

State Charts versus State Transition Diagrams

Why State Charts represent a superior design methodology and how to implement what they specify

GLO Intl Ltd

June 2004

Abstract	1
Why State Charts are superior	2
The concepts of state, event and transition	2
LitterBot's basic behaviour	2
Summary of State Charts	9
Implementing a state chart.....	10
What needs to be implemented	10
StateLogics - the solution	16
Your next step:	17
Call us now...	17

Abstract

This paper uses the example of a litter-collecting robot, LitterBot, to illustrate the difference between *state transition diagrams* and *state charts* as ways to specify and implement system behaviours. State charts allow hierarchic modelling of states and transition between such hierarchic states. This allows for the encapsulation of behaviours. They also provide an elegant expression of multiple, parallel behaviours. State charts remain humanly tractable even for complex systems. State transition diagrams lack these devices and quickly become incomprehensible, even for specifications of modest complexity, such as our LitterBot specification.

The second part of this paper illustrates the challenge of implementing the state behaviours specified by a state chart. States form a hierarchy, but transitions can occur at any level of that hierarchy and result in a new hierarchy. The implementation has to identify the correct transition to take and establish the new hierarchic state, as well as execute all relevant program code to effect actions upon transition, upon exit of the state hierarchy and upon entry to the new hierarchy. .

There are a number of variants of state charts, notably Harel's original and evolving versions of UML. For purposes of exposition, the present paper stays at a high level and introduces the common concepts.

A separate paper, available only under non-disclosure, describes GLO Ltd's solution to implementing the state behaviours specified by a state chart. The implementation requires minimal processor cycles and minimal memory – it is fast and small. This solution has the effect of turning state charts from specification tools into programming tools, maintaining a one-for-one mapping between chart and code. The solution can be adapted to any consistent semantics for state charts.

Why State Charts are superior

The concepts of state, event and transition

Both state charts and state transition diagrams specify the behaviour of a system in terms of the *states* the system can occupy and the allowable *transitions* between these states. Transitions are triggered by *events*, of which there are a finite number of types. Figure 1 illustrates these basic concepts.

1.

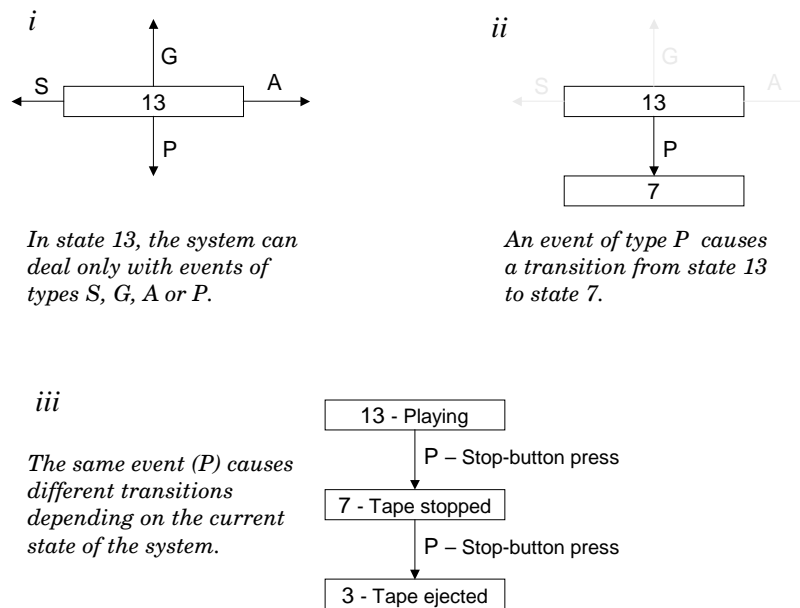
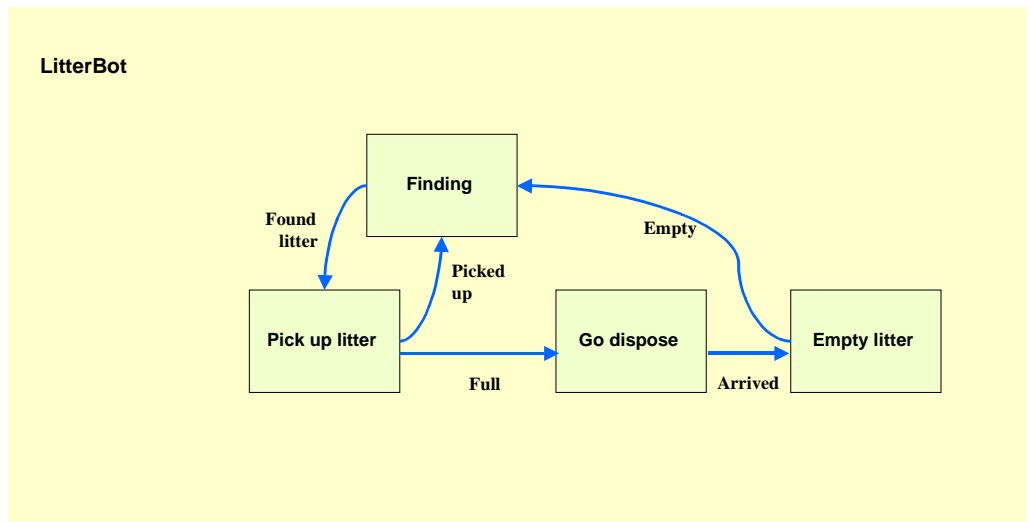


Figure 1 States, Events and Transitions

Part *i* of this diagram shows that, in a given state, the system's behaviour depends on which one of potentially many different inputs occurs. Part *iii* shows that, for one given input (P), the system responds differently depending on which one of potentially many states the system is currently in. The correct transition is determined by the combination of current state and event type, a many-to-many problem. This can be conceived as a matrix, with states as its column headings; inputs and events labelling the rows; and transitions in the cells – where the combination of state and input is a valid one.

LitterBot's basic behaviour

LitterBot is in the world to find and dispose of litter. Its basic behaviour is modelled in the state transition diagram of Figure 2. However, LitterBot runs on stored energy and this runs low after a variable period of operation. Its basic rule of survival is to drop everything and get a recharge, as soon as it senses its power is low. This additional behaviour is illustrated in Figure 3.



2.

Figure 2 *Litterbot's basic states*

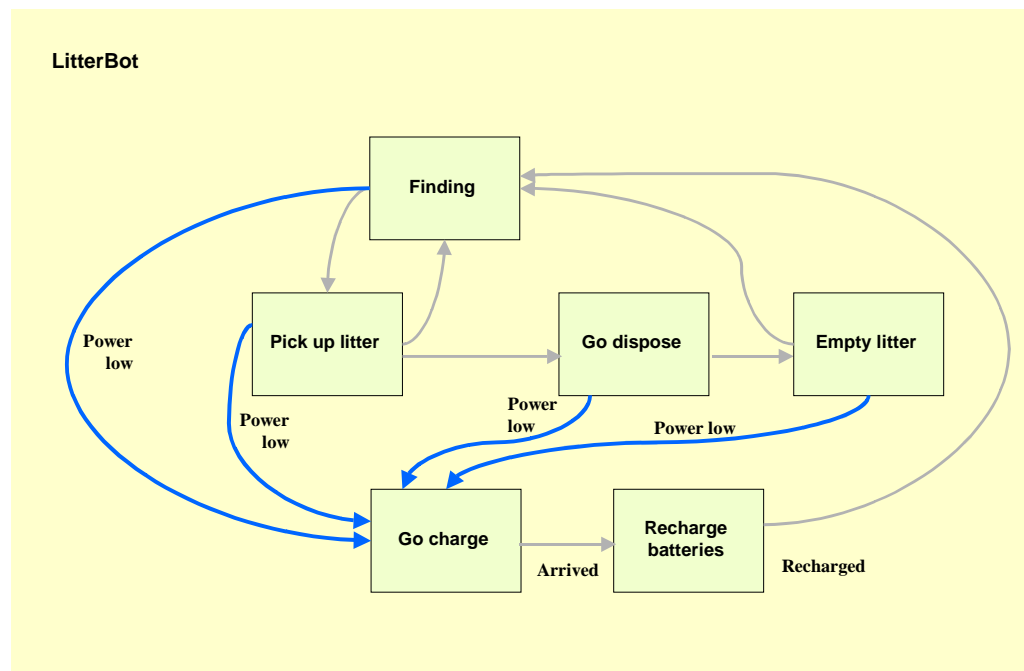


Figure 3 *Litterbot drops everything to get recharged*

Figure 3 shows that this simple survival requirement results in four transitions, triggered by the Power Low event, in addition to the transition needed to resume work (Recharged) and an intermediate transition (Arrived).

A State Chart, as opposed to a STD, uses hierarchy to express the same more clearly and elegantly. Figure 4 introduces the Collecting Litter state, to wrap up the four original states. Now a single Power Low transition suffices.

(For presentational purposes we alternate colours, for progressive levels in the hierarchy. Siblings have the same colour. The colours have no other significance.)

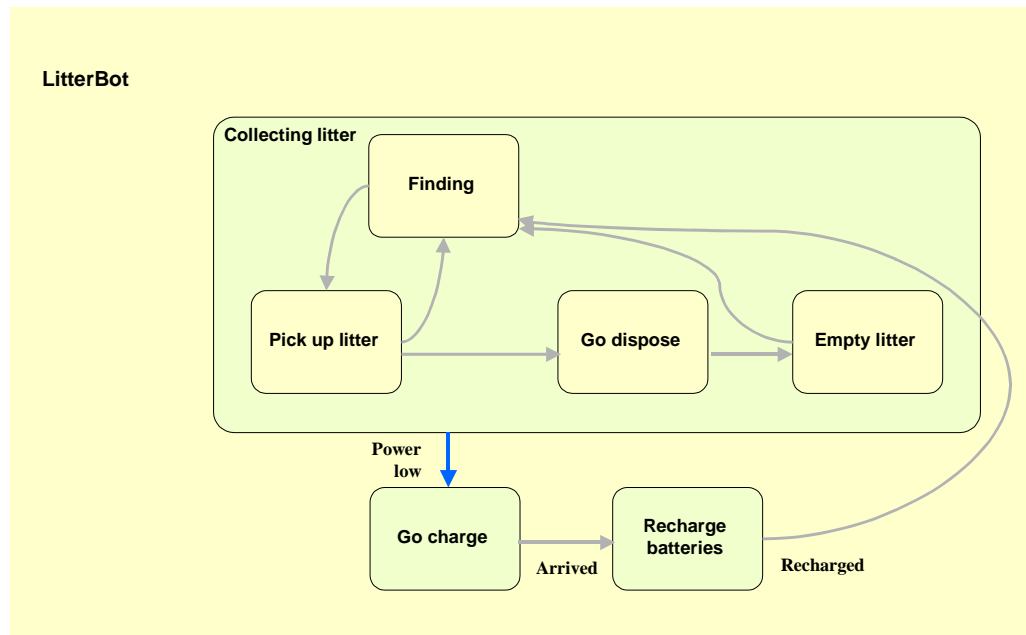


Figure 4 *A single transition quits all states in Collecting Litter*

Figure 4 can be further improved upon. We can declare Finding as the primary state (the initial state) within Collecting Litter. Then whenever any transition enables the Collecting Litter state, the sub-state Finding will also be enabled. This effectively encapsulates the behaviours within Collecting Litter and hides the detail from states outside it. Transitions are now *to* as well as *from* boundary, as shown in Figure 5

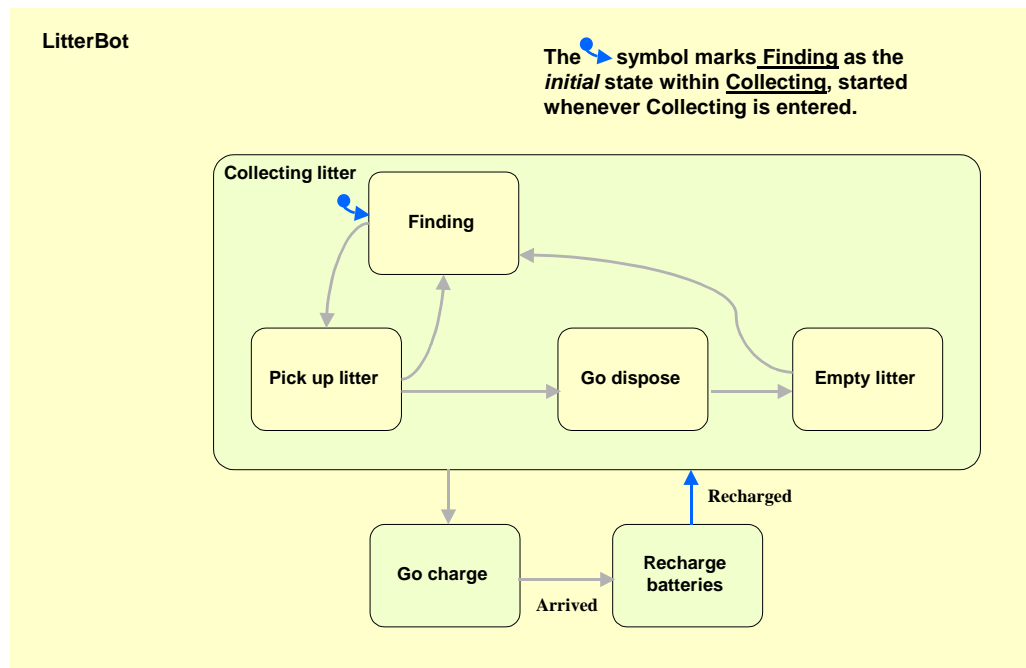


Figure 5 *Encapsulated detailed behaviours*

To find litter, dispose of it and charge itself, LitterBot needs to move around until it arrives where it needs to be, or receives a change of target. The chart for this behaviour is shown in Figure 6.

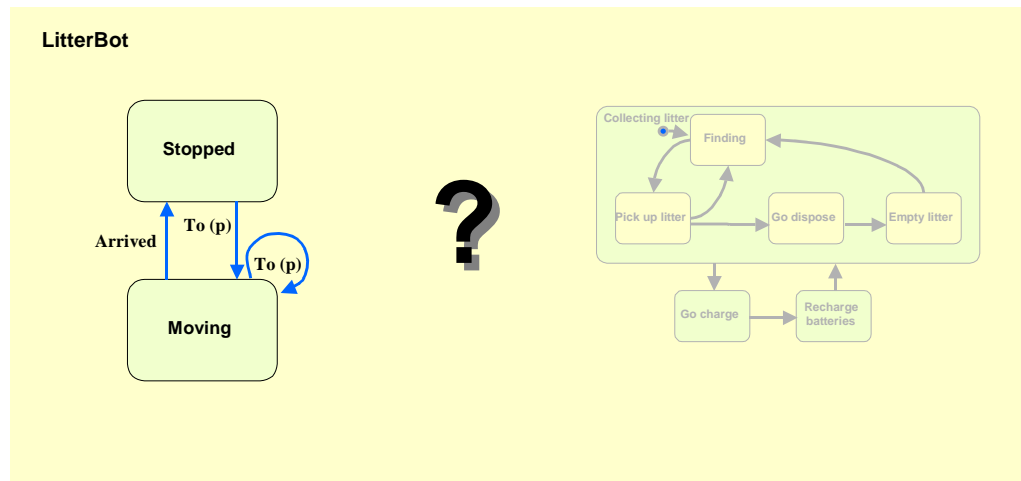


Figure 6 *Introducing parallel behaviours*

How do we integrate this parallel behaviour with the existing specification? In a state transition diagram, one would have to create a Moving and a Stopped variant of each previously specified state, doubling the number of states. The result is shown in Figure 7. This does not show transitions whose combined number would also double.

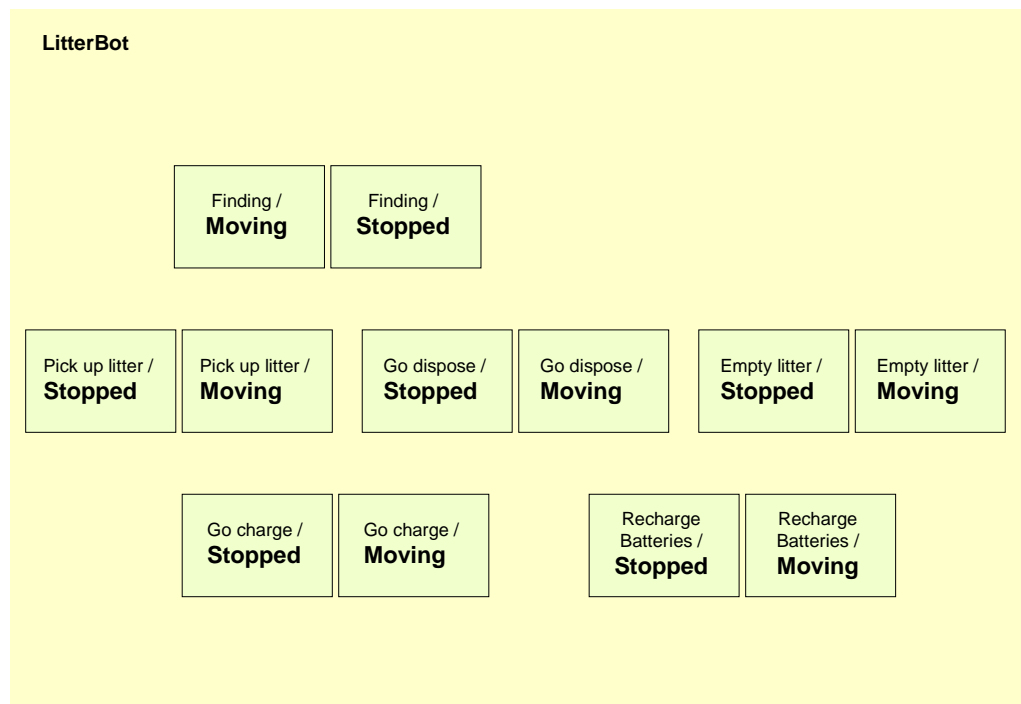


Figure 7 *States of parallel behaviours multiply*

A state chart deals with the integration of parallel behaviours without multiplying states. Instead the parallel behaviours are kept independent, as shown in Figure 8. The new, parallel behaviour is simply given a name (Advancing) and separated from the original behaviour (now named Operating) by a dotted line. Primary (initial) states are identified to show in which sub-state each parallel behaviour starts up when entered.

The specification of the original behaviour stays in tact. This is a key advantage of state charts: you can add behaviours with minimal impact on the existing specification

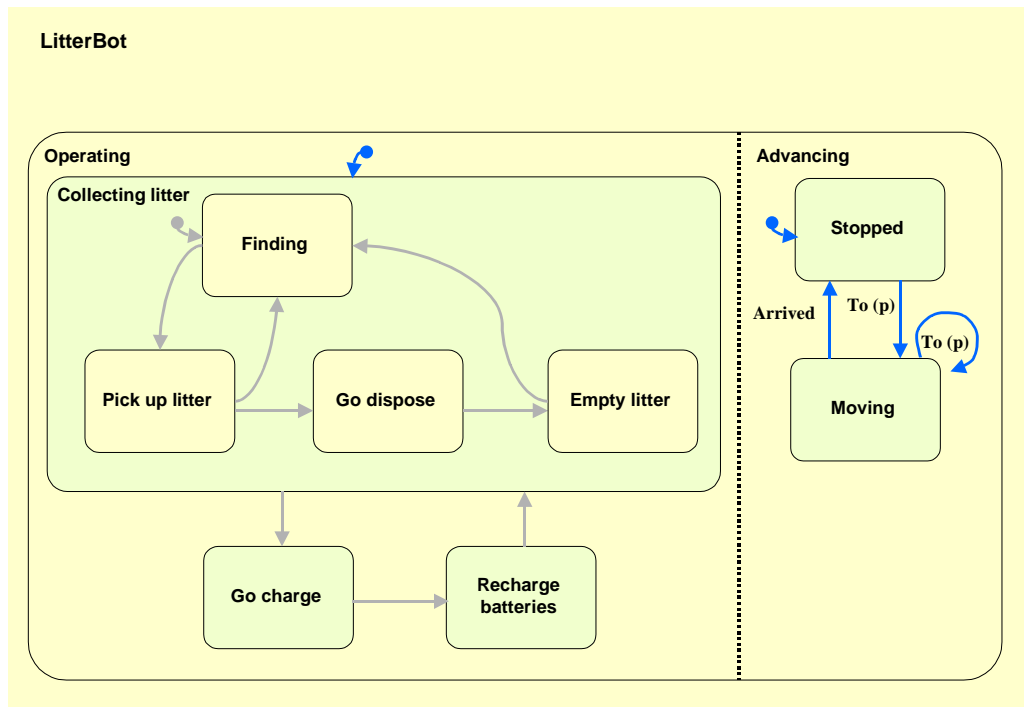


Figure 8 *Parallel behaviour is added with minimal impact*

To emphasise the point that you can evolve a state chart incrementally, without having to rework the entire specification, let's add some more behaviour: LitterBot may, from time to time, loose traction. The design assumption is that this is because it has flipped over – it's gone belly up – possibly because a competing bot has assaulted it. This is an event that can happen in any of its states. Its strategy for righting itself is not detailed here (presumably it involves vigorous thrashing and flailing), but it is survival behaviour that takes precedence even over recharging. Figure 9 shows the enhanced specification.

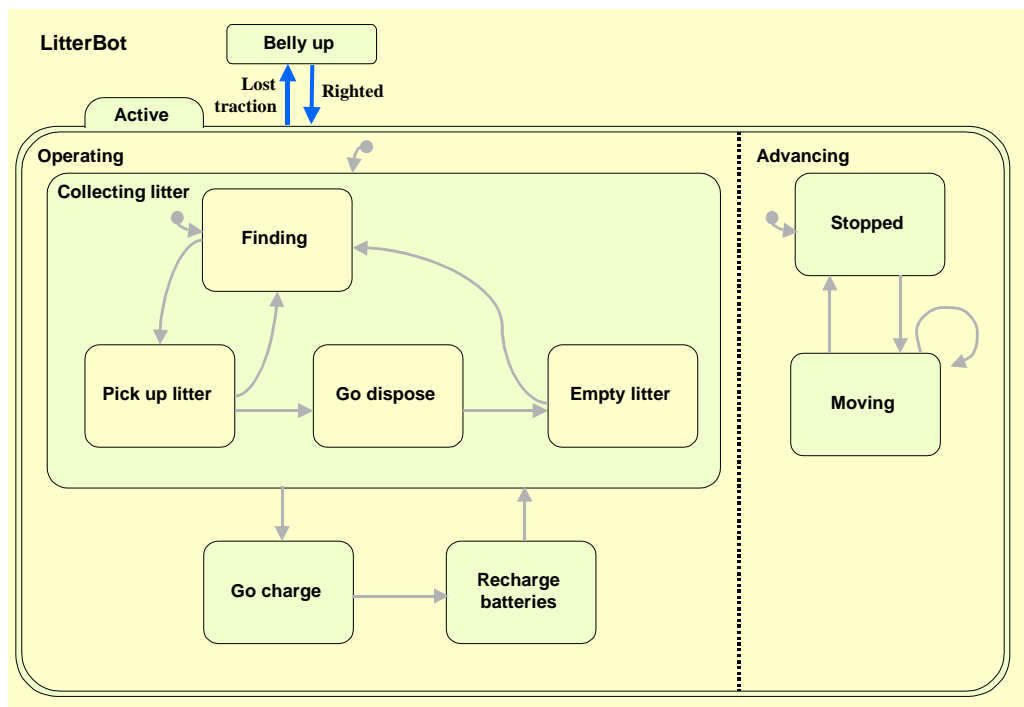


Figure 9 *Major survival behaviour added with little effort*

Note how little needed to be done to add the extra behaviour. We simply gave the combination of Advancing and Operating a new name (Active) and introduced just two new transitions, to or from the new Belly Up state.

Contrast this with the havoc this extra behaviour creates in a state transition diagram. Each of its multiplied states needs to be able to deal with the Lost Traction event, so each has a transition arrow going to Belly Up. Figure 10 shows our first attempt to draw the state transition diagram (STD) equivalent of Figure 9. This shows the lowest level states only, since a STD has no hierarchy. Each state has Stopped and Moving variants, to account for the parallel behaviour. The designer has eliminated two Moving variants as invalid. Note that we have left off all event names from the transitions. The diagram is already too cluttered.

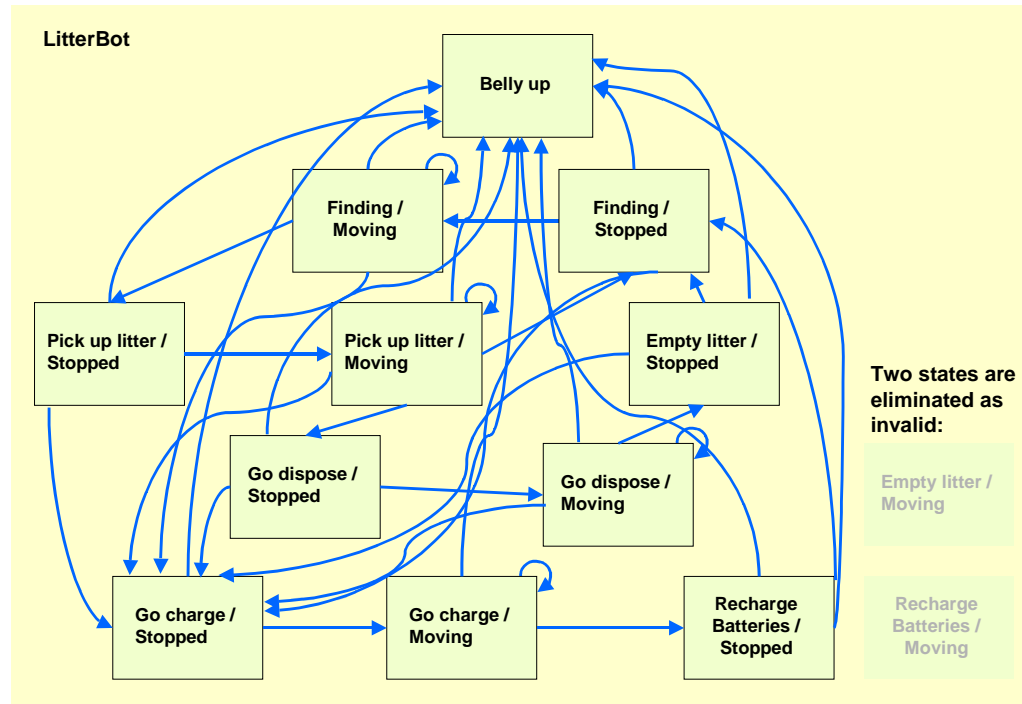


Figure 10 State transition diagram specifying the same behaviour as Figure 9

The long and crossing lines make this diagram incomprehensible. It can be tidied up by placing the most frequently targeted states (Go Charge and Belly Up) centrally on the diagram. This results in Figure 11. This is a complete reworking of the original. Not only is this a waste of the engineer's time, it militate against the best practices of incremental refinement and makes it very hard to ascertain whether the original behaviour has survived in tact. A dedicated tool could assist the engineer and calculate the optimum layout, perhaps piping connections in bundles to their common destination, but the result will remain hard to understand and hard to compare with earlier iterations.

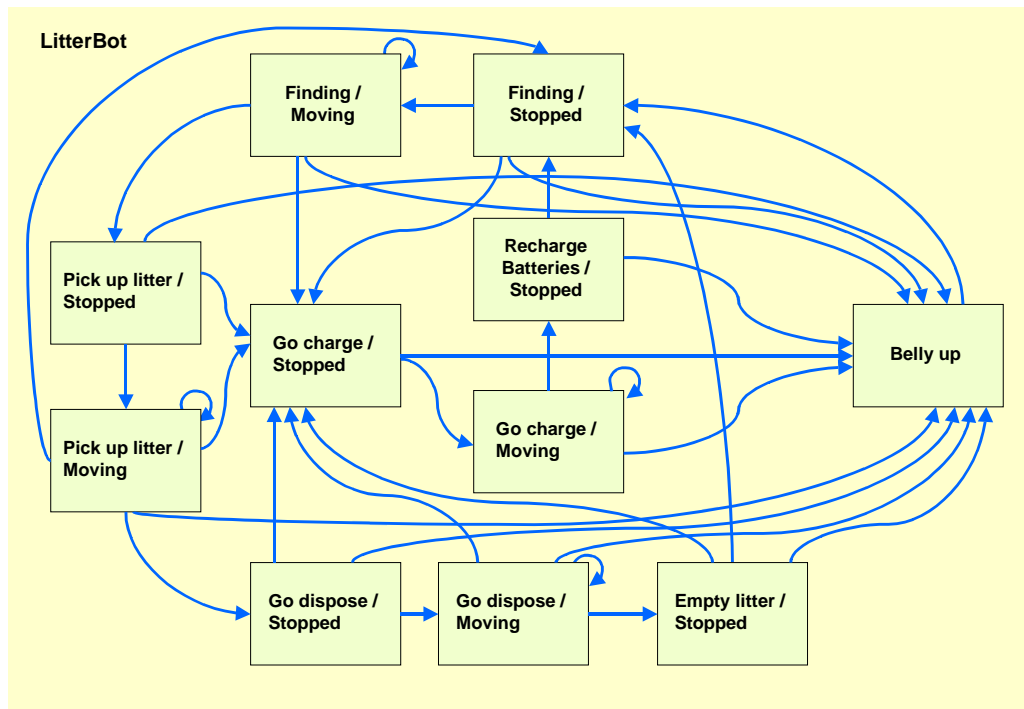


Figure 11 'Tidied up' STD – completely new layout

Figure 12 shows a further refinement of the LitterBot specification, adding additional detail to two of its existing states, including additional parallel behaviour. A dedicated state chart tool could hide or show the detailed behaviour within states on request, perhaps jumping to it via a hyperlink, or using sophisticated zooming behaviour to expand in context. This enhanced specification also features references to entry and exit code, to be executed when the machine reaches or leaves the state in question. Transition code could similarly be added to the transitions arrows.

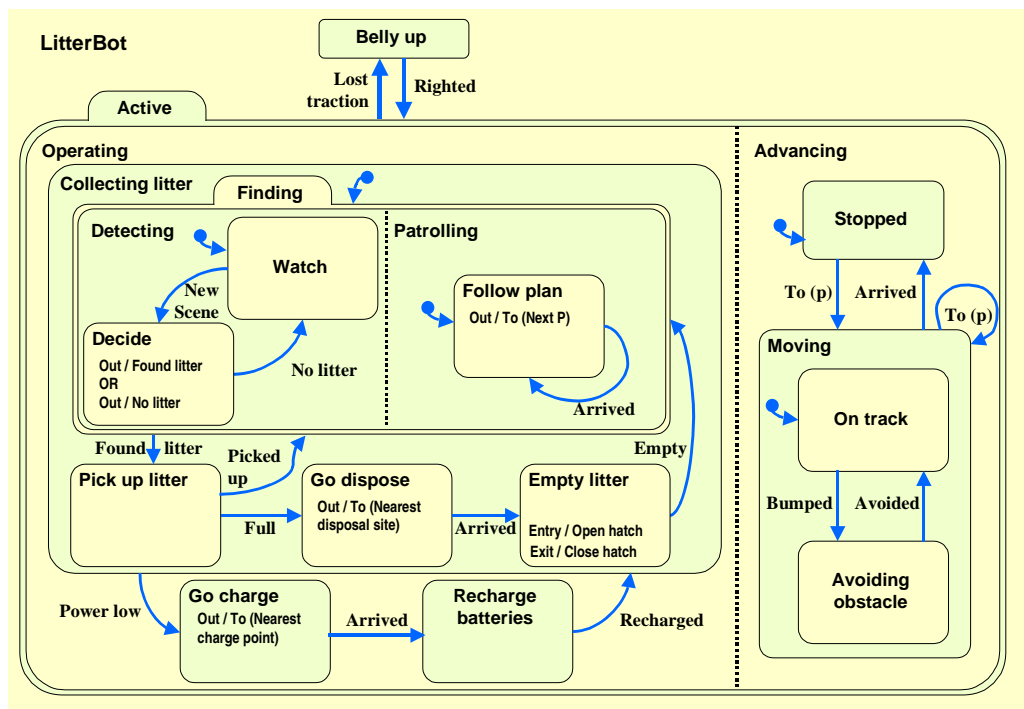


Figure 12 *A more detailed specification, leaving original in tact*

We did not attempt a state transition diagram equivalent of the behaviour specified in Figure 12. The additional state in Advancing would cause a multiplication of states not by two, as with the original Stopped and Moving, but by three. The additional parallelism in Finding would multiply the number of states further. Each resulting state would need transitions to Go Charge and Belly Up, as well as more local transitions. The result would be humanly intractable and probably involve a complete change of layout. In the state chart, by contrast, the additional behaviour is just a refinement of what was already there, the impact on the original specification is localised and minimal, and the result remains comprehensible and is easily compared with the earlier iteration.

Summary of State Charts

State charts are a superior to state transition diagrams as a formalism for specifying event driven behaviour. This derives from two innovation, pioneered by Harel and present in all subsequent variants, namely hierarchy and parallelism.

- Hierarchy delivers
 - Overview and drill-down
 - Minimal number of transition arrows
 - Scoping of transitions
 - Encapsulation of behaviours
 - Progressive refinement within a stable context.
- Parallelism delivers
 - Comprehensibility through the prevention of state and transition explosion
 - Extensibility, since parallel behaviours can be added without destroying the original specification

State charts provide an intuitive and elegant notation for hierarchies of states and for parallel live cycles (either of sub-systems or of co-existing roles of a system or sub-system). Consequently, state charts have gained almost universal acceptance, especially since their incorporation into UML.

However, the advantages of state charts are not carried forward into program code. The next section examines why implementing the specification in a state chart poses challenges.

Implementing a state chart

When an input arrives, the software has to determine the type of the input and the current state of the system. It must then identify the requisite transition, execute any relevant code and establish resulting state.

A state table or matrix is one of two common ways in which state charts are implemented. Effectively this flattens the hierarchic state chart to a finite state machine, multiplying out any parallel states, much like a state transition diagram. A state table quickly gets quite large. Dealing with the possibility of more than one valid transition (in independent sub-systems) makes it larger. Optimising it (sparse matrix compression) reduces the size, but adds machine cycles to the lookup. Because of the size of the table and the absence of clear structuring devices (to reflect the hierarchy and parallel behaviours) a state-table implementation is hard for engineers to debug.

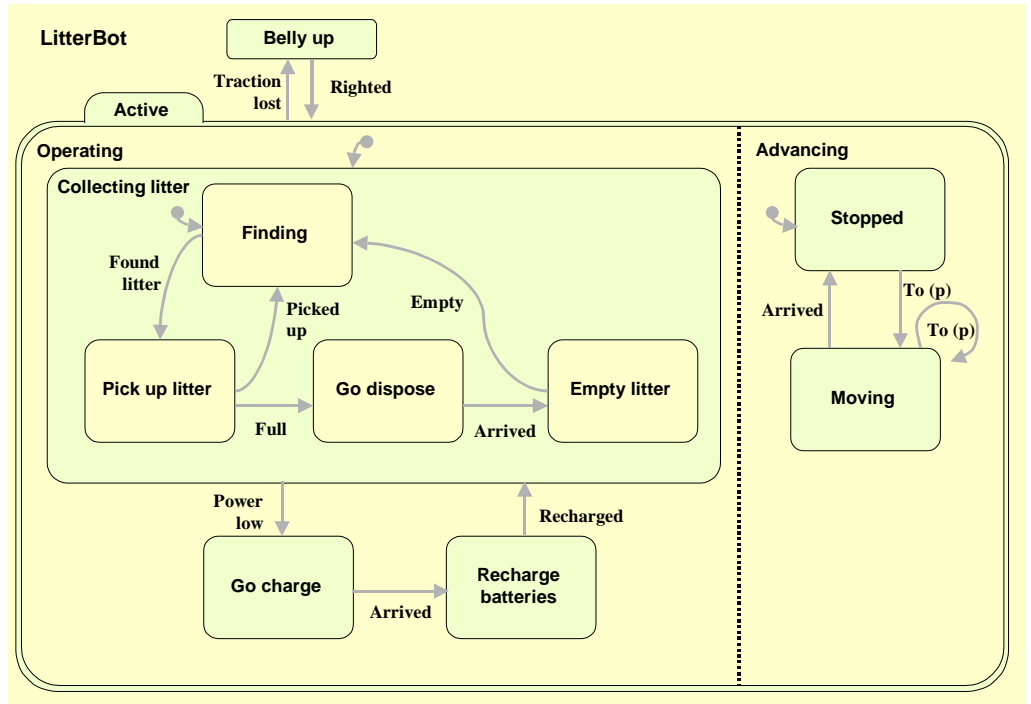
The other common implementation of state charts is as a hierarchy of branching statements (if then else; case of; switch) to test the input type and state in all valid combinations. A moderately complex state chart will give rise to a deeply nested hierarchy of branching statements. To deal with parallel states again requires them to be multiplied out. This destroys any resemblance the hierarchy of tests might have had with the hierarchy of states in the chart. It also increases the number of tests, resulting in a big and slow program. Nested branching statements are notoriously hard to write and debug – the bigger and deeper they are, the harder and more error prone they get.

While the state matrix allows for fast lookups, it is very large (and compression slows down the lookups). The deeply nested conditional statement may be parsimonious with memory, but takes many machine cycles to traverse. Neither device is readily comprehensible by an engineer.

Only GLO Ltd have an implementation that is both small and fast, using neither a matrix, nor a deeply nested conditional statement. It allows a one-for-one mapping between the code and the statechart, allowing either to be debugged and corrected.

What needs to be implemented

The hierarchy of states and the parallel behaviours, so elegantly expressed in the state chart formalism, also make it hard to implement. We will use the LitterBot example to set out what is required of an implementation. For ease of exposition, we will revert to an earlier evolution of LitterBot's specification, repeated here as Figure 13.



3.

Figure 13 *The LitterBot behaviour to be implemented*

This statechart represents a hierarchy of states, with parallel as well as mutually exclusive branches. Figure 14 makes this structure explicit as a tree. A double border around the parent node (Active) indicates parallelism. This is further emphasised by the horizontal line from which the parallel children (Operating and Advancing) are suspended.

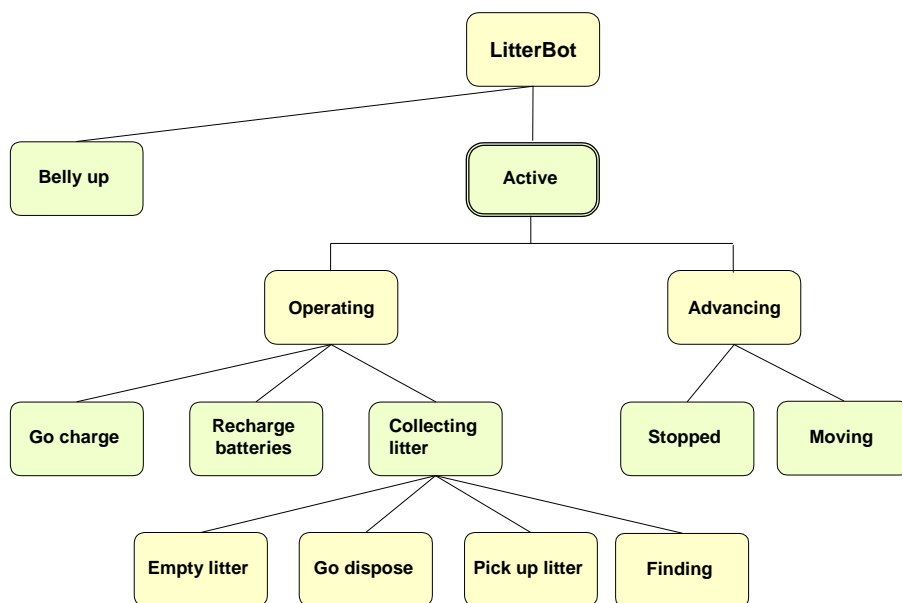


Figure 14 *The tree representing the structure of the state chart*

We can reflect in this tree the events that can be handled in each state. The Arrived event, for example, can be handled in three distinct places. (Figure 15.)

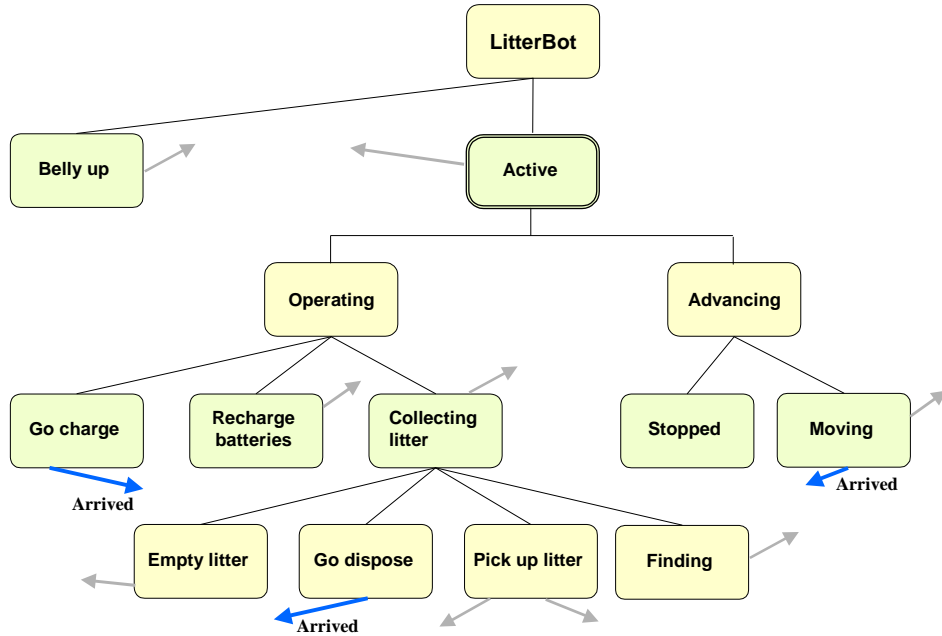


Figure 15 The Arrived event can cause three different transitions

The implementation must match the event with the *current* state in order to determine what (if anything) should happen next. In Figure 16 the statechart is marked to show the current state: Litterbot is moving to the nearest disposal chute. Events that can be handled (including Arrived) are highlighted.

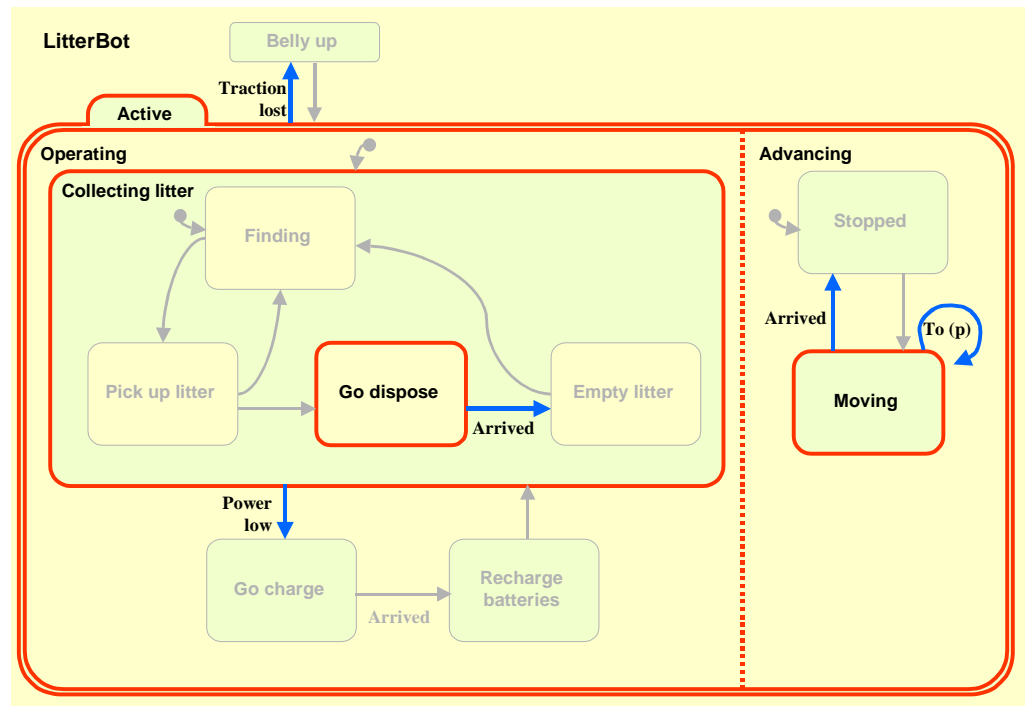


Figure 16 *The current state of LitterBot*

Unlike states in state transition diagrams, where each state corresponds to precisely one node, states in a state chart are complex. Figure 17 shows show the composition of the *current* state. Valid events can occur anywhere in the highlighted hierarchy.

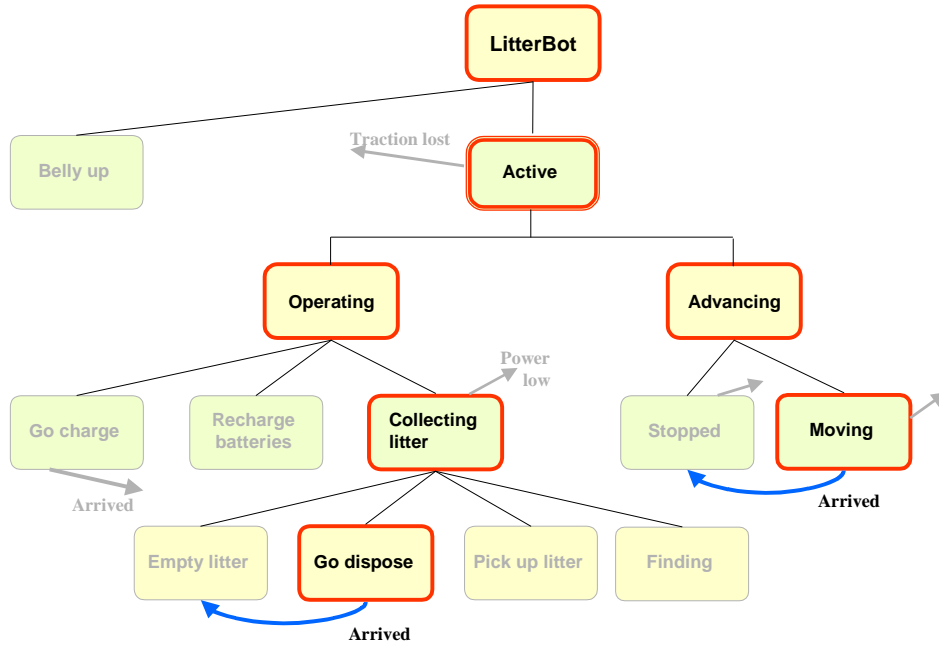


Figure 17 *Structure of the current state*

The implementation has to find the most specific valid state for an event. There may be more than one, as is the case here: both Moving and Go Dispose can (and do) handle the Arrived event. Figure 18 shows the resulting state.

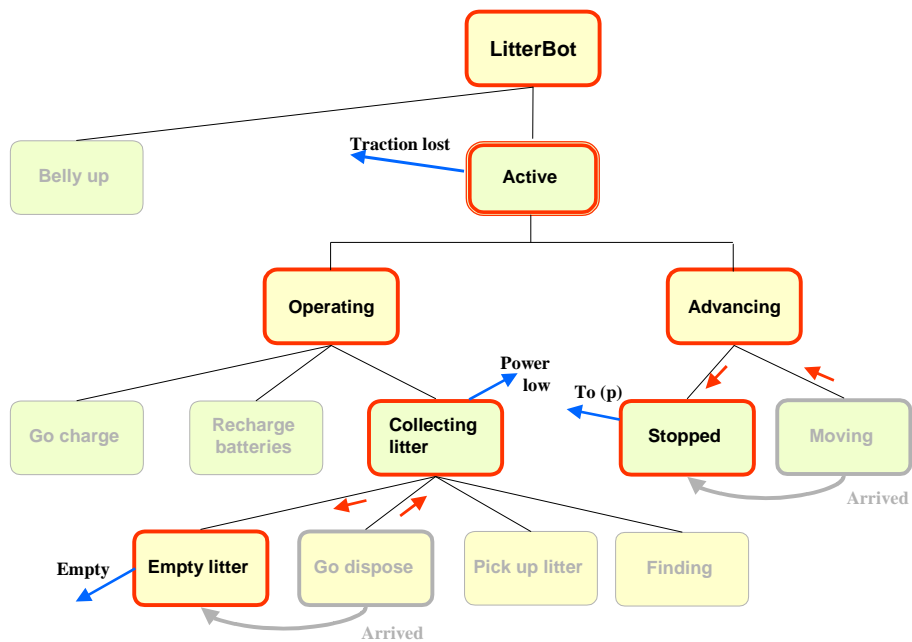


Figure 18 *Structure of the new state after Arrived occurred*

In this new state, LitterBot is ready to deal with four different events.

To complete this transition, the implementation must execute all relevant transition and exit code, assert the new (hierarchic) state and execute entry code.

Conceptually, you go up the hierarchy to the parent in common with the new state. (Advancing and Collecting are the relevant parents in parallel branches.). As you exit states you have to execute their exit code. From the parent in common, you then go down to the sub-state targeted by the transition, executing entry code as you go down and carrying on down if there are any primary states indicated. You do so for each relevant parallel branch. (There is no ordering amongst parallel branches.) You do not exit or enter the parent(s) in common and so do not execute their entry or exit code.

The state achieved in Figure 18 is marked on the state chart in Figure 19. Litterbot is stopped at a disposal chute. This chart has been augmented to show an entry and an exit action. Litterbot's hatch will now open. Possible transitions are highlighted.

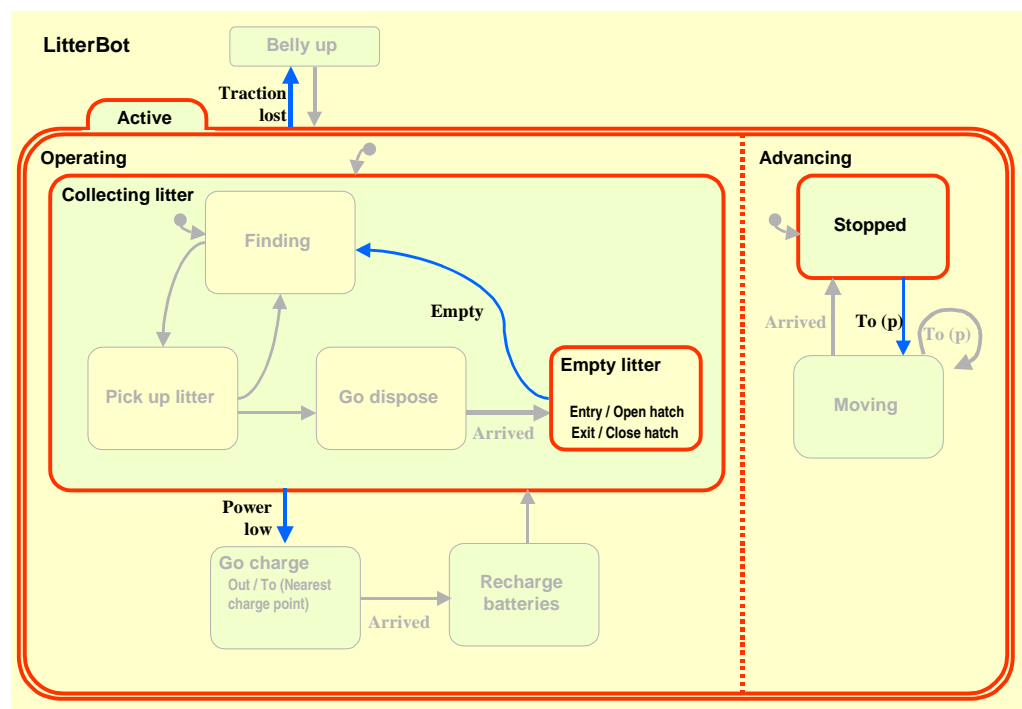


Figure 19 *The new state in plan view*

Let's say the Power Low event occurred. Figure 20 shows LitterBot's resulting state: it will go to the nearest charger. But this exits Collecting Litter and hence Empty litter. Before it goes anywhere, Litterbot must close its hatch.

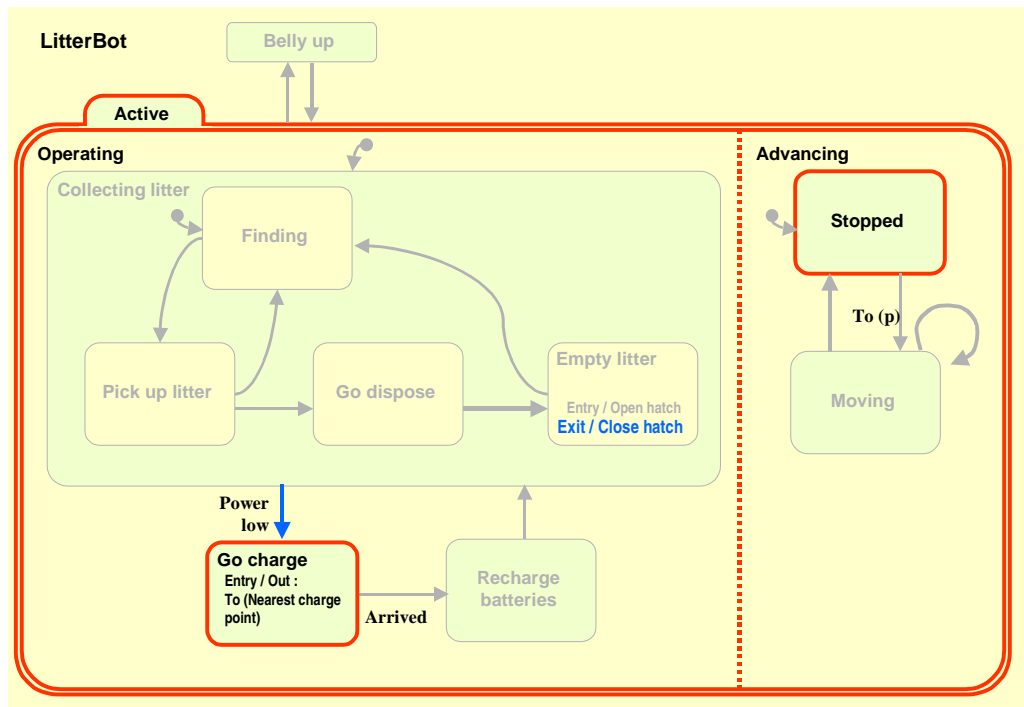


Figure 20 The state after a Power Low event

Again, in order to complete the transition to this new state, the implementation has to identify the relevant state/event combination to find the transition. It must execute *exit* actions (from the lowest up), assert the new hierarchical state and execute any entry code (from the highest newly enabled state down to the lowest primary state). Figure 21 visualises the transition as a move from one highlighted tree to another, with the highlights marking the old and new states.

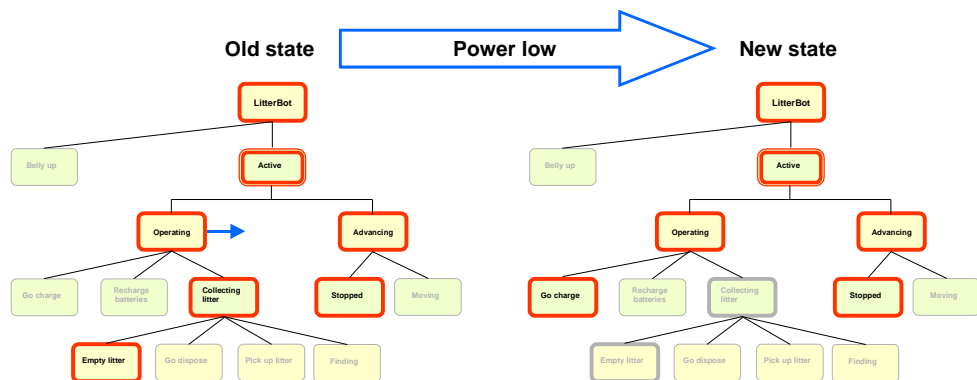


Figure 21

It is clear from this exposition that identifying the correct transition for an event is not trivial; that the transition involves a varying amount of work that must be executed in a certain order; and that asserting the resulting state must establish the correct hierarchic structure of that state.

StateLogics - the solution

4. The GLO algorithm represents the tree structure of the current state in a compact data structure. A number of highly efficient operations are defined on this data structure to match the valid states for a given event with the current state of the system at all levels of the hierarchy. When a match is found, the algorithm calculates a pointer into a simple table of transitions, where the code fragments to be executed are listed and the resulting state is indicated.
5. The hard work is done during a pre-processing stage, akin to compilation. This parses the state chart into the required data structure and constructs the table of transitions.
6. GLO's innovation takes state charts and implements the specified behaviour as a one-for-one mapping into code. The clear and concise structure of the state chart is thus echoed in the code, without a massive and incomprehensible state table or deeply nested conditionals.
7. The implementation can therefore be debugged, corrected and enhanced in program code that reflects the clarity of the specification. This improves quality, gets you to market more quickly and lets you evolve your product lines more rapidly: throughout the entire product lifecycle you stay at the analysis level.
8. This can be achieved today, without the need for code generation. Our algorithm is encapsulated in a library of C macros. Your software engineers can use these immediately and gain the benefit. Even if you draw a state chart using pencil and paper – you can then structure your C or C++ code precisely like the state chart you drew. There is an immediate gain in transparency and maintainability. You describe the state structure in macros and specify a function for any transition, entry and exit code. (Entry and exit are similar to the C++ constructor and destructor notions). Because of the close structural mapping, you can do round-trip engineering (from the code back to a pencil and paper drawing).

Your next step:

9. **StateLogics** - generates high quality software code directly from state chart specifications and has been implemented as a set of macros and generated header files within C+. It is of general applicability and of *particular* interest to the software engineering community engaged in building embedded and real-time systems.

- **Development partner** - work with us to incorporate, adapt and extend our innovation, in particular its integration with established or GLO developed tools. We look to the partner to contribute development resources, in return for significant influence on priorities and earliest access to any results.
- **Investors** – be enthused by amazing upside and patent protected technologies. GLO will consider third party investment from qualified institutions. Our commercial model has low running costs and is highly scalable with limited resources. It is a common and effective model in the software industry and, in addition to license fees; it generates maintenance, consulting and bespoke development revenues.
- **Academics** – get involved with these breakthrough technologies. We require technology validation and research assistance on a variety of exciting software technologies. Our current interest is in establishing research partnerships with academic institutions.

Call us now...

10.

William M Pearson Chief Technical Officer bill@gloltd.com +44 781 425 5901	Michael Tanaka Chief Executive Officer michael@gloltd.com +44 776 777 6141
--	--

11.

12.