Applied Knowledge Representation

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

Abstract

The number of knowledge representation (KR) formalisms grows exponentially. What most of those formalisms lack is a *successful application*. This is particularly true of logic-based KR systems, often created without taking tractability into account. Applied Artificial Intelligence (AI) systems, on the other hand, are often based on one specific formalism claimed to be *the* one.

In this paper we present several knowledge representation tools used within one project and show relationships between these languages. In this way we try to support the conjecture that when designing a *complex* AI system, one should use various 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 many other valuable tools. We are using one particular example of an AI system being designed within 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. For general purpose systems, which are designed to perform any imaginable sort of reasoning and are not tied to a particular application (e.g. CYC [GL90]), one can choose a formalism which would serve as a representational tool within the whole system. The only problem is to guarantee a "reasonable" system response time. Although different subsystems may use different representations, they, or their front-end processors, are responsible for translating knowledge chunks from the common representation to the one they actually use.

For systems tied to some particular application, and especially if a system consists of several non-homogeneous subsystems, one can imagine and justify a totally opposite approach. In this case the designs of subsystems (often constructed by different teams, and almost always aiming at effectiveness, and not at generality) constrain 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 these 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 methods within the following subsystems:

- *perception* and *execution* subsystems, which gather information from sensory inputs and drive 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);
- *knowledge acquisition* subsystem. We mean here both knowledge acquired during "being told" phase (in the simplest case this may degenerate to hand coding), as well as knowledge acquired during work phase (for example by some learning procedures);
- *communication* subsystems, which are used to interchange knowledge chunks among different subsystems.

Yet another language may be needed for *system specification*.

One possible approach would be to allow different subsystems to use their own KR tools, but to introduce a common *knowledge interchange* (KI) tool [San91, Gen90], implemented within the system. Another possibility is to use the KI tool during the specification and design phases but to eliminate it from the actual implementation. In both cases, however, the KI language has to be rich enough to express every form of information the system can deal with, yet also simple enough to be easily implemented, without introducing any computational overhead.

In this paper we present several KR languages used within one particular system, and point to relationships linking these languages. We will show by example that KR formalisms need not necessarily be the ones traditionally associated with AI — there are many other valuable tools. In particular, two of the three languages we present here are not typical KR formalisms. The first one is a pictorial language based on graph theory (Process Transition Networks, or PTNs, section 3). The second formalism is a very restricted temporal logic able only to distinguish facts that hold "now", from the facts that held "just a moment ago" (Restricted Elementary Temporal Feature Logic, or RETFL, section 4). The third language is a regular one (Configuration Language, or CL, section 5), and its interpretation is performed using classical accepting automata. Most of the examples will refer to our major application domain, namely road traffic, because the prototype system we are developing is intended to control an intelligent autonomous car. Configuration Language has been chosen especially for the purpose of this application domain. The first two languages, however, and the system architecture described in the next section, are much more general. Some of the examples illustrate the behavior of a can-collecting robot, HERBERT, constructed in the MIT AI Lab [Bro90].

2 Architecture of the System

The detailed architecture of the system employing KR tools to be discussed in this paper has been presented in [San90]. It consists of three layers, each

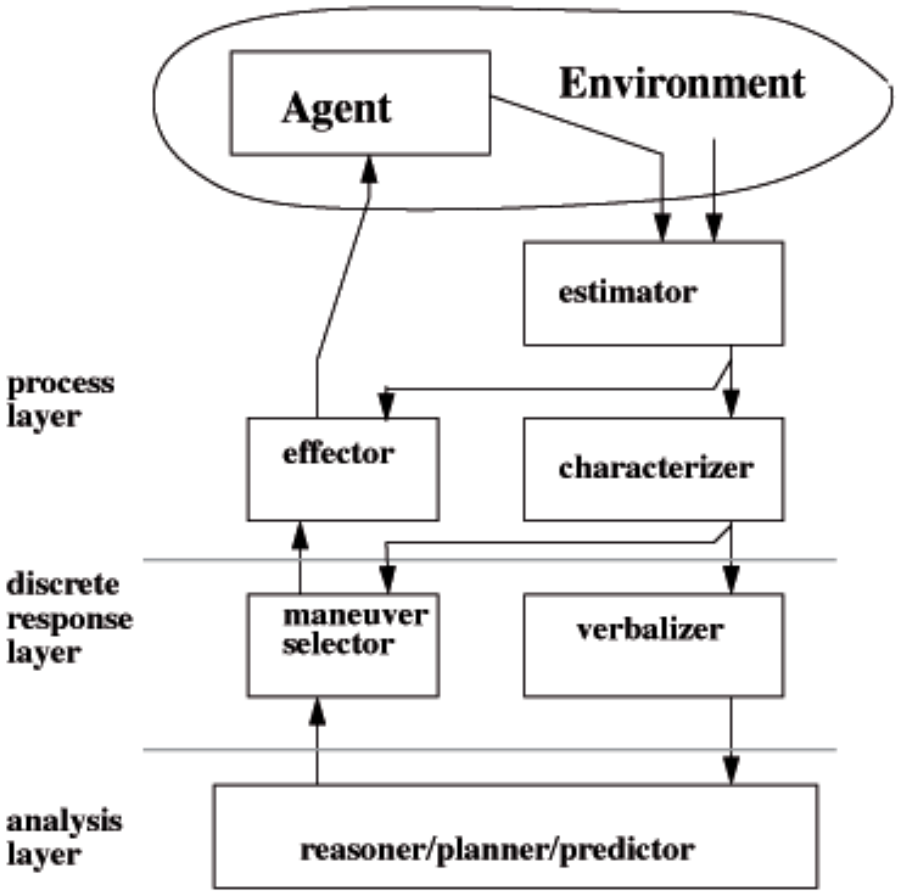


Figure 1: A three-layer architecture of an autonomous system.

defined to perform a different task ([HNS89], see Figure 1). The first layer, called the *process layer*, is responsible for controlling the robot's actuators, and consists of processes running and communicating with each other in real-time. The second layer, called the *rule layer*, contains the knowledge about reasons for and methods of selecting activities of the system. This knowledge is encoded using a rule-based representation paradigm. The *reasoning layer* is intended to perform all the tasks requiring symbolic reasoning (e.g. planning), and will be provided all relevant information about the environment and about 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 in this paper are related to the rule layer. PTNs are used to specify and illustrate the desired behavior to be encoded in a set of rules in this layer. RETFL is used to actually code (as logical formulae) the rules, and to verify their formal properties (such as consistency). Moreover, RETFL formulae are used for communication between the layers. Finally, CL is used to describe a very important subset of the application domain: car

configuration at an intersection. From the RETFL point of view, a CL statement denotes an object, i.e. it refers to the interpretation domain of the logic. When a RETFL formula containing an embedded CL expression is being interpreted, the CL-expression part is handled by a special sub-interpreter able either to accept or to reject this expression.

Only a subset of the languages used within the system is described in this paper. The three that have been chosen are interesting because they illustrate different but equally important aspects of KR techniques employed within *one layer* of a multi-layer system. The mutual dependence of the languages is enforced by assuming a fixed application domain, but both PTNs and RETFL can be seen as totally domain-independent tools for specifying and implementing discrete event-driven control systems.

3 Process Transition Networks

For the purpose of describing complex real world domains, such as road or air traffic, manufacturing processes, systems maintenance, etc., one needs to develop techniques for:

1. describing both the static and dynamic aspects of the environment in a sound, systematic, and user oriented way (taking not only formal precision but also legibility for domain experts into account);

2. formally analyzing the descriptions developed using techniques mentioned in the previous point, in order to verify their consistency, completeness, and computational complexity (the last one being especially important in real-time domains of application);

3. assisting the implementation of tools such as simulators, planners, and controllers for such environments. These tools should be "tied" to the description languages from the point 1, and conform to the requirements mentioned in the point 2.

In this section we present Process Transition Networks (PTNs) which are intended to serve two purposes:

- make the knowledge acquisition process easier: it is easier to model process dependencies with PTNs than with other representation tools, e.g. logic formulae or Petri nets;

- serve as a tool for knowledge explanation to non-experts.

PTNs are a language designed for the purpose of describing complex environments that, although are well organized by causal dependencies, are hard to capture using quantitative parameters. Currently our main application area consists of typical traffic scenarios, but possible application areas of PTNs are much more numerous. The design of the language is motivated by requirements (1) — (3) listed above, although only (1) is covered in this paper.

The PTN language is based on graph theory and its expressive power is less than that of Petri nets. The number of existing knowledge representation tools, including many pictorial languages, may raise the question whether yet another graphical knowledge representation tool is necessary. This section is an attempt

to answer this question. First, both the syntax and semantics of the language are defined formally, which makes it possible to compare the PTNs with other similar tools, and to prove their useful properties. Second, PTNs are well suited for expressing common-sense conceptualizations of complex environments. The ability to easily express *causal dependencies* between described entities is the major factor distinguishing PTNs from other formalisms.

3.1 PTN Language Primitives

We assume that the world consists of objects (some of which we call *agents*), and that the state of the world (including the state of the agent or agents) may be described using sets of object properties. Values of object properties may (and usually do) change with time. For most properties one of the following two statements is true:

- The set of property values is finite, and the value function is piecewise constant.
- The set of property values is infinite, and is usually continuous. The property's value may change with time, but except for some distinguished time points, the change is continuous. Thus the value function is piecewise continuous.

A set of adjacent time tokens during which a property value *continuously* changes (this includes values that remain constant) will be called a *circumstance*. So, for example, the state of a robot performing some operation (such as polishing a nearly-finished kitchen sink, or searching for an empty soda can), traffic lights being red, a car following another one, an airplane following some air corridor, or a computer being hung, are all examples of a circumstance.

In the PTN language we introduce the notion of a *process*, corresponding to some circumstance. A process will be depicted by a box. We are interested in those moments in which discontinuities of a property occur, because discontinuities convey important information about changes of the current situation.

We will call such discontinuities *events*. Events are represented in the PTN language by hexagonal boxes. Since discontinuities (by definition) may only appear between periods of continuous change of some property, events only appear between two processes. Such a *transition* (from process to event to another process) is denoted by two thick arrows, the first pointing from the first process to the event, and the second pointing from the event to another process.

Causal dependencies between discontinuities are represented by *enabling arrows*, denoted by thin lines pointing from either an event alone, or both an event and its subsequent process, to some other event. The first possibility corresponds to the case where one discontinuity *forces* another one to occur; the second possibility describes the case of an actual *enabling*.

A simple PTN constructed from these primitives is shown in Figure 2. This PTN models the behavior of a system consisting of three robots cooperating during the finishing phase of sink production.

A process that describes the behavior of some agent is called an *activity*. Activities that have implicit termination conditions (i.e. their termination condition is not explicitly stated in the PTN) are called *actions*. Activity boxes are distinguished by doubling their vertical borders; each action contains an

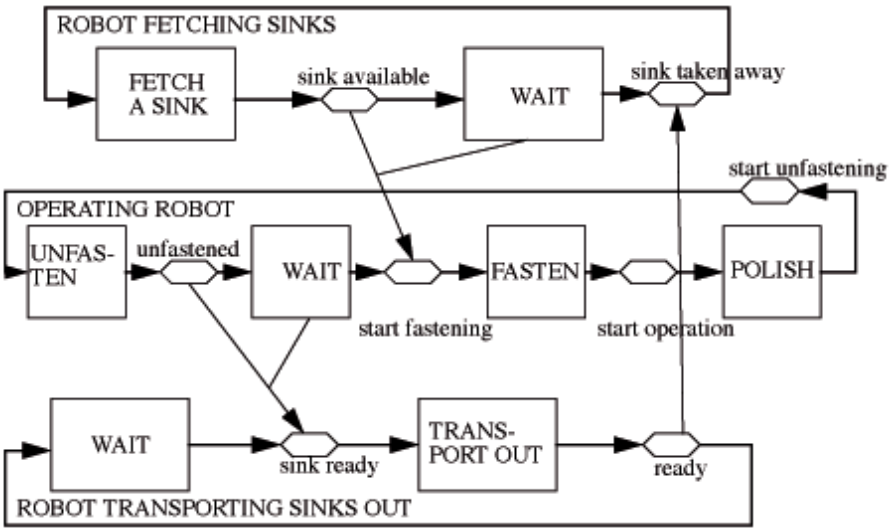


Figure 2: Three cooperating robots.

additional internal box denoting its termination condition. Actions are not followed by events — the event is "encoded" in the termination conditions of the action. However, if it is desired that this event enables some other event, an enabling arrow may be drawn directly from the box representing the termination condition. Figure 3 illustrates the use of activity and action boxes.

Activities and actions may be distinguished in an arbitrary way. They are used to focus attention on processes ascribed to some agent or agents. Later on we will assume that every PTN cycle containing an activity consists only of activities. The rationale behind this is that if some process (circumstance) is associated with some property of an agent, then all the subsequent values of that property (and thus, all the subsequent processes or circumstances) are also associated with the agent.

3.2 Creating more complex descriptions

The main reason for distinguishing actions is to keep PTNs as simple as possible. Moreover, making such a distinction allows us to draw the enabling arrows in an intuitively clear fashion. However, every action box can be seen as a way of encoding an activity and its subsequent event, together with the whole PTN cycle describing the triggering conditions for the event. Figure 4 illustrates the correspondence between action boxes and activity boxes enabled from some external cycle. The termination conditions of an action are just the conditions described by this external cycle. This example illustrates a fragment of high-level description of behavior of the robot HERBERT constructed in the MIT AI Lab. The robot is able to wander around the lab and to collect empty soda cans [Bro90].

In the PTN language, one may combine enabling arrows using both **and** and **or** connectives. The **and** of two enablings is expressed by two consecutive events on one transition between two processes. The **or** may be represented with two

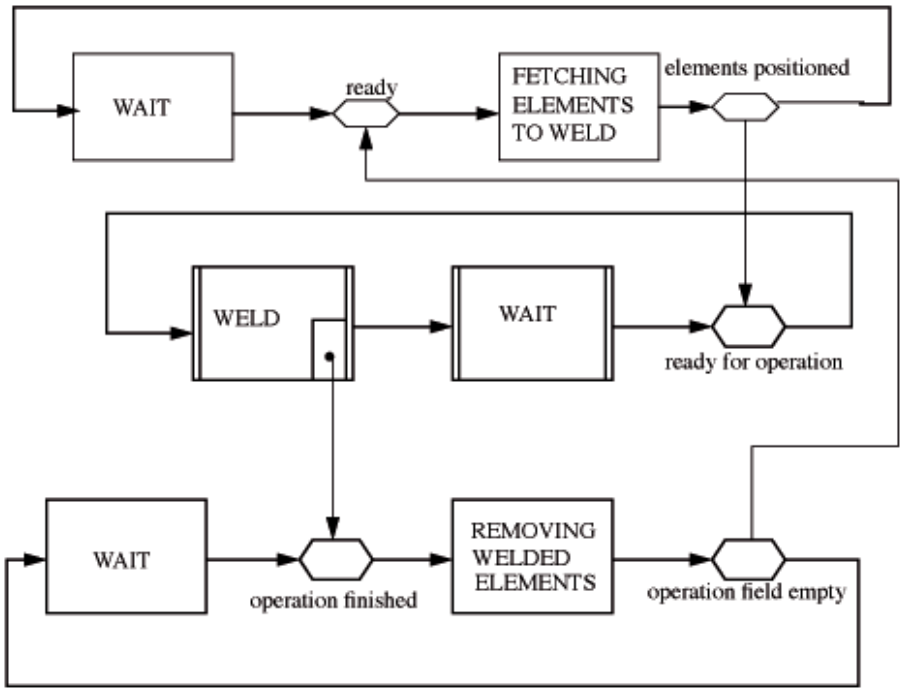


Figure 3: Welding robot activities.

different paths leading from one process to another, but is usually depicted by merging the enabling arrows themselves. See Figure 5.

Another primitive of the PTN language is *decision*. It is an action performed by an agent faced with several possibilities. Like actions, decisions encode some external condition which is not shown in the network. Decision box is pentagonal; an example of its usage, together with the equivalent network consisting only of the primitives defined previously, are illustrated in Figure 6.

In the example below, taken from our application domain, PTNs have been used to conceptualize the scenario of passing an intersection. Details can be found in [Mal91a]. Figures 7 and 8 contain two PTNs describing, respectively, a very general overview of passing an intersection, and a part of a much more detailed analysis of this scenario. An informal description of contents of these PTNs is given below.

When an intersection becomes visible (corresponding to the transition No intersection in sight → Intersection visible), the car switches from the Drive Freely activity to the Approach action. The Approach action consists of an appropriate deceleration (down to stopping, if needed), checking the intersection to determine if crossing is safe, and preparing to choose the direction of further travel (left, right, or straight). Upon concluding the Approach (ie., coming to the meeting point of the current road segment and the intersection) the car switches to the Pass action, which concludes passing through the intersection and causes the intersection to "vanish". After Passing the intersection the car switches to the Drive Freely activity again, continuing it until it meets the next intersection.

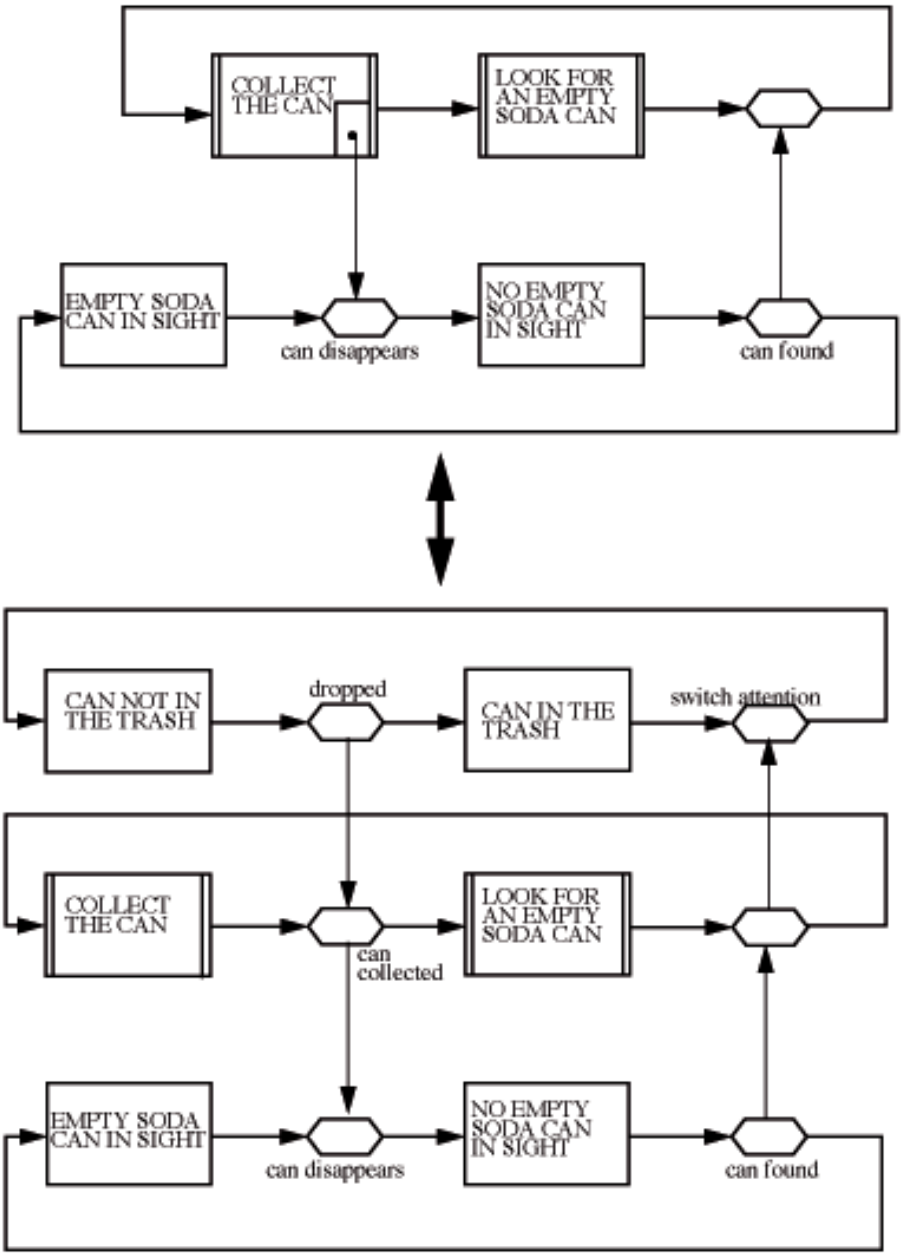


Figure 4: Action as an activity plus additional enabling cycle.

Note that the only conceptual difference between the lower ("intersection") and the upper ("robot activity") cycle is that we ascribe the latter to the behavior of some agent, whereas the former models the agent's environment. Note also that causal dependencies point both ways, so that changes within the environment can cause some behavior change of the agent, and vice versa.

The Approach—Pullup—Stop—Pass—Drive Freely path in Figure 8

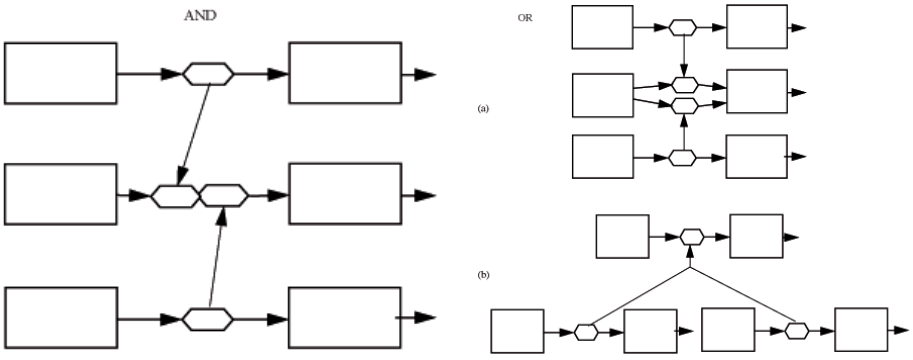


Figure 5: The **and** and **or** connectives.

contains the activities describing the robot's behavior. They correspond to the changing value of some circumstance, say STATE-OF-MYSELF, which, in turn, corresponds to some slot in the state vector containing all the sensory input data.

In Figure 8 the events are not explicitly marked, only the transitions between activities are. Thus, rather than introducing enabling arrows pointing to transition arrows, we have labelled each transition arrow. Unfortunately, all the triggering events belong to this part of the PTN which is not included in the figure (for the full set of PTNs describing the intersection passing scenario, see [Mal91a]). The labels denote triggering events called *configurations*, and correspond to different processes (in the remaining part of the PTN) describing configurations of cars in vicinity of an intersection. These labels should be read as follows:

> A – no car approaches the intersection;
>
> RL (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. In those two-letter labels, the first letter corresponds to the direction from which the car approaches the intersection, and the second letter to the direction it intends to turn. R = right, L = left, S = straight.
>
> R (F, L) – there is at least one car with either an RL, RS, or RR (either FL, FS, or FR; either LL, LS, or LR) property among the cars approaching the intersection from other directions.

It should be stressed that PTNs encourage a hierarchical approach to the task of domain description. One can start with a very rough conceptualization, distinguishing only the major properties of the world and their most important values, then successively refine this first approximation to obtain a sufficiently detailed (and, in the same time, ready to implement) specification of all the important interactions between an agent (or agents) and its (their) environment. Figures 7 and 8 illustrate to some extent this top-down approach, where one box from Figure 7 is split into several ones in Figure 8.

Another advantage of PTNs is that they can be used to graphically present and explain the contents of a knowledge base. Although this factor is not

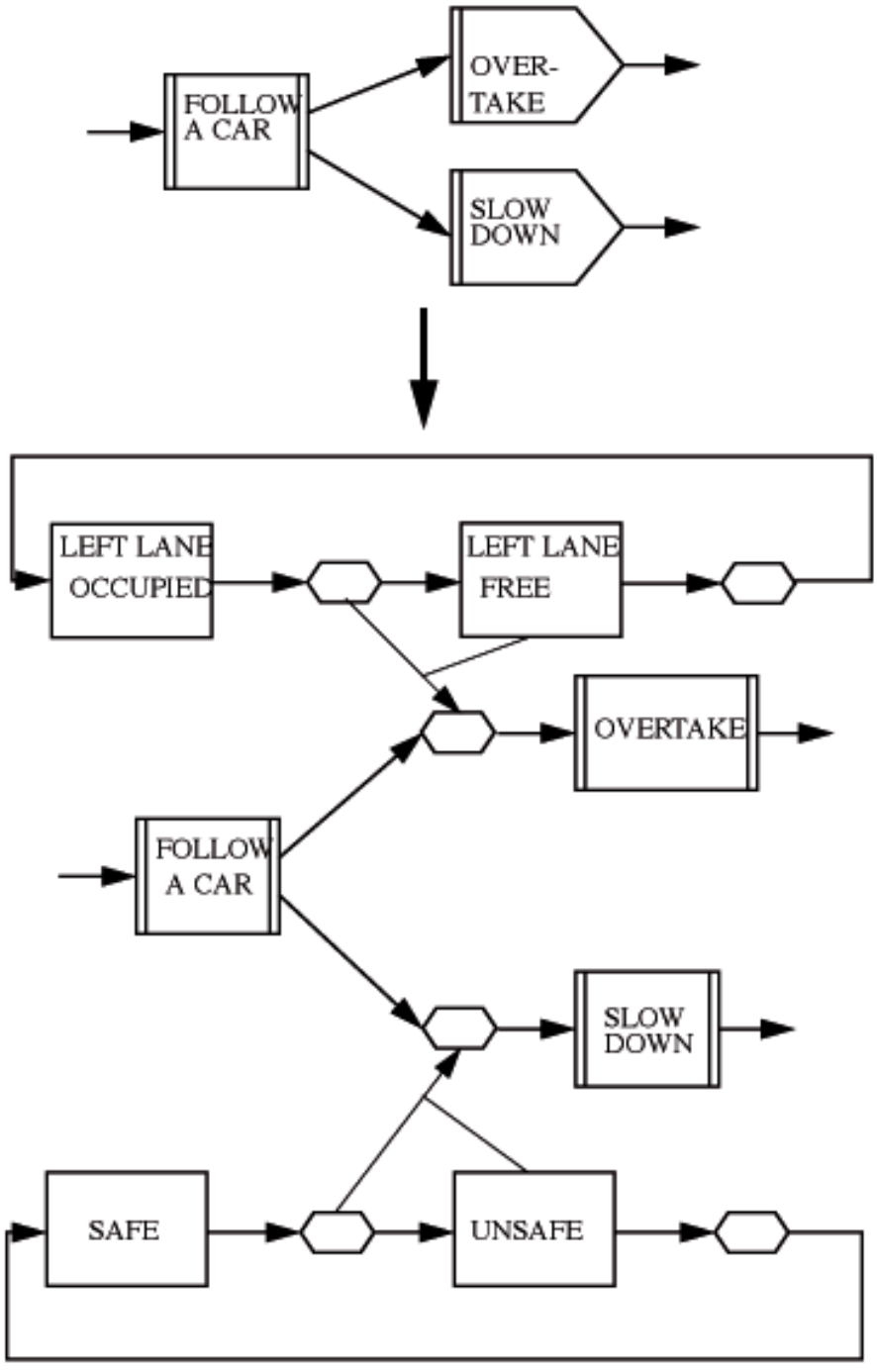


Figure 6: Alternative decisions to be taken by an agent.

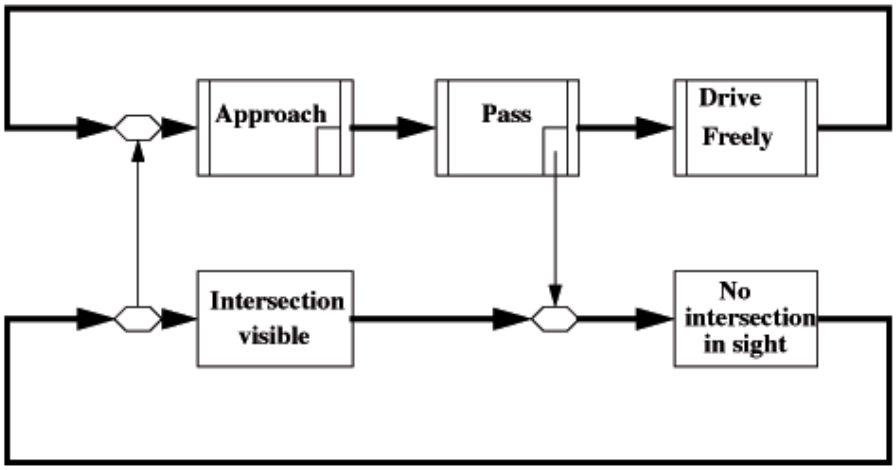


Figure 7: How to pass through an intersection.

significant from a formal point of view, it becomes very important in cases where domain experts, not necessarily being experts in formal specification languages, need to be involved in the process of creating, validating, and expanding a large and complex knowledge base.

3.3 Formal definition of a PTN

A *PTN* is a directed, labeled, finite graph $< N, A, L >$, where N is a finite set of *nodes*, A is a finite set of *arcs*, and L is a *labeling function*. It is assumed that $N = P \cup E$, where P and E are disjoint sets denoting a set of *processes* and a set of *events*, respectively. $P \supseteq A_v \supseteq A_n$, where A_v is a set of *activities* and A_n a set of *actions*.

The *labeling function* $L : N \rightarrow F$ associates with every node a label (which resembles a slot of a frame structure). $L : n \mapsto f \in Attr \times Val$, where $Attr$ is a finite set of *attribute names*, $Val = \bigcup_{a \in Attr} Va(a)$ is a set of all possible attribute values, and Va is a function assigning to every attribute name the set of all possible values for that name. $\forall(a \in Attr)(card(Va(a)) \in \mathrm{N})$, i.e., each set of attribute values should be finite.

$A \subset N \times N$, moreover $A = Ta \cup Ea$ such that $Ta \cap Ea = \emptyset$, where Ta is a set of *transition arrows*: $Ta \subset (P \times E) \cup (E \times P)$, and Ea is a set of *enabling arrows*: $Ea \subset E \times E$.

The set of transition arrows Ta fulfills the following constraints:

- Every event has at least one preceding process and at least one succeeding process:

$$\forall(n \in E)\exists(p \in P)(< n, p > \in Ta) \tag{1}$$

$$\forall(n \in E)\exists(p \in P)(< p, n > \in Ta). \tag{2}$$

- Every event can have at most one successor:

$$\forall(< e, n_1 > \in Ta)(e \in E \Rightarrow \forall(< e, n_2 > \in Ta)(n_1 = n_2)). \tag{3}$$

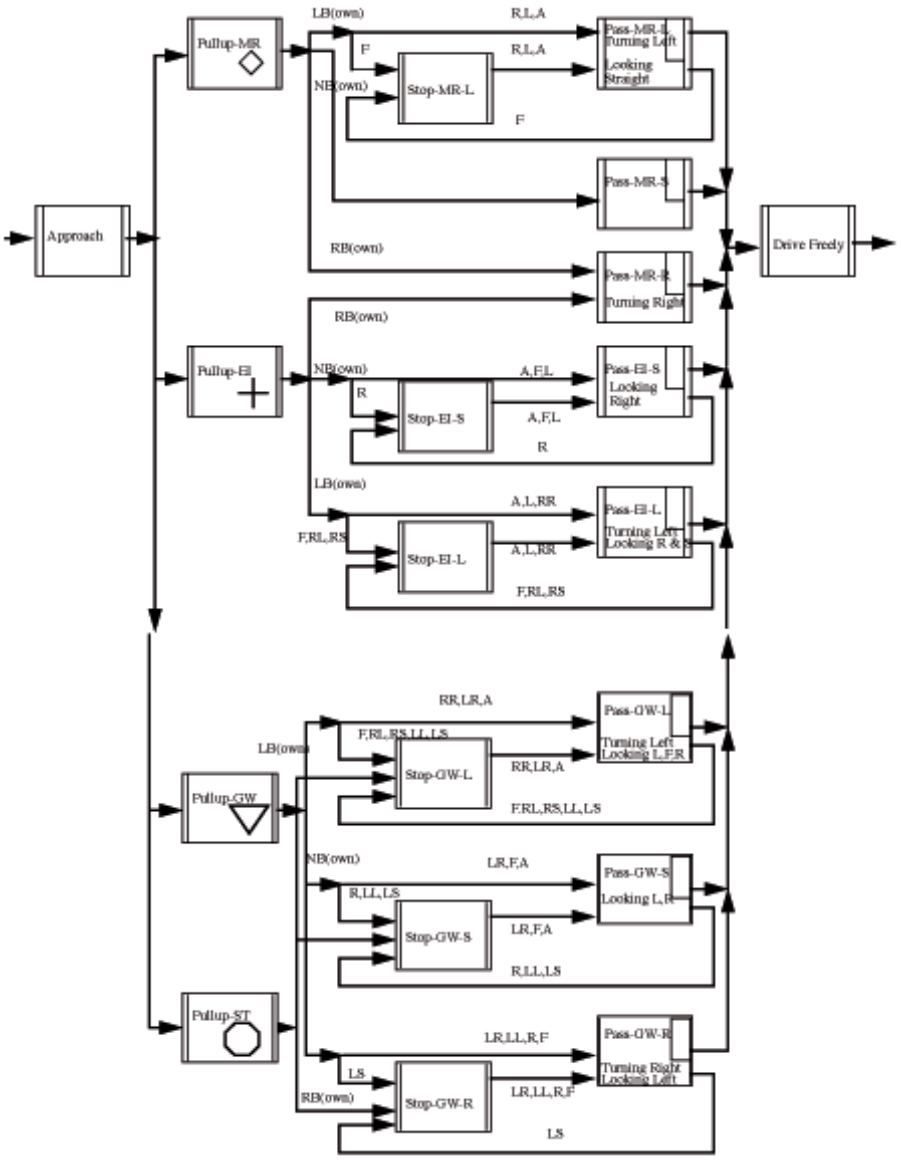


Figure 8: A refined fragment of "How to pass through an intersection".

- Every event can have at most one predecessor:

$$\forall(< n_1, e >\in Ta)(e \in E \Rightarrow \forall(< n_2, e >\in Ta)(n_1 = n_2)). \quad (4)$$

The set Ea of enabling arrows consists of two subsets: $Ea = Epa \cup Eea$, such that $Epa \cap Eea = \emptyset$. Epa denotes the set of *proper* enablings (one event enabling another event), and Eea denotes the set of *extended* enablings (both the event and its succeeding process enable some other event). The distinction will be explained in the next section, where we assign semantics to them.

A *transition* is a pair of transition arrows of the form $<< p_1, e >, < e, p_2 >>$, where $p_1, p_2 \in P$, and $e \in E$. It follows from constraints (1) — (4) that for each pair p_1, p_2, if such an e exists then it is unique. Thus we can treat transitions as pairs from $P \times P$.

Let T denote the set of all transitions in a PTN. Let $t \in T$. We will use $s(t)$ to denote the *source* of the transition t, i.e., $s(t) = s(< p_1, p_2 >) \stackrel{\text{df}}{=} p_1$, and $d(t)$ to denote the *destination* of the transition t, i.e., $d(t) = d(< p_1, p_2 >) \stackrel{\text{df}}{=} p_2$. In other words, $s(t)$ and $d(t)$ are projections of t to the first and second axes, respectively.

$C = \{t_1, t_2, ..., t_n\}$ will be called a *chain* of transitions iff $\forall i \in \{1, ..., (n-1)\}(d(t_i) = s(t_{i+1}))$. A chain will be called *proper* iff $\forall i, j \in \{1, ..., n\}(i \neq j \Rightarrow (s(t_i) \neq s(t_j) \,\&\, d(t_i) \neq d(t_j)))$, i.e., if it properly contains no cycles. Note, however, that a proper chain may itself be a cycle. Let $\mathcal{CP}$ denote the class of all proper chains.

Let us introduce two functions, $first$ and $last$, defined on $\mathcal{CP}$ and yielding the first and the last process of a chain, respectively:

$$p = first(c) \stackrel{\text{df}}{\Leftrightarrow} (c = \{t_1, t_2, ..., t_n\} \Rightarrow p = s(t_1))$$

$$p = last(c) \stackrel{\text{df}}{\Leftrightarrow} (c = \{t_1, t_2, ..., t_n\} \Rightarrow p = d(t_n)).$$

The following constraints must be fulfilled by a PTN:

- Every process is a member of some cycle:

$$\forall (p \in P) \exists (c \in \mathcal{CP})(p = first(c) \,\&\, p = last(c)). \tag{5}$$

- Every transition belongs to some cycle:

$$\forall (p, q \in P) \{\exists (c_1 \in \mathcal{CP})(p = first(c_1) \,\&\, q = last(c_1)) \Rightarrow \\ \Rightarrow \exists (c_2 \in \mathcal{CP})(q = first(c_2) \,\&\, p = last(c_2))\}. \tag{6}$$

Let C_p, called the *cycle set* of p, denote the set of *all* chains fulfilling (5), i.e., the set of all cycles containing p.

- Every cycle consists of at least two transitions:

$$\forall (p \in P) \forall (c \in C_p)(card(c) \geq 2) \tag{7}$$

where the function *card* yields the cardinality of its argument.

In order to write down the next two properties of PTNs, we introduce the notion of *cycle group* of some process p (denoted by $CG(p)$). $CG(p)$ consists of all the processes connected with each other via some proper chain (see Figure 9):

$$CG(p) \stackrel{\text{df}}{=} \{q \in P \mid \exists (c \in \mathcal{CP})((p = first(c) \,\&\, q = last(c)) \vee \\ \vee (q = first(c) \,\&\, p = last(c)))\}. \tag{8}$$

Then we can formulate the following two requirements:

- Every cycle containing an activity consists only of activities:

$$\forall (p \in P)((\exists (q \in CG(p))(q \in Av)) \Rightarrow (\forall (q \in CG(p))(q \in Av))). \tag{9}$$

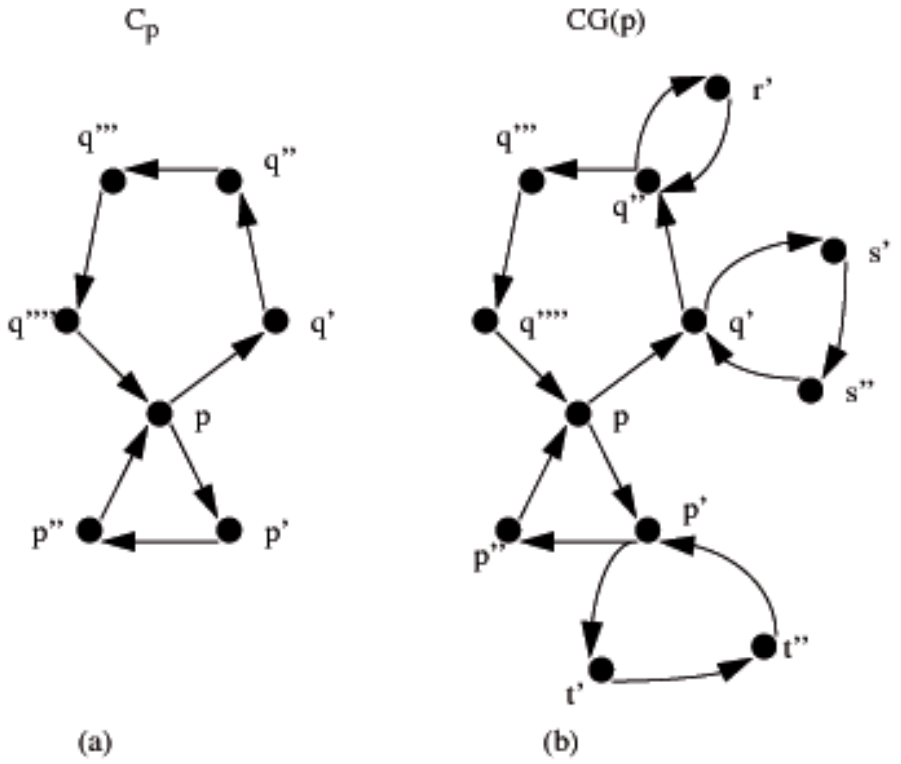


Figure 9: Cycle set (a), and cycle group (b) of a process p.

- Enabling arrows link only different cycle groups:

$$\forall(e \in Ea)\forall(p_1,p_2,p_3,p_4 \in P)$$

$$((e =< e_1,e_2 > \ \& \ < p_1,e_1 >,< e_1,p_2 >,< p_3,e_2 >,< e_2,p_4 > \in Ta) \Rightarrow$$

$$\Rightarrow (CG(p_1) = CG(p_2)\,\&\,CG(p_3) = CG(p_4)\,\&\,CG(p_1) \cap CG(p_3) = \emptyset)). \quad (10)$$

Finally, we put three constraints on the labeling function.

- Processes belonging to one cycle group are assigned the same attribute name:

$$\forall(p \in P)\forall(q \in CG(p)) \\ ((L(p) =< a_1,v_1 > \ \& L(q) =< a_2,v_2 >) \Rightarrow a_1 = a_2)\,. \quad (11)$$

- All value labels within one cycle group belong to the same value set:

$$\forall(p \in P)\forall(q \in CG(p))\,(L(q) =< a,v > \Rightarrow v \in Va(a))\,. \quad (12)$$

- Value labels of two different processes within one cycle group are different:

$$\forall(p \in P)\forall(q_1,q_2 \in CG(p)) \\ ((L(q_1) =< a,v_1 > \ \& L(q_2) =< a,v_2 > \ \& q_1 \neq q_2) \Rightarrow v_1 \neq v_2)\,. \quad (13)$$

This concludes the definition.

In [Mal92] we have constructed a declarative semantics for PTNs, based on classical notions of *domain* and *interpretation*. This allows us to treat PTNs as a fully defined language enjoying all the advantages of logic formalization. One of them is the ability to mechanically translate a PTN into RETFL rules, which will be shown in section 4.

3.4 PTNs vs. Petri nets

To show that any PTN can be reformulated as a Petri net (PN) [Pet81], we provide below a translation procedure yielding for any PTN a PN having the same behavior (meaning) as the one defined by the original PTN. The procedure consists of three steps:

1. **Simplification**

   According to the description provided in sections 3.1 and 3.2, more complex PTNs (using the full set of language constructs) can be reduced to ones using the primitives mentioned in section 3.1, in the following manner:

   (a) an **action** becomes an activity followed by an event enabled by an external cycle expressing the state of termination conditions of the action;

   (b) a **decision** is expanded to a set of events, each leading to an appropriate part of the activity cycle. The events are enabled from external cycles representing state of mind of the agent;

   (c) an event enabled by an **or-enabling** is split into separate events enabled by the disjuncts;

   (d) a series of **and-event**s become one event expressing the whole conjunction.

   After this step the PTN consists only of **processes** (and also activities as their special cases) and **events** as basic elements, and **transition** and **enabling arrows** as links between these elements.

2. **Translation to an extended PN**

   (a) processes are mapped 1–1 into PN conditions (places, states);

   (b) events are mapped 1–1 into PN events (transitions);

   (c) transition arrows are mapped 1–1 into arcs between corresponding conditions and events;

   For example, the simple PTN shown in Figure 10, would be translated into the PN shown in Figure 11.

   (d) proper enablings are mapped into two ordinary arcs (between the condition preceding the enabling event and the enabled event, and between the enabled event and the condition preceded by the enabling event) plus one inhibiting arc (pointing from the condition preceding the enabled event to the enabling event). This mapping is illustrated in Figures 12, and 13. The reason for introducing those

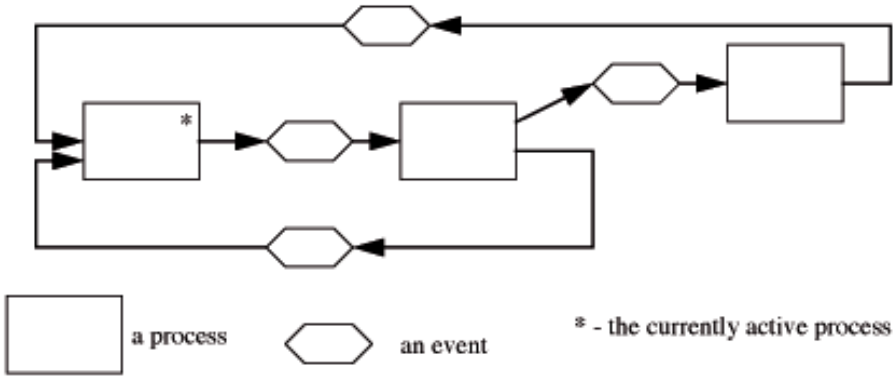


Figure 10: A very simple PTN (with no enabling arrows).

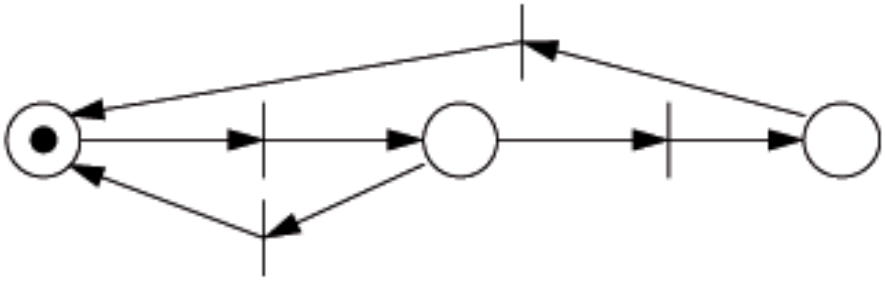


Figure 11: The Petri Net corresponding to Figure 10.

additional prerequisites (arcs) to the enabled event is to prevent it from occurring when the enabling event does not occur. At the same time, the inhibiting arc allows the enabling event to occur when the enabling condition is true but the enabled one is false (so that the event being enabled cannot occur anyway).

(e) extended enablings may be viewed as consisting of two parts: a proper enabling (from an event) and an additional enabling from the subsequent process. The proper enabling is translated in the manner described above. The additional part of the enabling is translated into a new event plus four arcs which are arranged as follows:

1. One pointing from the condition preceding the enabled event to the new event;
2. One pointing from the new event to the condition preceded by the enabling event;
3. One pointing from the condition preceded by the enabling event to the new event;
4. One pointing from the new event to the condition preceded by the enabled event.

This construction is illustrated in Figures 14, and 15.

3. **Translation to a classical PN**

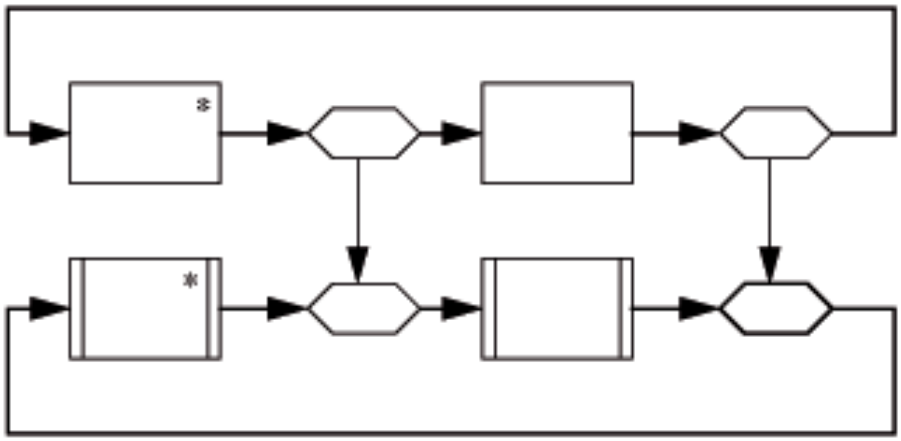


Figure 12: A simple PTN with proper enabling arrows.

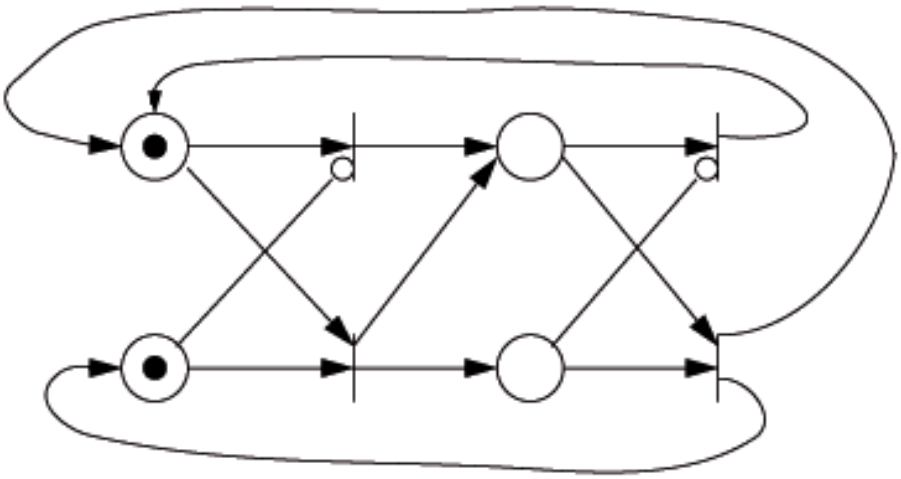


Figure 13: The Petri Net with inhibitor arcs corresponding to Figure 12.

> This step consists of translating inhibiting arcs into new conditions with appropriately chosen incoming and outgoing arcs, as described in [Pet81]. The idea of this transformation is to introduce a condition complementary to the inhibiting condition. This new condition can be used in all the situations where the inhibition from the original condition is required. The results of this transformation (as applied to the PNs shown on Figures 13 and 15) is illustrated in Figures 16 and 17.

Although the version with the inhibitor arcs is much simpler, the version without them demonstrates that the classical PN formalism may be used. (It is known that the inhibitor arcs are a substantial extension of the PN formalism [Pet81].)

This completes the translation algorithm. It can be shown that a PTN and the PN given by the procedure described above are behaviorally equivalent (in

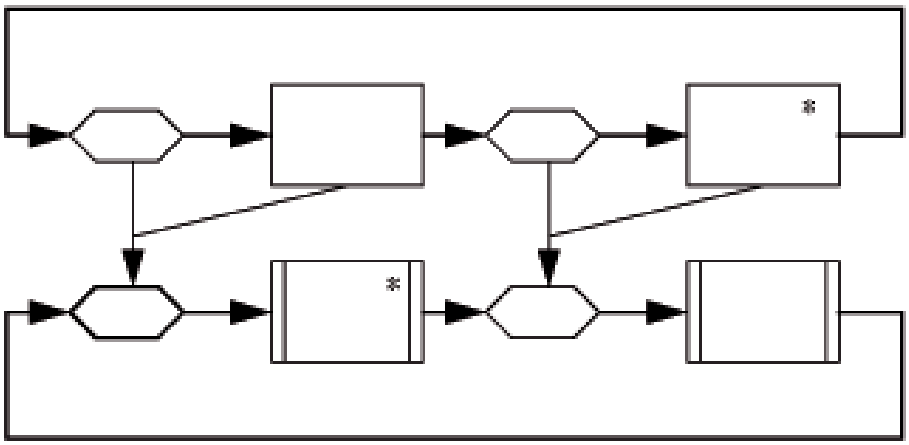


Figure 14: A simple PTN with extended enabling arrows.

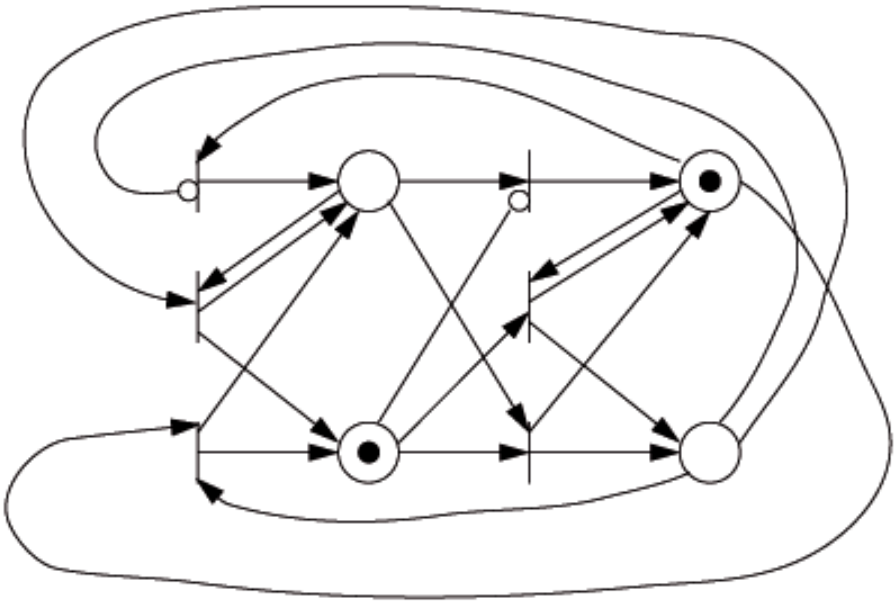


Figure 15: The Petri Net with inhibitor arcs corresponding to Figure 14.

the sense that they describe the same dynamics of the underlying system).

4 RETFL

Next we introduce a logic which is a modification of Elementary Feature Logic (EFL), originally defined in [San94]. The modification consists of both an extension and a restriction. EFL is augmented with a simple temporal operator able to distinguish "now" from "just before now". The restriction removes most of the classical connectives, thus constraining the shape of formulae. The result-

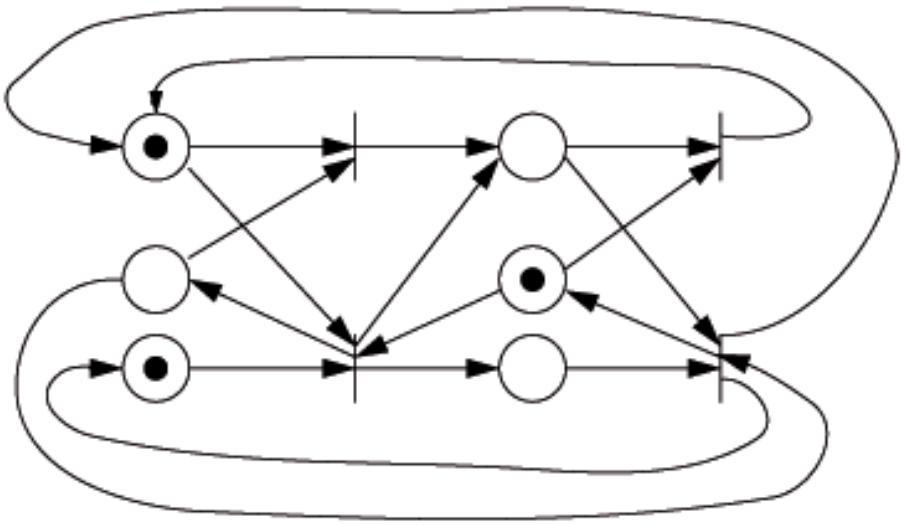


Figure 16: The Petri Net without inhibitor arcs corresponding to Figure 12.

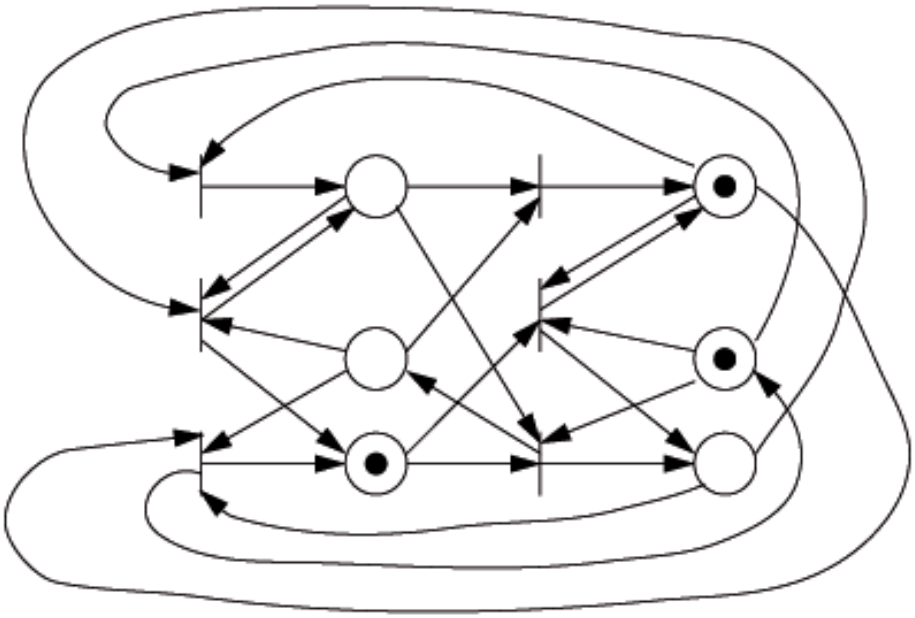


Figure 17: The Petri Net without inhibitor arcs corresponding to Figure 14.

ing logic, Restricted Elementary Temporal Feature Logic (RETFL), appears to be complex enough to express any possible PTN, yet also simple enough to be treated as a programming language. We have implemented an interpreter for this language (see section 4.3).

4.1 Restricted Elementary Temporal Feature Logic

4.1.1 Ontology

The ontology underlying RETFL is the same as that of the PTN language. Since it has been described in section 3.1 we will only briefly summarize it here.

The world consists of objects having certain properties, and involved in different relationships with other objects. Both the properties of an object and the relations it is involved in may change with time. Therefore a description of the world at a particular time point consists of data structures describing the objects, their properties, and their relations with other objects. Because contents of these data structures may change with time, they can be treated as *fluents*, i.e., functions mapping time into data structures. In a general case, values of different fields of such a data structure may belong to arbitrary value spaces. However, many of the fields characterizing an object will be numerical, and moreover, their value space will usually be dense (e.g., rational or real numbers, or vectors). In such cases we may distinguish certain time intervals during which value of some field will change in a continuous manner, and *breakpoints* between these intervals when discontinuities occur. In the next step we introduce *circumstances*: functions which are constant on those intervals during which the corresponding field value changes in continuous manner. Circumstances may change values only at breakpoints, so they are piecewise constant. Obviously, PTN processes correspond to circumstances, as do the *features* introduced in EFL. The value space of a feature corresponds to the set of possible values of the circumstance. Value domains are assumed to be finite sets of objects.

4.1.2 Syntax

A *similarity type* for RETFL is a mapping:

$$\sigma = \{f_1 : \mathcal{D}_1, f_2 : \mathcal{D}_2, ..., f_n : \mathcal{D}_n\}$$

where $n \geq 1$, each $f_1, f_2, ..., f_n$ is a *feature symbol*, and each of $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_n$ is a corresponding non-empty, finite *feature domain* consisting of *items*. Items will be written here in lower case; feature names will be capitalized. $\mathcal{X}$ will usually denote a set of feature items, i.e., $\mathcal{X} \subseteq \mathcal{D}_i$ for some $\mathcal{D}_i$.

An *elementary formula* in RETFL with similarity type σ is an expression either of the form

$$f \hat{=} \mathcal{X}$$

or of the form

$$f \bullet\!= \mathcal{X}$$

to be read "value of f belongs to $\mathcal{X}$", or "value of f becomes a member of $\mathcal{X}$", respectively, where f is a feature symbol defined in σ and $\mathcal{X}$ is a subset of the domain assigned to f by σ.

An *and-type formula* in RETFL with similarity type σ (or *and-formula* for short) is either:

- an elementary formula of the form $f \hat{=} \mathcal{X}$, or
- a formula composed from two and-formulae using the logical connective $\wedge$ (in the standard way), or

- a formula built from another and-formula by preceding it with unary connective λ, i.e., if F is an and-formula, then $\lambda(F)$ is also an and-formula.

An *arrow-free formula* in RETFL with similarity type σ (or *af-formula* for short) is either:

- an elementary formula $f \bullet\!\!= \mathcal{X}$, or
- an and-formula, or
- a formula composed from af-formulae using the logical connective $\wedge$ (in the standard way).

An *or-type constituent* in RETFL with similarity type σ (or *or-constituent* for short) is either:

- an elementary formula of the form $f \hat{=} \mathcal{X}$, or
- a formula composed from two or-constituents using the logical connective $\vee$ (in the standard way).

A *well-formed formula* in RETFL with similarity type σ (*logic formula*, or simply *formula*) is defined as follows:

1. An arrow-free formula is a logic formula;
2. If F is an af-formula, and G is an elementary formula, then $F \Rightarrow G$ is a logic formula;
3. If F is an elementary formula of the form $f \bullet\!\!= \mathcal{X}$ and G is an or-constituent, then $F \Rightarrow \lambda(G)$ is a logic formula;
4. Only the formulae constructed according with (1)—(3) are logic formulae.

4.1.3 Semantics

Let $\sigma = \{f_1 : \mathcal{D}_1, f_2 : \mathcal{D}_2, ..., f_n : \mathcal{D}_n\}$ be a similarity type for RETFL. Let T be a totally ordered set (of time tokens) and let $\leq$ denote the ordering relation. An *interpretation* for this similarity type is an assignment

$$I = \{f_1 : d_1^T, f_2 : d_2^T, ..., f_n : d_n^T\}$$

where each d_i^T is a mapping: $d_i^T : T \rightarrow \mathcal{D}_i$.

Let σ be a similarity type for RETFL, let α be a logic formula, and let I be an interpretation for σ. The value of α in I, written $val\,[\alpha, t, I]$, is defined as either $\mathcal{T}$ or $\mathcal{F}$ in the usual recursive manner:

- $val\,[f \hat{=} \mathcal{X}, t, I] = \mathcal{T}$, provided that $[I(f)]\,(t) \in \mathcal{X}$,
- $val\,[f \hat{=} \mathcal{X}, t, I] = \mathcal{F}$, provided that $[I(f)]\,(t) \notin \mathcal{X}$,
- $val\,[f \bullet\!\!= \mathcal{X}, t, I] = \mathcal{T}$, provided that $[I(f)]\,(t) \in \mathcal{X}$, and $[prevI(f)]\,(t) \notin \mathcal{X}$,
- $val\,[f \bullet\!\!= \mathcal{X}, t, I] = \mathcal{F}$, provided that $[I(f)]\,(t) \notin \mathcal{X}$ or $[prevI(f)]\,(t) \in \mathcal{X}$,
- $val\,[\lambda\alpha, t, I] = prevval\,[\alpha, t, I]$,

- $val\,[\alpha \wedge \beta, t, I] = min(val\,[\alpha, t, I]\,, val\,[\beta, t, I])$,
- $val\,[\alpha \vee \beta, t, I] = max(val\,[\alpha, t, I]\,, val\,[\beta, t, I])$,
- $val\,[\alpha \Rightarrow \beta, t, I] = max(neg(val\,[\alpha, t, I]), val\,[\beta, t, I])$.

Here $neg(\mathcal{T}) = \mathcal{F}$, $neg(\mathcal{F}) = \mathcal{T}$, max and min are defined with respect to the ordering $\prec$, where $\mathcal{F} \prec \mathcal{T}$, and $prevval\,[\alpha, t, I]$ and $[prevI(f)]\,(t)$ are defined as follows: let $T' = \{t' \in T | t' < t\}$, and let τ be its least upper bound. Then

$$prevval\,[\alpha, t, I] \stackrel{\text{df}}{=} \begin{cases} val\,[\alpha, \tau, I] & \text{if } \tau \in T' \\ \lim\limits_{t' \to \tau} val\,[\alpha, t', I] & \text{otherwise} \end{cases}$$

$$prevI(f)(t) \stackrel{\text{df}}{=} \begin{cases} [I(f)]\,(\tau) & \text{if } \tau \in T' \\ \lim\limits_{t' \to \tau} [I(f)]\,(t') & \text{otherwise} \end{cases}$$

The *entailment* relation is defined in the standard way.

4.2 PTNs as RETFL formulae

PTNs are transformed to RETFL formulae in two stages. The first is *simplification* defined in section 3.4. The second stage consists of application of the actual translation algorithm, which takes as input a simplified PTN and yields a set of RETFL formulae. This algorithm consists of 5 steps:

Step 1. Distinguish all the process cycles in the given PTN $< N, A, L >$, and assign to each of them a feature symbol. In order to maintain the correspondence between denotation of processes in the PTN, as defined by the labeling function L, the feature symbol assigned to a process cycle is further assumed to be equal to the first component of the value of the labeling function for an arbitrary process of this cycle (it must be the same for every process in a process cycle). So, if p is an arbitrary process belonging to a given cycle group and $L(p) =< attr, val >$, then the feature symbol f is equal to $attr$.

Step 2. For each feature symbol, define its domain by creating the set of all the second components of the labeling function applied to every process in the corresponding cycle group. Let p be an arbitrary process and let f be the feature symbol assigned to the cycle group $CG(p)$. Then if $L(p) =< attr, val >$ then $f = attr$ and $\mathcal{D}_f = \{v \mid \exists(q \in CG(p))(L(q) =< attr, v >)\}$.

Step 3. For each process $p \in P$ create the set of incoming transitions

$$TR = \{tr \mid tr =< q, p > \ \& \exists(e \in E)(< q, e > \in Ta \,\&\, < e, p > \in Ta)\}.$$

For p, and for each q_i such that $< q_i, p > \in TR$, take its label $L(p) =< attr, val >$ and $L(q_i) =< attr, val_i >$, respectively, and find their common feature name $f = attr$ and their feature values $\mathcal{Y} = val$ and $\mathcal{X}_i = \{val_i\}$, respectively. Now the RETFL formula describing all possible transitions incoming to the process p can be written as:

$$f \bullet\!\!= \mathcal{Y} \Rightarrow \lambda(f \doteq \mathcal{X}_1 \vee f \doteq \mathcal{X}_2 \vee ... \vee f \doteq \mathcal{X}_n).$$

Step 4. For each proper enabling arrow $pe \in Epa$ find its "source" and "destination" transitions, i.e., transitions $<< p_1, e_1 >, < e_1, p_2 >>$ and $<< p_3, e_2 >, < e_2, p_4 >>$ such that $pe =< e_1, e_2 >$. For each $p_i, i = 1, ..., 4$, take its label $L(p_i) =< f_i, val_i >$ and find its corresponding feature name f_i ($f_1 = f_2$, $f_3 = f_4$) and feature value $\mathcal{X}_i = \{val_i\}$. Now the RETFL formula corresponding to this enabling can be written as:

$$\lambda(f_1 \hat{=} \mathcal{X}_1) \wedge f_1 \bullet\!= \mathcal{X}_2 \wedge \lambda(f_3 \hat{=} \mathcal{X}_3) \Rightarrow f_3 \bullet\!= \mathcal{X}_4.$$

Step 5. For each extended enabling arrow $ee \in Epa$ find its "source" and "destination" transitions, i.e., transitions $<< p_1, e_1 >, < e_1, p_2 >>$ and $<< p_3, e_2 >, < e_2, p_4 >>$ such that $ee =< e_1, e_2 >$. For each $p_i, i = 1, ..., 4$, take its label $L(p_i) =< f_i, val_i >$ and find its corresponding feature name f_i ($f_1 = f_2$, $f_3 = f_4$) and feature value $\mathcal{X}_i = \{val_i\}$. Now the RETFL formula corresponding to this enabling can be written as:

$$f_1 \hat{=} \mathcal{X}_2 \wedge \lambda(f_3 \hat{=} \mathcal{X}_3) \Rightarrow f_3 \bullet\!= \mathcal{X}_4.$$

The collection of RETFL formulae created in this way has the same meaning as the original PTN.

4.3 Implementation

The RETFL language defined above is a tool used for expressing a set of rules governing the behavior of one of the rule layer subsystems, namely the *maneuver selector* (which selects the appropriate control algorithm to be applied in the process layer). The implementation of this subsystem, called Discrete Fluent Management System, or DFMS, is being loaded with a set of RETFL rules: a "program". Then for any change of input circumstance this program computes appropriate actions (or changes of output circumstances) to be effected by the process layer.

The intended use of those tools (PTNs, RETFL, and DFMS) is as follows: the user (the system designer) conceptualizes the domain employing PTNs. He can easily consult domain experts, as PTNs are relatively comprehensible. Then PTN is translated into RETFL program, which in turn is loaded into the DFMS shell. DFMS programmed in this way becomes a direct implementation of the control algorithm expressed by the original PTN.

5 Configuration description

5.1 The language

One of the very specific problems to be solved during the design of an autonomous vehicle was choosing an appropriate way of describing a configuration of cars at an intersection. Such a description is stored as a value of some slot, say CONFIGURATION, of an intersection frame. This value should inform about the current configuration of cars coming from each direction and intending to make some specific maneuver at the intersection. In order to be able to describe an arbitrary configuration of cars approaching an intersection, we have defined a special regular language which we call *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 in the classical recursive way:

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 denoting "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 denoting "a_1 and then a_2");
5. If $a \in \mathcal{CL}$, then also $(a)^* \in \mathcal{CL}$ (called *iteration* and denoting "concatenation of arbitrary number of a-s, including 0");
6. Only the strings constructed according to $1 - 5$ belong to $\mathcal{CL}$.

We will omit parentheses when that will not lead to ambiguity.

In addition, we introduce the standard abbreviation $a^+ = a(a)^*$, plus several 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 the syntax of $\mathcal{CL}$, let us exemplify the intended meaning of the language. First, some obvious configurations:

- λ — the roads coming to the intersection are empty;
- l — a car is coming from the left side;
- l_l — a car is coming from the left side and intends to turn to its left;
- $l_s, l_r, f_l, f_s, f_r, r_l, r_s, r_r, r, f$ — similarly.

Several slightly more difficult expressions:

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

One can even state interesting properties of some particular car. For example, "The second car from the left side intends to turn right" would be written as: $(r + f)^* l (r + f)^* l_r q^*$.

Saying something about both the second car from the left side *and* the second car from the right side becomes a bit complicated. "The second car from the left side intends to turn right, and the second car from the right side intends to turn right" would have to be written as follows: $f^* r f^* r_r f^* l f^* l_r q^* +$

$f^*rf^*lf^*r_rf^*l_rq^* + f^*rf^*lf^*l_rf^*r_rq^* + f^*lf^*rf^*r_rf^*l_rq^* + f^*lf^*rf^*l_rf^*r_rq^* + f^*lf^*l_rf^*rf^*r_rq^*$. However, in this particular application we usually state the properties of *all* the cars coming from some direction, so the problems with saying something specific about "the n-th car in a row" need not bother us.

We have designed and implemented (and built into DFMS) an algorithm that constructs a *nondeterministic finite state automaton* accepting a given regular language. It allows the rule interpreter to check, for any given configuration, whether it belongs to the set of "acceptable" ones (in terms of safeness, or-the rights of way), or not. The next section provides some details about this implementation.

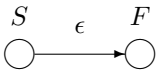
5.2 Implementation

Below we will present an algorithm for constructing a *nondeterministic finite state automaton* accepting a given regular language. An alternative approach could be based on *regular expression derivatives*, but since the languages we deal with are very simple, the straightforward algorithm presented below suffices for our purposes.

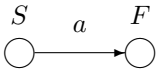
The algorithm consists of the following three steps:

1. For the language appearing as the right-hand-side argument of the $\hat{=}$ (or $\bullet=$) relation, build a nondeterministic automaton $\mathcal{A}$ with ϵ-moves accepting this language. The automaton is constructed recursively, according to the following steps appropriately many times:

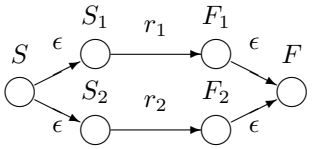
   (a) for the empty expression λ the automaton consists of two states, the initial S, and the final F, linked with a transition labelled with the empty letter ϵ:

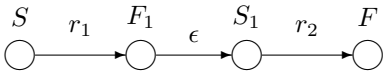
   (b) for the language consisting of one letter of the alphabet, say a, the automaton consists of two states, the initial S, and the final F, linked with a transition labelled with the letter a:

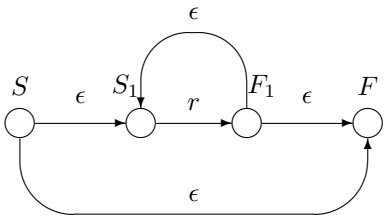
   (c) for the language consisting of the sum of two regular expressions, say $r_1 + r_2$, the automaton is built out of two subautomata, each recognizing one of them:

   (d) for the language consisting of the concatenation of two regular expressions, say r_1r_2, the automaton is built out of two automata, each recognizing one of them:



(e) for the language consisting of the iteration of a regular expression, say r^*, the automaton is created from the automaton recognizing r, together with appropriate transitions among surrounding states:



2. Take the left-hand-side argument of the relation $\hat{=}$ (or $\bullet=$) and feed it to the input of the automaton $\mathcal{A}$.

3. If the automaton reaches the final state F, then the test succeeds (i.e. the relation is true), otherwise it fails.

This approach allows us to incrementally parse a description of the current configuration: if some additional information about the configuration becomes known (some letter is appended to the end of the appropriate expression) then we can proceed with parsing it, instead of re-parsing the first part of the description.

6 Conclusions

In this paper we have presented several KR languages used within one particular system, namely the autonomous car control system, being designed within the PROMETHEUS project. We have shown relationships linking these languages. In our opinion, this example shows that KR formalisms used by a complex, heterogeneous AI system need not be the ones traditionally associated with AI — there are many other valuable tools. In particular, we have presented one language based on graph theory (PTNs) and one language being just a regular language ($\mathcal{CL}$), interpretation of which is done by classical accepting automata. We expect that this example may suggest possibility of usage of various kinds of such representational tools.

Process Transition Networks are a pictorial language suitable for the conceptualization and coding of knowledge about complex, often hard to describe quantitatively environments. The word "coding" is justified by the fact that sentences in this language (i.e. nets) can be translated into a restricted temporal logic language RETFL, which may then be treated as a programming language, because it has been provided with a working interpreter.

PTNs are closely related to several well-known formalisms which rely on graphical representation. One of the best known formalisms is Petri nets [Pet81]. PTNs are a proper subset of Petri nets, but they have been designed in a way that permits causal (and thus temporal) dependencies to be expressed using single symbol, something which is impossible to achieve using Petri nets. On

the other hand, the properties of Petri nets have been well studied, so we hope to get more insight into PTNs after having established formal correspondence between the semantics of both languages.

Another well-designed graphical formalism has been developed by Harel [Har87]. His *statecharts* have been designed for similar purpose as PTNs. Harel also provides a formal semantics for statecharts [HPSS87]. However, his approach seems to have one major drawback as compared with ours (at least within our application area). Statecharts are well suited to describe entities that fit the paradigm of automata theory. This implies that one has to distinguish an input alphabet (set of events), a set of states of the described entity, and a transition function mapping the current state and some input value to the new state (and possibly to the output). In our case we want to use dependencies both of the form *input* $\rightarrow$ *state*, and *state* $\rightarrow$ *input*, where *state* corresponds to some state of an agent and *input* is a description of its environment. In the automata theory paradigm, the input alphabet, the output alphabet, and the set of states are unrelated. Of course, there are close links between both formalisms, so it might be interesting to look at their relationship more carefully.

The advantages of the PTN language are simplicity, the ability to express causal dependencies, the ability to describe behavior of several agents acting in the same environment, modularity, and the possibility of using it for subsequent refinements of an initial conceptualization. These properties have been confirmed during the conceptualization of several non-trivial traffic-behavior scenarios.

The obvious disadvantage of the PTNs is the fact that we have introduced yet another language for expressing knowledge about the real world, moreover a very restricted one. However, none of the existing formalisms is suitable for our purpose, that is modeling complex dynamic environments, either because of too much generality, or because of lack of expressive power. In our opinion the comprehensibility of PTN language, together with its ease of use, justify its introduction.

The Restricted Elementary Temporal Feature Logic is an example of a KR language based on formal logic. It is, however, tailored to be easily computable (up to implementation level). RETFL is the counterpart for PTNs in the sense that a specification expressed with a PTN can easily be translated into a program in RETFL. One of the remaining questions is whether the reverse procedure is possible (i.e. whether each reasonably constrained set of RETFL formulae can be translated into a PTN).

The last of the presented languages, Configuration Language, albeit simply a regular language, plays a crucial role in representing knowledge about the environment our robot is to act in. Automata theory provides algorithms for any imaginable kind of computations that may be required from such a representation.

Finally, referring to two of the questions addressed by this volume:

1. Which AI techniques have survived the experimental verification?
2. Which KR schemes are of particular importance for the engineering research society?

we claim that the proper answer is: *computable*, and we hope that this paper contributes to this answer.

Acknowledgments

This research was supported by the IT4 research program.

Brant Cheikes, Patrick Doherty, Feliks Kluźniak, Witold Łukaszewicz, Hua Shu, and Jacek Wrzos-Kamiński helped much to improve various parts of this text. Erik Sandewall has been the source of continuous inspiration for most of the ideas presented in this paper. Anne Eskilsson helped to prepare one of the figures.

References

[Bro90] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[Gen90] Michael R. Genesereth. Knowledge Interchange Format Version 2.0, Reference Manual. Report Logic-90-4, Computer Science Department, Stanford University, 1990.

[GL90] R. V. Guha and Douglas B. Lenat. CYC: A midterm report. *AI Magazine*, 11(3):32–59, 1990.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HNS89] 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.

[HPSS87] David Harel, Amir Pnueli, J. P. Schmidt, and Rivi Scherman. On the formal semantics of statecharts. In *IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, 1987. IEEE Computer Society Press.

[Mal91a] 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.

[Mal91b] 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.

[Mal91c] Jacek Malec. Process Transition Networks: What Are They For? In *Proceedings of the IEEE Systems, Man and Cybernetics Conference, Charlottesville, VA*, pages 1177–1182, October 1991.

[Mal92] Jacek Malec. Process Transition Networks: The Final Report. Research Report LiTH-IDA-R-92-07, Department of Computer Science, Linköping University, 1992. This report summarizes and extends the results presented previously in two conference papers: [Mal91b] and [Mal91c].

[Pet81] James L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, 1981.

[San90] Erik Sandewall. System Block Specifications and their PROMETHEUS applications. In *Prometheus: Proceedings of the Fourth Workshop*, 1990. Available also as LAIC-IDA-90-TR22.

[San91] Erik Sandewall. The proposal for a ProArt Datastructure Exchange Format (PAD). In *Proc. 5th Prometheus Workshop*, pages 136–145, Munich, Germany, October 1991. Available also as LAIC-IDA-91-TR11.

[San94] Erik Sandewall. *Features and Fluents.* Oxford University Press, 1994.