

Skeleton-based Agent Development for Electronic Institutions

Wamberto W. Vasconcelos

Division of Informatics
University of Edinburgh
80 South Bridge, EH1 1HN, Edinburgh, UK
wvasconcelos@acm.org

Jordi Sabater, Carles Sierra, Joaquim Querol

IIIA - Artificial Intelligence Research Institute
CSIC - Spanish Scientific Research Council
Bellaterra, Catalonia, Spain
jsabater,sierra,joaquin@iiia.csic.es

ABSTRACT

In this paper we describe work in progress concerning the (semi-)automatic support for developing agents. We focus on the scenario in which agents have to be designed to follow an electronic institution. An initial design pattern is automatically extracted from a given electronic institution and offered to programmers willing to develop agents for the specific purpose of joining and performing in the electronic institution. We resort to logic programming as our underlying computational framework, explaining and justifying this decision.

1. INTRODUCTION

Electronic institutions (e-institution, for short) have been proposed as a formalism with which one can specify open agent organisations [17]. 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 should 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.

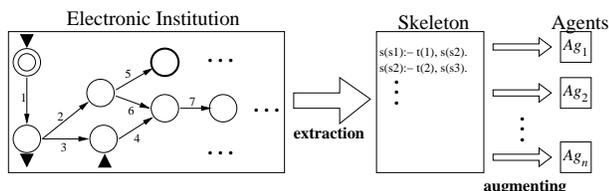


Figure 1: Overview of Proposed Approach

The idea 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 its 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 **Fig. 1** a diagram describing the general approach: starting from a given e-institution, a skeleton is extracted; the skeleton is augmented in different ways, yielding different agents; these agents can then perform in an enactment of the e-institution they were designed for.

Skeletons should ideally be in a computer processable format. By this we mean that the behaviours represented by them should be reproducible by a computer. This way we do not have to perform further translations from an abstract format onto more computer-oriented representations – skeletons, after all, should guide designers in the development of their agents. We propose that our skeletons be simple logic programs: the terse syntax, the precise declarative and procedural meanings and the ease with which one can write meta-programs to obtain alternative executions are some of the advantages of our proposal. We explain in more detail our representation for skeletons in Section 3 below.

A skeleton should define the basic behaviours agents should possess to successfully perform in the e-institution they are designed for. Our skeletons are simple logic programs with very limited functionality: the state of the computation, in the form of a label, is stored and transitions are followed. However, e-institutions are non-deterministic and there might be states of the computation from which more than one next state is possible. When a rational agent follows an e-institution, any non-determinism should be tackled by formal reasoning and decision-making procedures. The augmenting process which skeletons undergo (see **Fig. 1**) is aimed at filling in such capabilities. Reasoning and/or decision-making procedures have to be appropriately added to the initial skeleton, yielding correct and efficient/competent agents.

Any variability or customisation of messages to be performed by components 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 representation for a skeleton, that is, a simple logic program, is also supported by the wealth of research and results on automatic support and environments for logic programming development [3, 6, 7, 9, 16, 29]. Of particular

importance to our proposal is the work on the systematic approach to logic program development using *skeletons* and *programming techniques* [3, 4, 15, 16, 21, 23]. According to this approach, an initial simple program, a *skeleton*, defining the flow of execution, 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 shall inherit the results and experience from that previous research work.

The activity to augment the skeleton shown in **Fig. 1** is thus given support: programming techniques are added at the designer’s will conferring on the program additional capabilities. Program editing environments can be offered for this purpose by means of which designers shift the focus of their attention: rather than seeing programs as sequences of characters (and adding/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 below.

In the next section we introduce a “lightweight” version of e-institutions and provide a computer-processable means of representing. 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

Electronic institutions are means to formally specify interactions among heterogeneous agents in an open organisation [17]. It is assumed that the interaction among components is only by means of messages being exchanged. 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 basically NDFSM [13] with different states and edges. We shall present them here in a “lightweight” version in which those features not essential to our investigation will be left out. For a complete description of e-institutions refer to [22, 18]. Our lightweight e-institutions are defined as sets of *scenes* related by *transitions*. We shall assume the existence of a communication language CL among the agents of an e-institution. We shall first define a scene:

Def. (Scene) A scene is the tuple

$$\mathbf{S} = \langle R, W, w_0, W_f, WA, WE, \Theta, \lambda \rangle$$

where

- $R = \{r_1, \dots, r_n\}$ is the set of agent *roles* of the scene;
- $W = \{w_0, \dots, w_m\}$ is a finite, non-empty set of *states* of the scene;
- $w_0 \in W$ is the *initial state* of the scene;
- $W_f \subseteq W$ 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 messages of the agreed language M to the edges.

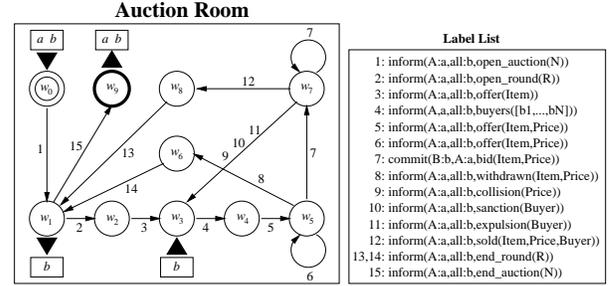


Figure 2: Auction Room Scene

To illustrate this definition we provide in **Fig. 2** a simple example representing a virtual auction room. A more friendly visual rendition of the formal definition is employed. Two roles a , for auctioneer, and b , for bidder, are defined. The initial state w_0 is denoted by a pair of concentric circles (top left corner of scene); the only final state w_9 is represented by a thicker circle. Access states are marked with 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 “▲” 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 in the **Labels** box.

In our definition below, we have equated an e-institution with a *performative structure* [17]. We are aware that these are not equivalent, but since the features that differentiate them will be left out of our present discussion, we can equate them.

Def. (E-Institution) An e-institution is the tuple $\mathcal{E} = \langle S, T, \mathbf{S}_0, \mathbf{S}_\Omega, E, \lambda_E \rangle$ where

- $S = \{\mathbf{S}_1, \dots, \mathbf{S}_n\}$ is a finite and non-empty set of scenes;
- $T = \{t_1, \dots, t_m\}$ is a finite and non-empty set of *transitions*;
- $\mathbf{S}_0 \in S$ is the *root scene*;
- $\mathbf{S}_\Omega \in S$ is the *output scene*;

- $E = E^I \cup E^O$ is a set of arcs such that $E^I \subseteq WE^S \times T$ is a set of edges from exit states WE^S of scene S to transitions, and $E^O \subseteq T \times WA^S$ is a set of edges from access states WA^S of scene S to transitions;
- $\lambda_E : E \mapsto p(x_1, \dots, x_k)$ maps each arc to a predicate representing the arc's constraints.

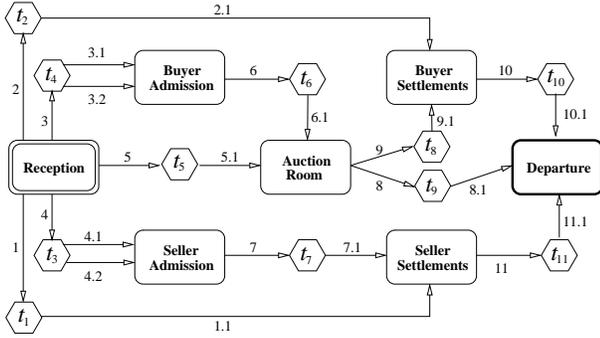


Figure 3: E-Institution for Auction House

Transitions can be seen as special states in between distinct scenes. We illustrate the definition above with an example comprising a complete virtual auction house. This e-institution has more than the above scene: before agents can take part in an auction they have to be admitted; after an auction session, *i.e.* the auction room scene is finished, buyers and sellers must proceed to settle their debts. We show in Fig. 3 a graphic rendition of an e-institution for our auction house. The scenes are shown in the boxes with round 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 visual 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, 24]: 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

```

roles(auction_room,[bidder,auctioneer]).
states(auction_room,[w0,w1,w2,w3,w4,w5,w6,w7,w8,w9]).
initial_state(auction_room,w0).
final_states(auction_room,[w11]).
access_states(auction_room,bidder,[w0,w3]).
access_states(auction_room,auctioneer,[w0]).
exit_states(auction_room,bidder,[w1,w9]).
exit_states(auction_room,auctioneer,[w9]).
theta(auction_room,[w0,inform(A:auctioneer,all:bidder,
open_auction(N)),w1]).
theta(auction_room,[w1,inform(A:auctioneer,all:bidder,
open_round(R)),w2]).
...
theta(auction_room,[w1,inform(A:auctioneer,all:bidder,
end_auction(N)),w9]).

```

Figure 4: Prolog Representation for Auction Room Scene

```

scenes([reception,auction_room,...,departure]).
transitions([t1,...,t11]).
root_scene(reception).
output_scene(departure).
arc([[reception,w3],p1,t1]).
arc([[t1,p1.1],[seller_settlements,w0]]).
...

```

Figure 5: Prolog Represent. for Auction House

see below. We show in Fig. 4 our Prolog representation for the auction room scene graphically depicted in Fig. 2 above. Each component of the formal definition has its corresponding fact. Since many scenes will co-exist within one e-institution, the facts are parameterised by a scene name (first parameter). The Θ 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 simply collections of scenes in this format, plus the extra components of the tuple comprising its formal definition. We show in Fig. 5 a Prolog representation for the auction house e-institution. Of particular importance are the arcs connecting scenes: these are represented as `arc/1` facts storing a list the first element of which holds (as a sublist) the exit state of the scene, the second element holds the predicate (condition) for the arc and the third element is the transition. Another `arc/1` shows the arc leaving a transition and entering an access state of a scene.

3. SKELETONS AS LOGIC PROGRAMS

E-institutions are based on non-deterministic finite state machines (NDFSM) [13]. Its states are connected by directed edges labelled with messages that are sent by components (agents) or the arcs are 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 specialised 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. Nevertheless, it will still be a NDFSM.

The information of a NDFSM can be efficiently represented with any of the classic data structures employed with graphs [5]. 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. NDFSM are abstract models that can be given different computational interpre-

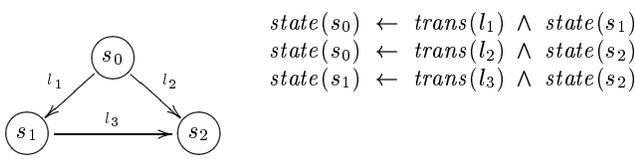


Figure 6: State Transitions and Corresponding Horn Clauses

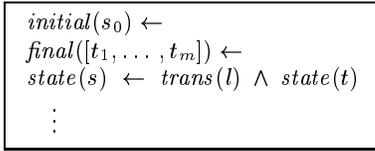


Figure 7: NDFSM as Logic Program

tations [13]: 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 to employ 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 $s_1 \xrightarrow{l_1} s_2$ is an edge, then when the flow of the execution is in s_1 , it should make l_1 happen and move to state s_2 ; alternatively, the flow of execution should *wait* until l_1 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 *trans* that checks if a transition is enabled can give rise to the different meanings.

Other forms of representation [5] 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 NDFSM which should be used as an initial design for a program.

There are other advantages in using our clauses. The simplicity of this notation is complemented by the procedural meaning given by sound and complete proof procedures such as SLD(NF) resolution [26], efficiently implemented in different logic programming systems [27]. If we assume this procedural interpretation, then it is enough to show the clauses that comprise our NDFSM. Our representation is thus an actual, albeit simple, logic program with a precise semantics. Given a NDFSM $M = (S, \Sigma, \delta, s_0, T)$, where $S = \{s_0, \dots, s_n\}$ is the set of states (or vertices), $\Sigma = \{l_1, \dots, l_m\}$ 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 [13]), $\delta : S \times \Sigma \mapsto S$ 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 $s, t \in S, l \in \Sigma$, such that $\delta(s, l) = t$, then we have $state(s) \leftarrow trans(l) \wedge state(t)$ in our clause representation. Additionally, we can include clauses to record the initial and final states, completely defining a NDFSM. We show in **Fig. 7** the general format of a logic program representing an NDFSM.

The agent skeleton are composed of two pieces of code. One piece contains the prolog predicates generated from the formal specification of the institution. The second piece of code is a set of predicates that implement basic functionalities such as message exchange and that is fixed for all e-institution specifications. The generation process is divided in two steps.

The first step consists of a syntactic transformation from the formal specification of an institution into a suitable set of prolog predicates to be used in the second step. For instance, **Fig. 8** shows a simple e-institution with two scenes and one transition and the correspondent prolog representation of one scene and the e-institution.

The second step uses the prolog representation (**Fig. 8.b**) to generate prolog code (**Fig. 8.c**) that structures the agent message generation and reception according to the NDFSM with the scenes and the transitions between scenes. Appropriate hooks are leaved within the code to permit the late addition of specific decision making code when more than one path can be followed from a given state in the NDFSM (see Section 4) and when other actions associated to specific transitions in the NDFSM can be made.

States of an e-institution are uniquely defined by their label and the scene they are part of. The role that a component plays within an e-institution is also employed when we want to follow an e-institution. In **Fig. 8.c** we show a portion of an e-institution for an auction room: an auctioneer agent initially in state s_0 broadcasts a message offering an *Item* to all bidders (first clause of $st/1$) then it moves to state s_1 . A bidder agent stays in the initial state s_0 until it gets a message from the auctioneer offering an item (second clause of $st/1$). A list was used to store all relevant components – this way only one argument is required in the skeleton.

4. PROGRAMMING WITH SKELETONS AND TECHNIQUES

Designers willing to develop agents for a specific e-institution can make use of the provided skeleton as explained above. The skeleton can be altered manually, that is, the designer proceeds to edit it, adding the functionalities by means of a text editor, inserting parameters, variables, extra literals, and so on. However, this approach is not very efficient for a number of reasons. Editing a program by means of line-editing commands is both time-consuming (for instance, inserting a parameter in one predicate definition requires the programmer to go through the whole program, making sure the arity is consistent throughout) and error-prone (for instance, in the same case of inserting a parameter, if a single occurrence is left out, an error may occur).

Automatic programming [2] has been a long-term goal of

a.1) Scene “ar”

```
R = {auctioneer, bidder}
W = {w0, w1, w2}
W0 = w0
Wf = {...}
θ = {(w0 w1), (w1 w1), (w1 w2)}
λ((w0 w1))={m(auctioneer, all,
               offer(Item)),
            m(auctioneer, bidder,
               offer(Item))}
λ((w1 w1))={m(bidder, auctioneer,
               offer(Item, price))}
λ((w1 w2))={m(bidder, auctioneer,
               offer(Item, price))}
```

⇒

b.1) Scene “ar” predicates

```
ar([(w0,w1,m(auctioneer,all,
              offer(Item))),
    (w0,w1,m(auctioneer,bidder,
              offer(Item))),
    (w1,w1,m(bidder,auctioneer,
              offer(Item,price))),
    (w1,w2,m(bidder,auctioneer,
              offer(Item,price))
    ]).

access-state(w0).
exit-state([w0,w1,w2]).
```

⇒

```
initial(w0).
terminal(...).
st([w0,ar,auctioneer]):-
  sendM([myself,all,
         offer(Item)]),
  st([w1,ar,auctioneer]).
st([w0,ar,bidder]):-
  recM([auctioneer,myself,
        offer(Item)]),
  st([w1,ar,bidder]).
st([w1,ar,bidder]):-
  sendM([myself,auctioneer,
         offer(Item,Price)]),
  st([w2,ar,bidder]).
st([w1,ar,auctioneer]):-
  recM([bidder,myself,
        offer(Item,Price)]),
  st([w1,ar,auctioneer]).
```

a.2) E-institution

```
S={scn1,scn2}
T={tr1}
S0={scn1}
S1={scn2}
E={{(scn1 tr1)},{(tr1 scn2)}}
λe(({scn1 tr1})=p(auctioneer,bidder)
λe(({tr1 scn2})=p(auctioneer,bidder)
```

⇒

b.2) E-institution predicates

```
e_institution([
  (scn1,tr1, p(auctioneer,bidder)),
  (tr1,scn2, p(auctioneer,bidder))
]).
s0(scn1).
sw(scn2).
```

⇒

c.2) E-institution

```
initial-scn(scn1).
terminal-scn(scn2).

e_inst([scn1,auctioneer]):-
  e_inst([tr1,_]).

e_inst([tr1,_]):-
  e_inst([scn2,auctioneer]).

e_inst([tr1,_]):-
  e_inst([scn2,bidder]).
```

a) Formal specification

b) Prolog specification

c) Prolog implementation

Figure 8: Example prolog Skeleton for an E-Institution

computer science in general [14] and, in particular, software engineering: programs can be obtained via the 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 [10, 20].

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 [31] 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, 4]. Logic program development can thus be seen as a *transformation* activity [25, 29] in which legal operations on a program (adding techniques) must preserve desirable properties (such as, for instance, termination) [19].

4.1 Prolog Programming via Skeletons and Techniques

To illustrate the skeletons and techniques approach, using a particular form of logic programming, *viz.* Prolog [1], we show in Fig. 9 an example. 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 “•” operator appropriately joins the two fragments, making sure that the base-case (non-recursive) clauses appear together, and that the recursive clauses get “blended” correctly. In order to match the respective recursive clauses together, the *X* 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 Horn 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 fulfil 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 “•” operator is an abstraction. It 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, 29] 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. Although a tool could offer a

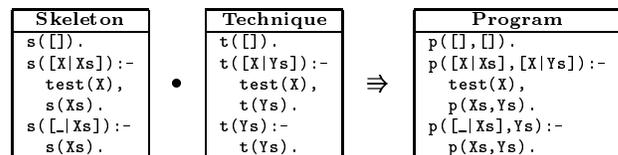


Figure 9: Skeleton • Technique ⇒ Program

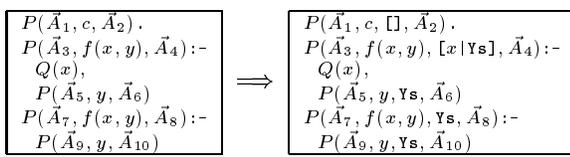


Figure 10: Technique as Rewrite Rule

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, 29]. Although we do not claim here this would be impossible in languages like C++ or Java, one cannot deny the effort involved in writing such a tool in these low-level languages.

It is important, though, to separate a technique from its application process. Since a technique is aimed at altering code, it is very tempting to represent it as an ordered sequence of editing steps, mixing together the actual technique (as shown in Fig. 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 (although closely related) components. Techniques are thus represented in a declarative way, free of implementational details and 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 recognise and understand them.

An environment that makes such a distinction is defined in [29]. Techniques are represented as simple program transformation schemata [8, 28, 30]. These are rewrite rules with program “templates”, that is, abstract constructs that stand for *classes of programs*. We show in Fig. 10 an example of a technique represented as a program transformation. The \bar{A}_i constructs stand for *vectors of arguments*, that is, a possibly empty sequence of terms. P and Q 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 Fig. 9 above. The application of transformations such as the one

above is precisely defined by a semi-unification algorithm [30]. 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 at the user’s will. Different presentations of skeletons and techniques can be offered to our 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. The program building activities can be supplemented with means to run the prototypes, debug and/or explain them, and have their efficiency analysed and improved [29].

4.3 Working Example: Intelligent Agents for Auctions

In this section we show how the skeleton extracted from the auction e-institution can be gradually augmented onto a more sophisticated program. We show in Fig. 11 an edited version of the sequence of steps followed.

To save space we only show portions of the original skeleton and how these get altered. The first box contains the skeleton. The second box 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 of for both these new goals have to be supplied. The third box shows the addition of another technique – the underlined constructs – to assemble a data structure `Msgs`. 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 offer items to be auctioned, the auctioneer needs to have a representation for its stock. The bidder, on the other hand, needs a structure in which to store the items it successfully bids for. 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 bidders place a bid. If an agent is to make adequate decisions about if/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

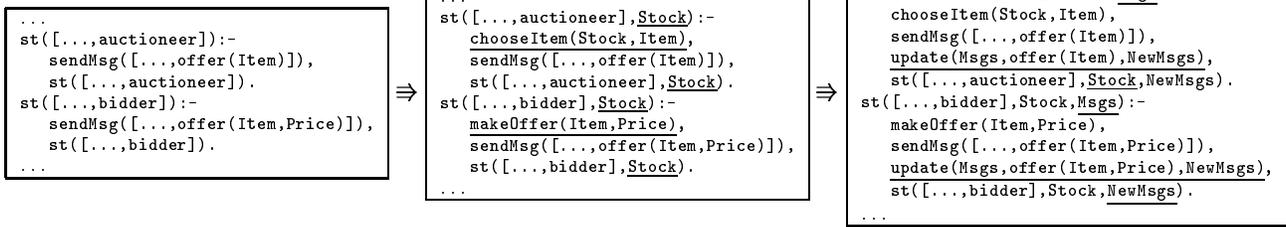


Figure 11: Auction Agent as a Stepwise Enhancement of Skeleton

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, a runnable 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.

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 same 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 [11]. 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 Fig. 12 a simple meta-interpreter for our needs. We aim here only at those programs with one single argument in the definition of their `st/1` predicates. The program should be *preceded* by the meta-interpreter shown. This way we take over Prolog's top-down clause selection: whenever the execution control tries to prove `st/1` it employs our extra clause for `st/1`

```

st(State):-
  setof(Body,clause((st(State):-Body)),Bodies),
  chooseB(Bodies,ChosenBody),
  meta(ChosenBody).
meta((G,Gs):-
  meta(G),
  meta(Gs).
meta(G):-
  system(G),
  call(G).
meta(G):-
  clause((G:-B)),
  meta(B).

```

Figure 12: Meta-Interpreter for Agents

shown at the very beginning of our meta-interpreter. This clause prescribes that when trying to prove `st/1` then collect via built-in `setof/3` the bodies `Bodies` of all those clauses `st(State):-Body` the head of which unifies with `State`, that is, all the transitions leaving `State`. After this, choose `ChosenBody` from `Bodies` via predicate `chooseB/2` (to be defined by the designers) and meta-interpret it via `meta/1`. The remaining clauses follow the usual definition for a Prolog meta-interpreter [1, 11, 24].

6. CONCLUSIONS AND DIRECTIONS OF RESEARCH

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 electronic institution (e-institution, for short). An initial design pattern is automatically extracted from a given e-institution \mathcal{E} and offered to programmers willing to develop agents for the specific purpose of joining and performing in \mathcal{E} .

Although agents programmed in any language can partake 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 “blue print”. We have provided an initial means to extract these behaviours and a way of representing them via *skeletons*. Our skeletons are very simple logic programs that can be augmented in order to cope with non-determinism and idiosyncratic variations on behaviour.

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 fulfil 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 composed onto more sophisticated programs.

Given an e-institution, there might be properties that ought to be maintained in skeletons. We want to investigate such properties possibly with model-checking techniques [12] and guarantee that these are preserved during the skeleton extraction stage.

Acknowledgements

This work has been sponsored by the European Union, contract IST-1999-10208, research grant **Sustainable Life-cycles in Information Ecosystems** (SLIE) and the Spanish project “Transacciones comerciales automatizadas por agentes autónomos inteligentes mediante instituciones electrónicas” (TIC2000-1414).

7. REFERENCES

- [1] K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.
- [2] A. W. Biermann. Automatic Programming. In *Encyclopedia of Artificial Intelligence*, volume 1. John Wiley & Sons, 1992.
- [3] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994.
- [4] P. Brna, P. Bundy, T. Dodd, C. K. Eisenstadt, M. Looi, H. Pain, D. Robertson, B. Smith, and M. Van Someren. Prolog Programming Techniques. *Instructional Science*, 20(2):111–133, 1991.
- [5] T. H. Cormen, Leiserson C. E., and R. L. Rivest. *Introduction to Algorithms*. MIT Press, USA, 1990.
- [6] Y. Devilles. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
- [7] M. Ducassé and J. Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *Journal of Logic Programming*, 19, 20:351–384, 1994.
- [8] N. E. Fuchs and M. P. J. Fromherz. Schema-Based Transformations of Logic Programs. In *Proc. LoPStr’91*. Springer-Verlag, 1992.
- [9] T. S. Gegg-Harrison. Learning Prolog in a Schema-Based Environment. *Instructional Science*, 20:173–192, 1991.
- [10] A. T. Goldberg. Knowledge-Based Programming: A Survey of Program Design and Construction Techniques. *IEEE Trans. on Soft. Eng.*, SE-12(7):752–768, July 1986.
- [11] P. M. Hill and J. Gallagher. Meta-Programming in Logic Programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 421–498. Oxford University Press, January 1998.
- [12] G. J. Holzmann. The SPIN Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
- [13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, U.S.A, 1979.
- [14] W. H. Kautz, E. A. Voorhees, and T. A. Jeeves. Automatic Programming Systems. *Comm. of the ACM*, 1(8):6–8, August 1958.
- [15] M. Kirschenbaum, A. Lakhota, and L. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Computer Engineering and Science Department, Case Western Reserve University, Ohio, U.S.A., 1989.
- [16] M. Kirschenbaum, S. Michaylov, and L. Sterling. Skeletons and Techniques as a Normative Approach to Program Development in Logic-Based Languages. OSU-CISRC-5/94-TR25, Department of Computer and Information Science, Ohio State University, Ohio, U.S.A., 1994.
- [17] J. A. Rodríguez Aguilar, F. J. Martín, P. Noriega, P. García, and C. Sierra. *Towards a Formal Specification of Complex Social Structures in Multi-Agent Systems*, pages 284–300. Number 1624 in LNAI. Springer, 1997.
- [18] Pablo Noriega. *Agent-Mediated Auctions: The Fishmarket Metaphor*. PhD thesis, IIIA-CSIC, 1997.
- [19] M. Proietti and A. Pettorossi. Transformations of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19, 20:261–320, 1994.
- [20] C. Rich and Y. A. Feldman. Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. *IEEE Trans. on Soft. Eng.*, 18(6):451–469, June 1992.
- [21] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conference on Logic Programming*, Edinburgh, Scotland, April 1991. Springer-Verlag.
- [22] Juan Antonio Rodriguez. *On the Design and Construction of Agent-mediated Electronic Institutions*. PhD thesis, IIIA-CSIC, 2001.
- [23] L. Sterling and M. Kirschenbaum. Applying Techniques to Skeletons. In *Constructing Logic Programs*. John Wiley & Sons Ltd, London, England, 1993.
- [24] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 2nd edition, 1994.
- [25] H. Tamaki and T. Sato. A Transformation System for Logic Programs which Preserves Equivalence. Technical Report TR 83-18, ICOT Research Center, 1983.
- [26] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of ACM*, 23(4):733–742, October 1976.
- [27] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, May/July 1994.
- [28] W. W. Vasconcelos and N. E. Fuchs. An Opportunistic Approach for Logic Program Analysis and Optimisation using Enhanced Schema-Based Transformations. volume 1048 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
- [29] W. W. Vasconcelos and N. E. Fuchs. Prolog Program Development via Enhanced Schema-Based Transformations. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1996. Presented at the 7th Workshop on Logic Programming Environments, held in conjunction with ILPS’95, Seattle, U.S.A.
- [30] W. W. Vasconcelos and E. X. Meneses. A Practical Approach for Logic Program Analysis and Transformation. volume 1793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
- [31] N. Wirth. Program Development by Stepwise Refinement. *Comm. of the ACM*, 14(4):221–227, 1971.