to fire the rules which lead to their decisions based on mental models of the status of the aircraft. One of these rules is explicitly to assert to the database of the manager the analyst's decision. In the third elaboration cycle, the manager perceives that in the previous elaboration cycle all of the analysts reported their decisions and so the manager now has all of the clauses required directly and indirectly to fire the rules required to reach the manager's own decision about the status of the aircraft.

A property of SDML is that all of these declarations can be made in parallel and, when that happens, the whole process is logically consistent and sound relative to the fragment of strongly-grounded autoepistemic logic that excludes classical negation but entails not-inferred negation.

5.2 Results

We ran experiments with two simulation setups, both adopting specification from Carley and Lin (forthcoming). In the first, the probability of an aircraft being in any one of the three states was equal and the status was determined on the assumption that each property of the aircraft independently contributed to its status. Mapping the property levels {low, medium, high} into {1, 2, 3}, an aircraft is friendly if the sum of those values is less than 16, neutral if in the range 17 to 19 inclusive and hostile if the sum is at least 20. In the second setup, the contributions of each property to the state of the aircraft was non-decomposable. The formula was

$$(2 \times F1 \times F2 \times F3) + (2 \times F4 \times F5) + F6 + F7 + (2 \times F7 \times F8 \times F9)$$

where F1 is the numerical value of the first property, F2 of the second, and so on. As in Radar-Soar, the first analysis section could observe the first three properties, the second section the second three and the third the last three. The values of this formula (which Carley and Lin found to yield equal probabilities of aircraft state) were adopted.

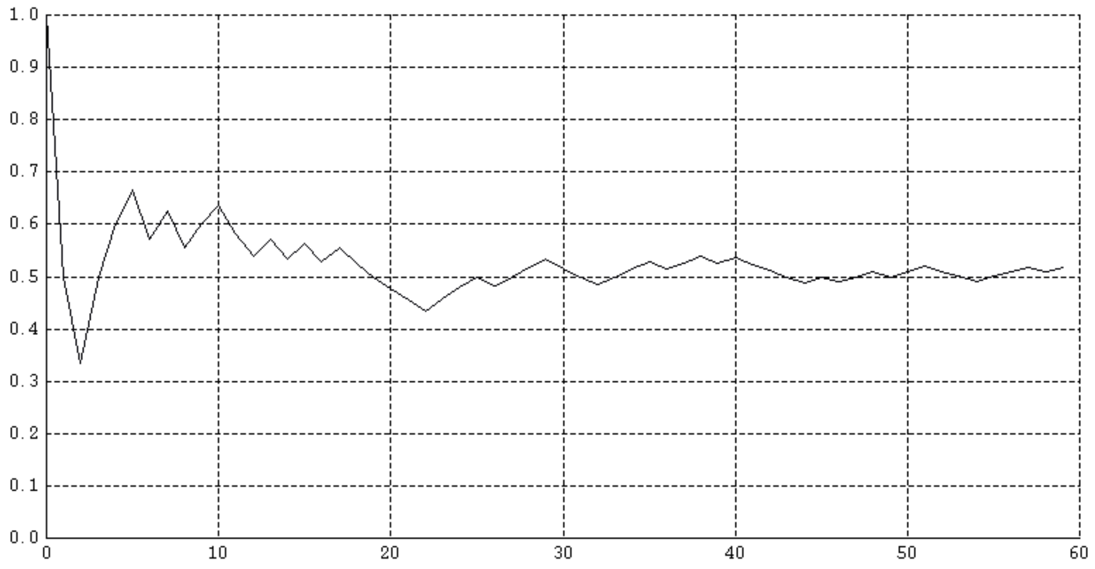


Figure 8: Cumulative proportion of correct decisions — decomposable properties

The results from an experiment with decomposable aircraft properties are indicated in Figure 8. The success statistic is the cumulative proportion of correct decisions by the manager. The expected success rate from choosing a state at random is a third. The manager in every experiment run with this setup achieved a success rate of about half. Since no experiment yielded a success rate of less than half, learning in this model clearly had some effect.

The manager did even better in the experiments with the non-decomposable properties. As seen in Figure 9, the success rate was in excess of 60 per cent which is the sort of success rate that Ye and Carley (1995) reported.

5.3 The efficiency effect of declarative cognitive representations

Ye and Carley report that the analysts and the manager each require some three hours of CPU time to complete an experiment with 60 aircraft where learning from feedback ceases after the first 30 are identified. The transcript from our experiments show that an experiment of the same length but with learning throughout takes less than 30 minutes including all decisions by all agents computed sequentially. Although our experiment was designed from the problem space map and flow charts reported by Ye and Carley, the much more declarative setup required one decision cycle for each aircraft and three elaboration cycles within each decision cycle. These cycles are not the same as in Soar since within each elaboration cycle the largest possible set of logically consistent steps are made. The SDML assumptions mechanism eliminates the inconsistent steps.

Apparently, the forward-chaining mechanism of SDML together with its correspondence to strongly-grounded autoepistemic logic renders the whole process far more efficient than Ye and Carley achieved with Soar. Whether this is a property of Soar or the Carley-Ye setup or some more pervasive property of declarative and procedural modelling approaches is a matter for further research. The Soar-to-SDML compiler reported in section 4 will provide a framework within which to address that question.

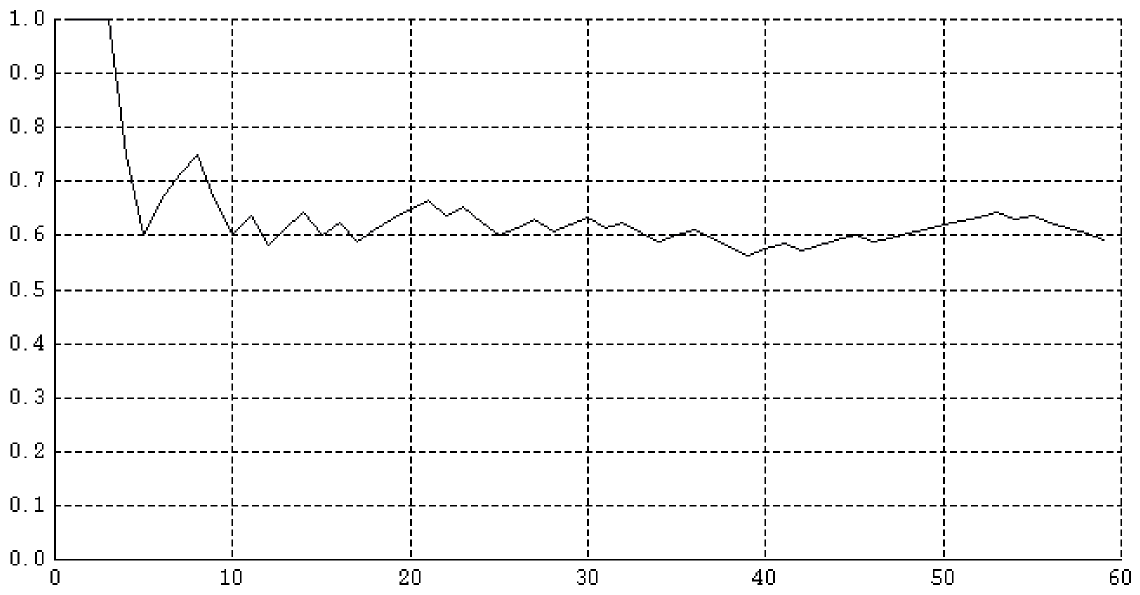


Figure 9: Cumulative proportion of correct decisions — non-decomposable properties

5.4 Extending Soar cognition

Ye and Carley did not allow the Soar chunking mechanism to operate in Radar-Soar because it is difficult to code, it only takes effect when meeting the same conditions encountered previously, that when used “Soar tried to learn from the degree to which the previous model matched, rather than from the features of the model (or current situation) that matched" and, finally, previous results suggest that "the main mechanism for learning in Soar, learning through chunking, is insufficient to capture social behaviour and organizational learning that occurs in the face of novel situations." (Ye and Carley, 1995, p. 244)

An alternative is to be found in the endorsements mechanism described in section 3. Agents could endorse their models and the components of the models. In effect, the Ye-Carley Radar-Soar implementation endorses a model which has been selected and yielded the correct status identification as having done so by replicating the model — effectively adding to the weight of that model in determining the probability of choosing it again when it is among the most applicable. It is also possible for the manager to endorse individual analysts according to the number of times their individual decisions have been correct or incorrect. The analysts could endorse the individual aircraft properties or even property values for their predictive power. there is no reason not to look for non-decomposability (*cf.* Moss, 1995). The reusability of code supported by the object-oriented features of SDML render the introduction of the endorsements mechanism straightforward, though endorsing rules will have to be written.

Moreover, a little experimentation with the existing SDML version of Radar-Soar indicates that the most successful mental models that imply a common decision also share a subset of conditions so that generalizing these models to have fewer conditions (thereby to cover a larger number of situations) is a promising direction to pursue.

These natural extensions of Radar-Soar together with the apparently huge difference in computational expense with a declarative version of Soar cognition supports our belief that SDML is a suitable platform from which to overcome the shortcomings of Soar as a vehicle for computational organization modelling without necessarily modifying the individual behaviour implied by Soar cognition. If we choose to modify that behaviour, then that, too, is straightforward in SDML.

6 Conclusion

For sophisticated computational models of organizations, two requirements are primary. These include the need for cognitive models of agents, together with multi-agent representations which enable us to capture the effects of agent interaction. These requirements are basic given such aims as capturing the relationship between individual cognition and emergent phenomena at the social level (such as norms), or modelling the effects of organization-level phenomena which necessarily impinge upon the individual, such as downsizing and business process reengineering. Related to these basic needs are further considerations we have outlined above, such as the need easily to represent the effects of change over time.

We have introduced SDML, a programming language designed to meet the requirements of multi-agent modelling. Its development has focused upon this need, rather than that of agent cognition, for a number of reasons. Members of the Soar community, we have seen, point to the lack of a suitable architecture in which easily to develop and modify multi-agent models as a major impediment to the development of sophisticated organizational models. Overcoming this impediment with declaratively programmed cognitive models of agents will enable us to determine how quickly (and whether) more complex phenomena emerge at the organizational level as well as the nature of emergent organizational responses to, for example, shifts in business strategy regimes. Soar already provides a widely used general model of problem solving and, where required, this can be easily implemented in SDML or, indeed, compilers written for Soar code in SDML.

The promise of SDML as a vehicle for more elaborate organizational modelling with cognitive agents is supported by our demonstration that Radar-Soar re-implemented in SDML and using SDML's efficient forward-chaining declarative capabilities yields simulation setups that achieve the same goals in many fewer steps but without loss of functionality. This raises the question of whether more declarative implementations of cognitive theories are inherently more efficient than procedural implementations and, if they are, whether they are less compelling as representations of actual cognition.

From the point of view of organization theory, there are clear benefits from representations of cognitive agents that are efficient within an artificial organization. If their cognition is less realistic, then this is yet another trade-off of the sort that scientists face all of the time. Whether there is such a trade-off and the extent to which we are constrained by that trade-off in the analysis of real organizational processes is an important question to be addressed in further research.

References

Barnard, P.J. (1985), “Interacting Cognitive Subsystems: a Psycholinguistic Approach to Short-term Memory”, in A.Ellis (Ed.) *Progress in the Psychology of Language*, Hillsdale, NJ: Lawrence Erlbaum, chapter 6, 197-258.

Kathleen M. Carley & Zhiang Lin (forthcoming), “A Theoretical Study of Organizational Performance under Information Distortion”. *Management Science*.

Carley, K.M. and D.M. Svoboda, (1996) “Modeling Organizational Adaptation as a Simulated Annealing Process”, *Sociological Methods and Research*, 25(1), 138-168.

Cohen, P.R. (1985), *Heuristic Reasoning: An Artificial Intelligence Approach*, Boston: Pitman Advanced Publishing Program.

Cooper, R., J. Fox, J. Farringdon, and T. Shallice, (1996) “A Systematic Methodology for Cognitive Modeling”, *Artificial Intelligence*, 85, 3-44.

Hunt, E. and R. Luce, (1992) “Soar as a World-view, Not a Theory”, *Behavioral and Brain Sciences*, 15(3), 447-448.

Johnson-Laird, P.N. (1983) *Mental Models*, Cambridge, UK: Cambridge University Press.

Moss, S. (1995), “Control Metaphors in the Modelling of Decision-making Behaviour”, *Computational Economics*, 8(4), pp. 283-301.

Moss, S., B. Edmonds and H. Gaylard, “Modelling R&D Strategy as a Network Search Problem”, *The Multiple Linkages Between Technological Change and the Economy*, Rome: CEIS.

Newell, A. (1990) *Unified Theories of Cognition*, Cambridge, MA: Harvard University Press.

So, Y. and E.H. Durfee, (1996) “Designing Tree Structured Organizations for Computational Agents”, *Computational and Mathematical Organization Theory*, 2(3), Fall 1996.

Tambe, M. and P.S. Rosenbloom, (1996) “Architectures for Agents that Track Other Agents in Multi-agent Worlds”, *Intelligent Agents, II, Springer Verlag Lecture Notes in Artificial Intelligence* (LNAI 1037).

Verhagen, H. and M. Masuch, (1994) “TASCCS: A Synthesis of Double-AISS and Plural-SOAR”, in *Computational Organization Theory*, K.M. Carley and M.J. Prietula (Eds.), Hillsdale NJ: Lawrence Erlbaum

Ye, M. and K.M. Carley, (1995) “Radar-Soar: Towards an Artificial Organization Composed of Intelligent Agents”, *Journal of Mathematical Sociology*, 20(2-3), 219-246.

Appendix 1 Using SDML features to represent technology: valuing composite activities

The determination and allocation of the factors used to calculate the values of composite activities use the same features of SDML that are used to represent communication among agents. They also make use of the facilities which support changing the agents which are active within containing agents.

The factors used to multiply the sums of the components of a composite activity to determine the composite activity value are generated for every organization by the setup agent. The setup agent creates a number of feasible abstract activity trees. These include an abstraction of the actual activity tree of each organization in the model. In addition, the user is asked to specify a number of additional abstract activity trees and this number of trees is created by the setup agent. The abstract activity trees are built up by a backward-chaining clause which takes a randomized list of the organization's activities and builds it up into a binary tree structure.

One composite activity is generated for every node of every abstract activity tree. Because these composite activities are generated by the model, they are subagents of the model which are never asserted to be active subagents. Because they are never activated, the rules of these composite activities never fire. They can therefore serve as abstract composite activities which are themselves comprised of abstract activities and none of them have any direct effect on the processes being simulated.

In order to match actual (that is, active) activities with abstract activities, each abstract activity and each actual activity is given an identifier. The individual activities in each activity tree are put in alphanumerical order by symbol and given a serial number which is determined by their respective positions in that order. The composite activities are given identifiers constructed from the identifiers of the individual activities from which, directly or indirectly, they are constructed. The actual activities are created by departments while the abstract activities are created by the model. Whenever two individual activities wants to establish a new composite activity with one another, they are permitted to do so only if the there is an abstract activity with the same identifier as the new, actual composite activity would have.

The model agent assigns a realization of the U(0,3] random number to each abstract composite activity. This number is the factor used to multiply the sum of the values of the constituent activities to yield the value of the actual counterparts of the abstract composite activity. The actual counterparts will be those with the same identifiers as the abstract composite activity. Because the clauses asserted with that number and the clauses recording the identifier of each abstract composite activity are defined in the model agent type, both the number and the identifier of each abstract composite activity can be accessed by every agent contained by the model. The identifiers and values of the actual activities are captured by clauses defined in the department agent type. since they are internally readable, no other department can read the values of activities of any other department.

It is not actually necessary to store valuation factors of the abstract composite activities on a database (though in some cases to do so might be efficient). When a new actual composite activity is created, a valuation factor is calculated for the corresponding abstract composite activity using a primitive clause randomNumber <number> <clause>. the number is instantiated by the primitive. The clause uniquifies that random number so that if the random number primitive is called again with the same uniquifying clause, the same random number will be unified with the first argument of the clause. This is a property of a strictly declarative modelling environment: once a statement has been proved true it is always true. So, if the uniquifying clause is distinguished by the identifier of the actual composite activity, then any actual composite activity with the same identifier generated by any organization in the model will have the same valuation factor.

In summary, we represent the technology of activity combination by the abstract activity trees. Because of the container hierarchy and the facility for defining the readability and writeability of clauses, it is straightforward to create and store the abstract activity trees and their characteristics so that all organizations are subject to the same technological environment but they can develop and use the technologies in different ways. In particular, in the model under discussion here, the organizations can exploit the technology to a greater or lesser extent by R&D while organizing their use of the technology differently by adopting different activity combinations and by building up the activity combinations in different orders.

Acknowledgements

We are grateful to Kathleen Carley for answering a string of questions about details of the implementation of Radar-Soar. SDML has been developed in a succession of Smalltalk-80 environments produced by ParcPlace, most recently VisualWorks 2.5.1. ParcPlace-Digitalk (UK) Ltd. sponsor the Centre for Policy Modelling and support the distributution of SDML with the VisualWorks object engine without charge for purposes of academic research. This research was also funded in part by the Economic and Social Research Council of the United Kingdom under contract number R000236179.