How to Pass an Intersection, or Automata Theory Is Still Useful

Jacek Malec*
Department of Computer and Information Science
Linköping University, S–581 83 Linköping, Sweden, jam@ida.liu.se

Abstract

In this paper I present several knowledge representation (KR) languages used within one project, and show relationships between these languages. I try to persuade that when designing a complex AI system, one should use different KR formalisms, namely the ones most suitable for purposes of different subsystems, and devise an appropriate tool for data (or knowledge) interchange between these subsystems. The KR formalisms need not be the ones traditionally associated with AI — there are lots of other valuable tools. The argument is presented using one particular example of an AI system being designed within the ProArt part of the PROMETHEUS project.

1 Introduction

Any complex AI system requires some knowledge representation (KR) formalism(s) to be chosen as a basis of its reasoning. In case of general purpose systems, which are designed to perform any imaginable sort of reasoning, and are not tied to any particular application (as e.g. CYC [1]), one can (and probably should) choose one KR formalism which would serve as a representational tool within the whole system. Although several subsystems may use different representations, they, or their front-end processors, are responsible for translating knowledge chunks from the "basic" representation to the one they actually use.

In case of systems tied to some particular application, and especially if a system consists of several non-homogeneous subsystems, one can imagine (and justify) totally different approach. In such case the subsystems (often constructed by different teams, and almost always aiming at effectiveness, and not at generality) govern the overall design in the sense that KR methods used by some specific subsystem cannot (or should not) be changed. The problem consists therefore of linking such non-homogeneous systems by paths allowing both interchange of knowledge, and translation between one representation and another. For example, a system able to act in the real world (e.g. an intelligent robot) would probably use different KR tools within the following subsystems:

- *knowledge acquisition* subsystem. I mean here both knowledge acquired during "being-told" phase (in the simplest case this may degenerate to hand-coding), and knowledge acquired during work phase (for example by some learning procedures);

*This research was supported by the IT4 research program.

- *perception* and *execution* subsystems, which gather information from sensory inputs and affect effectors (and when using both, the system is able to communicate with the environment);
- *reasoning* subsystems (in case of more complex systems one can expect several such systems, used for different purposes, and using different KR tools);
- *communication* subsystems, which are used to interchange knowledge chunks among different subsystems.

In this paper I present (however very sketchily) several KR languages used within one particular system, and point to relationships linking these languages. I'm trying to show (by pointing to an example) that the KR formalisms used need not be the ones traditionally associated with AI — there are lots of other valuable tools. In particular, two of the three languages I present here are not "AI-type" KR formalisms. The first one is a pictorial language based on graph theory, called the Process Transition Networks (PTNs). The second language is regular, and its interpretation is done by classical accepting automata. The system I use as my example is Co-Driver, being designed within the ProArt part of the PROMETHEUS project.

2 Architecture of the System

The detailed structure of Co-Driver system has been presented in [10]. Generally speaking, it consists of three layers defined to perform three different tasks [3]. The first layer is called the *process layer* because it's responsible for controlling the car, and consists of processes running and communicating in real-time with each other. The second layer is called the *rule layer*, as the knowledge about reasons and ways of selecting activities of the system is encoded there using a rule-based systems paradigm. The *reasoner layer* is intended to perform all the tasks requiring symbolic reasoning (as e.g. planning), and will be provided all relevant information about the environment, and the lower levels of the system (in the form of temporally annotated facts). It is expected to produce controlling sequences (which again are just temporally annotated facts) for the rule layer.

All the languages described later in the paper are tied with rule layer. PTNs are used to define (and illustrate) a set of rules to be encoded. Assignment Statements language (ASL) is used to actually code (as logical formulae) the rules, and to verify their formal properties (such as e.g. consistency). Moreover ASL serves as communication language between the layers. Finally, the configuration language is an element of implementation of an interpreter for ASL, and helps efficiently handle information about car configurations.

3 Process Transition Networks

Process Transition Networks (or shortly PTNs) have been proposed by Sandewall in [8]. They are intended to serve two purposes:

- make knowledge acquisition process easier (much less effort is needed to construct PTN diagrams describing particular traffic situations than e.g. when doing the same using logic formulas);
- serve as a tool for knowledge explanation to non-experts.

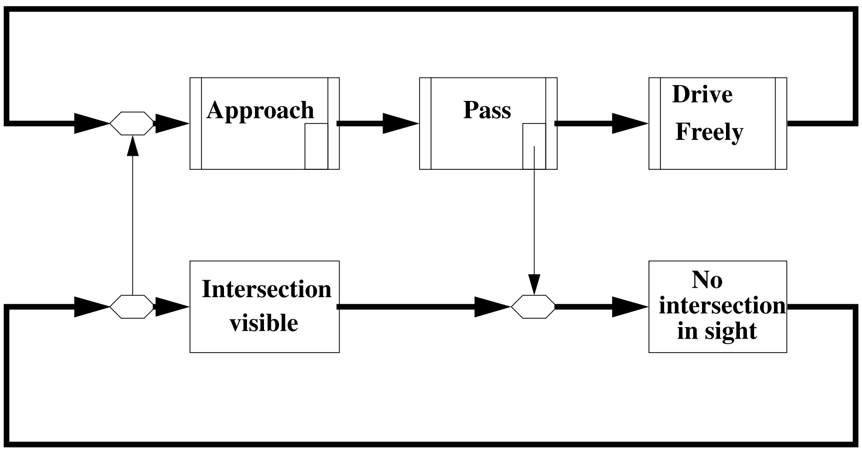

Figure 1: “How to pass an intersection” net.

PTNs are closely tied to several well-known formalisms exploiting advantages of graphical representation. The best known are of course Petri nets [6]. PTNs are proper subset of Petri nets, but it appears to be their advantage — they could have been designed in such way that expressing complex causal (and thus temporal) dependencies requires only one symbol, what is impossible to obtain when using Petri nets. Another well designed graphical formalism has been proposed by Harel [2]. His *statecharts* seem to be similarly motivated as PTNs. Both formalisms have very much in common, but have also some major differences, discussed in [5].

Let's begin with some terminology. We assume that state of the world (including our agent, or agents) may be described using a set of properties, where property values may change along the time axis. In PTN language a box represents a *process*, or more formally, a period of time when some property of environment *continuously* changes its value. We are interested in these moments, in which discontinuities (of the property value seen as a function of time) occur, and which we call *events*. Events are represented here just by a thick arrow from one process to another. *Activities* are processes describing behavior of some *agent*, and *actions* are such activities which have "inherent termination conditions". E.g. *Drive Behind A Car*, or *Drive Alone* are activities, whereas *Slow Down by 10 km/h* is an action finishing when the car's speed becomes 10km/h less than the initial one. Activity boxes are distinguished by doubling their vertical borders, and actions contain additional internal box denoting their termination condition. Causal dependencies are represented by s.c. *enabling arrows* shown with thin lines, and pointing from either event alone, or event and subsequent process, to some other event.

In the following simple example PTNs have been used to conceptualize the scenario of passing an intersection. Details can be found in [4]. Figures 1 and 2 present two PTNs describing, respectively, very simplified view of passing an intersection, and a part of much more detailed analysis of this scenario. Below an intuitive description of contents of depicted PTNs is given.

When an intersection becomes visible (what corresponds to the transition "No intersection in sight" → "Intersection visible", the car switches from the "DriveFreely" activity to the "Approach" action. The "Approach" action consists of appropriate slowing down (up to stopping, if needed), checking intersection state in order to determine if crossing is possible, and preparing to choose the outcoming road segment (left, right, or straight). After concluding the "Approach" (ie. coming to the meeting point of the

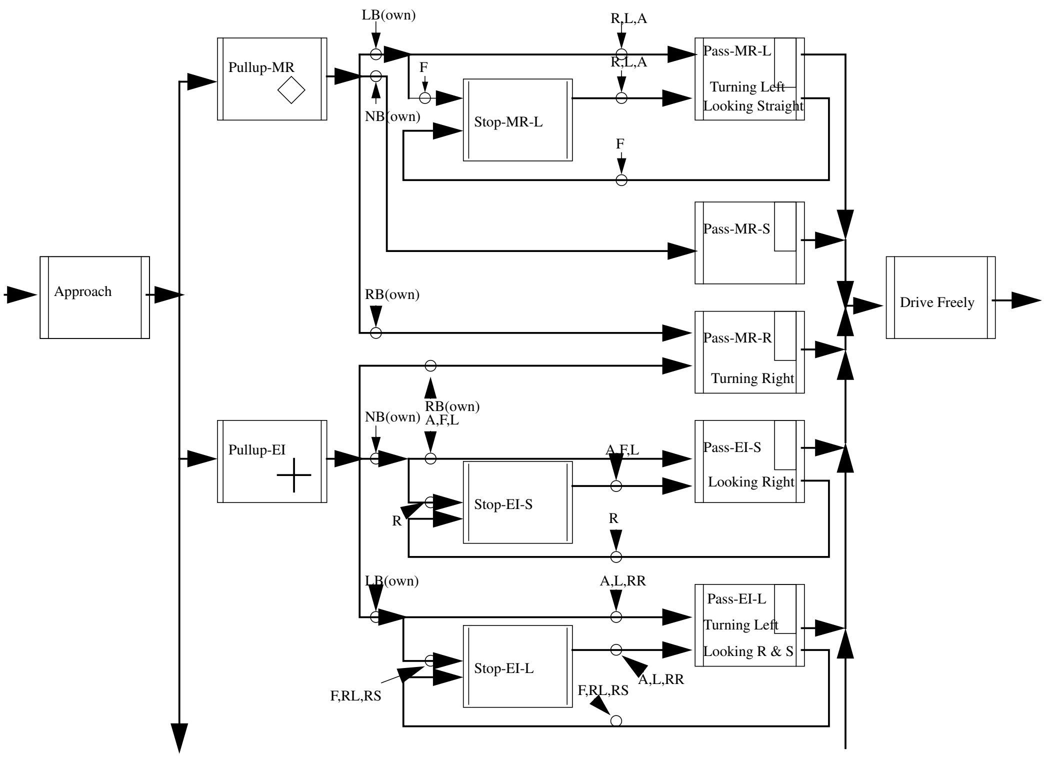

Figure 2: A fragment of "How to pass an intersection" net, with more details shown.

road segment we are on and the intersection) the car switches to the "Pass" action, which concludes passing through the intersection and causes the intersection to vanish. After "Passing" intersection the car switches to the "DriveFreely" activity again, until it meets the next intersection.

It's worth pointing that the only conceptual difference between the "intersection" cycle, and the "car activity" cycle is that we assume that the latter corresponds to the behavior of some intelligent agent, whereas the former describes the agent's environment. We should note also that causal dependencies point both ways, so changes within environment can cause some change of agent's behavior, and vice versa.

"Approach–Pullup–Stop–Pass–DriveFreely" path (cf. fig. 2) contains the processes (in fact, activities) describing behavior of our own car. They correspond to changing value of some fluent, say "State-of-my-own-car", which, in its turn, corresponds to some slot in the state vector containing all the sensory input data.

Because we don't explicitly mark events, but only transitions between processes (thick arrows on the pictures), enabling arrows (thin ones) are drawn between transitions, and not explicitly named events. All the labels on fig. 2 denoting triggering events (called configurations) correspond to different processes in some other part of the PTN, and describe configurations of crossing cars. These labels should be read as follows:

A – no crossing cars;

RL[1] (RS, RR, FL, FS, FR, LL, LS, LR) – there is at least one car having the RL (RS, RR, FL, FS, FR, LL, LS, LR) property among the cars approaching the intersection from other directions than the OWN car does.

[1]In this one, and in the following two-letter labels, the first letter corresponds to the direction the crossing car comes from, the second letter — to the direction it intends to turn.

R (F, L) – there is at least one car with either RL, or RS, or RR (either FL, or FS, or FR; either LL, or LS, or LR) property among the cars approaching the intersection from other directions than the OWN car does.

It should be stressed that PTNs strongly encourage hierarchical approach to the task of domain description. We can start with very rough conceptualization, distinguishing only major properties of the world around us, and their most important values, and then we can successively refine this first draft obtaining accurate (and, what's important in our case, ready to implement) description of (hopefully) all the important interactions among some agent(s) and its (their) environment. Figures 1 and 2 illustrate this top-down approach, where one box from fig. 1 is split into several ones in fig. 2.

Another advantage of PTNs is the following: they can be used to graphically present and explain the actual contents of a knowledge base. Although this factor is negligeable from formal point of view, it becomes very important in case where domain experts, not necessarily sharing our admiration of logic, have to be involved in the process of creation, validation, and expansion of a large and complex knowledge base.

A formal definition of PTN as a graph, and its rephrasing as a Petri net, have been presented in [5].

4 Assignment Statements

As the three layers of the proposed architecture have to communicate intensively, for the purpose of information transfer we have chosen some restricted temporal logic language called the *assignment statements language* (ASL). It shares the simplicity of the process layer (where the only kind of information passed to and from the rule layer are changes of some predefined parameters) and logical strictness of the reasoning layer (which, we assume, will be based on some multi-type temporal first order logic). Several details of ASL, together with description of its implementational counterpart (which is called *Discrete Fluent Management System*, [7]), are described below.

4.1 Language

Let us first present the definition of an *assignment*. It is grounded on the fact, that the world perceived by the system is described using *dynamic frames*, i.e. knowledge structures with slots, values of which may change over time.

The first kind of objects referred to by assignment statements are *propositional fluents*, or *circumstances*, ie. propositions, truth value of which change over time. Examples of circumstances: $tralite_2 \in tralite$ meaning "object (or frame) $tralite_2$ is of type *tralite*"; $color(tralite_2) = red$ meaning "property *color* of object $tralite_2$ has value *red*"; $ispart(moveleft_4, overtake_4)$ meaning "association of the role *ispart* holds between the two actions".

These are the only types of propositional fluents considered. Then we introduce their syntactic counterparts allowing us to express such propositions, namely the relations $\rhd$, and $\bullet\!\!\rhd$ (for $\in$), $:=$, and $\bullet\!=$ (for $=$), and $:\!\uparrow$, and $\bullet\!\uparrow$ (for role associations), where the bullet means that some relation *starts to be true precisely at the current time point*, and the colon that the relation simply *is true*. So as atomic formulas we consider such propositional fluents which are constructed using the above six (binary) relations. Of course, they can bind only terms (objects) of appropriate type.

Having defined the atoms we can define formulae as follows:

An *assignment statement* is formed from the abovementioned atomic formulae using standard logical connectives & and $\Rightarrow$, and two unary connectives λ and ρ (intended to express temporal scope of their argument).

Intuitively, λ denotes the left (in the temporal sense) limit of some value, and ρ the right limit, so e.g. $\lambda(tralite_2 := red)$ means that $tralite_2$ has been red until the current time point, but not necessarily now. Precise definitions of these operators can be found in [9].

4.2 Implementation

The language sketched above is being used to express the set of rules governing the behavior of one of the rule layer subsystems, namely the *selector* (which selects the appropriate control algorithm to be applied in the process layer). The implementation of this subsystem, Discrete Fluent Management System (DFMS) [7], is given a set of ASL rules describing the dependencies between the input fluent changes and required actions (or output fluent changes).

As PTNs are semantically subsumed by ASL, one can translate the former (which are easier to write, and helpful during the domain conceptualization phase) to the latter (which are the programming language for the DFMS). Moreover, this translation can be mechanized, so the final user's description of the domain would be a set of PTNs, rather than the ASL statements themselves.

5 Regular languages as a sufficient tool of environment description

One of the particular problems to be solved during the Co-Driver design was choosing an appropriate way of describing a configuration of cars on an intersection (such description is the value of some slot of an intersection frame). Its value should inform about configuration of cars coming from each direction, and intending to make some specific maneuver on the intersection. In order to be able to describe an arbitrary configuration of cars approaching an intersection, we have defined for this purpose a special regular language called *configuration language* ($\mathcal{CL}$).

Assuming that $C = \{\lambda, l_l, l_r, l_s, r_l, r_r, r_s, f_l, f_r, f_s\}$ is the alphabet, the *configuration language* $\mathcal{CL}$ can be defined using the classical recursive approach:

1. $\emptyset \in \mathcal{CL}$
2. $a \in \mathcal{CL}$ for every $a \in C$
3. If $a_1, a_2 \in \mathcal{CL}$, then also $(a_1 + a_2) \in \mathcal{CL}$ (called *alternative* or *sum*, and intended to mean "either a_1 or a_2")
4. If $a_1, a_2 \in \mathcal{CL}$, then also $(a_1 a_2) \in \mathcal{CL}$ (called *concatenation*, and intended to mean "a_1 and a_2")
5. If $a \in \mathcal{CL}$, then also $(a)^* \in \mathcal{CL}$ (called *iteration*, and intended to mean "concatenation of arbitrary number of a-s, including 0")
6. Only the strings constructed according to $1 \div 5$ belong to $\mathcal{CL}$.

We will omit parentheses if it does not lead to misunderstanding.

Moreover, we introduce the usual abbreviation $a^+ = a(a)^*$, plus a number of useful, domain-specific abbreviations: $l = (l_l + l_r + l_s)$, $r = (r_l + r_r + r_s)$, $f = (f_l + f_r + f_s)$, $q = (l + r + f)$.

Having defined syntax of $\mathcal{CL}$, let's say a few words about its intended meaning. First, some obvious configurations:

- λ — the roads coming to the intersection are empty (except our own car);
- l — some car is coming from our left;
- l_l — some car is coming from our left and intends to turn to its left;
- $l_s, l_r, f_l, f_s, f_r, r_l, r_s, r_r, r, f$ — analogously.

Something a bit more difficult:

- $lfrr$ — there are four cars approaching to the intersection: one from left, one from front, and two from right;
- $rfrl$ — same as above;
- $frrl$ — same as above;
- $l_r l$ — there are two cars coming from left, the first of them is going to turn right;
- $(r + f)^*$ — there is no car coming from our left;
- $(r + f)^+$ — there is no car coming from our left, but there is at least one car approaching the intersection from some other direction.

One can even say something interesting about some particular car, for example "The second car from left intends to turn right": $(r + f)^* l (r + f)^* l_r q^*$.

Saying something about the second car from left and about the second car from right becomes a bit complicated. E.g.: "The second car from left intends to turn right and the second car from right intends to turn right" can be written as follows: $f^*rf^*r_rf^*lf^*l_rq^*+ f^*rf^*lf^*r_rf^*l_rq^*+f^*rf^*lf^*l_rf^*r_rq^*+f^*lf^*rf^*r_rf^*l_rq^*+f^*lf^*rf^*l_rf^*r_rq^*+f^*lf^*l_rf^*rf^*r_rq^*$. However, in our particular application we need to state the properties of all the cars coming from some direction, so the problems with saying something specific about "the nth car in a row" needed not bother us.

We have devised (and built into the rule layer) an algorithm constructing a *nondeterministic automaton with ϵ-moves* accepting given regular language. Moreover, this approach allows us to parse incrementally the current configuration: if some additional information about the configuration is known (some letter is being appended to the end of an expression) then we can proceed with parsing it, not needing to reparse the first part of this description. Details of this approach can be found in [4].

6 Conclusions

In this paper I have presented very sketchily several KR languages used within one particular system, namely the Co-Driver, being designed within the ProArt part of the pan-european PROMETHEUS project. I was trying to show relationships linking these languages. In my opinion, this example shows that the KR formalisms used by a complex, heterogeneous AI system need not be the ones traditionally associated with AI — there are lots of other valuable tools. In particular, among others I have presented one language based on graph theory (PTNs), and one language being precisely a regular language ($\mathcal{CL}$), interpretation of which is done by classical accepting automata. I think that this example advocates usage of different kinds of such representational tools.

The paper covers only some subset of languages used within our project. Moreover, the presented languages were described in a very sketchy and informal way (because of the space constraints). Detailed descriptions are listed in references. For some of the languages used, however, some work has still to be done in order to establish their precise, formal semantics, and relations among them (currently ASL semantics is "operational", in the sense that speaking about meaning we have the implementation as a reference in mind). This is planned to be done in the nearest future.

Acknowledgment

Thanks to Jacek Wrzos-Kamiński for valuable remarks.

References

[1] R. V. Guha and Douglas B. Lenat. CYC: A midterm report. *AI Magazine*, 11(3):32–59, 1990.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[3] Johan Hultman, Anders Nyberg, and Mikael Svensson. A software architecture for autonomous systems. In *Proc. 6th International Symposium on Unmanned Untethered Submersible Technology*, pages 279–292, 1989.

[4] Jacek Malec. How to pass an intersection. In *Proc. of the Prometheus Pro-Art Workshop on Intelligent Co-Pilot*, pages 167–182, Grenoble, France, 1991.

[5] Jacek Malec. Process Transition Networks: A formal graphical knowledge representation tool. In Z. W. Ras and M. Zemankova, editors, *Methodologies for Intelligent Systems, 6th International Symposium*, pages 193–202. Springer-Verlag, October 1991. Lecture Notes in Artificial Intelligence 542.

[6] James L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[7] Erik Sandewall. Discrete Fluent Management System, a program in the HIRW Workbench. Unpublished report, 1990.

[8] Erik Sandewall. Proposal for a ProArt specification platform. Technical Report LAIC-IDA-90-TR18, IDA, Linköping University, 1990.

[9] Erik Sandewall. Reasoning about the world as perceived by an agent. In L. C. Aiello, editor, *Proceedings ECAI-90*, pages 579–584, Stockholm, 1990. Pitman. Available also as LAIC-IDA-89-TR30.

[10] Erik Sandewall. System Block Specifications and their PROMETHEUS applications. In *Prometheus: Proceedings of the Fourth Workshop*, 1990. Available also as LAIC-IDA-90-TR22.