Skeleton-based Agent Development for Electronic Institutions

Wamberto W. Vasconcelos
Centre for Intelligent Systems and their Applications (CISA)
Division of Informatics, University of Edinburgh,
80 South Bridge, EH1 1HN, Edinburgh, UK.
Phone +44-(0)131 650 2718 Fax +44-(0)131 650 6513
wvasconcelos@acm.org

Jordi Sabater, Carles Sierra, Joaquim Querol
Artificial Intelligence Research Institute (IIIA)
Consejo Superior de Investigaciones Científicas (CSIC)
Campus UAB, 08193, Bellaterra, Catalonia, Spain.
Phone +34 935 80 96 70 Fax +34 935 80 96 61
{jsabater, sierra, joaquim}@iiia.csic.es

ABSTRACT

In this paper we describe an approach for semi-automatic agent development. We focus on the scenario in which agents are designed to follow an electronic institution, a formalism to specify open agent organisations. In our approach, an initial design pattern is automatically extracted from a given electronic institution. This pattern is then offered to programmers willing to develop agents to perform in the electronic institution. Our approach supports developers when modifying the initial simple design pattern into more sophisticated programs.


General Terms: Design, Languages, Experimentation.

1. INTRODUCTION

Electronic institutions (e-institutions, for short) have been proposed as a formalism with which one can specify open agent organisations [20]. In the same way that social institutions, such as a constitution of a country or the rules of a club, are somehow forged (say, in print or by common knowledge), the laws that should govern the interactions among heterogeneous agents can be defined by means of e-institutions. Agents programmed in any language, using whichever design principles or methodologies, can join in an e-institution, adequately interact with other agents and achieve particular and global goals. However, we have detected an opportunity to provide semi-automatic support to agent designers.

The basic idea in our approach is to automatically extract from an e-institution an account of the behaviours agents ought to have. This simplified account is called a skeleton: it provides the essence of the agents to be developed. Skeletons should be simple, and represented in a suitable form that would encourage their use as the initial design for sophisticated reasoning agents. Engineers willing to develop agents to perform in e-institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs. We show in Figure 1 a diagram describing the general approach: start-

suitable form that would encourage their use as the initial design for sophisticated reasoning agents. Engineers willing to develop agents to perform in e-institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs. We show in Figure 1 a diagram describing the general approach: start-

Figure 1: Overview of Proposed Approach

1. INTRODUCTION

Electronic institutions (e-institutions, for short) have been proposed as a formalism with which one can specify open agent organisations [20]. In the same way that social institutions, such as a constitution of a country or the rules of a club, are somehow forged (say, in print or by common knowledge), the laws that should govern the interactions among heterogeneous agents can be defined by means of e-institutions. Agents programmed in any language, using whichever design principles or methodologies, can join in an e-institution, adequately interact with other agents and achieve particular and global goals. However, we have detected an opportunity to provide semi-automatic support to agent designers.

The basic idea in our approach is to automatically extract from an e-institution an account of the behaviours agents ought to have. This simplified account is called a skeleton: it provides the essence of the agents to be developed. Skeletons should be simple, and represented in a suitable form that would encourage their use as the initial design for sophisticated reasoning agents. Engineers willing to develop agents to perform in e-institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs. We show in Figure 1 a diagram describing the general approach: start-

suitable form that would encourage their use as the initial design for sophisticated reasoning agents. Engineers willing to develop agents to perform in e-institutions could then be offered a skeleton which would be gradually augmented into a complete program. Depending on the way skeletons are represented, semi-automatic support can be offered when augmenting them into more complex programs. We show in Figure 1 a diagram describing the general approach: start-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'02, July 15-19, 2002, Bologna, Italy.
Copyright 2002 ACM 1-58113-480-0/02/0807 ...$5.00.

696
ing correct and competent agents. Furthermore, any variation to be performed by the components such as the customisation of messages are not specified in the e-institution. If, for instance, a message offering an item is to be sent, the actual item which is offered is to be defined by whichever agent that actually partakes the e-institution. This variability is another capability that ought to be added to the initial skeleton.

Our choice of simple logic programs to represent skeletons is also supported by the wealth of research and results on automatic support and environments for logic programming development [3, 5, 6, 24]. Of particular importance to our proposal is the work on the systematic approach to logic program development using skeletons and programming techniques [3, 12, 18, 21]. According to this approach, an initial simple program which defines the flow of execution (a skeleton) is added with more features (the programming techniques). These are extra computations to be performed as the flow of execution, defined by the initial skeleton, is followed. Choosing to call our initial simple schema obtained from the e-institution a skeleton was deliberate: by doing so we inherit the results and experience from that previous related research work.

The activity of augmenting the skeleton shown in Figure 1 is thus given support: programming techniques are added at the designer's will conforming on the program additional capabilities. Program editing environments can be offered for this purpose, by which designers shift the focus of their attention: rather than seeing programs as sequences of characters (and adding or deleting them), programs are seen as "chunks" of constructs and operations over them. The addition of a parameter to a predicate, for instance, rather than requiring the appropriate editing of lines and characters to include the new parameter (with the likely risk of missing out on recursive calls or predicates with multiple definitions), becomes one single command which adequately alters all relevant portions of the program. We explain in more detail the augmenting activity in Section 4.

In the next section we introduce a "lightweight" version of e-institutions and provide a computer-processable means of representing them. In Section 3 we describe how this representation can be used to extract an initial design pattern, the so-called skeletons for our agents. In Section 4 we show how the initial design patterns can be used for developing a complete program and the existing work we shall build upon. In Section 5 we explain how skeletons and programs can be executed. Finally, in Section 6 we draw conclusions and give directions for further work.

2. "LIGHTWEIGHT" E-INSTITUTIONS

E-institutions are a means to specify interactions among heterogeneous agents in an open organisation [20]. It is assumed that the interaction among components is only through the exchange of messages. An e-institution is a formal description of the kinds and order of messages to be exchanged among agents within a specific scenario for well-defined purposes.

E-institutions are in essence non-deterministic finite-state machines (NDFSMs, for short) [10] comprising different states and edges. We shall present them here in a "lightweight" version in which those features not essential to our investigation will be omitted. For a complete description of e-institutions readers should refer to [14, 19]. Our lightweight e-institutions are defined as sets of scenes related by a transition. We shall assume the existence of a communication language $CL$ among the agents of an e-institution. We first define a scene:

**Def. (Scene)** A scene is the tuple $S = (R, W, \omega_0, W_A, WA, WE, \Theta, \Lambda)$ where

- $R = \{r_1, \ldots, r_n\}$ is the set of roles;
- $W = \{w_0, \ldots, w_n\}$ is a finite, non-empty set of states;
- $\omega_0 \in W$ is the initial state;
- $W_f \subseteq W$ is the non-empty set of final states;
- $WA$ is a set of sets $WA = \{WA_r, \subseteq W, r \in R\}$ where each $WA_r$, $r \in R$, is the set of access states for role $r$;
- $WE$ is a set of sets $WE = \{WE_r, \subseteq W, r \in R\}$ where each $WE_r$, $r \in R$, is the set of exit states for role $r$;
- $\Theta \subseteq W \times W$ is a set of directed edges;
- $\Lambda : \Theta \mapsto CL$ is a labelling function associating edges to messages in the agreed language $CL$.

To illustrate this definition we provide in Figure 2 a simple example of a scene for an agora room [15] in which an agent

![Figure 2: Simple Agora Room Scene](image)

willing to acquire goods interacts with a number of agents intending to sell such goods. This agora scene has been simplified - no auctions or negotiations are contemplated. The buyer announces the goods it wants to purchase, collects the offers from sellers (if any) and chooses the best (cheapest) of them. The simplicity of this scene is deliberate, so as to make the ensuing discussion and examples more concrete. A more friendly visual rendition of the formal definition is employed in the figure. Two roles $b$, for buyer and $s$, for seller, are defined. The initial state $\omega_0$ is denoted by a pair of concentric circles (leftmost state of scene); the only final state $W_f$ is represented by a thicker circle (rightmost state). Access states are marked with a "a" pointing towards the state with a box containing the roles that are supposed to enter the scene at that point. Exit states are marked with a "e" pointing away from the state, with a box containing the roles that may leave the scene at that point. For the sake of presentation, we have labelled the edges with numbers, which are defined on the right-hand side of the figure as Labels. A special label "nil" has been used to denote edges that can be followed without any action/event.

In our definition below, we equate an e-institution with a *performative structure* [20]. We are aware that these are not equivalent, but since the features that differentiate them will be left out of our present discussion, they can be safely considered to be the same.

**Def. (E-Institution)** An e-institution is the tuple $E = (SC, T, S_0, S_n, E, \Lambda_0)$ where

- $SC = \{S_1, \ldots, S_m\}$ is a finite and non-empty set of scenes;
- $T = \{t_1, \ldots, t_n\}$ is a finite and non-empty set of transitions;
- $S_0 \in SC$ is the root scene;
- $S_n \in SC$ is the output scene;

697
E = E' U E'' is a set of arcs such that E' ⊆ W^B × T is a set of edges from exit states W^E of scene S to transitions, and E'' ⊆ T × W^A is a set of edges from access states W^A of scene S to transitions;
• AS : E → p(x_1, ..., x_k) maps each arc to a predicate representing the arc's constraints.

Transitions can be seen as special states in between distinct scenes. We illustrate the definition above with an example comprising a complete virtual agric market [13]. This e-institution has more components than the above scene: before agents can take part in the agora they have to be admitted; after the agora room scene is finished, buyers and sellers must proceed to settle their debts. We show in Figure 3 a graphic rendition of an e-institution for our market. The.

Figure 3: E-Institution for Simple Agoric Market

Scenes are shown in the boxes with rounded edges. The root scene is represented as a double box and the output scene as a thicker box. Transitions are represented as hexagons. The arcs connect exit states of scenes to transitions, and transitions to access states. The labels of transitions have been represented as numbers. The same e-institution is, of course, amenable to different renditions.

2.1 Representing E-Institutions

We can devise a simple and natural way to represent institutions almost directly translating the definitions above. We shall employ Prolog [1]: this way our representation will already be in a computer-processable format. This would make it easier to extract our design patterns, as we shall see below. We show in Figure 4 our Prolog representation for the agora room scene graphically depicted in Figure 2.

roles(agra, [buyer, seller]). states(agra, [wd, w1, w2, w3]). initial_state(agra, w0). final_state(agra, w03).
states(agra, buyer, [w0, w1, w2]). states(agra, seller, [w0, w2]).
exit_states(agra, buyer, [w3]). exit_states(agra, seller, [w1, w2]).

theta(agra, [w0, request(buyer, all, seller, buy), (tomon, x11)].
theta(agra, [w0, offer(seeloler, bimber, seeloler, $gain, $price), w2]).
theta(agra, [w0, n1, w2]).
theta(agra, [w2, offer(seeloler, bimber, seeloler, $gain, $price), w2]).
theta(agra, [w2, inform(bimber, seeloler, reject($gain, $price), w3]).
theta(agra, [w2, n1, w3]).
theta(agra, [w3, inform(bimber, seeloler, reject($gain, $price), w3]).

Figure 4: Representation of Agora Room Scene

above. Each component of the formal definition has its representation. Since many scenes may co-exist within one e-institution, the components are parameterized by a scene name (first parameter). The G and Λ components of the definition are represented together in theta/2, where the second argument holds a list containing the directed edge as the first and third elements of the list and the label as the second element.

Any scene can be conveniently and economically described in this fashion. E-institutions are collections of scenes in this format, plus the extra components of the tuple comprising its formal definition. In Figure 5 we present a Prolog representation for the agora market e-institution. Of particular

importance are the arcs connecting scenes: these are represented as arc/3 facts stating the first argument of which holds (as a sublist) an exit state of a scene, the second argument holds the predicate (condition) p_i which enables

scenesis([admission, agora, settlement, departure]).
transitions([t1, t2, t3, t4, t5]).
root.scenesis(admission). output.scenesis(departure).
arc([admission, w0], p1, t1). arc([t1, p1, [1, 2]], departure, w0).
arc([admission, w0], p2, t2). arc([t2, p2, [1, 2], agora, w0], p3, t3).
arc([t3, p3, [1, 2], settlement, w0], p4, t4). arc([t4, p4, [1, 2], agora, w0], p5, t5).
arc([t5, p5, [1, 2], [departure, w0]].
arc([t6, p6, [1, 2], [departure, w0]].

Figure 5: Representation of Agoric Market

the arc and the third argument is the destination transition. Another arc/3 shows arcs leaving a transition and entering an access state of a scene.

3. SKELETONS AS LOGIC PROGRAMS

E-institutions are based on NDFSMs [10]. Their states are connected by directed edges labelled with messages that are sent by components (scenes) or by arcs labelled by predicates (transitions between scenes). Given an e-institution, we want to automatically extract essential information determining the behaviour of individual agents that will join in and interact with each other for some specific purpose(s). We shall call the representation for this essential information a skeleton of an agent.

The information obtained is to be used to restrict or define the possible behaviours of agents joining an e-institution. The very same e-institution can be employed for this purpose, but we want simpler and more specialized versions aimed at the individuals that will populate the enactment of the e-institution. The simplification and/or specialisation of an e-institution, however, is in the sense of obtaining portions of the original NDFSM that are relevant for specific agents. This process yields a hopefully smaller NDFSM.

The information of a NDFSM can be efficiently represented with any of the classic data structures employed with graphs [4]. However, we need to add to the static information of states and transitions the dynamics of a flow of execution. This flow of execution captures the informal mechanism we use when we try to follow a NDFSM. NDFSMs are abstract models that can be given different computational interpretations [10]: the very same automaton can be understood as a generator of correct output strings or as a device that accepts or rejects input strings. We also want to add to our representation some form of operational "meaning" of what happens when a transition is followed or triggered.

We propose logic programming for this purpose. The Horn clauses of logic programs are a compact formalism with precise declarative and procedural meanings. It provides a simple and natural means to represent our NDFSM as well as the flow of execution of such devices. Our proposal is exemplified in Figure 6: a non-deterministic state transition diagram is shown with its associated clauses. The meaning associated to NDFSMs is the following: if s_1 trans to s_2 is an
edge, then when the flow of the execution is in $s_1$, it should make $l_3$ happen and move to state $s_2$; alternatively, the flow of execution should wait until $l_3$ happens and then it should move to state $s_2$. Both these possibilities can be captured with our (Horn) clause proposal: the appropriate definition of predicate $\text{trans}$ that checks if a transition is given enable to rise to the different meanings.

Other forms of representation [4] will address separately the static information, i.e., the states and transitions, and the dynamic aspects of the model, i.e., how the static information is employed during the computations. Although such representations may be equivalent in terms of expressiveness, they are not so adequate for our needs, that is, a minimal representation for a NDFS that should be used as an initial design for a program.

There are other advantages in using clauses as a representation. The simplicity of this notation is complemented by the procedural meaning given by sound and complete proof procedures such as SLDNF resolution [22], efficiently implemented in different logic programming systems. If we assume this procedural interpretation, then it is enough to show the clauses that comprise our NDFS. Our representation is thus an actual, albeit simple, logic program with a precise semantics. Given a NDFS $M = (S, \Sigma, \delta, s_0, T)$, where $S = \{s_0, \ldots, s_n\}$ is the set of states (or vertices), $\Sigma = \{l_1, \ldots, l_m\}$ is the set of labels of transitions (we have used the more generic term "label of transitions" instead of an alphabet, as is the case in automata [10]), $\delta : S \times \Sigma \rightarrow S$ is a (partial) transition function, $s_0 \in S$ is a special state, the initial state, and $T \subseteq S$ is the set of terminal (or acceptance) states, then we can provide an automatic translation to our clause representation. For any $s, t \in S, \delta(s, t)$, such that $\delta(s, t) = t$, then we have $s \leftarrow \text{trans}(t) \land \text{state}(t)$ in our clause representation. Additionally, we can include clauses to record the initial and final states, completely defining a NDFS.

3.1 Extracting Skeletons from E-Institutions

It is straightforward to build a Prolog program to extract skeletons from our e-institutions represented as above. We show in Figure 7 portions of one such program. Predicate

\begin{verbatim}
cl.arc(R, arc((Sc,St),T),Clause):-
satisfy(R),
Clause = (\text{arc}(Sc,St,T)):-\text{holds}(\text{arc}(Sc,St,T)).
cl.arc(R, arc((Sc,St),T),Clause):-
satisfy(R),
Clause = (\text{arc}(Sc,St,T)):-\text{holds}(\text{arc}(Sc,St,T)).
cl.theta(R, theta(Sc,St,L,NSR1),Clause):-
L = \ldots,\ldots,L
Clause = (\text{theta}(Sc,St,L,NSR1)):-\text{send}(L,\text{theta}(Sc,St,L,NSR1)).
cl.theta(R, theta(Sc,St,L,NSR1),Clause):-
L = \ldots,\ldots,L
Clause = (\text{theta}(Sc,St,L,NSR1)):-\text{recv}(L,\text{theta}(Sc,St,L,NSR1)).
cl.theta(R, theta(Sc,St,L,NSR1),Clause):-
Clause = (\text{theta}(Sc,St,L,NSR1)):-\text{recv}(L,\text{theta}(Sc,St,L,NSR1)).
\end{verbatim}

Figure 7: Portion of Program to Extract Skeletons

4. PROGRAMMING WITH SKELETONS & TECHNIQUES

Automatic programming [2] has been a long-term goal of computer science in general [11] and, in particular, software engineering: programs being obtained via rigorous manipulation (i.e., by a computer) of intermediate formalisms. Programming is an activity that can be given different degrees of (automatic) support. We can see a spectrum of possibilities, ranging from completely automatic programming environments through to the completely manual and unsupported text-editing scenario. Somewhere in between these two extremes, lie the programming assistants. These are tools that support human programmers developing, reusing, documenting and maintaining their code [7, 17].

Logic programming, with its terse syntax, concise semantics and formal underpinnings, is particularly suitable for such support tools. One particular approach incorporates the classic methodology proposed by N. Wirth [26] by means of which an initial simple program is gradually refined and customised to the user's needs. The initial program is a skeleton and the refinements are techniques added to it [3]. Logic program development can thus be seen as a transformation activity [24] in which legal operations on a program
4.1 Prolog Programming via Skeletons & Techniques

To illustrate the skeletons and techniques approach, using a partial form of logic programming, viz. Prolog [1], we present an example in Figure 9. In this example, an initial skeleton for a Prolog program, s/1, to traverse a list (left-hand side) and test for specific components, is augmented with a technique t/1 to collect the specific components (middle box) in order to compose Prolog program p/1. The "a" operator appropriately joins the two fragments, making sure that the base-case (non-recursive) clauses appear together, and that the recursive clauses get "blend"ed" correctly. In order to match the respective recursive clauses together, the variable appearing in both skeleton and technique ought to be the same — although a variable X appears both in the skeleton and in the technique, the scope of a variable in both clauses is the clause in which it appears [1]. The resulting program p/1 joins the functionalities of skeleton (a list is traversed and its items are tested for some property) and the technique (those items that fulfill the test for some property are assembled together as a list). The flow of control, that is, the program's execution, is defined by the skeleton, whereas the computations to be performed as the execution proceeds are added by the technique.

The above "a" operator stands for the low-level operations on the programming constructs that take place for a complete program to be built. Although substantial support can be offered [3, 24] human intervention is still needed at points. For instance, in the example above it is required that the test/1 predicate be defined in order to have a complete program. Even though a tool could offer a library of likely tests, users may still want to develop their own routines. Again, help could be offered when auxiliary predicates are being developed, and so on, until the program is complete. The programming activity is thus redefined: an initial skeleton is chosen from an existing collection and techniques are applied to gradually obtain a program with the desired flow of execution and that performs the expected computations.

4.2 A Techniques-Based Prolog Programming Environment

It is possible to develop a programming environment incorporating the skeletons and techniques approach. In the case of Prolog, a high-level symbolic language that allows programs to be conveniently manipulated by a program written in the very same language, this has been successfully done [3, 24]. It is important, though, to separate a technique from its application. Since a technique is aimed at altering code, it is very tempting to represent it as an ordered sequence of editing steps, combining the actual technique (as shown in Figure 9 above) and its effects on the skeleton or program. Alternatively, we can dissociate these concepts, that is, consider a technique and its application as two independent (though closely related) components. Techniques are thus represented in a declarative way, free of implementational detail or particular usage in mind. The process of applying a technique is independently defined. An immediate benefit of this approach is that techniques have a clean and concise presentation format that would enable both engineers of the tool and its future users to quickly recognize and understand them.

An environment that makes such a distinction is defined in [24]. Techniques are represented as simple program transformations

\[
P(\bar{A}_1, \bar{A}_2) \quad \Rightarrow \quad P(\bar{A}_3)
\]

Figure 10: Technique as a Rewrite Rule

700
Stage 1
Figure 11: Agora Agent as a Stepwise Enhancement of a Skeleton

Stage 2

Stage 3

\[
\begin{align*}
\text{Stage 1} & \quad \text{Stage 2} \\
\text{Stage 3}
\end{align*}
\]

ation schemas \[23, 25\]. These are rewrite rules with program "templates", that is, abstract constructs that stand for classes of programs. We show in Figure 10 an example of a technique represented as a program transformation. The \( A \) constructs stand for vectors of arguments, that is, a possibly empty sequence of terms. \( F \) and \( G \) are meta-variables that abstract the actual predicate names. Construct \( c \) stands for a generic constant name and \( f(x, y) \) for a generic functor with two arguments, the second one of which is recursively defined. A program transformation is defined if a program matches the left-hand side schema, then it can be rewritten as the right-hand side. On the right-hand side schema, arguments are appropriately added to the program, with the same effect of the programming technique of Figure 9 above. The application of transformations such as the one above is precisely defined by a semi-unification algorithm [26]. Actual programming constructs are matched against the schematic constructs. This match will yield the new program with the prescribed added parts. Additional constraints on the schematic constructs can be defined, so as to narrow the possible matches.

An extensible programming environment has been defined using the above proposal. New techniques and skeletons can be added as needed. Different presentations of skeletons and techniques can be offered to the users, such as a brief explanation in English or a visual representation. The history of the preparation of a program can be recorded, and users may backtrack to previous points in order to change their design decisions. Program building is supplement with means to run the prototypes, debug and/or explain them, and have their efficiency analyzed and improved [24].

4.3 Working Example: Intelligent Agents for the Agora

In this section we show how skeletons extracted from e-institutions can be gradually augmented into more sophisticated programs. We show in Figure 11 an edited version of the sequence of steps followed. For brevity we only show portions of the original skeleton and how these get altered. The first box (Stage 1) contains the skeleton. The second box (Stage 2) shows the original skeleton added with constructs (underlined variables and goals). A technique which carries a stock data structure around as the execution proceeds has been added to the skeleton - this data structure is employed to obtain via predicate `chooseItem/2` the value of `Item` in the first clause of the part of the skeleton shown. Another technique has also been inserted to obtain via predicate `makeOffer/2` the value of variable `Price` in the second clause shown. The definitions for both these goals have to be supplied. The third box (Stage 3) shows the addition of another technique - the underlined constructs - to assemble a data structure `Mega`. This data structure stores the messages sent and received and is updated by means of calls to predicate `update/3` which should be defined.

In our example above we can see the gradual addition of functionalities to the initial skeleton. In order to sell items, the seller needs to have a representation for its stock. The buyer, on the other hand, needs a structure in which to store the items it needs to buy. The first technique, shown in the second box above, addresses these two requirements. The second box also shows another technique, the insertion of a predicate to obtain the value of variable `Price` when sellers make an offer. If an agent is to make adequate decisions about its when to send a message, then it should have an account of those messages already received and sent. The technique added in the third box above caters for this need.

The original set of behaviours of the skeleton is preserved in our extended program above. Ideally this should always happen, making sure that agents will perform correctly and efficiently/intelligently. If, however, we allow manual editing operations to be carried out, which sometimes cannot be avoided, the result may be a program in which certain original behaviours will not be reproduced. This could be the case, for instance, if a clause is deleted or an argument is inserted in the head goal of a clause. The behaviour represented in that clause will not be present in the execution of the extended program.

5. EXECUTING SKELETONS AND COMPLETE PROGRAMS

A skeleton should be, in principle, an executable logic program. Any non-determinism and/or missing design choices should ideally be dealt with by an a priori policy. This would guarantee that at least one possible behaviour could be automatically programmed, that is, a default behaviour without any sophistication but correct with respect to the e-institution it is aimed at.

For instance, assuming the usual SLDNF resolution as implemented in Prolog [1] whereby clauses are used in a top-down manner, any non-determinism involving two or more clauses depicting the transitions leaving one state would be deterministically solved. In such scenario the clauses are assumed to comprise an ordered sequence and the first clause obtained in this sequence which is successfully proven will be the one chosen. This feature, however, may not always be desired.

Logic programming has long been praised as a useful tool for meta-programming [8]. A meta-program is a program whose data denotes another (object) program, both of which
are in the same language. We can provide our programs with a meta-interpreter which allows the meta-reasoning about any non-deterministic choices. We show in Figure 12 a simple meta-interpreter for our needs. Its two initial clauses

\[
\text{meta}(0,0):= \\
\text{meta}(0), \\
\text{meta}(d0). \\
\text{meta}(0):= \text{system}(0), \\
\text{call}(0). \\
\text{meta}(0). \\
\text{meta}(Body, \text{clause}(d0, Body), Bodies), \\
\text{choose}(Body, \text{chosenBody}), \\
\text{meta}(\text{chosenBody}).
\]

Figure 12: Meta-Interpreter for Agents

are the usual meta-interpreter definitions for conjuncts (first clause) and system built-ins (second clause) [1, 8]. The third clause generalizes the usual meta-interpreter definition to handle user-created predicates: those clauses the head of which unify with goal 0 are all collected and one of their bodies is chosen via predicate choose/2 as the chosenBody that will be further used in the meta-interpretation. Predicate choose/2 ought to be defined by the agent's designer and should reflect the policies and attitudes of the agent regarding non-deterministic choices.

Interestingly, the use of meta-interpreters to control the execution of skeletons may provide us with another way to approach agent development. In this approach it is the meta-interpreter that is gradually augmented: the skeleton is kept unchanged and techniques are instead applied to the meta-interpreter, augmenting its capabilities. We show in Figure 13 the meta-interpreter of Figure 12 augmented with techniques for managing data structures for the stock and messages (received and sent) of the agent. This sec-

\[
\text{meta}(0,0, \text{Stock, NewStock, Ms, NewMs}):= \\
\text{meta}(0, \text{Stock, NewStock, Ms, NewMs}), \\
\text{meta}(0, \text{Stock, NewStock, Ms, NewMs}), \\
\text{update}(\text{send}(\text{N}), \text{Stock, NewStock}), \\
\text{meta}(\text{recv}(\text{N}), \text{Stock, NewStock, Ms, NewMs}):- \\
\text{update}(\text{recv}(\text{N}), \text{Stock, NewStock}), \\
\text{meta}(\text{Stock, Stock, Ms, Ms}):= \\
\text{agents}(\text{N}), \\
\text{call}(\text{N}). \\
\text{meta}(\text{Stock, Stock, Ms, NewMs}):= \\
\text{meta}(\text{body, clause}(\text{G-Body}), \text{Bodies}), \\
\text{choose}(\text{Body, chosenBody}), \\
\text{meta}(\text{chosenBody, Stock, NewStock, Ms, NewMs}).
\]

Figure 13: Augmented Meta-Interpreter

ond meta-interpreter has added features like those of our working example of Figure 11. One advantage of applying techniques to meta-interpreters is that these are normally compact pieces of code and hence are easier to alter and maintain. Another advantage lies in their potential for reuse: the very same (augmented) meta-interpreter can be used with distinct skeletons from disparate e-institutions.

Although agents programmed in any language can partake in an e-institution, we have detected an opportunity to help programmers design their agents. An account of the behaviours an agent ought to possess in an e-institution can be automatically extracted and used as a design "blueprint". We have provided an initial means to extract such behaviours and a way of representing them as "skeletons". Our skeletons are very simple logic programs that can be augmented in order to cope with non-determinism and idiosyncratic variations on behaviour (e.g. offering item a for sale in an auction scene).

The use of logic programming in our approach is backed by important advantages. Our skeletons which capture the agents' behaviours in the e-institution should be as simple as possible, yet they should provide enough information so as to allow the reproduction of the behaviours by a computer. Our representation of skeletons as logic programs fulfill both these requirements. Much support can be given in the task of customising the initial skeletons. Indeed, our choice of representing skeletons as simple logic programs means that we inherit a wealth of methods, research and results on providing support for logic program development. We have shown how skeletons can be gradually and safely altered onto more sophisticated programs.

As another means to cope with non-determinism, we have provided a simple meta-interpreter. This meta-interpreter can be augmented with programming techniques that exploit the underlying skeletons in their simple form, that is, without added techniques. A combined approach is also possible in which both skeletons and meta-interpreter are augmented with techniques.

We do not claim here that the actual format of our skeletons is the only possibility or indeed the best one. However, the information stored in them is necessary and sufficient for representing the behaviours of agents following an e-institution. The way we employ the e-institution specificiation to obtain the agents' behaviours is not dependent on the actual representation format we employ. A more graph-oriented representation with nodes, edges and labels could be used instead, with adequate graph-traversal programs to emulate the behaviours.

The extraction process always terminates after an exhaustive combination of the roles with each transition of the scene, at an exponential cost on the number of roles. The small number of roles should hopefully keep this exponential cost low. The extracted skeleton can, however, be used differently. It can, for instance, be the input for programs written in a language such as C++ or Java, which should be able to parse and "run" the skeleton. Such programs would play the role of the meta-interpreters of Section 5.

We are currently customising for our purposes the programming environment proposed in [24] and explained in Section 4.2. In our environment we offer users standard means to augment skeletons, such as adding parameters to store data structures and predicates to manage them. We also offer a library of meta-interpreters which users can experiment with. Our environment also supports the disciplined alteration of meta-interpreters via programming techniques. The newly altered meta-interpreters can be stored in the library and offered for later reuse.

We believe our proposal should make e-institutions more practical and useful in the design of heterogeneous multi-agent systems. There are, however, important related is-

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel manner of programming agents. We specifically address the context in which agents have to be designed to follow an e-institution. An initial design pattern is automatically extracted from a given e-institution and offered to programmers willing to develop agents for the specific purpose of joining in and performing in that e-institution.
7. REFERENCES