Towards Utility-based Programming

Alcino Cunha and José Barros

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
Tel: +351253604470, Fax: +351253604471
{alcino,jbb}@di.uminho.pt

Abstract. Many programs have an objective that can be precisely stated as the maximization of a function defined over its local variables. This is the case of utility-based software agents, which are reactive entities that try to maximize their welfare, usually accessed by an utility function. This paper introduces a programming language suitable for explicit programming with utility functions. Starting from a standard concurrent programming language, we added primitives to allow the parametrization of each process with an utility function that should be maximized. For the moment, using techniques of Markov decision problems, we can compile sequential programs, written in a restricted version of this new language, into equally behaved programs written in the original one. Major problems in developing such utility-based programming language are the need to compare the utility of infinite executions or the need to deal with uncertainty.

1 Introduction

Artificial Intelligence (AI) researchers do not agree about which aims one should follow when developing an intelligent system. Usually, it is possible to identify four major categories for this objectives [14]: to develop systems that think like humans, systems that act like humans, systems that think rationally or systems that act rationally. The main difference between an approach oriented towards the human behavior and one built upon the notion of rationality, is that, while on the later it is possible to define precisely the optimal way of acting and thinking, on the former we have to cope with the frequent "irrationality" and diversity of human behavior. Due to this difference, the AI community is increasingly orienting its efforts towards rationality centered approaches, and specially towards the development of useful systems that act rationally, usually denominated intelligent agents.

There are several ways which can be used to characterize precisely the notion of rational behavior. The most common consists of defining the agent's objectives as a set of goals, or states of computation, that must be reached. Although this approach is very well studied and frequently used, it is not sufficiently flexible in order to allow quality decision making in complex problems. A better approach consists of defining the objectives as an utility function, that assigns to each

state a numerical value which represents the level of satisfaction "perceived" by the agent in that state. With this approach, the rational behavior can be defined as the one that allows the maximization of the utility function. An agent whose objectives are defined by an utility function is denominated *Utility-based Agent*.

The above definition of rational behavior for an utility-based agent is still not very precise, since it does not take into account the horizon of the decision making process: nothing was said about when the utility function should be maximized. There are several acceptable possibilities for the so called decision criteria, such as, for example, to maximize the utility function at the next state or at the end of the execution. This problem is further enhanced if we think that agents are reactive entities, that interact continuously with the environment, and usually programmed with an infinite life horizon. In this setting there is not a consensual decision criteria to be used, and the choice of which one to use should be left to the agent designer.

In the last years there as been an effort to develop languages to program intelligent agents, giving rise to the *Agent-Oriented Programming* (AOP) paradigm. This term was first coined by Yoav Shoham, when developing his language AGENT0, in order to refer to a paradigm where agents are programmed directly in terms of the mentalistic or intentional notions developed by the AI theorists to represent their properties [15]. Within the AOP paradigm several other languages have been developed, such as, for example, PLACA [16], Concurrent METATEM [8], or AgentSpeak(L) [13].

The language introduced in this paper contributes to the AOP paradigm, and is specially targeted at the development of utility-based agents. Most of the typical agent programming languages try to match the high-level multimodal logical theory used to specify properties about agents. Our approach differs significantly, for it consists in choosing an existing concurrent language, and introduce the minimal number of constructs that allow one to specify the agent's "desires", represented by the utility-function and the decision criteria. The resulting language is rather low-level and resembles a typical concurrent and imperative programming language. A concurrent language is used due to the need of modeling the context where the agent interacts, usually populated by other similar entities. Although it started in the AOP context, we believe that this language can be used in many other contexts besides agent programming, since there are many programs whose objective can be precisely stated as the maximization of a function defined over its local variables.

The main problems in developing this language are the need to deal with infinite horizons and with uncertainty in the decision process implicitly modeled by a program. The first problem derives from the possible existence of infinite loops in the programs. The second is a consequence of the concurrency primitives. When several utility-based agents are modeled by different and concurrent processes, it is no longer possible for each of them to predict with certainty what is the present state of the world, and consequently, what is the precise result of executing an instruction. In order to simplify the language development, we divided the project into two major phases. First we study the sequential version

of the language, where only a process is allowed, in order to model an isolated utility-based agent. Later we shall introduce the concurrency primitives. This division allows us to deal with the mentioned problems one at a time. This paper presents some results of the first phase of the project.

This paper begins with the presentation of the syntax and semantics of the sequential version of the language. We then present the compilation technique, developed in order to translate utility-based programs into equally behaved ones written in the original language. Afterwards we introduce some very preliminary results concerning a refinement method for this new language. Finally, we present some conclusions and future work, namely a brief discussion about the major implications of introducing concurrency.

2 The Utility-based Simple Programming Language

The *Utility-based Simple Programming Language* (USPL) presented in this paper is based on a subset of the *Simple Programming Language* (SPL), introduced by Manna and Pnueli to specify reactive systems [11]. The main reasons to use SPL as a starting point to develop our language were:

- As mentioned in section 1, our long term objective is to develop an utility-based concurrent programming language. As such, it is convenient to start from a concurrent language. The SPL, being a rather generic specification language, accommodates several concurrent programming paradigms and primitives, namely both communication by message passing and shared memory. This flexibility is desirable because, at this point of developing, we do not know which paradigm will better suit our purposes.
- Sometimes it will be necessary to impose conditions upon the programs (see, for example, section 5 on the refinement process). These conditions will be expressed in linear temporal logic, which is the language chosen by Manna and Pnueli to specify and prove properties about SPL programs. Moreover, there exists already a system, the *Stanford Temporal Prover* (STeP) [4], that can be used to verify these conditions.

The sequential subset of SPL that we will use in USPL is very simple. There are only two basic statements - guarded assignment (`guard`), and active awaiting (`await`) - and three composite statements - concatenation (`;`), non-deterministic choice (`or`), and while loop (`while`). This subset can be used to implement almost every sequential instruction of SPL, such as conditionals, repeat loops, standard assignment, or skip. The major limitation of USPL consists of a single type for variables: the finite range of integers. This limitation was imposed at this stage of development in order to guarantee a finite set of states. The initial value of a variable can be restricted by a condition.

The major innovation of USPL is the possibility to parametrize the process with an utility function - an expression defined over the variables that expresses its preferences - and with a decision criteria, that defines the way these preferences are extended to infinite computations. This innovation changes the expected behavior of the non-deterministic choice: now it models a decision node,

program ::= [*declarations*] *process*
declarations ::= *declaration* {`;` *declaration*}
declaration ::= *mode variable* `:` `[` *int* `..` *int* `]` [`where` *boolexp*]
mode ::= `in` | `out` | `local`
process ::= *id* `(` *intexp* `,` *criteria* `)` `::` `[` *instruction* `]`
criteria ::= `look-at-end`
| `look-ahead` *int*
| `discounted` `0.`*int*
instruction ::= `[` *instruction* `]`
| *instruction* `;` *instruction*
| *instruction* `or` *instruction*
| `while` *boolexp* `do` *instruction*
| `guard` *boolexp* `do` *variable* `:=` *intexp*
| `await` *boolexp*

Fig. 1. The syntax of USPL.

where the choice should be done among the enabled instructions that lead to the most preferred executions. The syntax of USPL is presented in figure 1. The definition of some non-terminals is omitted, but it should be clear from context.

At present, only three decision criteria are allowed in USPL:

- `look-at-end`. With this criteria, the utility of running a program is the final utility in execution. This criteria can not be applied to all programs, since it requires that the utility remains constant after a certain point. If the program is non-terminating this may not happen.
- `look-ahead`. This criteria is parametrized by an integer. This integer denotes the depth of the decision tree that should be considered at each decision node. The most popular variant of this criteria is `look-ahead 1`, that corresponds to a greedy decision strategy.
- `discounted`. This criteria is parametrized by a discount factor γ. This factor is used to weight differently the increments in utility through time. In a decision node, an increment of utility that will occur t steps ahead will be weighted by γ^t.

A very simple (and useless) example of an USPL program can be found in figure 2. In this example, at each iteration of the cycle it is possible to either increment or decrement the variable `x`. Since the utility function is $-|x|$ and we have a greedy decision criteria, the variable will be decremented when its value is positive and incremented otherwise, therefore enhancing the convergence to the termination condition of the cycle.

3 The semantics of USPL

As expected, the semantics of USPL is also inspired by the semantics of SPL. A model for a SPL program is a *Fair Transition System* (FTS), whose major

```
local x:[-10..10]

P(0-abs(x), look-ahead 1) :: [
  while !(x=0) do
    [
      guard !(x=10) do x := (x+1)
    or
      guard !(x=-10) do x := (x-1)
    ]
]
```

Fig. 2. The running example.

difference to a traditional transition system is the concept of fairness, introduced in order to model correctly the concurrency.

Definition 1 *An* Utility-based Fair Transition System *(UFTS) is defined by a tuple* $\langle V, \Theta, \mathcal{T}, u, \rho \rangle$*, where:*

- V *is a finite set of system variables.* $\pi \in V$ *is a special control variable that will be used to store the control location of the program. Each variable has a type, and the states of the system are defined as type-consistent interpretations of this set. Let* Σ *denote the set of states.*
- $\Theta : \Sigma \to \mathbb{B}$ *is a predicate that determines the initial execution states.*
- $\mathcal{T}$ *is a finite set of transitions. Each transition* $\tau \in \mathcal{T}$ *is a function with type* $\Sigma \to \mathcal{P}(\Sigma)$*, that maps each state* $s \in \Sigma$ *into a (possibly empty) set of successor states* $\tau(s) \subseteq \Sigma$*. A transition* τ *is enabled at state* s *if* $\tau(s) \neq \emptyset$*. There is a special idle transition* $\tau_I \equiv \lambda s.\{s\}$ *that must always be present in* $\mathcal{T}$*.*
- $u : \Sigma \to \mathbb{R}$ *is an utility function, that defines the preferences of the process.*
- $\rho : (\Sigma \to \mathbb{R}) \to \mathcal{P}(\Sigma^\omega) \to \mathcal{P}(\Sigma^\omega)$ *is a decision criteria, that, given an utility function and a set of executions, determines which are the preferred ones. This function has the following restrictions:*
  - *It must only choose between the given executions:*

$$\forall u, \Omega \cdot \rho\ u\ \Omega \subseteq \Omega$$

  - *If possible, it should choose something:*

$$\forall u, \Omega \neq \emptyset \cdot \rho\ u\ \Omega \neq \emptyset$$

Aside the utility function and the decision criteria, the major difference between an UFTS and a FTS is the fact that all transitions, except the idle one, are just (or weakly fair), and so its not necessary to classify the transitions according to the different types of fairness. In a FTS, besides V, Θ and $\mathcal{T}$, we have the sets $\mathcal{J}$ and $\mathcal{C}$ that determine, respectively, the just and compassionate transitions of the program. The notion justice will be presented later, and since that, for the

chosen set of instructions, the compassion does not apply, this concept will not be explained.

Each transition $\tau \in \mathcal{T}$ can be represented by a first-order formula $\rho_\tau(V, V')$, called the transition relation of τ, that expresses the relation between a state s and its successors in $\tau(s)$. The transition relations may refer both primed and unprimed versions of the system variables, in order to distinguish the values in source and successor states. A state s' is a successor of s by τ if ρ_τ evaluates to true when one interprets each $x \in V$ as $s(x)$ and each $x' \in V'$ as $s'(x')$.

An infinite sequence of states $\sigma \equiv s_0, s_1, s_2, \ldots$ is a just execution of an UFTS if it satisfies the following requirements:

- Initiality: s_0 is an initial state.
- Consecution: for each $j = 0, 1, \ldots$ there exists a transition τ that occurs between s_j and s_{j+1}, that is, $s_{j+1} \in \tau(s_j)$.
- Justice: for each diligent transition $\tau \neq \tau_I$, it is not the case that τ is continuously enabled beyond some position j in σ without ever occurring.

Definition 2 *Let Ω be the set of all just executions of an UFTS. An execution $\sigma \in \Omega$ is a* computation *iff it belongs to the set $\rho\, u\, \Omega$.*

We will now present the method to calculate an UFTS from an USPL program. The set of variables V includes all variables declared in the program, plus a control variable π that is defined over an integer set with has as many values as different locations in the program. The initial condition Θ is defined as $(\pi = 0) \wedge \varphi$, where φ is the conjunction of all conditions that appear after the `where` clauses in the variable declarations.

In order to define the transition relations that characterize each USPL instruction, it is necessary to know the value of the control variable before and after its execution. Assuming that a program has labels before and after each instruction, we can define a function *pos*, that maps each label to its location, as the function that identifies at most the following labels:

- In a process *id* `(` *intexp* `,` *criteria* `)` `::` `[` l: *instruction* $\widehat{l}$: `]`

$$pos(l) = 0$$

- In an instruction l: `[` m: *instruction* $\widehat{m}$: `]` $\widehat{l}$:

$$pos(l) = pos(m) \wedge pos(\widehat{l}) = pos(\widehat{m})$$

- In a concatenation l: m: *instruction* $\widehat{m}$: `;` n: *instruction* $\widehat{n}$: $\widehat{l}$:

$$pos(l) = pos(m) \wedge pos(\widehat{m}) = pos(n) \wedge pos(\widehat{n}) = pos(\widehat{l})$$

- In a choice l: m: *instruction* $\widehat{m}$: `or` n: *instruction* $\widehat{n}$: $\widehat{l}$:

$$pos(l) = pos(m) = pos(n) \wedge pos(\widehat{l}) = pos(\widehat{m}) = pos(\widehat{n})$$

– In a loop l: `while` *expbool* `do` m: *instruction* $\widehat{m}$: $\widehat{l}$:

$$pos(l) = pos(\widehat{m})$$

Usually, in a transition relation only a few variables of the set V are changed. In order to simplify the definition of the transition relations, for a set $U \subseteq V$ we will define

$$pres(U) \equiv \bigwedge_{u \in U} (u' = u)$$

We then define the transition relations that correspond to each USPL instructions as follows:

– *Guard*. To an instruction of type $\boxed{l\text{: } \texttt{guard } c \texttt{ do } u \texttt{ := } e\ \widehat{l}\text{:}}$ corresponds the following transition relation:

$$c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge u' = e \wedge pres(V - \{\pi, u\})$$

This transition may only occur when c is true in the source state. The only variables that change its value are π, that will contain the new location, and u, the variable to which e is attributed.

– *Await*. To an instruction of type $\boxed{l\text{: } \texttt{await } c\ \widehat{l}\text{:}}$ corresponds the following transition relation:

$$c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge pres(V - \{\pi\})$$

This case is almost identical to the previous. However, only π changes its value.

– *While*. To an instruction of type $\boxed{l\text{: } \texttt{while } c \texttt{ do } m\text{: } \textit{instruction}\ \widehat{m}\text{: } \widehat{l}\text{:}}$ corresponds the following transition relations:

$$c \wedge \pi = pos(l) \wedge \pi' = pos(m) \wedge pres(V - \{\pi\})$$
$$\neg c \wedge \pi = pos(l) \wedge \pi' = pos(\widehat{l}) \wedge pres(V - \{\pi\})$$

In this case we have two transition relations. The first one models the transitions that originate a new iteration of the loop, when c is verified in the source state. The second one occurs when the termination condition $\neg c$ is true. Once again, only π changes its value.

– The choice and concatenation instructions do not originate any transition relations.

The utility function is defined by evaluating the expression presented after the process name. In order to formally specify the semantics of a decision criteria, usually we first define a preference relation on just executions using the utility function, and afterwards we choose the computations as the subset of most-preferred executions. Given a preference relation $\succ$, this set will be denoted by $\uparrow_{\succ} \Omega$ and is formally defined as:

$$\sigma \in \uparrow_{\succ} \Omega \qquad \text{iff} \qquad \sigma \in \Omega \wedge \forall\, \sigma' \in \Omega \cdot \neg(\sigma' \succ \sigma)$$

In the next section we will show that the idle transition should be ignored when other transitions are available. As such, in the definition of the preference relations those transitions are not considered, that is, the only state that can be repeated indefinitely in an execution is the last one[1]. The preference relation should be closed by idle transitions, that is, if $\ldots, s, \ldots \succ \ldots, t, \ldots$ then $\ldots, s, \ldots \succ \ldots, t, t, \ldots$ and $\ldots, s, s, \ldots \succ \ldots, t, \ldots$. For example, for the decision criteria `look-ahead` n and utility function u, we define the preference relation $\curvearrowright_u^n$ as:

$$s_0, s_1, \ldots \curvearrowright_u^n t_0, t_1, \ldots \qquad \text{iff} \qquad \forall j, k \in \mathbb{N}_0 \cdot s_j = t_k \supset u(s_{j+n}) > u(t_{k+n})$$

In the figure 3 we present the semantic model of the example of figure 2, that was determined according to the method described above.

$$\begin{aligned}
V &= \{x, \pi\} \\
\Theta &= (\pi = 0) \\
\mathcal{T} &= \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_I\} \\
\rho_{\tau_1} &= (\pi = 0) \wedge (x \neq 0) \wedge (\pi' = 1) \\
\rho_{\tau_2} &= (\pi = 0) \wedge (x = 0) \wedge (\pi' = 2) \\
\rho_{\tau_3} &= (\pi = 1) \wedge (x \neq 10) \wedge (\pi' = 0) \wedge (x' = x + 1) \\
\rho_{\tau_4} &= (\pi = 1) \wedge (x \neq -10) \wedge (\pi' = 0) \wedge (x' = x - 1) \\
u &= \lambda s. - |s(x)| \\
\rho &= \lambda\ u\ \Omega\ .\ \uparrow_{\curvearrowright_u^1}\ \Omega
\end{aligned}$$

Fig. 3. Semantic model of the running example.

4 Translation of USPL programs into SPL

For USPL to be an useful language, it is desirable that it could be used as a programming language for utility-based agents, or at least, as a specification language, where one could prove properties about this kind of entities. In order to address both objectives, we developed a mechanism to transform USPL programs into SPL. The goal is to obtain a SPL program whose valid computations are exactly the same as the valid computations of the original program. This transformation allows us to prove properties about utility-based agents, using the techniques developed for the SPL by Manna and Pnueli. Since SPL is only a specification language it does not have a compiler that allows one to execute its programs. This is a drawback if we intend to use USPL as a programming

[1] In fact, this happens because, given the chosen subset of instructions, the value of the control variable before and after the execution of an instruction is always different.

language. However, considering only the subset of SPL instructions used in the transformation process, it would be rather straightforward to develop a compiler to this language.

The most simple way to achieve this transformation relies on viewing an USPL program as a specification of a *Markov Decision Problem* (MDP)[2]. A MDP consists of a Markov decision process, that is, a model of a decision agent that interacts synchronously with its environment, plus an optimization criteria, which is used in a similar way to the decision criteria in the USPL. A Markov decision process is characterized by having the so called Markov property - all transitions and rewards depend only on the current state. To solve a MDP consists of finding an optimal decision strategy, that is, a function that tell us what action to choose at each decision node.

Definition 3 *A* Markov decision process *consists of a tuple* $\langle S, A, T, R\rangle$ *where:*

- S *is a finite set of possible states of the world.*
- A *is a finite set of possible actions that a decision agent can execute.*
- $T : S \times A \times S \to [0, 1]$ *is a state transition function, that maps each state and action to a probability distribution on successor states.*
- $R : S \times A \to \mathbb{R}$ *is a reward function, that captures the increase or decrease in utility an agent may get by executing each action at a given state.*

In order to transform an USPL program into a MDP, the non-deterministic choices must be considered as decision nodes, and the transitions enabled at those nodes as the actions from which the decision maker must choose the preferred ones. Considering $S \equiv \Sigma$ and $A \equiv \mathcal{T}$, the main differences between a MDP and an UFTS are:

- In a MDP the non-determinism is quantified by a probability distribution. Given the equivalences above, all the transitions of an UFTS can be stored in a function of type $S \times A \to \mathcal{P}(S)$. Without further restrictions, it would be necessary an external mechanism to quantify the non-determinism in order to convert an UFTS into a MDP. However, looking at the semantics of the instructions included in the USPL, we can see that they are all deterministic, that is, when a transition occurs at a state there is only one possible successor. Given this fact, T can be determined as follows[3]:

$$T(s, \tau, s') = \begin{cases} 1 \text{ if } s' \in \tau(s) \\ 0 \text{ if } s' \notin \tau(s) \end{cases}$$

- The preferences are represented by rewards instead of an utility function. The transformation between the two models is very simple:

$$R(s, \tau) = \begin{cases} u(T(s, \tau)) - u(s) & \text{if } T(s, \tau) \neq \bot \\ -\infty & \text{if } T(s, \tau) = \bot \end{cases}$$

[2] These problems are mentioned in many books from the operational research area. Our presentation of MDP concepts is based on [12]

[3] For convenience, and since the transition system is deterministic, we will some times use an isomorphic definition for T with type $S \times A \hookrightarrow S$.

Later we will see that the infinite negative rewards do not originate any kind of problem. This subtlety is used because the reward function is total, but we have to guarantee that the transitions that are not enabled in a state can not be chosen at that state. Due to the existence of the idle transition there is always at least one transition with finite reward:

$$\forall s \in S, \exists \tau \in A \cdot -\infty < R(s, \tau) < +\infty$$

– There are no fairness conditions in a MDP. This is the major problem in the transformation, because one must guarantee that the process of choosing the best transitions does not lead to executions that are not computations of the original program. However, this problem is simplified because all diligent transitions are just, and lead to a successor state that is necessarily different from its predecessor. Given these conditions, it suffices to exclude from MDP the idle transition in the states where there exists at least a diligent one that is enabled. If the idle transition was the only one chosen at one of those states, it would exist an execution that beyond some point would remain indefinitely at that state. However, due to the fairness conditions, that execution is not a computation of the original UFTS because a diligent just transition must occur at some point. Thereafter, the following condition must be imposed on T:

$$T(s, \tau_I) = \perp \text{ iff } \exists \tau \neq \tau_I \cdot \tau(s) \neq \emptyset$$

As seen above, to solve a MDP consists of finding an optimal decision strategy, that is, a function that tells the decision maker which are the best actions to execute at each decision node. In this case, the obtained strategies should be non-deterministic and memoryless. A non-deterministic strategy defines for each state a set of actions whose execution is indifferent to the decision agent. Strategies can also be pure, when only an action is selected, or probabilistic when the non-determinism is quantified by a probability distribution. However, if we choose to synthesize pure strategies the resulting program would only have one possible computation (and its variants induced by the idle transitions), and it would not be possible to have all computations of the original USPL program. Obviously, we also can not synthesize probabilistic strategies because our model does not allows us to quantify the non-determinism. The choice of memoryless strategies is less obvious. In order to implement the memory it would be necessary to increase the set of variables of the resulting program, which would lead to computations that would differ from the original ones. However, with memoryless strategies it may not be possible to synthesize all the computations generated by any decision criteria. That would be a major drawback, but for the criteria allowed for the moment, well known results from the MDP theory seem to indicate that memoryless strategies suffice to generate all the possible computations. In the future, if we decide to implement more decision criteria, probably we will have to relax the notion of equivalence between programs written in USPL and in SPL, in order to allow the inclusion of strategies with memory.

In order to calculate the optimal strategy π^* (a strategy that generates all possible computations of the original program), we use classical techniques de-

veloped to solve MDPs with minor modifications. One of the most usual is the *value iteration* algorithm, that was introduced by Bellman in [3], and that is used when the decision criteria is `discounted`. Given an UFTS $\langle V, \Theta, \mathcal{T}, u, \rho \rangle$ and an optimal strategy π^*, the equivalent FTS obtained after restricting the original transition system using that strategy is defined by the tuple $\langle V', \Theta', \mathcal{T}', \mathcal{J}', \mathcal{C}' \rangle$ where[4]:

$$
\begin{aligned}
V' &\equiv V \\
\Theta' &\equiv \Theta \\
\mathcal{T}' &\equiv \left\{ \rho_\tau \wedge \bigvee_{s \in \Sigma, \tau \in \pi^*(s)} val(s) \,|\, \tau \in \mathcal{T} - \{\tau_I\} \right\} \cup \{\tau_I\} \\
\mathcal{J}' &\equiv \mathcal{T}' - \{\tau_I\} \\
\mathcal{C}' &\equiv \emptyset
\end{aligned}
$$

After determining an FTS equivalent to the UFTS that models the original program, we need to obtain an SPL program whose semantic model is precisely that FTS. This process is done through a graph rewriting algorithm, and for the selected subset of instructions it can be proved that this reconstruction is always possible. In fact, the main reason to choose the syntax of figure 1 was precisely the guarantee that one could obtain valid SPL programs after determining the equivalent FTSs. The SPL program, obtained according to the method presented in this section, that is equivalent to the program of figure 2 is presented in figure 4.

```
local x:[-10..10]

P :: [
  while !(x=0) do
    [
      guard (x<=0) do x := (x+1)
    or
      guard (x>=0) do x := (x-1)
    ]
]
```

Fig. 4. SPL program equivalent to the running example.

[4] The function *val* maps each state to the boolean expression that represents the interpretation of the variables in that state, and is defined as:

$$val = \lambda s. \bigwedge_{x \in V} x = s[x]$$

5 Refinement

The transformation of USPL programs into SPL using MDP resolution techniques is rather inefficient. The number of arithmetic operations needed in order to determine the optimal strategy is polynomial in the number of states and actions [10]. However, the number of states grows exponentially in the number of variables, and in practice it is almost impossible to use these techniques to compile reasonable programs. In order to address this problem we are investigating two different approaches. In the first one we will continue to use MDPs, but we will try to apply abstraction techniques with the objective of reducing the number of states of the problem, such as the ones presented in [7] or [9]. The second approach, that will be described briefly in this section, does not imply the transformation of an USPL program into a MDP and is based on a particular notion of refinement.

The objective of the refinement process is to transform gradually an USPL program into the equivalent SPL one, using syntactical transformation rules that preserve the set of valid computations of the original program. There are two types of rules, namely, rules that allow the transformation of USPL programs directly into SPL ones, and rules that allow the transformation of USPL programs into more simple USPL ones, typically into programs where decision nodes have fewer branches. Usually the refinement rules can not be applied under all circumstances, and we need to impose conditions that must be verified in order to use them. Some of this conditions concern the decision criteria used in the program, but others restrict the behavior of the utility function through the executions. The later conditions will be specified using linear temporal logic, and (as was already mentioned in section 2) these can be verified using proving techniques developed by Manna and Pnueli for the SPL language. The study of this refinement process is still very preliminary, and for the moment there are still very few rules, some of which are presented below.

In the following rules, $\mathcal{P}'$ denotes the SPL program that one obtains from an USPL program $\mathcal{P}$ by simply removing the utility function and the decision criteria that parametrize the process. The set of computations of $\mathcal{P}'$ is composed of all the executions of $\mathcal{P}$ that verify the fairness conditions. U denotes the integer expression that defines the utility function. The first two rules are examples of rules that allow one to transform an USPL program into an equivalent SPL program. The following two rules fall on the second class mentioned above, and allow the transformation of USPL programs into equivalent, but more deterministic, USPL programs.

Rule 1 *If the utility function is constant, the resulting SPL program is identical to the original one. This rule can be applied if the following condition is verified:*

$$\mathcal{P}' \models \exists k \cdot \Box(U = k)$$

Rule 2 *In an USPL program with decision criteria* `look-at-end`*, if all executions end with the same utility, then the resulting SPL program is identical to*

the original one. This rule can be applied if the following condition is verified:

$$\exists k \cdot \mathcal{P}' \models \Diamond\Box(U = k)$$

Rule 3 *In an USPL program with decision criteria* `look-at-end` *or* `look-ahead`*, any constant part of the utility function expression can be eliminated. A program* $\mathcal{P}$ *with an utility function*

$$U = \Omega + \Delta$$

can be transformed into an equivalent USPL program with utility function

$$U = \Omega$$

if the following condition is verified:

$$\mathcal{P}' \models \exists k \cdot \Box(\Delta = k)$$

Rule 4 *If an USPL program with decision criteria* `look-ahead 1` *has a choice between guarded assignments (where* I_1 *and* I_2 *are arbitrary instruction blocks)*

l: `[guard` c_1 `do` x_1 `:=` e_1`;` I_1`] or [guard` c_2 `do` x_2 `:=` e_2`;` I_2`]`

and verifies the condition

$$\mathcal{P}' \models \Box\,(at\_l \wedge c_1 \wedge c_2 \supset U[x_1 := e_1] > U[x_2 := e_2])$$

then it is possible to replace it by the following instruction:

l: `[guard` c_1 `do` x_1 `:=` e_1`;` I_1`] or [guard` $c_2 \wedge \neg c_1$ `do` x_2 `:=` e_2`;` I_2`]`

6 Conclusions and Future Work

Even at this early stage, USPL shows that it is possible to develop an imperative programming language for utility-based agents. For the moment, the major difficulties in developing this language where derived from subtleties of the semantic model, such as the fairness conditions or the need of idle transitions. The need to compare the utility of infinite executions was avoided due to the inclusion of the decision criteria and the use of MDP resolution techniques. However, it seems that the need to specify a decision criteria makes programming with USPL a little more difficult. Its rather easy to translate the objectives into an utility function, but when the program can have infinite executions it is not very easy to decide which decision criteria to use, or to decide how to parametrize it.

USPL can also be seen as a language for specifying a particular class of MDPs, combining in an intuitive programming paradigm a decision-theoretic and agent programming flavor. We have little knowledge of similar work. The only language known to us that is being developed with the same objective is DTGolog [5], which is based on the situation calculus and on the logic programming paradigm. This language resembles more precisely a MDP structure, allowing probabilistic

transitions and specifying the objectives by rewards instead of an utility function. Although DTGolog is more powerful that USPL, we believe that programming is easier with the later, because of its imperative style and qualitative decision model. It is known that it is often very difficult or even impossible to quantify the probabilities of the transitions [6]. We also believe that specifying the objectives with an utility function is easier than through rewards.

All the techniques presented in this paper are being included in a prototype system, developed in the functional programming language Haskell [1]. So far, it allows one to translate sequential USPL programs to SPL, to apply interactively the refinement rules and generate the specifications to be proved on STeP.

There is still a lot to be done in order to make USPL a fully usable and well founded language. Concerning the sequential version presented in this paper, it is necessary to verify formally that the strategies determined by the process presented in section 4 are indeed optimal, in the sense that they generate all the valid computations according to definition 2. The same applies to the refinement rules. It is necessary to prove that the transformations preserve the set of valid computations. Obviously, it is also necessary to find more refinement rules, so that the refinement can be used only by itself. As mentioned in the section 5, we also intend to apply abstraction techniques in order to reduce the set of states of an MDP, so that we can apply the developed compilation method to real problems. Particularly, we believe that the factoring algorithm presented in [7] is specially suited to our framework. This algorithm tries to find the coarsest homogeneous refinement of an initial partition (where states are aggregated according to equal rewards) of the state space of an MDP. An homogeneous partition is one in which for each action, states in the same set have the same probability of being carried to each other set. We already have some preliminary results that show that this abstraction technique induces a huge reduction in the state space of most USPL programs.

Although there is still some work to be done in the sequential version of USPL, the major innovation will be the introduction of concurrency primitives in the language. As stated in section 1, the existence of more than one process will induce uncertainty in the decision model of each one of them. This fact will turn obsolete the compilation technique presented in this paper, since it was not conceived to deal with uncertainty. This evolution will be done in two stages. First it will only be allowed to model a single utility-based agent, that is, only one process will be parametrized by the utility function and the decision criteria. This means that the behavior of all other entities is perfectly know *a priori*. In this stage we believe that the optimization can be done using the standard extension of MDPs that deals with uncertainty, the *Partially Observable Markov Decision Problems*. In the last stage we will study the case where more than one process models an utility-based agent. Since that the decision process will have to deal not only with uncertainty, but also with the rationality of other entities, in order to correctly predict their behavior, we will surely need to use techniques from game theory, namely, from the theory of *Repeated Games with Incomplete Information* [2].

References

1. L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, S. Jones (editor), and J. Hughes (editor). Haskell 98: A non-strict, purely functional language. Technical report, 1998.
2. Robert Aumann and Michael Maschler. *Repeated Games With Incomplete Information*. The MIT Press, 1995.
3. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
4. Nokolaj Bjorner, Anca Browne, Eddie Chang, Michael Colón, Bernd Finkbeiner, Arjun Kapur, Zohar Manna, Henny Sipma, and Tomás Uribe. *STeP: The Stanford Temporal Prover Educational Release (User's Manual)*. Computer Science Department, Stanford University, October 1997.
5. Craig Boutilier, Ray Reiter Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 355–362, 2000.
6. Ronen Brafman and Moshe Tennenholtz. Modeling agents as qualitative decision makers. *Artificial Intelligence*, 94(1–2):217–268, 1997.
7. Thomas Dean and Robert Givan. Model minimization in markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 106–111, 1997.
8. Michael Fisher. A survey of concurrent metatem - the language and its applications. In D. Gabbay and H. Ohlbach, editors, *Temporal Logics - Proceedings of First International Conference*, volume 827 of *LNAI*, pages 480–505. Springer-Verlag, 1994.
9. Milos Hauskrecht, Nicolas Meuleau, Leslie Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of markov decision processes using macro-actions. In *Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence*, 1998.
10. Michael Littman, Thomas Dean, and Leslie Kaelbling. On the complexity of solving markov decision processes. In *Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence*, 1995.
11. Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.
12. Martin Puterman. *Stochastic Models*, volume 2 of *Handbooks in Operations Research and Management Science*, chapter 8, pages 331–434. North-Holland, 1990.
13. Anand Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Agents Breaking Away (Proceedings of MAAMAW'96)*, volume 1038 of *LNCS*, pages 42–55. Springer-Verlag, 1996.
14. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
15. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
16. S. R. Thomas. *PLACA, an Agent Oriented Programming Language*. PhD thesis, Computer Science Department, Stanford University, 1993.