Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 159 (2019) 202–211

Procedia Computer Science

www.elsevier.com/locate/procedia

23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

AIM: Designing a language for AI models

Ionuţ Cristian Pistol[a,*], Andrei Arusoaie[a,*]

[a]*Alexandru Ioan Cuza, University of Iaşi, General Berthelot 16, Iaşi, 700483, Romania*

Abstract

Describing an unambiguous model for an Artificial Intelligence (AI) problem has been a significant topic in AI for almost 70 years, with the main goal of formalizing a natural language description to allow the computer to solve it. Nowadays, an AI problem is usually modelled as a transitional system, by following four steps: identify a representation for a problem state, describe the initial and final states in that representation, describe valid transitions together with a search strategy that looks for a path between an initial and a final state using the available transitions. This paper proposes a new language for describing AI models with the goal to generate executable code. The proposed language is capable of representing all common types of problems, and models implemented can be adapted easily to any search strategy with specific requirements (such as score functions). The language is fully described in this paper together with several non-trivial examples, while the code generation feature is work-in-progress.

Keywords: artificial intelligence models; model specification; programming languages; code generation;

1. Introduction

An Artificial Intelligence (AI) problem ranges from (apparently) simple toy problems, such as the Hanoi towers or Queens problem, to protein sequencing and complex game AIs. Whatever the complexity of the problem, the main difficulty in approaching one is describing the data and the requirements is such a way as to allow the computer to solve it. Formalizing an AI problem (also referred as modelling an AI problem) has been done in various ways over the years, with proposed approaches such as the original neural networks [11], going through the first actual problem-solving computer model [15] and up to modern approaches, which not only allow a problem to be described and solved but also allow automated validation of the model, approaches well summarized in [17]. Nowadays, the most common approach to an AI problem is one established in its original form by [1] and [3]. Similar approach is described in recent works such as [17] and [6]. First, we identify a representation for a problem state which has to be simple enough so that the computer can analyze, modify and store it easily, but expressive enough so that all

* Corresponding author
E-mail address: ipistol@info.uaic.ro

1877-0509
10.1016/j.procs.2019.09.175

required data is included. Then, using the chosen representation, we describe the initial and final (goal) states, as well as the valid transitions within the problem space (which includes all valid states). A search strategy is then described, looking for a path between an initial and a final state using the available transitions.

Formal languages for describing such models have existed since the original STRIPS language (proposed almost 50 years ago) [3] to modern action query languages such as PDDL 3 [12] and Action Language C+ [5]. Although current languages allow the description of basically all types of AI problems, some even implementing search strategies to find a solution and model checkers to validate models, they have the disadvantages of being too complicated for certain types of problems and of being removed from programming languages. This paper proposes a new language, called **AIM** (Artificial Intelligence Models) with the goal of being well formalized and generic enough to allow easy description of any model, code generation to common programming languages and checking a model using modern SMT (Satisfiability Modulo Theories) solvers [2]. In this paper we describe **AIM** using the traditional Extended Backus-Naur Form (shorthanded as EBNF) [14] which is typically used for describing programming language syntax (e.g., C-Standard [4], The Java Language Specification [7]).

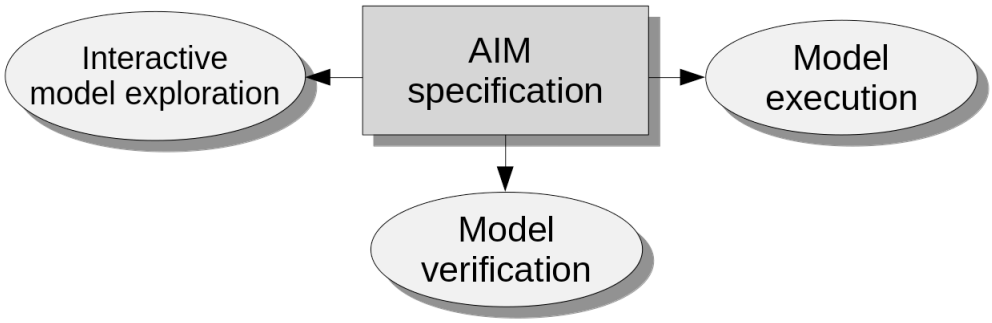

Fig. 1. Uses of AIM specifications

The motivation behind having **AIM** specifications is not only to generate executable code. As shown in Figure 1, **AIM** specifications are meant to serve as a basis for other tools which can provide interactive step-by-step model exploration, model execution tools, methodologies and tools for checking formal properties of the models. The main design principle of **AIM** is to write a *single* formal specification and everything else is generated or based on that specification. Therefore, the results obtained by various tools or methodologies are comparable and consistent due to the fact that there is only one specification. For example, if one uses model verification to prove that the current specification does not allow any solutions, then the model execution will not find any solution for *precisely* the same specification.

AIM usage could be further extended by providing automated conversion to and from other transitional systems description languages, such as STRIPS or PDDL, which are already familiar to many potential users and already include many implemented models.

1.1. Paper structure

Section 2 of this paper gives a couple of examples of classic AI problems and some models of representation for them. Section 3 introduces the AIM language, and section 4 exemplifies AIM on the previously described models. Section 5 concludes the paper and gives some ideas about future developments of AIM.

1.2. Contributions

We propose a new language to describe AI models. We introduce two classic AI problems (*nQueens* and *missionaries and cannibals* with two and respectively one model. We than describe the **AIM** grammar, defining a language able to represent virtually all commonly used types of AI models, including the three previous examples. For them, we provide the **AIM** implementations and we show how search strategies can be specified in **AIM**, specifying particular search functions or required parameters.

2. Artificial Intelligence problems

In order to exemplify AIM we introduce some models for a couple of classic AI toy problems: n-Queens problem and Missionaries and Cannibals. For each problem we identify the following items:

· **Instance** : the parameters required to build the initial state, and the variables within the problem statement;
· **State** : all the data required to describe the initial configuration and the problem goal, as well as to allow the search strategy to find a solution;
· **Initial state** : the problem state from which the search strategy starts looking for a solution, built from the instance data;
· **Final state** : a state which, when reached, indicates that a solution has been found;
· **Transitions** : the way in which the current state can be changed by the search strategy;
· **Valid transitions** : determines if a transition is valid considering the current state and the rules of the problem.

2.1. n-Queens

The n-Queens problem (*nQ*) can be defined as: Having an `nxn` table, place `n` queen chess pieces on it without having any queen attack another (as per the rules of chess).

Many proposed models for this problem exist, we have selected the couple below as examples.

2.1.1. nQ first model

We describe the *nQ first model*. A state is a list of Queens placements on columns, each list position corresponds to a table row. Since no two queens can be placed on the same row and the number of queens to be placed is equal to the number of rows, there is no additional need to identify the row on which a queen is placed. Final state is reached if all values are not zero (so a queen is placed on every row). A transition places a queen (currently not placed) on an empty row (value of `0` in the state) by changing `0` to a column in the table (between `1` and `n`). This transition is valid if the placed queen does not attack any other placed queen.

Instance	:	n - number of queens, size of $n \times n$ table
State	:	$(S_1, S_2, \ldots, S_n), 0 \leq S_i \leq n, 1 \leq i \leq n$
Initial state	:	a list $(0, 0, \ldots, 0)$ of size n
Final state	:	$(S_1, S_2, \ldots, S_n), 1 \leq S_i \leq n, 1 \leq i \leq n$
		$S_i \neq S_j$, for all $1 \leq i, j \leq n$;
Transitions	:	$(S_{11}, S_{12}, \ldots, S_{1n}) - > (S_{21}, S_{22}, \ldots, S_{2n})$,
Valid transitions	:	$S_{1i} = S_{2i}$ for all $1 \leq i \leq n$, except exactly one i for which $S_{1i} = 0$ and $S_{2i} \in [1..n]$
		$S_i \neq S_j, 1 \leq i, j \leq n, i \neq j$
		$\mid S_i - S_j \mid \neq \mid i - j \mid, 1 \leq i, j \leq n, i \neq j$

Fig. 2. nQ first model

2.1.2. nQ second model

In the second model of *nQ* we use the same representation of a state but the initial state is a list of randomly placed queens. A transition means changing the column on which a queen is placed (the row - position in the list - remain unchanged), and is valid if the the queen in the new placement doesn't attack another one.

2.2. Missionaries and cannibals

The missionaries and cannibals (MC) is another classic AI toy problem, one for which the building of an AI model was originally exemplified in [1]. A generalization of the problem statement can be: On the shore of a river there are `n` missionaries and `m` cannibals. There is a boat with `b` capacity on the same shore. Find, if it exists, a way to move all

Instance	:	n - number of queens, size of $n \times n$ table;
State	:	$(S_1, S_2, \ldots, S_n), 0 \leq S_i \leq n, 1 \leq i \leq n$;
Initial state	:	a list $(S_1, S_2, \ldots, S_n)$ of size n, where $S_i = random(1, 8)$, forall i where $1 \leq i \leq n$
Final state	:	$(S_1, S_2, \ldots, S_n), 1 \leq S_i \leq n, 1 \leq i \leq n$;
		$S_i \neq S_j$, for all $1 \leq i, j \leq n$;
Transitions	:	$(S_{11}, S_{12}, \ldots, S_{1n}) \rightarrow (S_{21}, S_{22}, \ldots, S_{2n})$,
Valid transitions	:	$S_{1i} = S_{2i}$ for all $1 \leq i \leq n$, except exactly one i;
		$S_i \neq S_j, 1 \leq i, j \leq n, i \neq j$;
		$\mid S_i - S_j \mid \neq \mid i - j \mid, 1 \leq i, j \leq n, i \neq j$

Fig. 3. nQ second model

people on the initial shore to the other shore, using the boat. Consider that the boat moves only with 1 to b people in it, and on neither shore there can be more cannibals than missionaries, if there is at least one missionary there.

2.2.1. MC model

In this paragraph we describe a possible model for the *missionaries and cannibals* problem. Here, a state is a list with the capacity of the boat, number of missionaries and cannibals (n_1 and m_1) on the left (initial) shore, placement of the boat (left shore - 1, right shore - 2), number of missionaries and cannibals on the right shore (n_2 and m_2). The initial state has all people and the boat on the left shore, while the final state has all people and the boat on the right shore. The only transition moves a number of people (equal or smaller than the boat capacity b, but greater than zero) to the opposite shore the boat is currently on. The transition is valid if the number of missionaries on either shore, if greater than zero, is larger than the number of cannibals on that shore.

Instance	:	n - number of cannibals, m - number of missionaries, b - capacity of the boat;
State	:	$(b, n_1, m_1, bp, n_2, m_2), b \in [1, 2], n_1 + n_2 = n, m_1 + m_2 = m$;
Initial state	:	$(b, n, m, 1, 0, 0)$;
Final state	:	$(b, 0, 0, 2, n, m)$;
Transitions	:	$(b, n_1, m_1, bp, n_2, m_2) \rightarrow (b, n_1 + v_1, n_2 + v_2, 3 - bp, n_2 - v_1, m_2 - v_2)$;
Valid transitions	:	$n_1, m_1, n_2, m_2 \geq 0$;
		If $m_1 > 0$ then $m_1 \geq n_1$;
		If $m_2 > 0$, then $m_2 \geq n_2$;
		If $v_1 + v_2 > 0$, then $bp = 1$, else $bp = 2$;
		$\mid v_1 + v_2 \mid \leq b$.

Fig. 4. MC model

The models described in this section serve as examples, and are designed to cover varied aspects frequently occurring in AI models: lists of varied and fixed length, validations (belonging, comparisons, conditional validations), random values and mathematical expressions. The models make no effort to facilitate particular informed strategies; they are however considered in the AIM translation in Section 4.

3. The AIM grammar

In this section we present the **AIM** grammar using the traditional EBNF style [14]. The grammar is shown in Figure 5. The language, hereafter denoted as **AIM**, accepted by this grammar is a declarative one and it is intended to be expressive enough to describe AI models. In addition, the language is designed to be extensible with user defined basic functions. An **AIM** specification consists in a list of items that describe an AI model:

- an `Instance` which is a list of input parameters declarations;
- the conditions to be satisfied by `ValidStates`;
- the `InitialState`;
- the conditions to be satisfied by the `FinalState`;
- the `Transition` function;
- the `Strategy` to be used.

To describe an instance of an AI problem, we use a list of declarations of the input parameters of the problem. For example, `#instance: m:int, n:int ; m = 10; n = 5;` specifies that the input parameters are `m` and `n`, and their values are 10 and 5, respectively.

The initial state is specified using the `#initial-state` keyword. The symbol ';' separates a list of declarations from a piece of code. The declarations are local variables visible in the code. The code is meant to initialize a state and return it. For instance, the following piece of code builds a list of size `n` where each element is 0, and then it returns it:

```
#initial-state: inits:list<int> ;
                foreach i <- [0,n]:
                     inits[i] = 0 ;
                end
                return inits;
```

It is worth noting that we always expect the value returned at `#initial-state` is always a state. Otherwise, the semantics of this language construct is undefined.

The `#valid-state` keyword introduces a function which validates a state. For instance, `#valid-state: s:list<int> ; size(s) == n` specifies that a state `s` is valid if its length is always `n`. Note that the list of declarations before the ';' symbol is in fact a list of input parameters for the validity function. Thus, one can specify more parameters if needed. The expression after the ';' symbol is always a boolean expression. Boolean expressions are described by the `Bool` non-terminal, and they include: usual boolean operations (negation, conjunction, disjunction, implication), comparisons over arithmetic expressions, and a special construct `"forall" RangeIter ":" Bool` needed to express universally quantified properties. Note that `Bool` captures the main first-order logical constructs, and thus, we are able to express complex logical formulas.

The code that we are allowed to write is fully specified by the `Code` non-terminal. `Code` defines a simple imperative language with assignments, branching statements, loops and return constructs. It allows one to use boolean expressions and arithmetic expressions: integers, identifiers (for variables), arithmetic operations (addition, subtraction, multiplication, division, and modulo), access to the i^{th} element of a list, and basic (user-defined) builtin functions. These set of builtin functions (i.e., `BasicFunctions`) is intentionally left unspecified because it is extensible, and may contain utility functions like `random`, `min`, `max`, `size`, etc. The only restriction we have over them is the return type which has to be a boolean or an integer.

Final states are specified using `#final-state`. The syntax resembles the syntax of `#valid-state`: a list of declarations, followed by a boolean condition. For example, the following piece of code specifies that a state `s` is final if it is a list with non-zero elements:

```
#final-state: s:list<int> ;
              forall i <- [0, n-1]: s[i] != 0
```

Transitions are specified using two keywords: `#transition` and `#valid-transition`. The former is meant to specify how the new state is created, while the latter is used to specify the condition that needs to be satisfied for a valid transition. The code below shows an example of a transition function: a transition from `s` to `s'` means to set value `v` on position `i` in `s`; the transition is valid only if `v` is greater than the previous element in `s`.

```
#transition: s:list<int>, s':list<int>, i:int, v:int;
             s' = s; s'[i] = v; return s';
#valid-transition: s:list<int>, i:int, v:int ;
                   0 < i && i < size(s) && s[i-1] >= s[i]
```

```
BasicFunctions   ::=  <builtin_functions>
Id               ::=  <id_regex>
Int              ::=  <integers>
Type             ::=  "int"
                   |  list "<" Type ">"
Decl             ::=  Id ":" Type
Decls            ::=  Decl
                   |  Decl "," Decls
Exp              ::=  Int | Id
                   |  Exp "[" Exp "]"
                   |  BasicFunctions
                   >  Exp "*" Exp [left]
                   |  Exp "/" Exp [left]
                   |  Exp "%" Exp [left]
                   >  Exp "+" Exp [left]
                   |  Exp "-" Exp [left]
Bool             ::=  BasicFunctions
                   |  Exp "==" Exp
                   |  Exp "!=" Exp
                   |  Exp "<" Exp
                   |  Exp ">" Exp
                   |  Exp "<=" Exp
                   |  Exp ">=" Exp
                   |  "forall" RangeIter ":" Bool
                   |  Bool "&&" Bool
                   |  Bool "||" Bool
                   |  Bool "->" Bool
                   |  "!" Bool
Code             ::=  Exp "=" Exp ";"
                   |  "if" Exp "then" Exp "else" Exp "endif"
                   |  "while" Bool "do" Code "end"
                   |  "foreach" RangeIter ":" Code "end"
                   |  "return" Exp ";"
RangeIter        ::=  Id "<-" Range
Range            ::=  "[" Exp ".." Exp "]"
Strategy         ::=  "random"
                   |  "hill-climbing"
Instance         ::=  "#instance:" Decls ";" Code
ValidState       ::=  "#valid-state:" Decl ";" Bool
InitialState     ::=  "#initial-state:" Decls ";" Code
FinalState       ::=  "#final-state:" Decls ";" Bool
Transition       ::=  "#transition-function:" Decls ";" Code
ValidTransition  ::=  "#valid-transition:" Decls ";" Bool
Strategy         ::=  "#strategy:" StrategyName
                   |  "#strategy:" StrategyName ";" Decls ";" Code
Specification    ::=  Instance State InitialState FinalState
                      Transition ValidTransition Strategy
```

Fig. 5. The AIM Grammar

The last element to be provided in an **AIM** specification is the strategy to be used. Here, we provide a list of common strategies, which can be extended: `random`, `BFS`, `uniform cost`, `DFS`, `backtracking`, `iterative`

`deepening`, `best-first`, `hill-climbing`, `A*`. The strategies are introduced by the `#strategy` keyword followed by the strategy name, and optionally by some declarations and code. These optional components are needed by:

- Uninformed strategies, such as `iterative deepening`, if the user wants to limit the maximum depth of exploration of the problem space.
- All informed strategies, such as `best-first`, `hill-climbing` or `A*`, which require a score function to evaluate states.

In Section 4 we show an example where `#strategy` is used to customize a hill climbing strategy.

4. Models in AIM

We describe the models in Section 2 in **AIM**. For each model we have a corresponding specification. For each model, a search strategy is described in **AIM** although no strategy was indicated in the model descriptions. This is done to show how any strategy can be used in an **AIM** specification, even informed strategies requiring the description of additional scoring functions.

4.1. nQ first model

For the first model of the 4-queens problem the input parameter is n, the number of queens to be placed on the board. The initial state is a list of zeros of size n and has type `list<int>`.

One important feature of **AIM** is that the language is expressive enough to capture logical conditions. For instance, the conditions that establish the validity of a state (i.e., $(S_1, S_2, \ldots, S_n), 0 \leq S_i \leq n, 1 \leq i \leq n$) are easily expressible in **AIM** using the `forall` construct: `(forall i <- [0,n]: 0 <= s[i] && s[i] <= n)`. Note that `[0,n]` produces a list of integers in the set $\{0, \ldots, n-1\}$. Using the same construct we describe the predicate that establishes when a state is final: `forall i <- [0, n]: s[i] != 0`.

The `#transition` function simply places a queen on a given position: row `r` and column `c`. It is worth noting that the conditions for a valid transition are implemented in just a few lines of code.

```
#instance:          n:int ; n = 4;
#initial-state:     n:int, inits:list<int> ;
                    foreach i <- [0,n]:
                         inits[i] = 0 ;
                    end
                    return inits;
#valid-state:       s:list<int> ;
                    size(s) == n && (forall i <- [0,n]: 0 <= s[i] && s[i] <= n)
#final-state:       s:list<int> ; forall i <- [0, n]: s[i] != 0
#transition:        s : list<int>, r : int, col:int, s' : list<int> ;
                    s' = s; s'[r] = col; return s';
#valid-transition:  s:list<int>, r:int, col:int;
                    (forall i <- [0,n]: s[i] != col) &&
                    (forall i <- [0,n]:
                        forall j<-[i+1,n]:
                            abs(s[i] - s[j]) != abs(i - j)) && s[r] == 0 && col != 0
#strategy:          hill-climbing ;
                    s : list<int> ; score : int ;
                    score = 0;
                    foreach i <- [0,size(s)]:
                        if (s[i] != 0)
                            then score = score + 1
                        endif
                    end
                    return score;
```

For illustrative purposes, we choose here to use an informed strategy (hill-climbing), as it requires an additional function that assigns a score to a given state. The scoring function is provided by the user, for this example we have implemented a basic score: the number of placed queens. The rest of the implementation for hill-climbing is not required, **AIM** expects only the custom components of a strategy, as mentioned in the previous section of this paper.

4.2. nQ second model

For the second model, the **AIM** implementation differs in two places: when generating the `initial-state` and when validating a transition. These minor changes are well reflected in the **AIM** implementation and have major consequences for the difficulty of finding a solution for some search strategies, for example `hill-climbing`, require an entire different scoring function. Strategies like `iterative-deepening`, used in the example below, will also generally work much slower than in the previous model.

```
#instance:           n:int ; n = 4;
#initial-state:      n:int, inits:list<int> ;
                     foreach i <- [0,n]:
                          inits[i] = random(0,n);
                     end
                     return inits;
#valid-state:        s:list<int> ;
                     size(s) == n && (forall i <- [0,n]: 0 <= s[i] && s[i] <= n)
#final-state:        s:list<int> ; forall i <- [0, n]: s[i] != 0
#transition:         s : list<int>, r : int, col:int, s' : list<int> ;
                     s' = s; s'[r] = col; return s';
#valid-transition:   s:list<int>, r:int, col:int;
                     (forall i <- [0,n]: s[i] != col) &&
                     (forall i <- [0,n]:
                         forall j<-[i+1,n]:
                             abs(s[i] - s[j]) != abs(i - j)) && col != 0
#strategy:           iterative-deepening; maxdepth : int; maxdepth = 10;
```

The fact that the significant difference with regards to search strategies behaviour translates into minor differences in the **AIM** implementation in welcomed, as it simplifies writing, validating models and generating code from them.

4.3. MC model

For the missionaries and cannibals example model, the **AIM** implementation includes more operations on lists and more logical expressions, with **AIM** code starting to look very similar to actual Python or Java. We exemplified the selection of the `random` strategy, which assigns as new current state a random valid state which can be reached from the current state.

Recall from Figure 4 that the state has the form $s = (b, n_1, m_1, bp, n_2, m_2)$, where: the number of missionaries and cannibals on the left shore are denoted by $s[1] = n_1$ and $s[2] = m_1$; the placement of the boat is denote $s[3] = bp$ and has value 1 when the boat is on the left shore, or value 2 when the boat is on the right shore; the number of missionaries and cannibals on the right shore is denoted by $s[4] = n_2$ and $s[5] = m_2$, respectively.

The initial state for the missionaries and cannibals is $(b, n, m, 1, 0, 0)$ and it is naturally encoded in **AIM** by simple assignments. The final state needs to check whether all the cannibals and missionaries have been moved to the other shore, and thus, we only need to check if the state has the form $(b, 0, 0, 2, n, m)$.

A valid transition requires checking some complex logical formulas as shown in Figure 4. **AIM** is expressive enough to describe such complex condition in a natural manner.

```
#instance:          m:int, n:int, b:int; n = 3; m = 3; b = 3;
#valid-state:       s:list<int> ;
                    s[1] + s[4] == n &&
                    s[2] + s[5] == m &&
                    (s[3] == 1 || s[3] == 2)
#initial-state:     m:int, n:int, initialstate:list<int> ;
                    initialstate[0] = b; initialstate[1] = n;
                    initialstate[2] = m; initialstate[3] = 1;
                    initialstate[4] = 0; initialstate[5] = 0;
                    return initialstate;
#final-state:       s:list<int> ;
                    s[0] == b && s[1] == 0 && s[2] == 0 &&
                    s[3] == 2 && s[4] == n && s[5] == m
#transition:        s:list<int> , v1:int, v2:int, st:list<int> ;
                    st[0] = s[0];
                    st[1] = s[1] + v1;
                    st[2] = s[2] + v2;
                    st[3] = 3 - s[3];
                    st[4] = s[4] - v1;
                    st[5] = s[5] - v2;
                    return st;
#valid-transition:  s:list<int> , v1:int , v2:int;
                    (forall i <- [1,6]: s[i] >= 0) &&
                    (s[2] > 0 -> s[2] >= s[1]) &&
                    (s[5] > 0 -> s[5] >= s[4]) &&
                    (v1 + v2 > 0 -> s[3] == 1) &&
                    (v1 + v2 <= 0 -> s[3] == 2)&&
                    abs(v1 + v2) <= s[0]
#strategy:          random
```

4.4. Towards generating executable code from **AIM** *specifications*

AIM is designed to be easily translatable into an existing programming language. Except for `#instance` and `#strategy`, all the other entries in an **AIM** specification are designed to be functions. For example, the corresponding function of an `#initial-state` for the cannibals and missionaries problem takes as parameters `m`, `n`, and `b`, while the body of the function is given by the assignments. We show here a possible translation to the Python language of the function which returns the initial state:

```
def  initial_state(m, n, b):
     initialstate = list(range(0,6))
     initialstate[0] = b
     initialstate[1] = n
     initialstate[2] = m
     initialstate[3] = 1
     initialstate[4] = 0
     initialstate[5] = 0
     return initialstate
```

Other functions, like the ones corresponding to predicates (`#valid-state`, `#valid-transition`, `#final-state`) return boolean results. The `#valid-transition` function needs to be combined with the `#transition` function, in order to compute valid transitions.

Depending on the target programming language and the decision of the user, an `#instance` can be translated using global variables, specialized functions, or command line parameters.

Strategies are completely decoupled from an **AIM** specification. By design, strategies are meant to be implemented directly in the target programming language. The implementation of these strategies can be parameterised by some other functions (like scoring functions).

5. Conclusions and future work

We have shown how we can use **AIM** to describe AI models with the goal to generate executable code. The language is proposed to be capable of representing all common types of problems, and models implemented can be adapted easily to any search strategy that requires it. The first programming language to which AIM models will be transferred to Python. Some consideration regarding this near future implementation were made in this paper. The benefits provided by getting an executable code from an **AIM** description of a model for any AI problem are evident, and go towards the established goal of AI: use computers to solve problems with minimum human intervention.

AIM specifications are also meant to serve as a basis for other tools which can provide interactive step-by-step model exploration, model execution tools, methodologies and tools for checking formal properties of the models. This will serve both didactic purposes as well as high-level model checking, which usually requires a high degree of competence in AI as well as in formal logic and mathematics.

AIM usage could be further extended by providing automated conversion to and from other description languages (e.g., STRIPS, PDDL), which are already familiar to potential users and already include many implemented models.

Natural language is ambiguous and its correct understanding requires addition information beside its content, such as context and general world knowledge. These are the initial difficulties faced by automated problem solvers - transferring the natural language statements into an unambiguous and informative representation allowing a computer to use a search strategy to look for solutions. Efforts to automate this process have been made, for instance in [18], [16] and more recently in [10], but they generally simplify the issue by expecting the problem to be formulated in a particular way, limiting the scope of the approach. We believe a possible new solution could be provided by **AIM**, using implemented problems and the expressive power of the grammar to generate new natural language statements by replacing grammar constructs with natural language equivalents and improving the resulted text's general cohesion and coherence, as well as localized grammar, by using established text generation techniques such as those described in [9], [13] or more recent one like [19] and [8] which use Deep Learning to train patterns. The resulted alternate text statements can be then used to train a machine learning system potentially able to recognize and map new problem statements to **AIM** grammar constructs.

Acknowledgements. This work was supported by a grant of the "Alexandru Ioan Cuza", University of Iaşi, Research Grants program, Grant UAIC, ctr. no. 6/03.01.2018 and by the UEFSCDI research project ctr. no. 73PCCDI/2018.

References

[1] Saul Amarel. On representations of problems of reasoning about actions. In *Readings in artificial intelligence*, pages 2–22. Elsevier, 1981.
[2] A Arusoaie and IC Pistol. Using SMT solvers to validate models for AI problems. Technical report, UAIC, March, 22 2019. 1903.09475.
[3] R Fikes and N Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.
[4] International Organization for Standardization. Information technology - programming languages - C - ISO/IEC 9899:2018, 2018.
[5] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.
[6] Adrian A Hopgood. *Intelligent systems for engineers and scientists*. CRC press, 2016.
[7] Guy Steele Gilad Bracha Alex Buckley Daniel Smith James Gosling, Bill Joy. The Java language specification - Java SE 12 Edition, 2019.
[8] Chloé Kiddon, Luke Zettlemoyer, and Yejin Choi. Globally coherent text generation with neural checklist models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 329–339, 2016.
[9] William C Mann. Discourse structures for text generation. In *Proceedings of the 10th international conference on Computational linguistics*, pages 367–375. Association for Computational Linguistics, 1984.
[10] C Matuszek, E Herbst, L Zettlemoyer, and D Fox. Learning to parse natural language commands to a robot control system. In *Experimental Robotics*, pages 403–415. Springer, 2013.
[11] W McCulloch and W Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of math. biophysics*, 5(4):115–133, 1943.
[12] D McDermott, M Ghallab, A Howe, C Knoblock, A Ram, M Veloso, and D Weld, D Wilkins. Pddl-the planning domain definition language. 1998.
[13] Kathleen McKeown. *Text generation*. Cambridge University Press, 1992.
[14] F.P. Miller, A.F. Vandome, and M.B. John. *Extended Backus-Naur Form*. VDM Publishing, 2010.
[15] Allen Newell, Herbert Alexander Simon, et al. *Human problem solving*, volume 104. Prentice-Hall Englewood Cliffs, NJ, 1972.
[16] F Pereira and D Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, 1983.
[17] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.
[18] Candace L Sidner. Plan parsing for intended response recognition in discourse 1. *Computational intelligence*, 1(1):1–10, 1985.
[19] Yizhe Zhang, Zhe Gan, Kai Fan, Zhi Chen, Ricardo Henao, Dinghan Shen, and Lawrence Carin. Adversarial feature matching for text generation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4006–4015. JMLR. org, 2017.