Complex Behavior Specification for Autonomous Systems

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

Abstract

The paper focuses on one of the problems of design of an autonomous reactive system, namely, specification of its behavior with respect to (often unpredictable) changes occurring in the real world. Several "behavior-oriented" languages are briefly presented, and some formal relationships between two of them — Process Transition Networks and Statecharts — are given. The goal of this presentation is to address the need for augmenting the "behavior-based design" paradigm with theoretical tools both adequate for expressing complex behaviors of autonomous reactive systems pursuing high-level goals, and applicable to both analysis and synthesis tasks for such systems.

1 Introduction

The problem of designing an autonomous system capable of acting in the real world has been the subject of much attention in recent years. The research has focused on designing systems with the following attributes:

- *reactivity*, in order to allow the system to cope with unpredictable changes in the dynamic environment while pursuing its mission;

*This research was supported by the Centrum for Industrial Information Technology (CENIIT).

- *robustness*, meaning the ability to function in a variety of situations, including failure of some of its subsystems;
- *selectivity of attention*, in order to effectively use existing resources of the system, such as computing power, or sensory equipment;
- *ability to pursue goals* defined either by the designer, or by the system itself.

A great deal of research on this topic is currently being done within the "behavior-based design" paradigm. This paradigm is the result of work initiated and pursued at the MIT AI Lab by Brooks and his co-workers [Ang89, Bro86, Bro90b, Bro91, Con89, FBIB89, Mat90], and augmented by research conducted in other places [Ark90, Pay90, KR90]. The main idea consists of building a controller for an autonomous (possibly intelligent) system out of a set of primitive reactive behaviors that couple sensory inputs of the system to its actuating outputs. Ultimately a system built this way should reveal the ability to cope effectively with dynamic and unpredictable changes in the real world while performing its task. The results obtained so far, both at MIT and at other places, are very promising and prove that this approach is worth further investigation.

The major open problems of the behavior-based approach, as Brooks states in his recent paper [Bro91], are the following:

- "Understanding the dynamics of how an individual behavior couples with the environment via the robot's sensors and actuators."
- "Understanding how many behaviors can be integrated into a single robot."
- "Understanding how multiple robots can interact as they go about their business."

These questions have been already addressed by the research done within the competing paradigm, which we might call symbolic-AI-based approach to the problem. The existing answers are, however, formulated with a totally different framework in mind, with the notable exception of Rosenschein and Kaelbling's research. In the behavior-based approach, behaviors are the basic building blocks, and the problem of constructing a complex autonomous system amounts to the synthesis problem. In the symbolic-AI-based approach, a final result (i.e., a complex behavior as a whole) is the

starting point, and the design of a system can be seen as the analysis problem. Of course, both approaches use different tools, and, to some extent, even different languages. This raises the question of what common ideas they share, and what could be the starting point towards integration of both approaches.

Assuming that the (set of) system's behavior(s) has been designed, one then has to solve the problem of choosing an appropriate architecture for the implementation. A whole spectrum of approaches is imaginable, as illustrated by the following list:

- an unstructured collection of loosely-coupled behaviors;
- a layered system, where higher layers implement conceptually more complex behaviors, but where all the layers are equally reactive (subsumption architecture [Bro86]);
- a layered system, where the reactive component resides in the bottom layer(s), and the reasoning (symbolic) component resides in the top layer(s) [Fir89, HNS89, MS90, Gat91];
- a classical "sense-think-act" architecture, where computations done on the path from the "sense" part to the "act" component realize behavior revealing the required degree of reactivity, but also appropriate complexity.

Of course, the extremes should be seen as idealizations, unreasonable to be used in their purest form. However, there exist approaches which are very close to these extremes. One can quote here Brooks' [Bro91] on one (collection of behaviors) end, and Georgeff and Lansky's PRS [GL87] on the other (sense-think-act cycle). Both of them have their obvious strong and weak points. The bottom-up direction guarantees that effective realization is always possible — the system is physically grounded from the very beginning. The top-down design, on the other hand, provides systems proven to pursue top-level goals requiring non-trivial amounts of intelligence. However, the question remains open whether systems possessing both these properties can be designed.

It is worth noting that quite a lot of recent publications (see e.g. [FBT89, KR90]) stress very strongly the need to *constrain* the design so that it is realizable by simple, and thus fast, hardware. This leads to the third, to some extent orthogonal to the previous two, major aspect of design: the problem of choosing the best medium of implementation. One can imagine solutions

ranging from general purpose machines with appropriate software implementing the adopted control strategy, as in the case of NAVLAB [Tho87], to collections of specialized processors designed for specific purposes (processing visual data, motor control, sensor fusion), to purely hardware realizations of control algorithms, as in case of many Brooks' robots.

This paper should be seen as a first attempt to review the state of the art in the area of specification of an autonomous agent's behavior. It also provides some basis, and suggestions for further research on unifying "classical" (pure automata theory) and AI-based approaches to this problem.

2 Overview of behavior languages

The goal of this short presentation is twofold: to sketch the state of research in the area of specifying behaviors for autonomous agents, and to identify the existing constraints within which one could investigate the following problems:

1. the expressive power and limitations of specification tools used so far (behavior language, REX, Gapps, Ruler, PTNs, automata theory)

2. compositional specification of behaviors, so that from a set of specifications of simpler behaviors one could formally derive a specification of their composition;

3. decompositional specification of behaviors, so that specification of a complex behavior would allow one to infer a set of specifications of hardware-realizable primitive behaviors, or at least would yield predictable software fitting the layered architecture;

4. correspondence between specifications of behavior(s) of an autonomous system, and software specification for the three-layered software architecture;

5. theoretical tools for modeling dynamics of a single behavior;

6. theoretical tools for modeling dynamics of a system of coupled behaviors.

In this section we have chosen to briefly present four selected specification methods: the Behavior Language, the situated automata approach, the statecharts, and the Process Transition Networks.

2.1 Behavior language

Brooks has created the Behavior Language: a LISP-based programming language for specifying behavior-based controllers [Bro90a]. Using the constructs of the language one can define a set of behaviors comprising the control system of a robot. Such a set of behaviors is usually intended (by the designer) to make the robot achieve some predefined set of goals. However, those goals are only *implicit* in the specification (thus the name "emergent functionality").

The behavior language is based on the assumption that every behavior is expressible as a set of so-called "real-time" rules. A real-time rule is an expression of the form:

(`whenever` *condition* `&rest` *body-forms*)

or

(`exclusive &rest` *whenever-forms*)

and is executed independently of other rules, in real-time, possibly by an Augmented Finite State Machine (AFSM). An AFSM is a finite state machine equipped with registers and timing elements.

Behaviors are defined either as stand-alone real-time rules or as groups of such rules. A stand-alone rule is named by:

(`defmachine` *name declarations rule*)

and complex behaviors are constructed using the `defbehavior` facility:

(`defbehavior` *name inputs outputs declarations rules*).

Then input and output registers of behaviors can be connected by "wires" according to the designer's intentions:

(`connect` *source dest1* `&rest` *more-dests*)

The core idea of the behavior-based approach is an asynchronous, independent activation of real-time rules and behaviors. The following mechanisms are used for activating behaviors [Bro90a]:

- thresholding,
- hormone system (conditions, releasers),
- spreading of activation.

The behavior language, however, guarantees only that *if* a (set of) behavior(s) can be specified using constructs of the language, *then* it can be automatically compiled down to the level of AFSMs directly realizable in hardware. No support is provided for checking whether such a set of behaviors is contradictory or not (either wrt the sensory input vs. expected motoric output, or wrt the

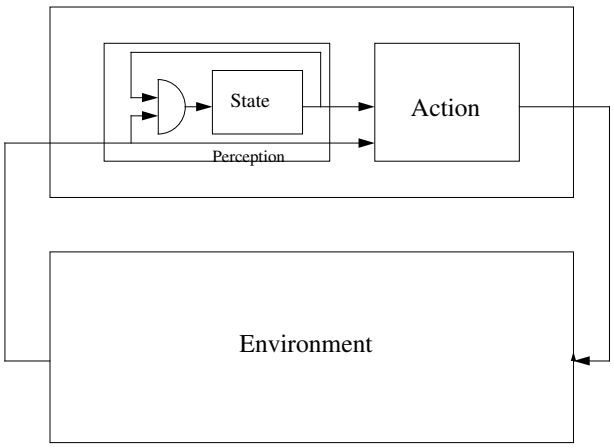

Figure 1: An automaton situated in its environment.

internal dependencies between interacting behaviors), or whether those behaviors can actually help achieve the assumed goals.

2.2 Situated automata

A somewhat more constrained approach to the problem has been proposed by Rosenschein and Kaelbling [Ros89, KR90]. Their work is based on the concept of situated automata (Figure 1, [Ros85]), where both an autonomous agent (or, more generally, a system) and its environment are modeled as finite state automata. This idea allows them to formally state dependencies between the state of the environment and the state of the system. They consider the information about the environment carried by the states of the system, and use it to formally prove the correctness of the system's design.

On the basis of the idea of situated automata Kaelbling and Rosenschein have created a hierarchy of languages: Rex, which provides a link from a set of reactive behavior rules to hardware implementation; Gapps, which is used for defining goal-oriented behaviors (for the action part of the system); and Ruler, which is used for describing the perception component of the system. Rex [RK86] is defined in Lisp and serves for specifying a "low-level description" of a machine. Using an appropriately defined logic which accepts specifications computed by Rex, one can prove properties of the behavior of the designed machine. Gapps [KR90] and Ruler [Ros89] expressions compile down to Rex statements, so all the theoretical machinery can be applied to Gapps and Ruler programs as well. Using these tools, one has the guarantee that

having specified the set of desired system's interactions with the world, *and having completely specified the possible set of situations occurring in the world, as perceived by the sensory input of the system*, the system will actually behave in the desired way, and, moreover, it will be compilable down to the gate level. However, the requirement of complete specification of all possible situations the system might find itself in (or might recognize), together with the limitations imposed on the specification languages in order to make them compilable to hardware, causes this approach to be rather restrictive.

2.3 Statecharts

Another interesting language for specifying behaviors is statechart formalism [Har87] or its more general incarnation, higraphs [Har88]. Statecharts are used to define complex, hierarchical finite state automata. Therefore, they seem to be a very plausible candidate for specifying autonomous agent's behavior. Their advantages are the following:

- well-defined (formal) semantics;
- graphical representation (which means user-friendliness);
- existence of implemented support tools (STATEMATE [H$^+$90]).

Due to their formal grounding in automata theory, statecharts have strong connections to the situated-automata approach of Kaelbling and Rosenschein; however, no tools are provided in the formalism for expressing goals of a system acting in some environment. This should not be seen as a deficiency, because statecharts were originally designed for specification of stand-alone automata, not pairs of coupled automata (i.e., the system and the environment). On the other hand, statecharts are very intuitive as a specification tool: one can easily express interdependencies between various elements of a system, so the conceptualization process is supported much more strongly than in other approaches.

2.4 Process Transition Networks

Yet another graphical language created for the purpose of specifying behaviors of autonomous agents are Process Transition Networks (PTN, [Mal91b]). They have been assigned a formal semantics [Mal91a], and tools for implementing the represented control

strategy using PTNs have been created. PTNs enable the designers to simultaneously specify behaviors of several independent agents *and* the environment. They are compilable (after making some necessary assumptions) to automata that describe behavior of a separate agent [SM91], and they can also be used to explicitly couple the goals of an agent to changes of its behavior. They are somewhat more "flat" than statecharts, since there are fewer provisions for hierarchical abstraction in the PTN language. However, they have the same expressive power as statecharts, what will be the topic of the next section.

3 Process Transition Networks vs. Statecharts

In order to show the equivalence of PTNs and statecharts, we will present an algorithm for translating either one of them into the other. First, however, we have to present the languages themselves in more detail.

Both descriptions presented below contain an informal presentation of language constructs, in order to provide the reader with necessary intuitions, together with some provisions for more formal definition. However, neither of the formal definitions is ultimately quoted here; the interested reader can consult [Mal91a] and [HPSS87], respectively[1].

3.1 Definition of a PTN

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 change with time. In most cases 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.

[1]We hope that the decision to sacrifice formal accuracy for (hopefully) improved readability will not give the impression that the languages are not formal.

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 primitive[2] 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 "source" 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 and pointing either from an event alone or from an event and its subsequent process, to some other event. The first case corresponds to the situation where one discontinuity forces another one to occur; the second case describes the situation of an actual enabling. Somewhat unfortunately, due to historical reasons, both are referred to as enabling arrows.

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 additional internal box denoting its termination condition. Actions are not followed

[2]As used here, *primitive* does not mean simple, but rather "basic", or "not compound" (with respect to some assumed level of analysis). So an operation that is primitive on one analysis level might appear to be a complex operation on some lower level.

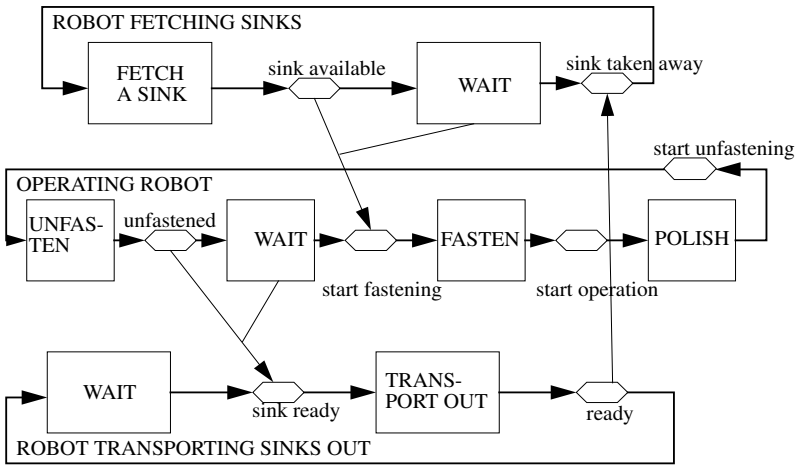

Figure 2: Three cooperating robots.

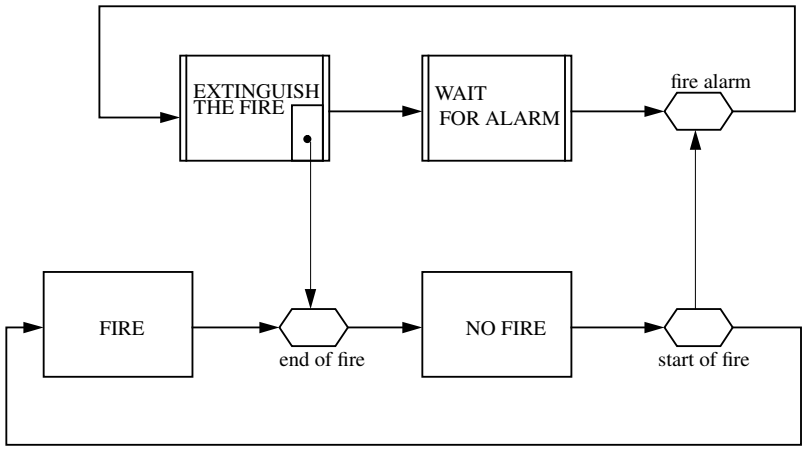

Figure 3: Fire brigade activities.

by events, rather, the event is "encoded" in the termination conditions of the action. However, if it is desired that this event enable 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. In our example, a fire brigade has been treated as single agent. Later on we will assume that every PTN cycle containing an activity consists only of activities. The ra-

tionale 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.

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. 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. 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.

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 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. Thus, by making this condition explicit, a decision box can be written using the primitives defined previously.

Definition. 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* $L : N \to F$ associates with every node a label. $L : n \mapsto f \in Attr \times Val$, 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. Each set of attribute values should be finite.

The set of arcs $A \subset N \times N$, 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 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

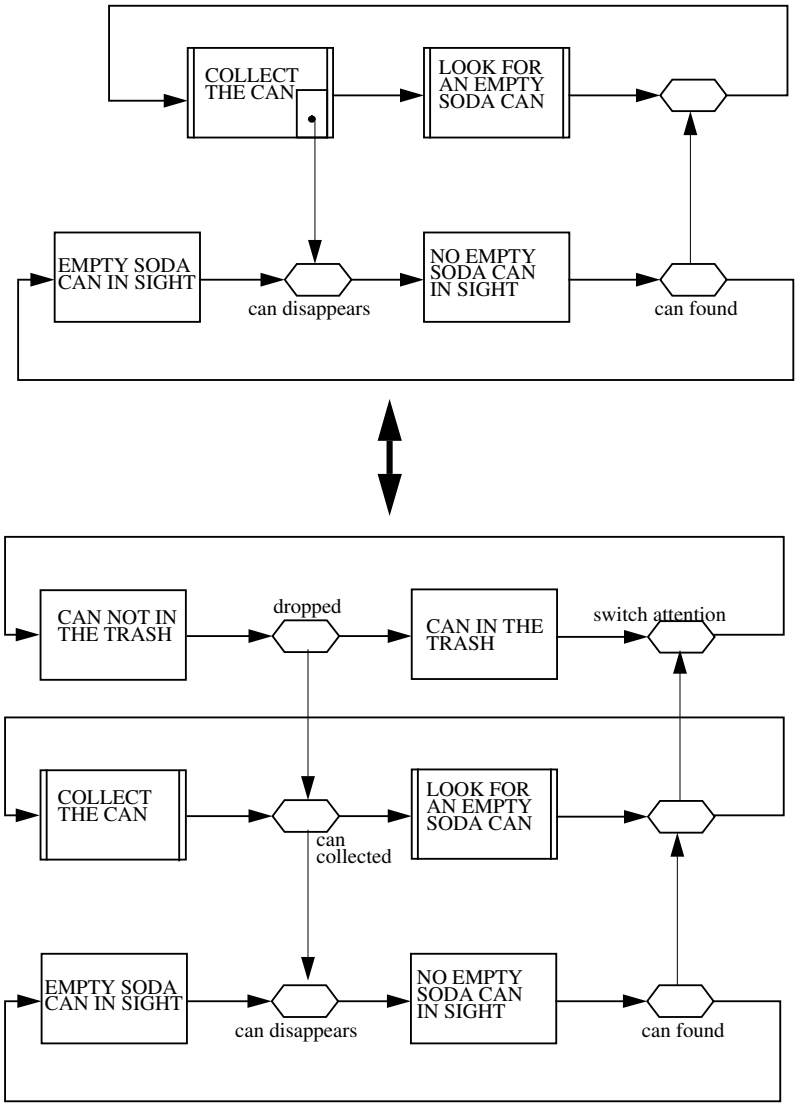

Figure 4: Action as an activity plus additional enabling cycle.

the set of *extended* enablings (both the event and its succeeding process enable some other event).

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 the previous constraint that if such an e exists then it is unique. Thus we can treat transitions as pairs from $P \times P$.

We define then a *cycle* of processes linked by transition, and a *cycle group of a process* consisting of all processes reachable from the given process and all processes from which the given process is reachable. Using those terms we can state all the properties of a

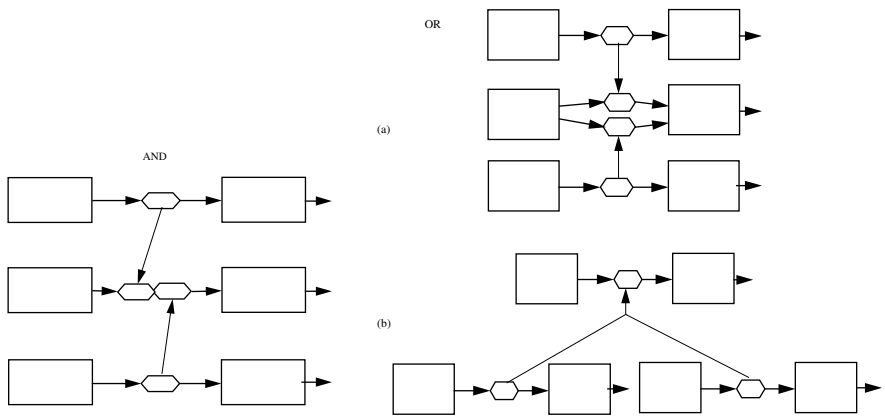

Figure 5: The **and** and **or** connectives.

well-formed PTN. The following set of constraints has been stated formally in [Mal91a]. Due to the space limitations we provide below only the natural-language formulation.

- Every event has exactly one preceding process and exactly one succeeding process.
- Every process is a member of some cycle;
- Every transition belongs to some cycle;
- Every cycle consists of at least two transitions;
- Every cycle containing an activity consists only of activities[3];
- Enabling arrows link only different cycle groups[4];
- Processes belonging to one cycle group are assigned the same attribute name;
- All value labels within one cycle group belong to the same value set;
- Value labels of two different processes within one cycle group are different.

This concludes the definition.

[3]This is to ensure that processes describing the activities of an agent are not intermingled with processes denoting the states of some external property of the environment.

[4]This is a necessary, but not sufficient, provision for ensuring that the described system is non-anticipative.

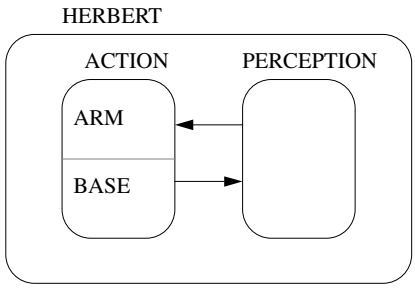

Figure 6: A higraph representing a hypothetical control system for a robot called HERBERT. The ACTION blob is a cartesian product of ARM and BASE.

3.2 Definition of a statechart

We begin the presentation of statecharts by quoting the formal definition of the more general construct of higraph [Har88].
Definition. A *higraph* is a quadruple $H = (B, \sigma, \pi, E)$ where B is a finite set of *blobs* and E is a set of *edges* (a binary relation on blobs). The *subblob* function $\sigma : B \rightarrow 2^B$ assigns to each blob $x \in B$ the set $\sigma(x)$ of its subblobs; it is assumed that this function is cycle free. The function $\pi : B \rightarrow 2^{B \times B}$ introduces an equivalence relation on blobs; these classes of equivalence for a given blob x will be denoted as $\pi_i(x)$. The *atomic blobs*, denoted by A, are those that do not have any subblobs (see Figure 6).

Statecharts are a particular subset of higraphs, namely the ones in which the E relation corresponds to the transition function of an automaton. Some of the more advanced constructs of statecharts, such as e.g. *history-dependent* transitions, require an additional relation on blobs to be defined; however, we will not concentrate here on the formal aspects of the language.

The basic idea distinguishing statecharts from other tools for specifying automata is their ability to easily capture hierarchies and abstractions. These notions are illustrated in Figure 7. Figure 7 (a) presents a simple three-state automaton (with incompletely specified transitions). Subfigure (b) presents a possible hierarchical view of this automaton, where states A and C are treated as a group of states (distinguished by the common transition β to the state B) named D. Note the fact that an arrow from/to a blob corresponds to a set of arrows from/to all its subblobs. Subfigure (c) illustrates possibility of abstraction, where state D is drawn without investigating its internal structure. Subfigure (d) presents another concept introduced in statecharts, namely

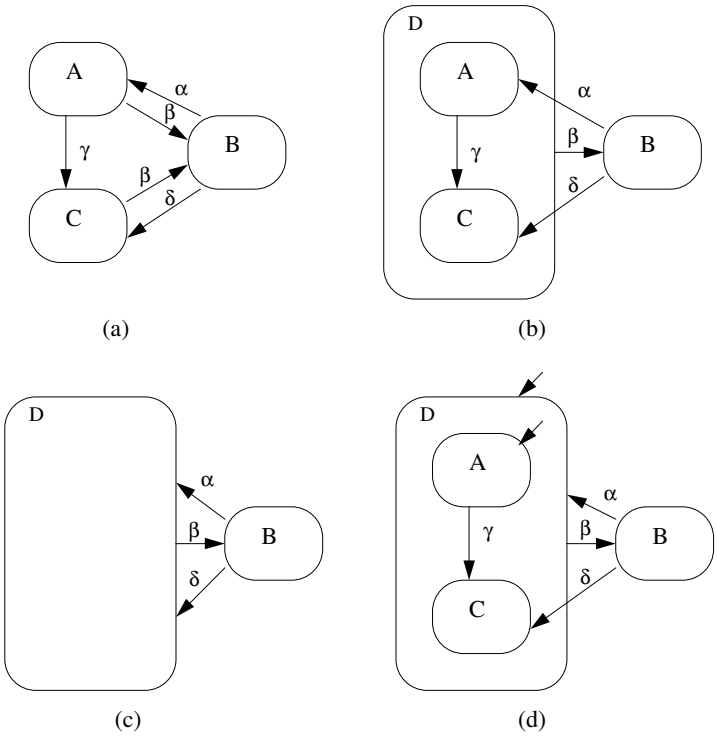

Figure 7: Abstraction and hierarchization in statecharts.

the *default* arrows. A default arrow describes the state in which the system will find itself in when a transition from the outside happens, and when more detailed information about the initial state is missing. So the outer default arrow pointing to the blob D (in subfigure (d)) says that in case of ambiguity between B and D the next state will always be D. The inner default arrow refers only to the subblobs of D, so in case of a transition pointing to D (as e.g. α in our example) the transition will occur to state A.

The next notion introduced in statecharts for simplifying specification of automata is a *history-dependent* default entry, illustrated in Figure 8. Subfigures (a) and (b) present two ways of using the history circle. Its meaning agrees with our intuition: the history circle causes a transition to the most recently visited state within a blob. In the (a) case it applies only to the incoming α transition, in the (b) case any entry using default will choose the most recent state. Obviously, when a blob is visited for the first time, the default state is chosen. Note that a history entry applies only to the uppermost level of a blob. So in subfigure (c), within G and F the usual default entry rules will be applied. On the other hand, using H* entry one can force application of the his-

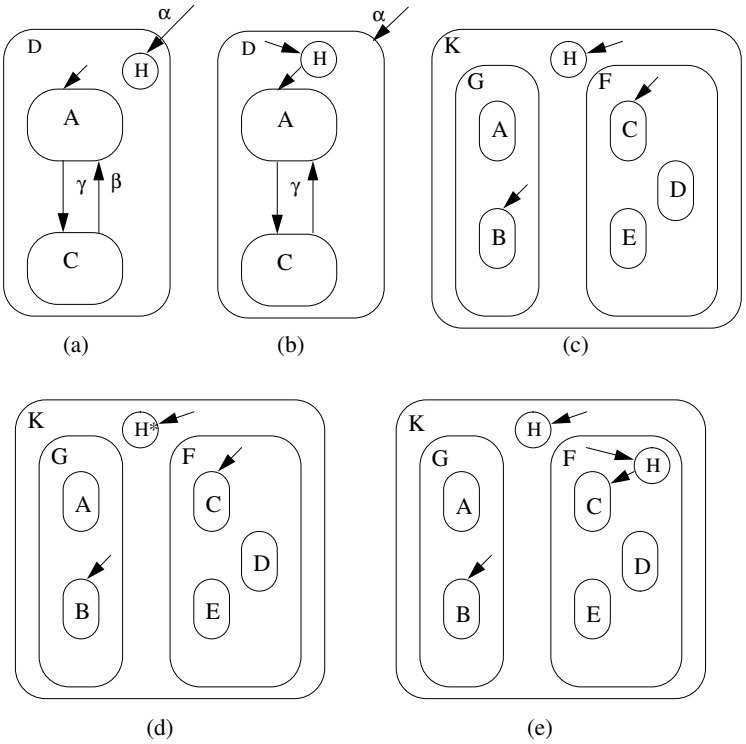

Figure 8: History-dependent entries in statecharts.

tory operator down to the atomic level. Subfigure (d) illustrates usage of H*. Finally, subfigure (e) illustrates intermediate usage of history entries, where entry to K and F is history-dependent, but within G only the standard default is applied.

The next important notion we would like to present here is *orthogonality*. It has already appeared in the formal definition of higraph in the form of the function π. Its meaning is the following: in the set of the states of a system, we can distinguish several subsets describing subsystems working in parallel. In terms of the state space this means that each global state is a tuple of local states of all subsystems, as illustrated in Figure 9. Subfigure (a) presents a blob (Y) divided into two orthogonal subblobs (A and D). The corresponding "flat" version of the state space is presented in subfigure (b). Note that the number of states and transitions in a non-orthogonal description is exponential with respect to the number of orthogonal components of the system.

Figure 9 (a) illustrates one more important feature introduced in statecharts: a *conditional transition*. The label β(in G) means that the transition (in this case from C to B) occurs only in case of input letter β appearing in the input **and** the subsystem D

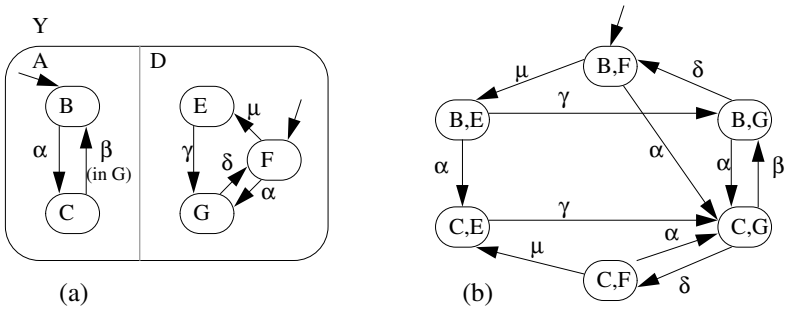

Figure 9: Orthogonality in statecharts.

being in state G.

Although Harel introduces several more concepts into the statecharts language (conditional dispatching, selectors, delays, and output functions), they do not pose any additional problems with respect to formalization or translation into PTNs. We will omit discussion of those concepts here.

3.3 Translation algorithm

To show that the PTN formalism and statecharts have the same expressive power (in terms of possibility of specifying different kinds of systems, not in terms of readability or of the number of language primitives) we have to present an algorithm for translating any PTN into a statechart, and any statechart into a PTN. Let us begin with the first, easier task.

In its most simple version, a PTN consists of a set of cycles of processes linked by transitions. Transitions may be linked with each other by enabling arrows (provided that the constraints of the definition are fulfilled).

The corresponding statechart will consist of atomic blobs corresponding 1–1 to the PTN's processes. Different cycle groups of the PTN will be represented by atomic blobs belonging to orthogonal subblobs of the main global blob of the statechart. (So there are three levels of the resulting statechart: one global blob, orthogonal subblobs corresponding 1–1 to cycle groups, and atomic blobs corresponding to processes.) Transitions between processes are mapped to corresponding transitions between the atomic blobs.

The proper enabling arrows are encoded as identical labels attached to enabling and enabled transitions, moreover, the enabled

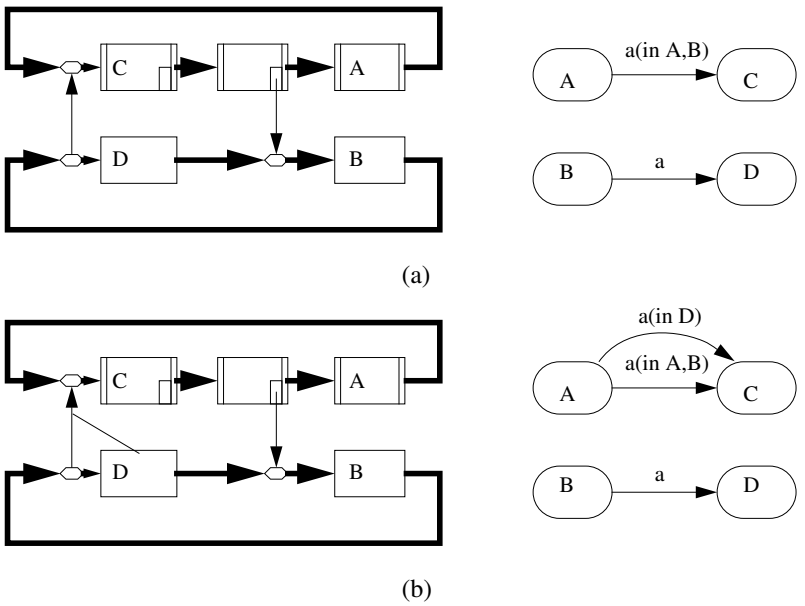

Figure 10: Translation of enabling arrows.

transition must also be labeled by a condition checking that both of the states preceding the transitions are active. See Figure 10 (a), where the transition from B to D enables the transition from A to C. The extended enabling arrows enable the target transition even if the source transition has already occurred. This case is illustrated in Figure 10 (b).

The statechart constructed this way corresponds behaviorally (in terms of state changes as a function of time/input) to the original PTN. Now we proceed to the translation of an arbitrary statechart into a PTN.

First, all atomic blobs are mapped into processes. All the transitions between the atomic blobs are mapped into appropriate transitions between processes. If the statechart contains orthogonal parts then they are translated into separate cycle groups of the PTN. We have to consider the translation procedure for the following constructs:

1. superblobs (i.e. non-atomic ones);
2. transitions to superblobs;
3. default entries;
4. history entries;
5. conditional transitions.

in order to complete the algorithm. Let us proceed in this order.

1. As superblobs are only abstractions, unless they have incoming or outgoing transitions, they can just be ignored.
2. Transitions pointing from superblobs have to be drawn separately from all the processes corresponding to subblobs of the considered blob. The incoming transitions have to be drawn to a single process corresponding to the initial state. In the case when there are neither default nor history entries, the process can be chosen in an arbitrary way. The other two cases are considered in the following points.
3. Default entries are pointing to the initial state, thus all the transitions to a superblob with a default entry should be directed to this state.
4. History entries have to be encoded by an additional PTN cycle. Processes of that cycle will correspond to all the internal states of a blob with a simple, i.e. H-type, history entry. Changes in that cycle correspond 1–1 to changes of the state within the blob. (Because such a cycle had to be constructed during previous steps, we can just use its copy.) Then any transition entering a blob via the history entry is dispatched to the appropriate process. Dispatching consists of multiplying the transition into an appropriate number of copies pointing to all the processes in the cycle, each transition being enabled from different place in the "history" cycle.

   As H*-type entries can be expressed using H-type ones, we don't present a separate construction for them.
5. Conditional transitions are translated to transitions with extended enabling arrows pointing from the processes corresponding to the condition.

The construction processes presented above (we reserve the term algorithm for more formal description) allow one to translate any PTN into a statechart, and any statechart into a PTN. This fact has interesting theoretical consequences for PTNs, as it has been shown [Zad90] that higraphs, the generalization of statecharts, correspond to a specific class of *datalog* logic programs.

4 Conclusion

We have presented four languages used for specifying behaviors of reactive autonomous systems. The common factor is that all those languages are based in some way on finite state automata. Two of the languages, Process Transition Networks and statecharts, have been chosen for closer presentation and it has been shown that they are equivalent.

The goal of this short presentation was twofold: to sketch the state of research in the area of specifying behaviors for autonomous agents, and to identify the existing constraints within which we would like to investigate the following problems:

1. the expressive power and limitations of specification tools used so far (behavior language, REX, Gapps, Ruler, PTNs, automata theory)

2. compositional specification of behaviors, so that from a set of specifications of simpler behaviors one could formally derive a specification of their composition;

3. decompositional specification of behaviors, so that specification of a complex behavior would allow one to infer a set of specifications of hardware-realizable primitive behaviors, or at least would yield predictable software fitting the layered architecture;

4. correspondence between specifications of behavior(s) of an autonomous system, and software specification for the three-layered software architecture;

5. theoretical tools for modelling dynamics of a single behavior;

6. theoretical tools for modelling dynamics of a system of coupled behaviors.

Acknowledgment

I would like to thank Brant Cheikes for very helpful comments.

References

[Ang89] Colin M. Angle. Genghis, a six legged autonomous walking robot. Master's thesis, MIT, 1989.

[Ark90] Ronald C. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2:14–23, 1986.

[Bro90a] Rodney A. Brooks. The behavior language; user's guide. Memo 1227, MIT AILab, April 1990.

[Bro90b] Rodney A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[Bro91] Rodney A. Brooks. Intelligence without reason. In *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence, Sydney*. Morgan Kaufman, 1991.

[Con89] Jonathan H. Connell. *A colony architecture for an artificial creature*. PhD thesis, MIT, 1989. AI Lab Tech Report 1151.

[FBIB89] Anita M. Flynn, Rodney A. Brooks, William M. Wells III, and David S. Barrett. The world's largest one cubic inch robot. In *IEEE Micro Electro Mechanical Systems*, pages 98–101, Salt Lake City, Utah, 1989.

[FBT89] Anita M. Flynn, Rodney A. Brooks, and Lee S. Tavrow. Twilight zones and cornerstones: A gnat robot double feature. AI Memo 1126, MIT, 1989.

[Fir89] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Department of Computer Science, Yale University, 1989.

[Gat91] Eran Gat. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*. PhD thesis, Virginia Politechnic Institute and State University, 1991.

[GL87] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings AAAI-87: Sixth National Conference on Artificial Intelligence*, pages 677–682. Morgan Kaufman, 1987.

[H+90] David Harel et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, April 1990.

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

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[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.

[KR90] Leslie Pack Kaelbling and Stanley J. Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6:35–48, 1990.

[Mal91a] 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.

[Mal91b] 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.

[Mat90] Maja J. Mataric. A distributed model for mobile robot environment-learning and navigation. Technical Report 1228, MIT AI Lab, May 1990.

[MS90] Magnus Morin and Erik Sandewall. A software architecture supporting integration of sub-systems. In *PROMETHEUS: Proceedings of the Fourth Workshop*,

1990. Also available as Technical Report LAIC-IDA-90-TR21, Dept. of Computer and Information Science, Linköping University.

[Pay90] David W. Payton. Internalized plans: A representation for action resources. *Robotics and Autonomous Systems*, 6:89–103, 1990.

[RK86] Stanley J. Rosenschein and Leslie Pack Kaelbling. The synthesis of digital machines with provable epistemic properties. In Joseph Halpern, editor, *Theoretical Aspects of Reasoning about Actions*, pages 83–98, Monterey, CA, 1986.

[Ros85] Stanley J. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3:345–357, 1985.

[Ros89] Stanley J. Rosenschein. Synthesizing information-tracking automata from environment description. In *Proceedings of the First International Conference on Principles on Knowledge Representation and Reasoning*, pages 386–393. Morgan Kaufman, 1989.

[SM91] Hua Shu and Jacek Malec. From Process Transition Networks to Behavior Automata. In *Proceedings of the 1991 IEEE International Symposium on Intelligent Control, Arlington, VA*, pages 257–262. IEEE Control Systems Society, 1991.

[Tho87] Charles E. Thorpe. Outdoor visual navigation for autonomous robots. In L.O. Hertzberger and F.C.A. Groen, editors, *Intelligent Autonomous Systems*, pages 530–544. North-Holland, 1987.

[Zad90] Włodek Zadrożny. Logical dimensions of some graph formalisms. In John F. Sowa, editor, *Principles of semantic networks*, chapter 12, pages 363–380. Morgan Kaufmann, 1990.