

# Representing and Interpreting Multiagent Plans with Statecharts<sup>1</sup>

**Xiaohong Wu, Benny Cox, and Albert C. Esterline**

*Department of Computer Science/NASA ACE*

*North Carolina A&T State University*

*Greensboro, NC 27411*

*email: {xiaohong, bennydc, esterlin}@ncat.edu*

## ABSTRACT

We show how to represent multi-agent plans using the statechart formalism. We emphasize an explicit approach for representing aggregate plans. The assignment of agents to roles in a plan and agents as position holders are also discussed. We present a textual syntax for plan-statecharts and sketch the design of a Java interpreter for plan-statecharts.

**KEYWORDS:** multiagent plans, statecharts, aggregation, concurrency

## INTRODUCTION

We represent multi-agent plans with statecharts [6], which represent both hierarchies of states and orthogonal (concurrent) components. Synchronization among components could be achieved implicitly, by modeling the aggregate through a web of references. Instead, we use an explicit approach, which makes plans more reusable and extendible. The next section introduces the statechart notation, describes the explicit approach to synchronizing aggregate plans, and describes agents as position holders. Subplans are not the agents but rather the role positions to which agents are assigned. The following section presents a textual, abstract syntax for plan-statecharts, and then we sketch a Java interpreter for plan-statecharts that assigns roles to different threads (agents).

## STATECHARTS

Statecharts [2] are a visual formalism to specify the behavior of complex reactive systems. They extend conventional state-transition diagrams to include the notions of hierarchy, concurrency (orthogonality), and communication. Statecharts also may be viewed at varied levels of detail through abstraction and refinement. Thus,

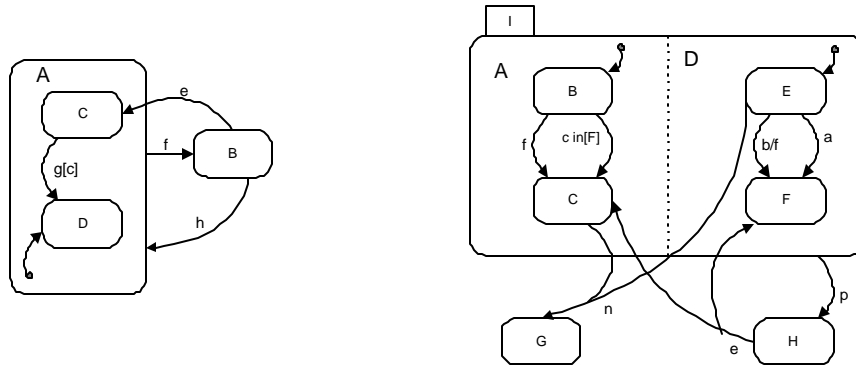
Statecharts = state-diagrams + depth + orthogonality + broadcast-communication

Hierarchy in statecharts comes from grouping states into superstates – an *XOR* (exclusive-or) decomposition as in Figure 1. This statechart includes two states, A and B. A has two substates, C and D. If the system is in A, then it must be either in substate C or D, but not both (exclusive or). The arrow indicates the transition direction; the label over it indicates the transition event. Suppose the system is in state B. When event *e*

---

<sup>1</sup> This research was supported by grant NAG 2-1150, “Motion Planning in a Society of Intelligent Mobile Agents,” from NASA Ames Research Center.

occurs, it goes to substate C. When event  $h$  occurs, it goes to A as a whole (the arrow labeled “ $h$ ” stops at A’s boundary). The unlabeled dotted arrow to substate D indicates that D is the default state of A. So, the system goes to D on event  $h$ . If when event  $f$  occurs, whatever substate of A the system is in, it goes to B (the arrow labeled “ $f$ ” starts at the boundary of A). Finally, if the system is in substate C, when event  $g$  occurs, it goes to D if and only if condition  $c$  holds (see the “ $c$ ” in square bracket after the label “ $g$ ”)



**Figure 1** Statechart with XOR-state      **Figure 2** Statechart with AND-state

Concurrency in statecharts is shown through orthogonality, which is an AND decomposition captured by the partition of the statechart as in Figure 2. There are three states, G, H and I. I has two orthogonal components, A and D (see the dashed boundary). When the system is in I, it must be in A and D. A and D also have XOR decompositions so that, if the system is in I, it must be in one of four possible pairs of substates  $\langle B, E \rangle$ ,  $\langle B, F \rangle$ ,  $\langle C, E \rangle$  or  $\langle C, F \rangle$ . If the system is in the state H when event  $e$  occurs, it enters I and ends up at substates C of A and F of D. If the system is in substate E of D and either B or C of A, then, when event  $a$  occurs, it goes to F and stays in whatever substate of A it was in. However, when event  $b$  occurs, a transition labeled  $b/f$  is taken, the action  $f$  is immediately activated, and is regarded as an (internal) event. Now, if the system is also in the substate B of A, then it goes to C; thus, the system goes to F and C when event  $b$  occurs. If the system is in substates C and E in I, when event  $n$  occurs, it leaves the state I and enters state G. The arrow labeled “ $p$ ” indicates that, if the system is in any one of four pairs of substates of I aforementioned, when  $p$  occurs, it goes to state H.

The communication mechanism in statecharts is based on broadcast, whereby the sender may proceed even if nobody is listening. It also forces all enabled listeners to accept and respond to the message simultaneously.

## PLAN-STATECHARTS WITH WHOLE-PARTS CHARTS

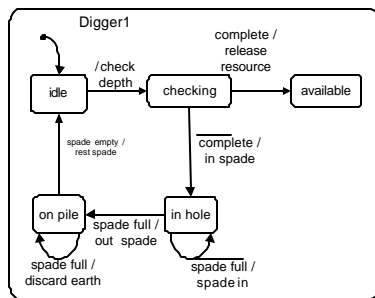
Multiagent plans can generally be represented straightforwardly with statecharts –called *plan-statecharts* – since these plans decompose into several subplans carried out sequentially (XOR) or concurrently (AND). Plan-statecharts generally need idle states, and they must clearly indicate the roles they define that can be assigned to agents.

Traditionally, interactions among orthogonal components are synchronized implicitly: different components of an AND-state synchronize by the combined effects of broadcast and triggering [4]. These implicit methods lack reusability, understandability and extendibility because implicit synchronization does not encapsulate the details of components – with any change in a part, the other parts and the AND-state must change.

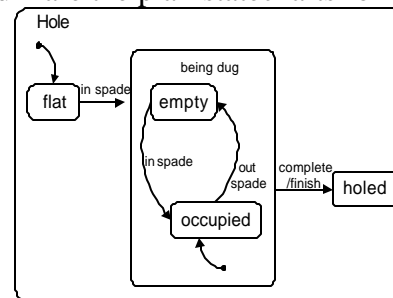
And, with any modification of the whole, the different parts must be modified. [4] proposed an explicit method that extracts synchronization aspects from the statechart and encapsulates them into an additional statechart, representing the whole. The *whole* section and *parts* sections evolve concurrently as in an AND-state. The following communication constraints enforce self-containment and encapsulation: (1) Events cannot trespass the boundary among the different *parts* sections. (2) Events can be broadcast both from the *whole* section to the *parts* sections and vice versa. So the *whole* knows its *parts* but the *parts* ignore each other. Synchronization aspects are expressed entirely in the *whole* section without affecting the behavioral descriptions of the parts.

In plan-statecharts, we adopt this explicit method. A *whole-parts* chart is associated with any AND-plan whose components interact. A whole-parts chart has a parts section and a whole section. The interacting components are expressed as parts in a whole-parts chart separated by solid lines to emphasize rule (1). The events which control the synchronization among the parts are listed in the parts. A dash line separates the whole component from the parts to emphasize rule (2). A solid circle in the whole component indicates its start, and a solid circle within a circle indicates the termination of the whole. A prefix notation indicates the source of events:  $A.\alpha$  denotes the event  $\alpha$  from plan  $A$ .

Salient subplans show plans for role positions, not agents. This allows flexibility in the agent assignment process. For example, we might have a dig-hole plan that is an AND-plan with two parts, diggers and a hole. Figures 3 and 4 are the plan-statecharts for the



**Figure 3** The Position-Related Plan-Statecharts for Digger1's Plan



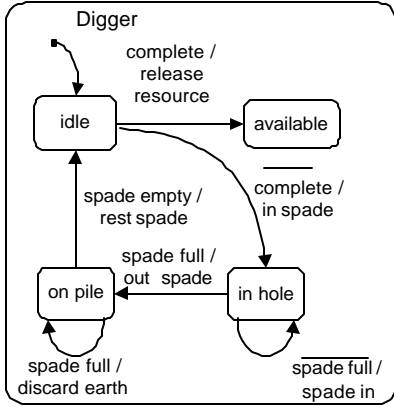
**Figure 4** The Position-Related Plan-Statechart fro Hole's Plan

*digger1* and *hole* position holders. The *dig-hole* plan may involve several diggers. The other diggers' positions (called *Digger*) differ from *Digger1* by lacking a checking step. Figure 5 shows the general digger's position plan. When several diggers are assigned to the *dig-hole* plan, the actions among the diggers are synchronized to avoid their spades colliding. Figure 6 shows a two-diggers whole-parts chart for *dig-hole*. The negotiation part in section *whole\_dig-hole* assigns two agents to the positions *digger1* and *digger* and initiates the plan. (For negotiaion, see, e.g., [1].) The collisions and modeled in the *whole* section, which explicitly expresses the synchronization among the components.

Explicitly expressing the synchronization aspects entirely in a *whole* section provides a natural context for extendibility. We can add one more digger in the *dig-hole* plan without any change to the parts already present and by including just one additional state in the whole.

## A TEXTUAL SYNTAX FOR PLAN-STATECHARTS

We sketch a textual, abstract, inductively-defined syntax for plan-statecharts with an intuitive operational semantics. This provides concepts that guide us in designing the interpreter. Some of the definitions are based on the syntax in [5] and in [3].

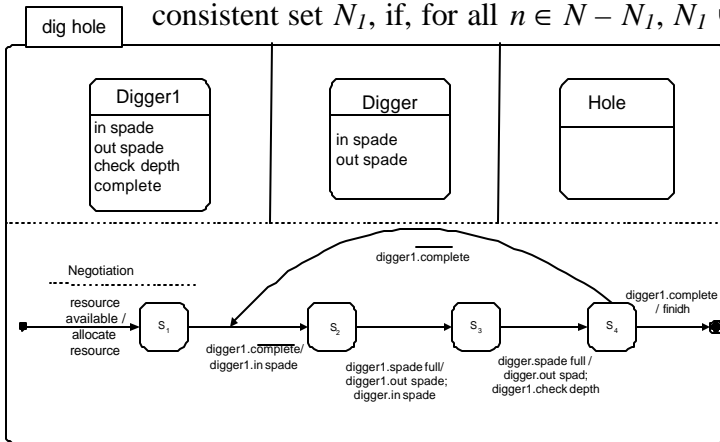


**Figure 5** The Position-related Plan-Statechart for Digger

We begin with the syntax of statecharts then enhance it to accommodate whole-part charts. A statechart is a tuple  $S = (\Pi, N, \mathcal{T}, r)$ , where  $\Pi$  is a finite set of primitive events,  $N$  is a finite set of (names of) states,  $\mathcal{T}$  is a finite set of transitions, and  $r \in N$  is the root. The states  $N$  are structured as a tree by the function  $children: N \rightarrow 2^N$ . If  $n_2 \in children(n_1)$ , then  $n_1$  is the *parent* of  $n_2$  and  $n_2$  is the *child* of  $n_1$ . We can also view  $children$  as a binary relation on  $N$ , and it is straightforward to define the reflexive and transitive closure of  $children$ , called *descendants*. Function  $type: N \rightarrow \{PRIM, AND, XOR\}$  identifies each state as a primitive (or leaf) state, an AND-state, or an XOR-state. Finally, the partial function  $default: N \rightarrow N$  is such that  $default(n) \in children(n)$  is

the initial state for XOR-state  $n$  when  $n$  (and not one of its substates) is the target of a transition.

Let  $lca N_1$  be the least common ancestor of the set  $N_1$  of states. States  $n_1$  and  $n_2$  are *orthogonal*, written  $n_1 \perp n_2$ , if neither is an ancestor of the other and  $type(lca \{n_1, n_2\}) = AND$ . A set  $N_1$  of states is an *orthogonal set* if,  $\forall n_1, n_2 \in N_1$ , either  $n_1 = n_2$  or  $n_1 \perp n_2$ . A set  $N_1$  of states is *consistent* if,  $\forall n_1, n_2 \in N_1$ , either  $n_1 \perp n_2$  or  $n_1$  and  $n_2$  are ancestrally related. Intuitively,  $S$  can be in two states at the same time if they are consistent. For a consistent set  $N_1$ , if, for all  $n \in N - N_1$ ,  $N_1 \cup \{n\}$  is not consistent, then  $N_1$  is



**Figure 6** The Whole-Parts Chart for Dig-Hole Plan

*maximally consistent*. A *configuration* of statechart  $S$  is a maximally consistent subset of  $N$  and represents a global state of  $S$ .

Let  $\bar{\Pi} = \{\bar{e} \mid e \in \Pi\}$ , where a negative event  $e$  indicates the absence of event  $e$ . A label  $l = (\varepsilon/\alpha)$  is an ordered pair where  $trigger(l) = \varepsilon \subseteq \Pi \cup \bar{\Pi}$  and  $action(l) = \alpha \subseteq \Pi$ . Let  $\mathcal{L}(\Pi)$  be the set of labels definable over  $\Pi$ .

$trigger(l) = \{e_1, \dots, e_i, \bar{e}_{i+1}, \dots, \bar{e}_k\}$  indicates that events  $e_1, \dots, e_i$  happen along with any other events consistent with them except for  $e_{i+1}, \dots, e_k$ .  $action(l) = \{a_1, \dots, a_j\}$  indicates that the events  $a_1, \dots, a_j$  happen (i.e., the actions are performed) along with any other events consistent with them. Now, a *transition*  $t$  is a triple  $(N_1, l, N_2)$ , where  $N_1, N_2 \subseteq N$  and  $l \in \mathcal{L}(\Pi)$ ; here  $source(t) = N_1$ ,  $target(t) = N_2$ , and  $label(t) = l$ . In simple cases,  $N_1$  and  $N_2$  are singletons, but the orthogonality of statecharts may require a transition that leads from a *set* of orthogonal components to another such set. The *arena* of  $t$ ,  $arena(t)$ , is the

smallest XOR-state containing all parts of  $t$ . That is,  $arena(t) = lca\ source(label(t)) \cup target(label(t))$ . Transitions  $t_1$  and  $t_2$  are *consistent* if either  $t_1 = t_2$  or  $arena(t_1) \perp arena(t_2)$  (i.e., if the one does not prevent the other, i.e., if they could fire together). A set  $T$  of transitions is *consistent* if,  $\forall t_1, t_2 \in T, t_1$  and  $t_2$  are consistent.

A plan-statechart  $P$  has one more component,  $N_w$ , than a statechart:  $P = (\Pi, N, \mathcal{T}, r, N_w)$ .  $N_w \subseteq N$  is the set of (names of) whole states (or “sections”). We assume that the children of any AND-state include exactly one whole state; the other children (the “proper children”) are part states. Let  $whole: N \rightarrow N$  be the partial function that maps an AND-state to its whole state. The partial function  $parts: N \rightarrow 2^N$ , mapping an AND-state to its set of part states, is defined so that  $parts(n) = children(n) - whole(n)$ .

Now, we can think of the subtree rooted at a whole state  $w$  as a sub-statechart whose set of states is  $descendants(w)$ . The only role of a whole state is to provide synchronization – there are no transitions into or out of the sub-statechart rooted at a whole state. More formally, for any transition  $t$  and whole state  $w$ , either  $arena(t) \cap descendants(w) = \emptyset$  or  $arena(t) \subseteq descendants(w)$ .

It remains to characterize the synchronizing events. For any part state  $p$ , let  $sort(p)$  be the synchronizing events visible to the whole section. For any state  $n$ , let  $Tr(n)$  be the set of transitions contained within the sub-statechart rooted at  $n$ :

$$Tr(n) = \{t \mid t \in \mathcal{T} \text{ and } arena(t) \subseteq descendants(n)\}$$

And let  $Ev(n)$  be the set of events that appear as triggers or actions in these transitions:

$$Ev(n) = \cup_{t \in Tr(n)} (trigger(label(t)) \cup action(label(t))).$$

Then, for a part state  $p$ ,  $sort(p) \subseteq Ev(p)$ . Since all the triggers and actions relating to a whole state  $w$  are involved in synchronization, we define  $sort(w) = Ev(w)$ . Then, for any AND-state  $n$ ,

$$sort(whole(n)) = \cup \{sort(p) \mid p \in parts(n)\}$$

## INTERPRETER IMPLEMENTATION

An interpreter for a state transition diagram is straightforward. The diagram itself is represented by a new-state table whose row indices are state identifiers and whose column indices are event identifiers. At each step, the next state is found from the table using an input event and the value of the current-state variable. If an action is performed, we include an action table similar to the new-state table. This simple approach does not work for plan-statecharts. Because of hierarchy, we in effect would need separate tables for the children of XOR-states. Because of orthogonality, the analogue of the current state is a configuration. Finally, the trigger and the action of a transition can each contain several events, and several transitions may fire at the same time – so a *set* of events must be remembered and subsets of it used as triggers.

In a whole-parts chart that is realized by a group of agents, the parts agents are now realized in Java as separate threads. The whole section is a shared resource having its own configuration, updated by the parts agents when they access it. To impose a discipline on the parts agents’ access of the whole, we use Java’s synchronization resources. The only transitions in or out of a group are when the entire group is created or terminated. Some whole-parts charts correspond not to a group but to several concurrent activities for one agent; the agent’s entire configuration must be remembered.

Within an agent, we flatten the state hierarchy, using the subset of a configuration with only primitive, leaf states. A statechart being a tree, we can recover the configuration.

We form the set of all the maximal consistent sets of primitive states descended from a transition's set of source states, and, in the set of target states, we replace each non-primitive state with the primitive state found by following the chain of default links starting from it. If there are  $k$  primitive source state sets, this gives  $k$  new transitions. These transitions are stored in a binary search tree indexed by the sets of primitive source states. Each node in this tree is the root of another binary search tree, indexed by sets of positive events that serve as triggers for transitions with the given source states. Transitions at a node in this tree differ only on the negative events in their triggers (and their target states). We find which transitions are consistent (which requires hierarchical information) and record this in a two-dimensional array indexed by transition numbers.

At each step, we have the current (primitive-state) configuration and set of events from the last step (including those from the parent agent and the visible events from the child agents). We use this information to find all enabled transitions, of which we randomly select a maximal consistent subset  $T$ . We then signal any agent groups within the current agent to make a step (recursion) and perform the transitions in  $T$ . To perform a transition, we remove its source states from the configuration and adjoin its target states. We also adjoin its actions to the set of events for the next step. If a target state is an AND-state representing a group of agents, this group must be started (threads created, etc.). When a group of agents all reach their end states, their threads are terminated. There may be several activations of a whole-parts chart represented by a group because negotiation is required each time it is entered. Negotiation is trivial when we use threads, but, if we use network resources, we must find the appropriate network agent for the role.

## CONCLUSION

We have shown how statecharts represent multi-agent plans and have emphasized an approach to modeling an aggregate plan that explicitly represents the whole plan. This makes plans more reusable and extensible. We have also presented a textual syntax for plan-statecharts and sketched a plan-statechart interpreter based on it. It remains to define an operational semantics for the textual language and to extend the interpreter to other forms of concurrency. Note that agents are not represented by parts of statecharts but rather are assigned such parts. This allows a useful notion of the state of an agent, namely, a pair consisting of the history of the agent and a set of pointers to the parts of its plan currently executing. The hierarchical nature of statecharts allows an agent's history to be summarized by remembering only higher-level states if details are unnecessary.

## REFERENCES

1. Davis, Randall and Smith, Reid G. "Negotiation as a Metaphor for Distributed Problem Solving." *Artificial Intelligence* 20 (1983), 63-109.
2. Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8 (1987), 231-274.
3. Harel, P., Pnueli, A., Schmidt, J. P., and Sherman, R. "On the Formal Semantics of Statecharts." *Proc. 2<sup>nd</sup> IEEE Symposium on Logic in Computer Science* (1987), 54-64.
4. Pazzi, L. "Extending Statecharts for Representation Parts and Wholes." *Proceedings of the 23rd EUROMICRO Conference* (1997).
5. Pnueli, A. and Shalev, M. "What is in a Step: On the Semantics of Statecharts." *Theoretical Aspects of Computer Software*. Edited by A. R. Meyers. Springer, Berlin (1991), 244-264.
6. Wu, X. The Formal Representation of Multi-Agent Plans. Master's thesis, Dept. of Computer Science, North Carolina A&T State University, Greensboro, NC (1999).