BetterState Pro 3.0

An Introduction to Designing with Extended State Diagrams

First Time User Tutorial

R-Active Concepts, Inc.
20654 Gardenside Circle
Cupertino, CA. 95014-5081
phone: (408) 252- 2808
fax: (408) 438-7684
email addresses:
Technical Assistance, doron@ractive.com
Sales Assistance, bweeks@ractive.com

Printed in the United States
Ver 2.0 October 1994, Rev 2.1 December 1994, Rev 3.0 July 1995

Disclaimer Notice:

R-Active Concepts Inc. and Co-Active Concepts do no guarantee or imply a guaranteed behavior of the code generated by their code generators. You are cautioned that such a behavior depends on the semantics of the underlying language and the behavior of the compiler, interpreter, or synthesis tool used to realize the code BetterState products generate, on a lower level platform, Operating system or hardware. For example, the behavior of a BetterState diagram for which C code is generated, depends on the semantics of the C language and the particular C compiler used, and the particular behavioral phenomena induced by the operating system under which this code will execute.

To be more explicit: your design is by no means verified until you complete thorough testing of the final application running on the target platform or hardware. For example, you should test the C code BetterState generates after compiling it into machine code, and running it within the final application on the target platform or hardware, using the operating system you will use in the field (if any).

BetterState Pro Tutorial
Table of Contents

Tutorial Overview

The purpose of this tutorial is to provide the reader with a quick overview of the BetterState Pro design environment and the design methods it includes. There are four parts to this tutorial. Each part builds upon the information provided in the preceding sections.

Part One *provides an easy to understand overview of the BetterState design environment. This section covers how BetterState Pro operates, what's required from you and what BetterState will do for you.*

Part Two *is for those who are new to designing with State Diagrams or those who just need a quick refresher on state diagramming design techniques. In this section we'll spend a little more time on the fundamentals of this popular design method, describing how to use it and working through some simple examples.*

Part Three *is our* ***Quick Start*** *section for those of you who just can't wait to get started. If you have a good understanding of state diagram design methods and you want to get a feel of how BetterState works, then Part Three is the place to start. In Part Three we'll get you familiar with the BetterState design area and show you where the various design tools are located. You will work through a very simple design, generate code and be on your way.*

Part Four *of this tutorial focuses exclusively on BS4VisualBasic. It will go through a simple design, covering every detail from initial design to interfacing with your Visual Basic front end.*

Part Five *covers BetterStates unique and powerful Visual Debugging capabilities.*

 In your BetterState directory there is an examples\ simple sub directory with example programs that can be used with this tutorial. For example, the telephon.dsd (this is a BetterState diagram) resides in the examples \ simple\ telephon directory. There is also a telephon.mak file for the front end of telephone example. By opening the telephon.dsd in BS4VisualBasic and the telephone front end design (either via Visual Basic or directly by double clicking on telephon.exe) simultaneously, you'll be able to see BetterStates animation taking place.

If you would like a little extra help, there are **Hot Spot Tutorials for Extended State Diagrams** and **VB 4 VisualBasic** accessible from the Help Menu. The context sensitive Help can be opened by pressing the right mouse button or by pressing Control - F1.

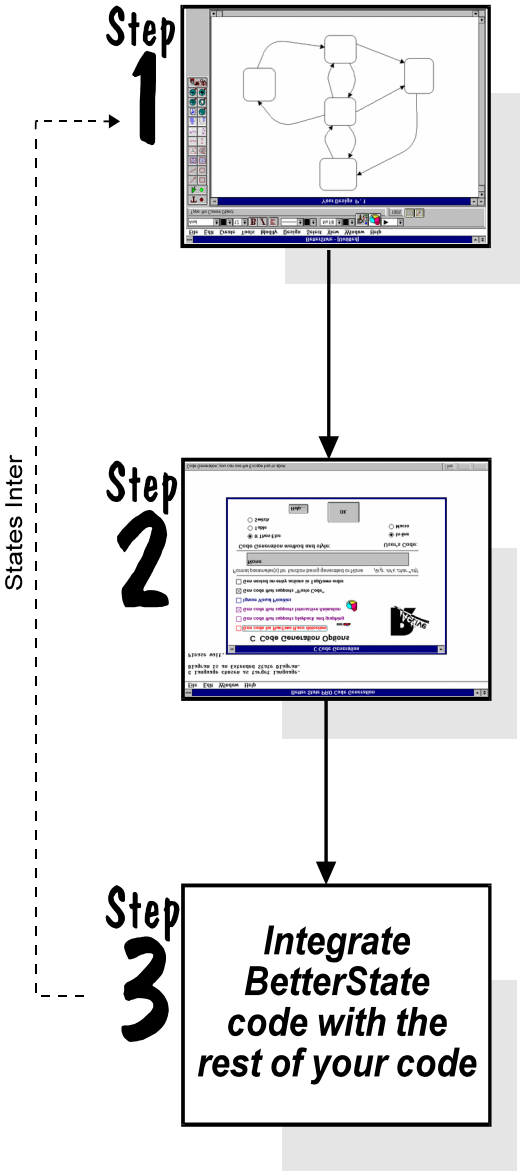
Part One, *Designing with BetterState Pro*

BetterState Pro is extremely easy to use. In fact, in less than 15 minutes you can be generating code for your first design. There are three simple steps between your conceptual idea and the code required to implement it. The following is a quick overview of how to design with BetterState. Remember, if you have ever hand drawn a state transition diagram you already know how to use BetterState.

Step 1. Draw your design. To start your design go to the File Menu and open a New Project. When the Chart Dialog box appears, select the programming language you will be using, the type of diagrams you want to create, name your project and start to draw your design. If you need assistance, BetterState has an extensive HELP system with HOT Spot Tutorials and Context Sensitive Help text.

Step 2. Generating code with BetterState is simple. Just select the Code Generation option from the Tools Menu and select style of code you want to generate. Once you press OK, everything else is automatic. After your code is generated you'll be asked to give it a file name and that's it.

Step 3. Integrating the BetterState generated code is straight forward. How the code is integrated depends upon the language you are using but basically all you need to do is tell your main code when you want to call the BetterState generated code. *Go to Part Three of this tutorial for language specific examples of how to use BetterState generated code.*


Step 1
Step 2
Step 3
Integrate BetterState code with the rest of your code
States Inter

Part Two, An Introduction to State Diagram Design Methods

Virtually all software can be thought of as a state machine specification. A state machine is a common term used to describe what a system should be doing under certain conditions and the order they should be done in. Just like the statements a programmer writes will be executed in a certain order and each statement specifies the changes in the computer's state.

The concept of describing systems behavior with state machines using their visual counterpart, state diagrams, is a very popular design method. State diagrams can be used to model the behavior of systems and sub systems ranging from simple business applications to the most sophisticated communications protocol.

There are two categories of system behavior. The two categories are Transformational and Reactive. Understanding the differences between these two types of system behavior is important to learning the basics of state diagrams.

Transformational (sub) Systems

Transformational (sub) Systems are those which have all inputs ready when invoked and the outputs are produced after a certain computation period; see Figure 2.

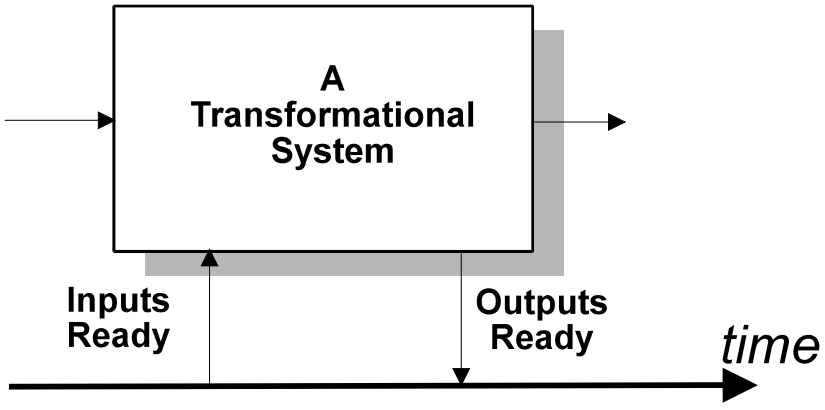


Figure 2 A simple Transformation System

Examples of Transformational Systems are data acquisition systems and voice-compression systems (software and hardware) or even a simple procedure which calculates the square root of an input. Top-down decomposition is a natural design methodology for transformational systems because it breaks down complex input/output (functional) relationships into simpler, more manageable ones. Similarly, conventional programming and system-level specification languages are transformational oriented and cater to top-down functional design.

Reactive Systems

A well understood reactive system is a traffic-light controller. It never has all its inputs ready--the inputs arrive in endless and perhaps unexpected sequences. It is virtually impossible to write a transformational program that implements a controller such as this. In fact, most controllers are by definition reactive, not transformational, with application domains ranging from process control, military, aerospace, and automotive applications to DSP, ASIC design, medical electronics, and similar embedded systems.

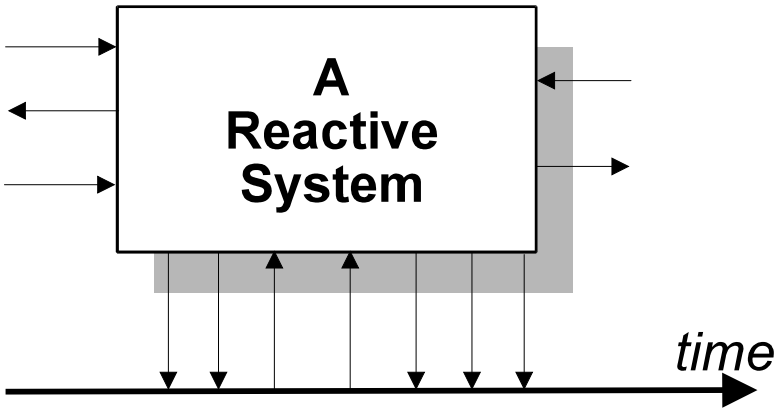


Figure 3 A simple Reactive System

Just about every system has a reactive component, because a system is seldom isolated from its environment. On the contrary, the reason the system exists is typically to collaborate or interact with some entity or entities in its environment. Such collaboration is done by sending, receiving, recognizing and subjecting sequences of symbols--a reactive behavior.

State Diagrams (as well as Statecharts and Petri Nets) relate to Reactive Systems. Reactive Systems are BetterStates forte. Throughout our documentation we often call a reactive subsystem a "controller". This should not be confused with classical control theory.

State Diagramming Basics

Types of State Diagrams

Before we get started discussing state diagramming design methods it is important to note that there are two commonly used types of state diagrams; State Diagrams and Extended State Diagrams (the second being an extension of the first). We will discuss both types. BS4VisualBasic and BetterState Pro supports both types, however our focus will be on Extended State Diagrams.

Finite state machines (FSMs) and their diagrammatic counterpart, state diagrams have traditionally been used to specify and design reactive (sub)-systems. They are well known, well accepted, highly visual and intuitive. Their ability to describe finite and infinite sequences, combined with their visual appeal, has made FSMs one of the most commonly accepted formalisms in the electronic industry.

Traditional State Diagrams

State diagrams are easier to design, comprehend, modify and document than the corresponding textual approach. Traditional State Diagrams haven't changed much over the past years and suffer from limitations when applied to today's reactive applications. Later on in this section you'll discover how Extended State Diagrams resolve these limitations. Some of the limitations of Traditional State Diagrams include:

☑ They are **flat**, they do not cater to top-down design and information hiding. Moreover, top-down design concepts require interactive software to enable the user to manipulate and browse through complex designs.

☑ Traditional FSMs are purely sequential, whereas applications are not. Modern reactive subsystems (which we call controllers) need to react to signals to and from a plurality of entities in their environment. Consider an answering machine controller specified to cater to a "second call waiting" situation in addition to the "first caller." A conventional FSM needs to account for all possible combinations of states catering to the first and second callers, which leads to the well-known state-blowup phenomenon.

Extended State Diagrams

Compensating for these limitations are Extended State Diagrams designed by David Harel and described in his paper "Statecharts: a Visual Approach to Complex Systems" published in Science of Computer Programming (1987). While addressing the hierarchy, concurrence, priorities, and synchronization within state diagrams, Extended State Diagrams retain the visual and intuitive appeal inherent to Finite State diagrams.

The following text will discuss basic design methods that apply to both traditional and extended state diagrams. The discussion in the latter part of this section discusses design methods unique to extended state diagrams.

State Diagram Notation

The major components of a state diagram are states and arrows representing state changes or transitions. There are a number of different types of symbols used to graphically represent a state. The symbols are circles, rectangles and rectangles with rounded corners. BetterState uses rectangular boxes with rounded corners to graphically represent a system state.

System States

Webster's New World Dictionary defines a "state" as follows:

"A set of circumstances or attributes characterizing a person or thing at a given time; way or form of being;..."

State Diagrams are used to graphically show the states and the interaction between states at any given time. The simplest state diagram will have a source state and a transition to a target state.

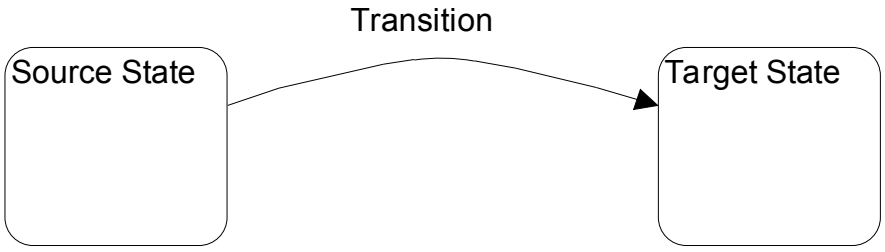


Figure 4 A simple state diagram

An observable state that the system is in can only correspond to a period of time when:

- it is waiting for something in the external environment to occur or

- it is waiting for a current activity in the environment (like mixing, washing, filling, calculating) to change to some other activity.

This does not mean that our systems are incapable of taking action. We will discuss actions later on in this section. However, it is important to note that actions are not the same as states which represent observable conditions that the system can be in. Thus a state must represent some behavior in the system which is observable and that lasts for some finite period of time.

Note: In BetterState, when you have finished drawing a state a dialog box appears. In this dialog box you can enter the states name, the type of action required (more on this topic later) and select other options relative to the state. This topic is discussed in detail in the BetterState Pro Users Manual, Chapter 4.

Changing States

A system typically has rules governing its behavior. It's these rules that specify when the system will change from one state to another state. A valid state change is called a transition. A transition will connect relevant pairs of states. The notation for a transition is a line that ends with an arrowhead. The arrow head points in the direction of the transition. The state diagram in Figure 5 shows that the system can change from state 1 to state 2. It also shows that when the system is in state 2 it can change to either state 3 or back to state 1. However according to this state diagram the system cannot change from state 1 directly to state 3. On the other hand the diagram tells us that the system can change directly from state 3 back to state 1. Note that state 2 has two successor states (states 1 and 3). This is common in state diagrams. Indeed any one state might lead to any number of successor states, but usually under different circumstances.

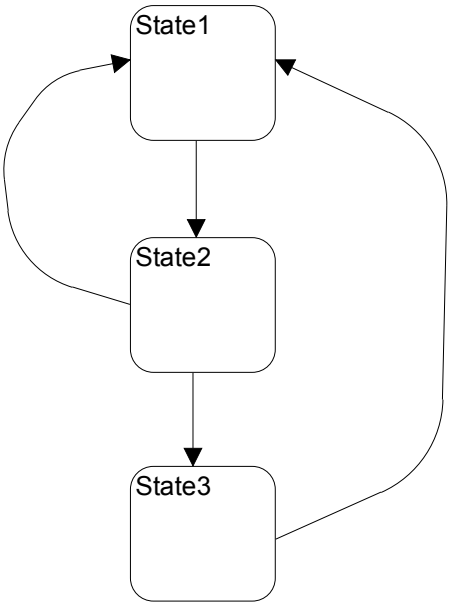


Figure 5 Simple three state FSM

Figure 6 gives us some interesting information about the time dependent behavior of a system. However it leaves out a very important element of our system. That is, what is the systems initial states? Indeed Figure 6 is a model of a system that has been active forever and will continue to be active forever. Realizable systems must start operating somewhere.

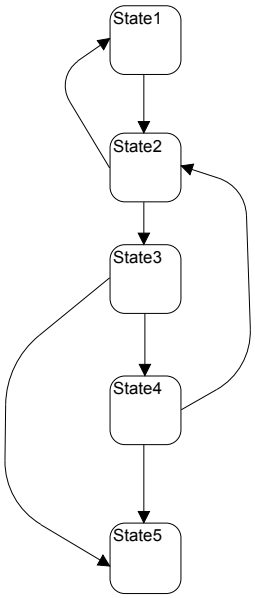


Figure 6 illustrates a simple state diagram without a specific start state.

With BetterState an initial or start state is identified with a default symbol. The default symbol is a D with a circle around it. Figure 6.1 shows the same state diagram (Figure 6) with a specified Default state.

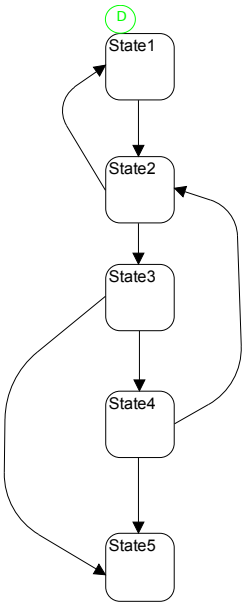


Figure 6.1 Simple state diagram with specified start state.

Note: A simple FSM might only have a single initial state. However, extended state diagrams, due to hierarchy and concurrency, can have multiple starting states and multiple state machines running at any given time.

Conditions and Actions

To make our state diagram complete there are two additional options you may want to add:

- the (optional) conditions that cause the change of the state and
- the (optional) action that the system takes when it changes states.

> **Note:** In the case of Visual Basic, it is a Visual Basic event that causes the change of state. *More information on this subject is located in the BS4VisualBasic Users Manual.*

A transition condition is some condition in the external environment that the system is capable of detecting. It might be a signal, an interrupt, or the arrival of a packet of data. This can be anything that will cause the system to change from one state *waiting for x* to a new state of *waiting for y*, or carrying out *activity x* to carry out *activity y*.

As part of the change of states, the systems can take one or more actions. It will produce an output, display a message on the users terminal, carry out a calculation, and so on. Actions shown on the state diagram are either responses sent back to the external environment or they are calculations whose results are remembered by the system in order to respond to some future event.

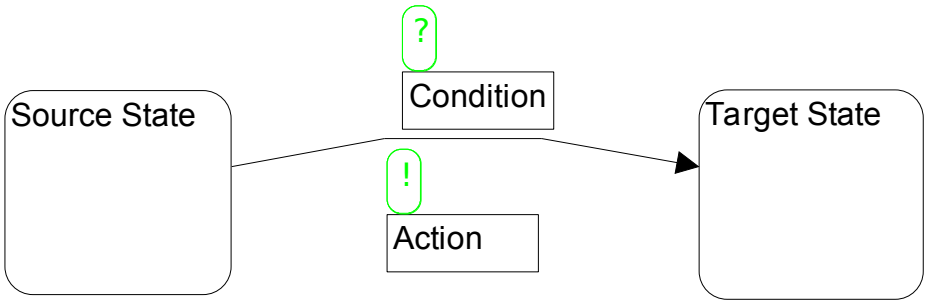


Figure 7 Simple state diagram with specified Conditions and Actions.

> **Note:** In BetterState, when you are finished placing a transition between two states, a dialog box will appear. In this dialog box you define the conditions for the transition and what action is required. This topic is discussed in detail in the BetterState Pro Users Manual, chapter 4.

Your FIRST State Diagram design

You now have enough knowledge of state diagram design methods to model the behavior of our first system. Our first design is going to be challenging. **We are going to design an Ice Cream Stand Logic that detects the sale of 3 ice cream cones.** You can draw this design with pencil and paper so you can get a feel of how it was done in the old days or you can jump right in and use BetterState.

> Please keep in mind that the purpose of Part Two of this tutorial is to introduce you to designing with extended state diagrams. As such we spend very little time in this section on how to use BetterState. This topic is discussed in detail in Part Three. If you want to enter this design using BetterState we suggest you review Part Three which will walk you through a similar design example.

In this simple example, you might start by first drawing the initial state called **Start**. Then draw a second state and name it **One Sold**. The state **One Sold** represents the state your system will be in after you have sold one cone. Now you need to traverse from **Start** to **One Sold**. However, you don't want your system to make the transition (from Start to One Sold) unless one ice cream cone is sold. So the condition to transition from **Start** to **One Sold** is ***Cone Sold***. This process continues until three cones are sold. In this example we are not using actions but we could. We could add an action that records the sale in the cash register everytime a cone is sold.

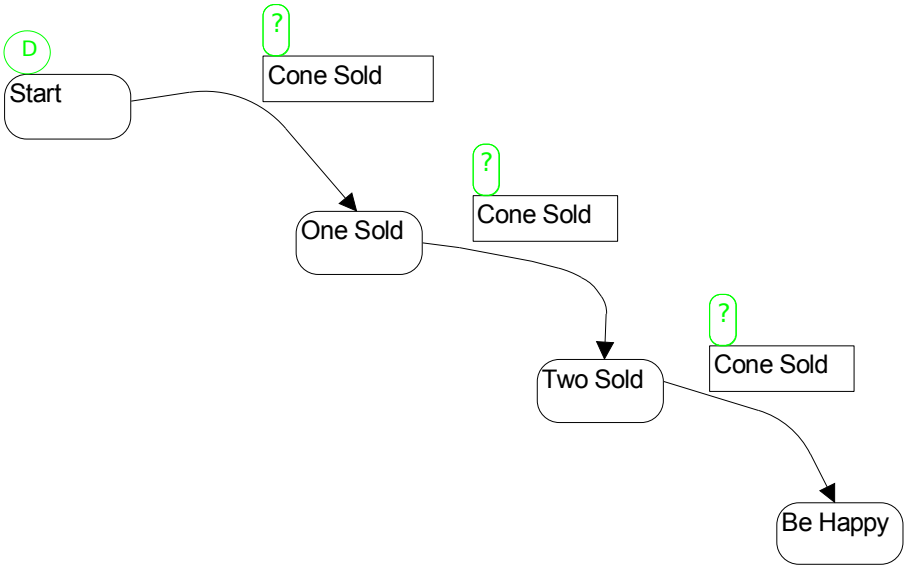


Figure 8 Illustrates the Ice Cream Stand computer system specification.

So far we have discussed the basic principals of state diagrams. These principals hold true for both state diagrams and extended state diagrams. Now the fun part. From this point on Part Two focus on the enhanced capabilities of Extended State Diagrams. We will briefly cover these enhanced capabilities and try to give you a good introductory understanding of the inherent design horsepower of BetterState Pro. For complete detailed discussion on these and other Extended State Diagram capabilities, please see the BetterState Pro Users Manual.

Design Hierarchy

In a complex system, there could be dozens of distinct system states. It would be difficult if not impossible to show all of them is a single diagram. Hierarchy is a well accepted approach for designing and managing complex designs. Hierarchy is used to group sets of states together. Hierarchy helps the human abstraction process and is generally accepted as a basic feature of modern computer software.

In BetterState hierarchy is drawn by placing states within states. Figure 9 shows an example of what we are talking about. In this example, hierarchy is used to group sets of states together. High Level transitions have an existential meaning, in this example, the transition is traversed if the High Level State is **either** State1 **or** State2. If it's State2 it is **either** State3 **or** State4.

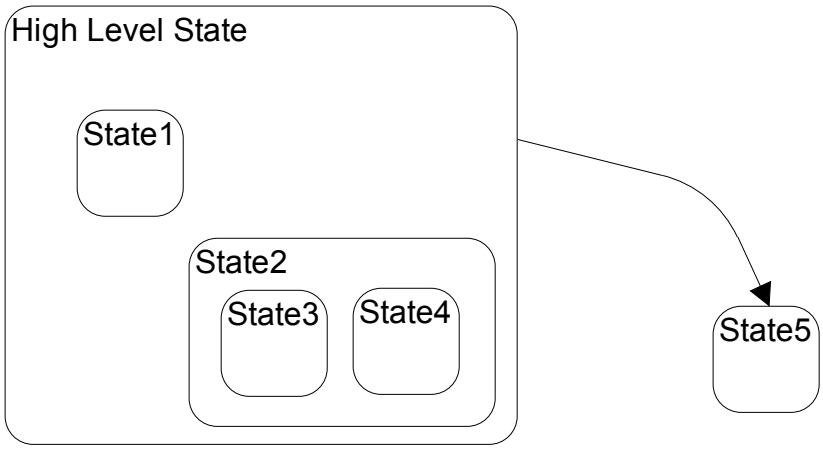


Figure 9 Shows a hierarchical state with 2 sub-states, one of which has 2 additional sub-states.

So how is hierarchy applied in the real world? Let's go back to our Ice Cream Stand Computer and change the specifications. **Now instead of our computer just detecting the sale of 3 ice cream cones, we want it to detect the sale of 3 ice cream cones OR the sale of one or more cones followed by a sale of a container of ice cream.**

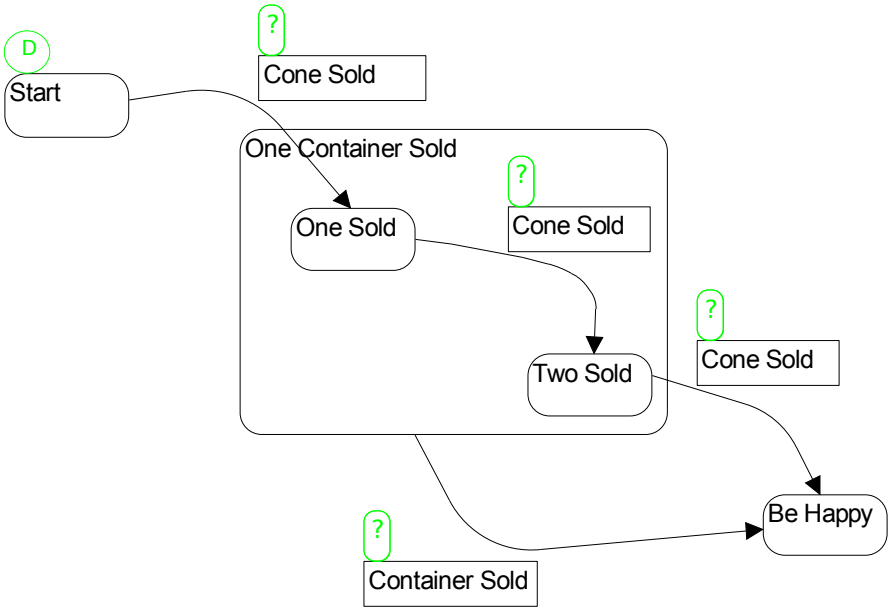


Figure 10 Illustrates the design of our modified specification using hierarchy.

By simply placing a new state around states **One Sold** and **Two Sold** we have created a high level state that in this example represents the sale of a container of Ice Cream. The condition *Container Sold* is called a High-Level transition. High-level transitions have an existential meaning. In this example, the high level transition is traversed if the High Level State is in either state **One Sold** or state **Two Sold and** the **Container Sold** condition is satisfied. In our example that means if one or more cones are sold followed by a container of ice cream being sold, then transition to the state **Be Happy**.

Independent Threads of Control

Extended State Diagrams also provide a way to capture **independent** sequences of input events. This means that when a state is entered it can fork into two or more threads of control. Each thread of independent control behaves like a state machine on its own right. Hence, these threads of control traverse their transitions in parallel. ***Independence (also called Concurrency) is a very powerful design capability. Please see the BetterState Users Manual, chapter 4 for complete details and more examples of using Concurrency / Independence.***

To illustrate this example, lets expand our ice cream system model to include processing *hot dogs*. In this example, in addition to counting ice cream cones and container of ice cream sold we want to accommodate those customers who ask for *hot dogs*.

Now you could design a new state diagram and have two separate diagrams that might look like Figure 11.

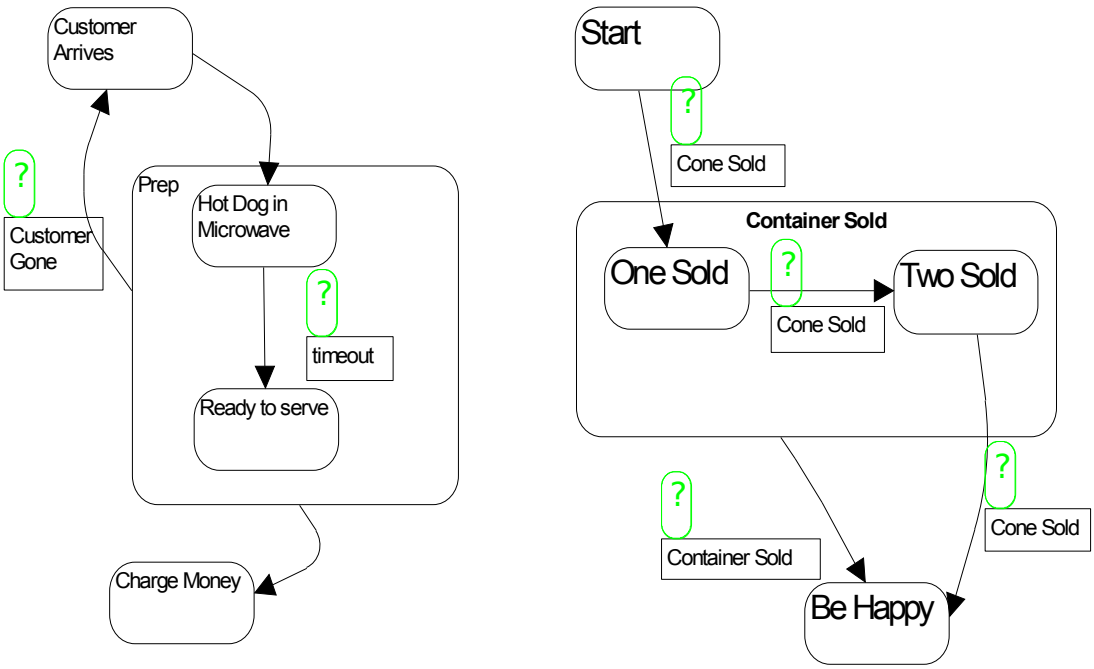


Figure 11 Illustrates the two separate tasks that are performed while at work.

Independence gives you a powerful design tool that can significantly simplify the design illustrated in Figure 11. To design with independence, draw a state and then select the Concurrency / Independence option from the Create Menu. A dialog box appears so you can name your newly created "threads of control". Now you can draw state diagrams within the new thread.

Figure 12 shows our ice cream / hot dog design, now drawn as independent threads of control within the "**Work**" state.

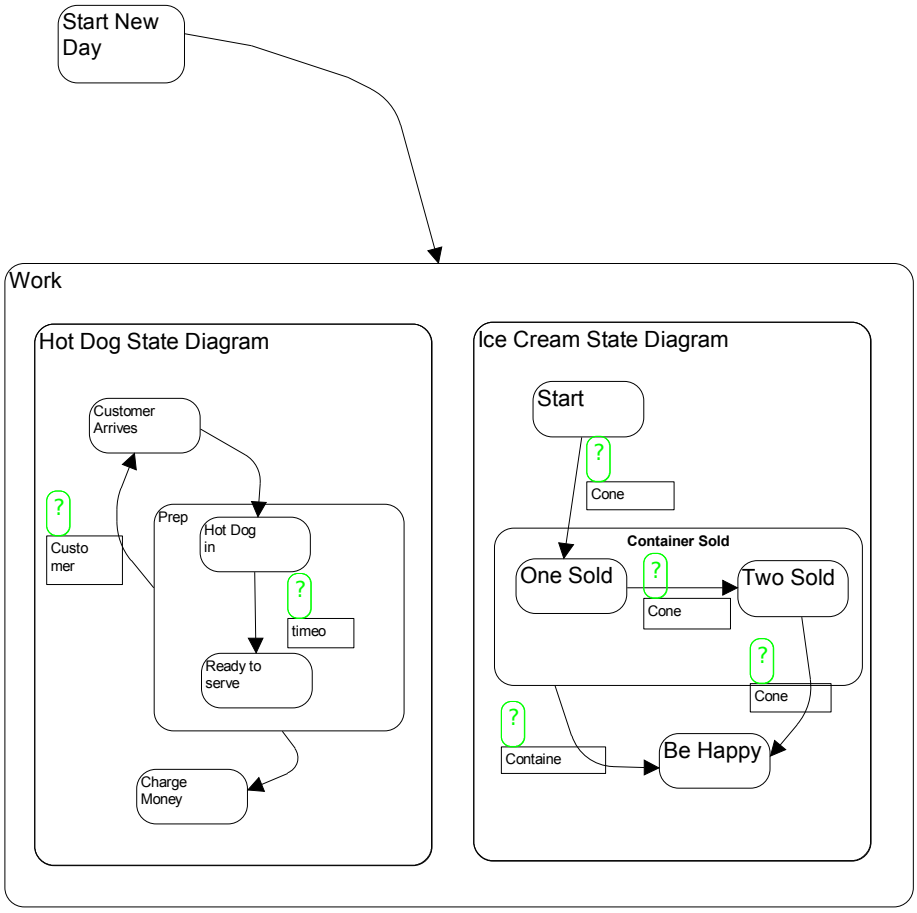


Figure 12 Shows how the two threads of control could be designed.

Notice how much easier it is to read Figure 12. Before (Figure 11) we had two separate state diagrams but there was little information on how (if at all) the diagrams were related. In Figure 12 it is easy to see how the two sub-state diagrams relate. That is, you go to **work** at the start of the new day. While at work, at any given time, you can be selling ice cream and preparing hot dogs. Then, when **it's dark**, you shut down both activities.

With Hierarchy and Independence we can take complex tasks and break them down to smaller more manageable diagrams.

Visual Synchronization

Visual Synchronization gives us a graphical method to show how independent threads interact with each other. Visual synchronization is powerful because no text based message passing mechanism is used; the precise behavior is induced from the diagram alone. For example, let's change our design in Figure 12 to force the **hot dog thread** to charge money when the **ice cream thread** has finished a "**happy**" sale. Figure 13 shows what our new diagram would look like.

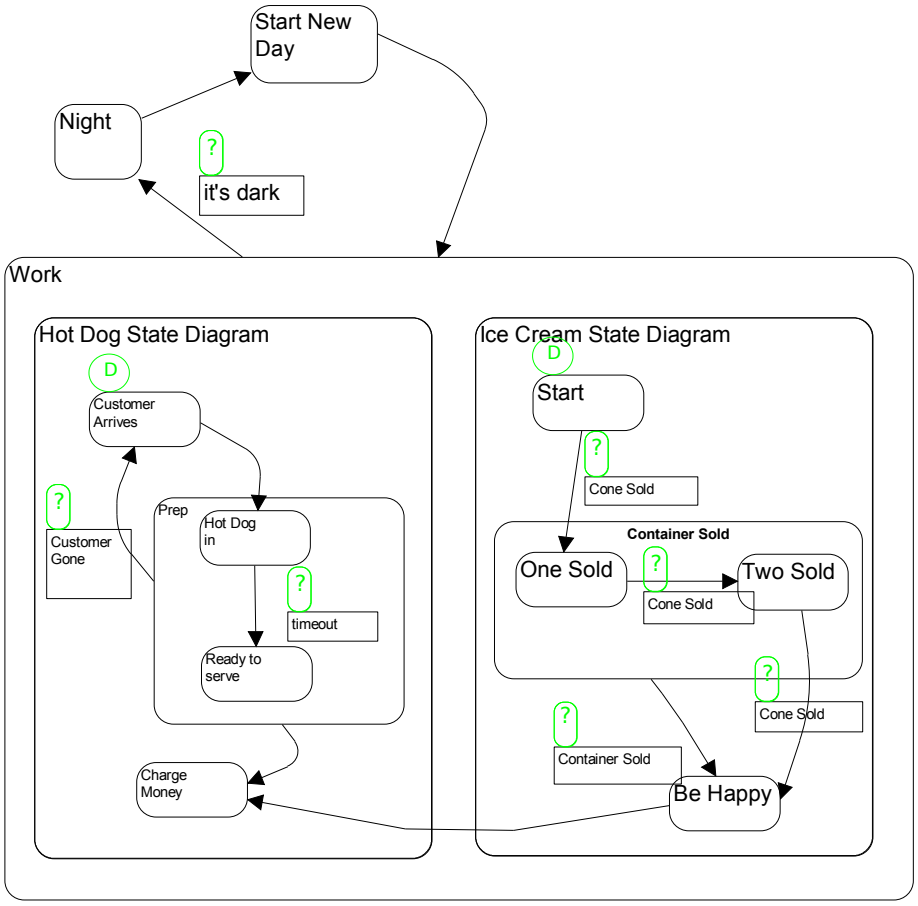


Figure 13 shows how the state ***Be Happy*** is synchronizing state ***Charge Money***.

Visual Switch / Case (decision polygon)

In a traditional Finite State Diagram, a condition, *X* for example, can have only one of two possible outcomes. That is the condition evaluation to True or False. The Visual Switch/ Case capabilities in BetterState functions like a decision polygon in a flow chart and can have one of several outcomes. You're not restricted to one of two outcomes, it can support virtually any number of outcomes.

A Visual Switch / Case visually represents a decision that will be made between two or more resulting target states. The transition leading into the *Visual Switch/Case* specifies the expression that is to be evaluated. Depending on the value of the expression, the diagram will go to one of a number of resulting next states.

At any state, at any level in the Extended State Diagram the designer can draw a *Visual Switch/Case* which is a diagrammatic representation of a C / C++ / Verilog switch, Visual Basic Case Select, and VHDL Case Statements**.**

Figure 14 shows an example of how a Visual Switch/ Case would be used. **We'll use a state diagram with a Visual Switch/ Case to show our behavior on a rainy day.** In this diagram, our behavior will depend upon how much it is raining. If it's raining alot we're going back home. If its raining moderately; we'll just open our umbrella and continue on with our business. If its light rain, we'll just ignore it altogether and be on our way.

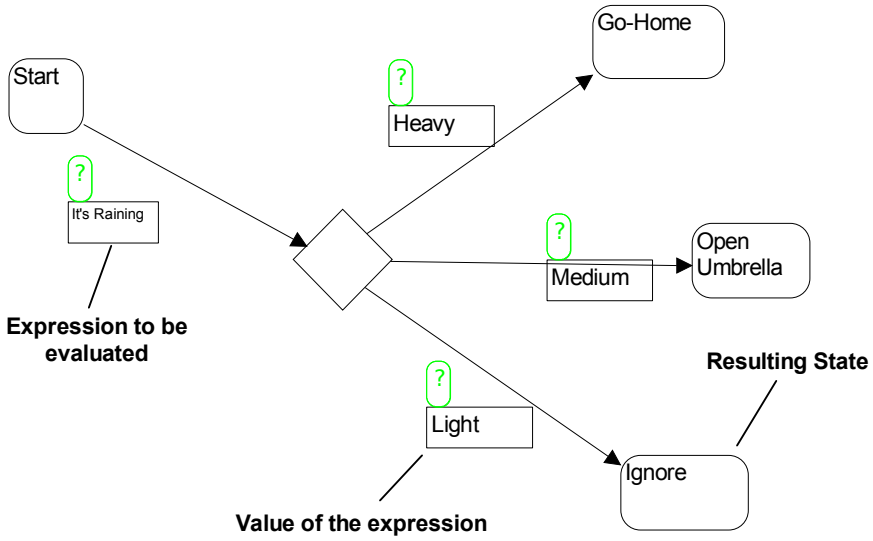


Figure 14 Illustrates an extended state diagram that shows your behavior on a rainy day.

Design Exercise

Here's a fun exercise you can try. Draw a system describing this Telephone Answering Practice: **When I'm busy, I skip 2 out of 3 calls, unless it's my BOSS!!** Again, you can try it the old fashion way with pencil and paper or you can do it the BetterState way.

Here's some helpful hints you might want to consider before you start the exercise.

- What do you do when you are busy?

- What are the possible states in your system ? Start, Waiting For Call, One Call and etc.

 When you finally answer the phone, is it your boss?

Your finished behavioral model regarding the Telephone Answering Practices may look something like the one in Figure 15.

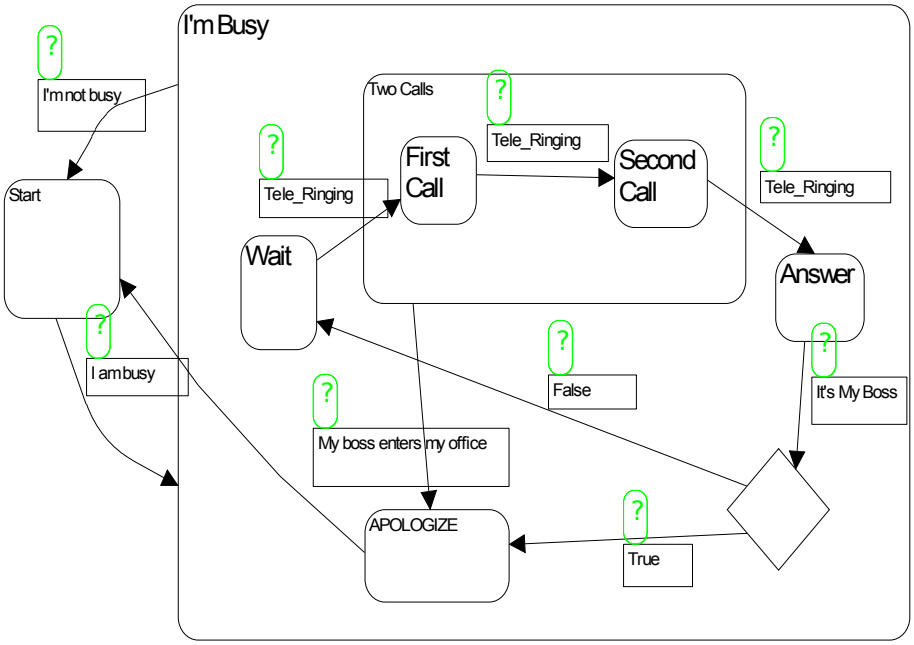


Figure 15 Illustrates one possible design of the Telephone Answering Practices specifications.

Consistence Check Guide Lines

When you have finished building your preliminary state diagram you should carry out the following consistence check guidelines:

 Have all the states been identified? Look at the system closely to see if there is another observable behavior or any other conditions that the system could be in besides the ones you have identified.

 Can you reach all the states? Have you defined any states that do not have paths leading into them?

 Can you exit from all states? There are only two types of states that you cannot, or may not, exit from, one is a History State and the second is a Terminal State. The use of these states is beyond the scope of this tutorial however they are discussed in detail in Chapter 4 of the Users Manual.

 In each state does the system respond properly to all the possible conditions? This is one of the most common errors when building state diagrams. The designer identifies the state changes when normal conditions occur but fails to specify the behavior of the system for an unexpected condition. Sup-

pose the designer has modeled the behavior of a system and expects the user will press a function key on his terminal to cause a change from state 1 to state 2 and a different function key to change from state 2 to state 3. But what if the user presses the same function key twice in a row? Or they press some other key?

Before we end this section of the tutorial, we would like to discuss one last important topic called Critical Regions.

Critical Regions

Critical Regions (drawn as purple boxes around groups of states), overlap two or more threads and place limitations on simultaneous visits to states in the Critical Region. For example, a Critical Region overlapping states s1,s2 in thread A and states s3,s4 in thread B means that if s1 or s2 are visited in thread A, then thread B cannot visit s3 and s4, and visa-versa. Figure 16 illustrates the capabilities of Critical Regions.

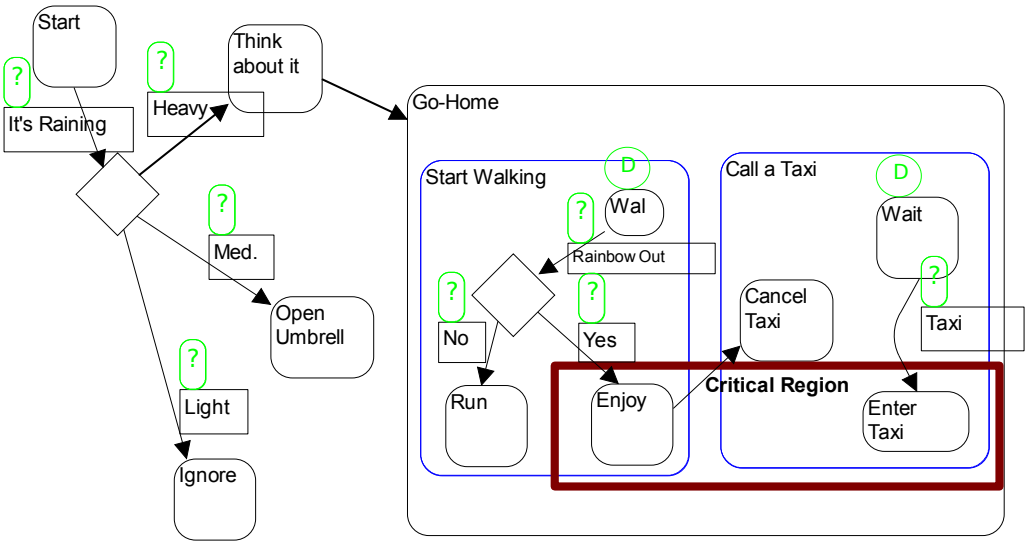


Figure 16 Illustrates how Critical Regions work.

The example in Figure 16 shows all the extended state diagramming capabilities we have discussed so far. In an earlier example we modeled our behavior to rain using the decision polygon. Figure 16 is a model of that behavior, that is, if ***it's raining*** is the expression to be evaluated, ***Heavy*** is one of the evaluations and if it is raining ***Heavy*** we ***Go Home***.

Figure 16 details how we are going home. We're either going to walk or call a cab. Note the independent threads ***Start Walking*** and ***Call a Cab*** are simultaneous.

We have also added a ***Critical Region*** to this design. The Critical Region sets limits on the number of simultaneous states that can be visited within the region. In this example you cannot ***Enjoy*** the walk home while you are riding (***Enter Cab***) in the cab.

So far we have discussed the basics of designing with extended state diagrams. BetterState Pro's extended state diagrams also support History States, InterActive State Animation, and many other important design and debug capabilities. And remember, this is just the design front end of BetterState Pro and BS4VisualBasic. Once your initial design is completed (or anytime during the design process) you can automatically generate code using BetterStates proprietary code generators. For detailed information on Code Generation and BetterStates unique animated debug and analysis capabilities, please see the BetterState Pro Users Manual.

Now you are an expert on designing with Extended State Diagrams. Go on to Part Three and discover how easy BetterState is to use.

Part Three, *BetterState Quick Start*

This section has been developed to help you jump right into diagrammatic programming using BetterState Pro. In this section we assume that you are familiar with extended state diagramming methods.

Throughout these sections be on the look for the **+** symbol. Should you have some questions or want to learn more about a topic, this symbol will point you to the chapters covering this topic in the BetterState Pro's User Manual.

> **Note:** Of course you can by-pass the manual and try BetterStates Hot Spot Tutorials and on-line help (**+** select *Help*).

Now to get started, make sure BetterState is loaded on your computer, **(+** see the section on *Installation* in **Introduction Chapter)**. These are the items you will be working with in Part One.

The Quick Start Project

To show how easy BetterState is to use, we have selected a simple counter for our Quick Start Project.

The specifications for the counter are as follows: **A simple *car counter* that raises a Flag if 3 cars are detected. Assume that the variable, "new_car" is a 1.**

(Obviously this is an example that could be easily hand coded. However for training purposes we picked an example that was easy to understand.)

Getting Started

The first step in using BetterState is to start a project and to select an associated programming language. To do so:

 Select *New Project* from the **File Menu**. (**+** for more information on BetterState Projects see *Working with Projects* in the **Introduction Chapter** and *Starting a Project* in **Chapter 4, Extended State Diagrams).**

 When the Chart Dialog box appears, name your chart (for example: Simple Counter).

 Select the programming language you want to relate to (this information is user information that is used by the code generator). In this example we are going to use C. However, if you do not have the C code generator you can select C++, VHDL or Verilog HDL. This example is not for Visual Basic. Visual Basic is used in Part Four.

> **Note:** The code generators installed in your copy of BetterState depend upon what you purchased. BS4VisualBasic just has the Visual Basic code generator. Visual Basic is included in all other configurations of BetterState Pro.

 The diagram type should be Extend State Diagrams.

Click on OK when you are done.

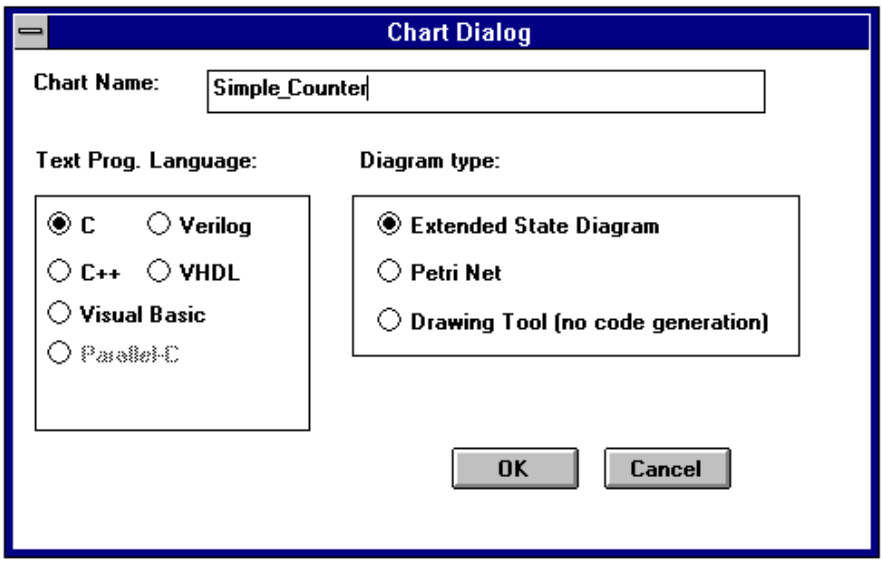


Figure 18 This is the BetterState Chart Dialog box that appears everytime you start a new project or a new chart to an existing project.

 To place your first "state" move your cursor to the vertical Icon Bar and click once on the icon showing a box with rounded corners. Notice how your cursor changes. Move your cursor to the location where you want to place the first state and press your left mouse button once.

 Once you have drawn the state a Dialog Box appears (note, you can always go back to the Dialog Box by double clicking on a specific state). Make this state the default state with the name **Start**. You also want to specify if the C or C++ action specified is *on-entry or on-exit.* In this state the action is *Flag=0* and it is an *Entry* Action. When you have completed the Dialog Box entries click on OK. (**+** For more information see *States and Transitions* in **Chapter 4, Extended State Diagrams).**

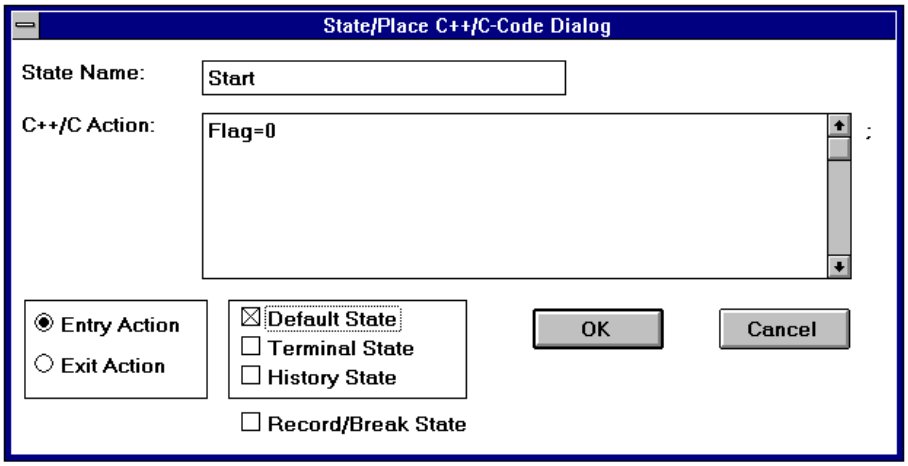


Figure 19 This is a BetterState State dialog box that appears when you draw a State.

Now that you have selected OK from the dialog box, notice in the design area how the Action descriptions are displayed with the state and an arrow show the type of action (on-entry or on-exit). The "on-entry" action text associated with the start state may be moved and resized but it will automatically remain a region of the "start state"(+ for more information see *States and Transitions* in **Chapter 4, Extended State Diagrams)**.

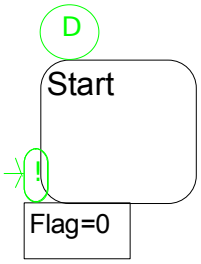


Figure 19.1 Shows the initial start state with the on-entry action Flag=0

To move or resize the action dialog, select the green arrow icon on the vertical menu bar to change your cursor to a pointer. Click once on the "on-entry" dialog box (the box with Flag=0) and notice how a group of small solid color outline boxes appear. These boxes are called handles and they allow you to resize the selected item. You can also move the box by placing your cursor anywhere within the box, and pressing the left mouse button down while moving the box to it's new location.

 Since we are designing a 3 car counter let's now place three more states, one for each event (car1, car2 and car3 being counted). Note there is no need to add any action dialog to these states.

 Once we have drawn the states we need to define the transitions between the states. This is done by selecting the "transition" button (the icon button with an arrow pointing down to the right) from the vertical menu bar. Once you have selected this button, move your new cursor to some location anywhere inside the first state (***start state***) and press the left mouse button. While holding the mouse button down, move your cursor to some location anywhere inside the next state (***Car1***) and release the mouse button. A transition should have been drawn between the two states with the arrow head pointing in the direction you moved the mouse. (+ see *States and Transitions* in **Chapter 4, Extended State Diagrams** and *Creating Connectors* in **Chapter 3, Using BetterState)**.

 Once you have drawn the "transition", the transition Dialog Box appears and you should enter *New_Car* as the condition for the transition to fire. A similar transition should be drawn between the **Car1** and **Car2** states and the **Car2** and **Car3** states.

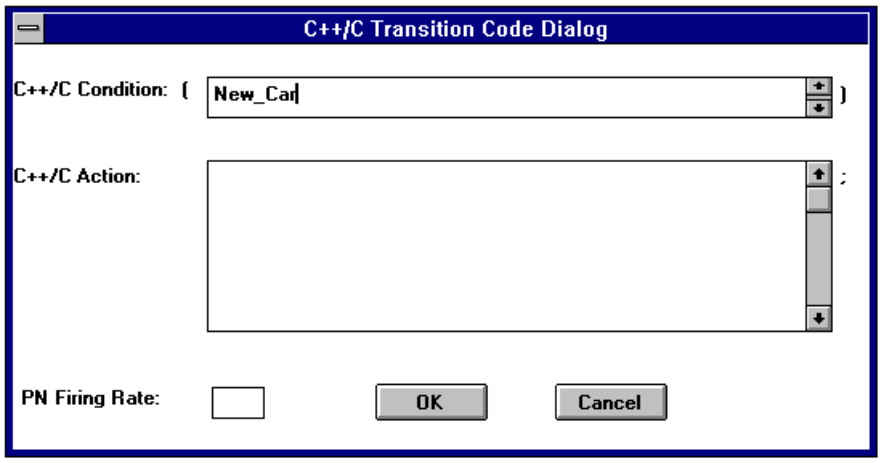


Figure 20 Shows a BetterState C++/C Transition Dialog box.

 We stated in our specification that we wanted a flag raised (Flag=1) when three cars were detected. Go to the **Car3** state and specify that action. To do this, place your cursor on the state **Car3** and double click on the state using the left mouse key. This will bring up the state dialog box for that state and you can enter Flag=1 in the Action Field.

Your finished state diagram should look similar to this one:

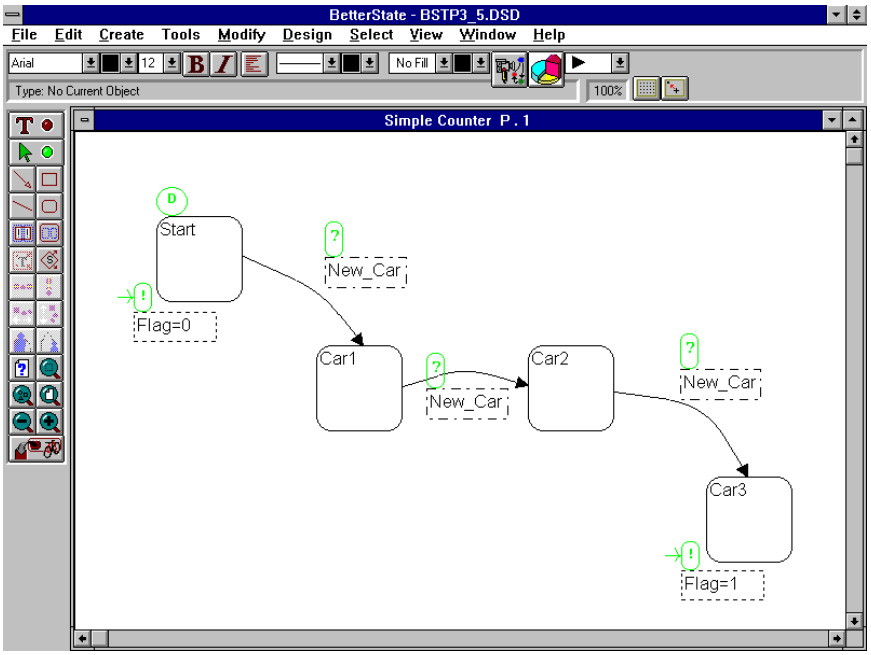


Figure 21 Shows the design of our *car counter* project.

Code Generation

To generate code for this design do the following;

 From the Menu Bar select Tools.

 From the Tools Menu select Code Generation. After the Code Generation option is selected you might see a message page that will tell you if you have any design errors that need to be corrected before code can be generated. If there are no errors the next screen to appear will be the Code Generation Menu. To select the default conditions press OK. (+ for more information on code Generation see **Chapter 6**)

You can now view or compile the code that was just generated.

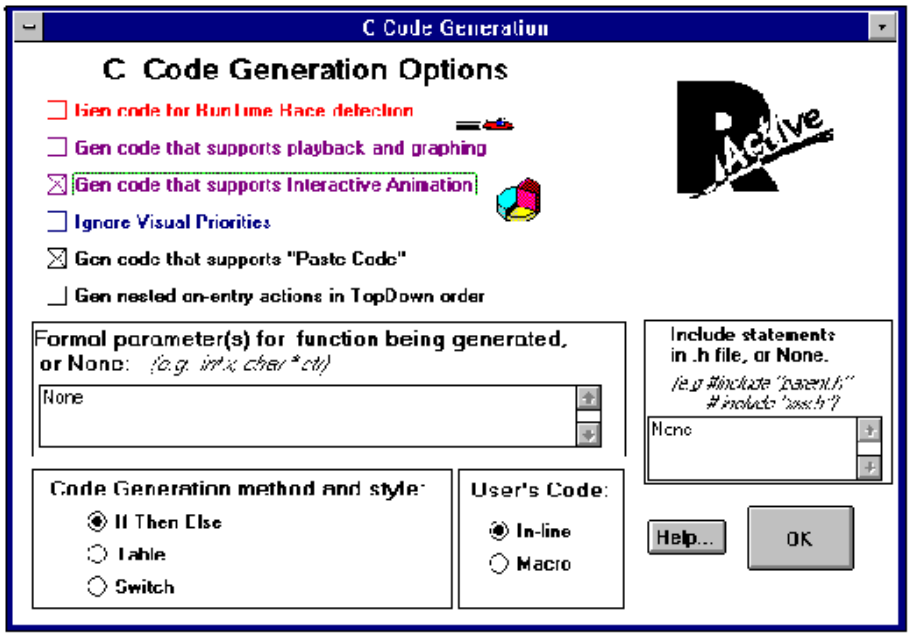


Figure 22 Shows the C Code Generation Menu that was selected at the beginning of the project

Let's take a look at a slightly more complex example. The new specification is as follows:

Now instead of raising the flag when three cars are detected, the specification now states that the flag will be raised if one or more cars is detected followed by a truck.

Drawing Hierarchy

In BetterState hierarchy is drawn by placing states within states. Hierarchy is used to group sets of states together. In this example we can create a hierarchical state around the states ***Car1*** and ***Car2*.** We can name this new high level state ***Truck*.** This gives your diagram the following meaning. *If three cars are counted* **OR** *if one or more cars followed by a Truck are counted a Flag will be raised.* The following will show you how easily your original BetterState diagrams can be modified to accommodate the new specification.

 Simply select the *state* button from the vertical menu bar. Place the state cursor somewhere on your design just above the ***Car1*** state and while holding the left mouse button down, drag the cursor down and over the ***Car1*** and ***Car2*** states. Then release the mouse button. When you release the mouse button a State Dialog Box appears and you can name your new state ***Truck*.** You've now created a hierarchical state with two sub states, ***Car1*** and ***Car2***. (**+** for more information on hierarchy see *Hierarchy* in **Chapter 4, Extended State Diagrams)**.

 Now place a transition between your high level state and the ***Car3*** state. Specify the condition as ***New_Truck.***

Your design should look similar to this one:

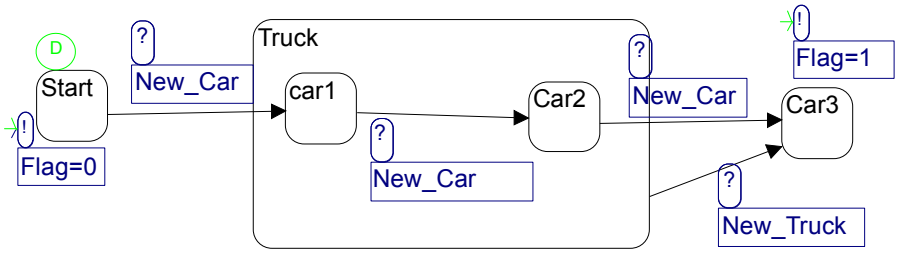


Figure 23 Illustrates our new design with the higher level state ***Truck.***

Now generate code for your newly modified design. Again, go to the tools menu and select Code Generation. **It's that simple!**

Integrating your BetterState Generated Code

The code generated by BetterState is typically a component of your overall system-level design. Thus it needs to be integrated into your system level code. In C, this component is a function; in C++, it's a class, in Visual Basic its a module, in VHDL its an architecture, and in Verilog it's a “task” or “always” statement. In each language (except Visual Basic), this component needs to be invoked by the system-level code.

For example, in C/C++ this is done by a function call, where each call to the function realizes one pass over transitions in the diagram, firing one or more concurrent / independent transitions, and then returning control to the calling program. (**+** for more information on Scheduling see **Chapter 6).**

The BetterState code generator generates code that can be scheduled in a flexible way. This enables very powerful possibilities for realizing your overall system. In addition to the code being generated from the chart, you should have an envelope program in which the external variables, signals, I/O, and functions being used in your chart are defined. (**+** Some examples exist in the examples sub-directory of your installation directory.)

When C is the underlying language

Then the envelope program should call the controller(s) using the function call *CHRT_xxx(int reset)* (where xxx is the user defined name of this chart). Each such call acts like a clock tick; it enables one cycle of execution where the controller advances its state in all active threads of computation.

(+*Simple.c* in the examples sub-directory of your installation directory is an example of such an envelope program developed under Microsoft's Visual C.)

When C++ is the underlying language

For C++, BetterState generates a class for each controller. The class constructor resets the object upon creation. The *BS_Fire()* member function fires the controller for one cycle, in which all enabled transitions may fire once, after which program control is returned to the calling program. Hence, the C++ code supports the same flexible scheduling capabilities offered by our C product.

(+ An example of using your BetterState generated C++ code is located in the examples sub-directory of your installation directory. Select *its_cpp* sub directory.)

When VHDL is the underlying language

In VHDL the envelope is an entity. The controller's VHDL code generated by the code generator is an architecture for that entity. The controller is scheduled by an input **CLOCK** and has an (asynchronous) reset signal

(+An example for a VHDL entity is listed in the examples sub-directory of your installation directory. Select *VHDL* sub directory.)

When Verilog HDL is the underlying language

When Verilog is the underlying language, the envelope module should call the controller(s) using the task call **CHRT_xxx (reset,terminal)** (where xxx is the user defined name of this chart). Each such invocation is typically caused by the CLOCK, but can be invoked by any event; it enables **one cycle of execution** where the controller advances its state in all active threads of computation.
Always @ (posedge clock)
CHRT_xxx (reset, terminal)

Alternatively, you can choose to realize your Verilog controller as an always @ (posedge CLOCK) statement.

(+An example of using the Verilog HDL is listed in the examples subdirectory of your installation directory. Select *Verilog* sub directory. In this directory there is also a sample of BetterState generated code that was synthesized using Synopsys, *see synopsys.n*)

When Visual Basic is the underlying language

For complete details on using Visual Basic as the underlying language, see the BetterState for Visual Basic Users Manual.

Congratulations!

You just completed your first BetterState design.

You can continue to Part Four and see how easy it is to put Visual Basic to work for you as a visual prototyping tool.

In this section we will go through a simple design example using Visual Basic and BetterState For Visual Basic. This design example will focus on designing the state diagrams within the BetterState environment. It is up to the user to set up the Visual Basic front-end portion of this design.

Note: The Visual Basic Code Generator is also available in every version of BetterState Pro.

Consider a simple example, where a Visual Basic form consists of a standard Visual Basic Timer control named Timer1 and a custom bitmap control representing a traffic light, named Light1, which can be either Green or Red. Suppose we want the light to toggle between Green and Red every time a Timer event occurs. This behavior is conveniently modeled as a State Machine, and visually depicted as a state diagram, as illustrated in Figure 24.

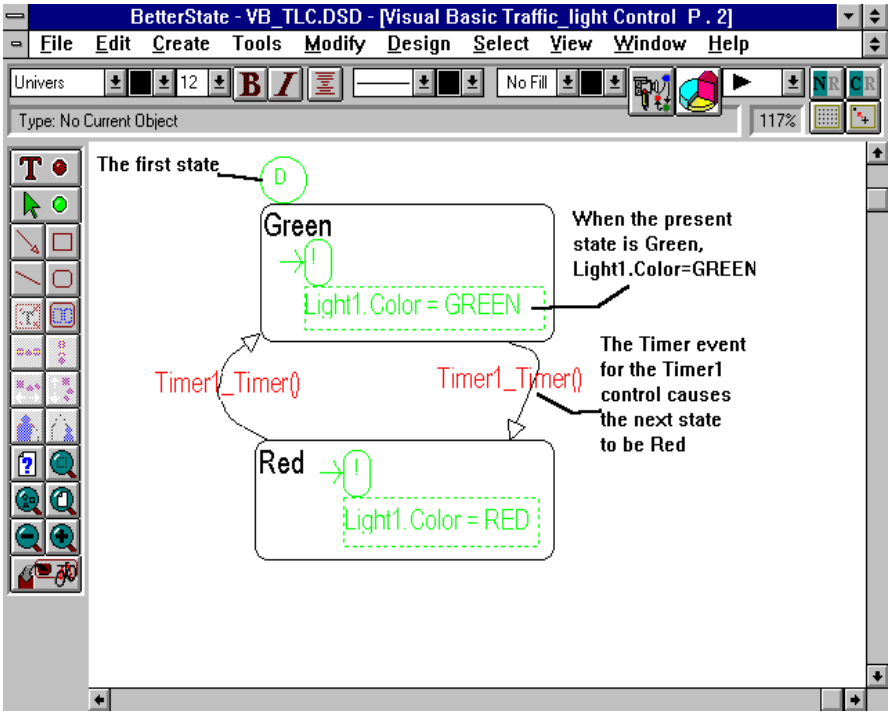


Figure 24 Shows our simple traffic light controller with two states.

The state machine starts operating in a state labeled *Green*, and each time the *Timer1_Timer* event occurs, which happens at fixed time intervals, the state machine toggles between the *Red* and *Green* states, thereby asserting *Light1*'s color property appropriately. Obviously, we could achieve the same result using plain Basic code within the *Timer1_Timer* event Subroutine, us-

ing a variable, say *LightState*, being 0 or 1, and using an *if-then-else* statement that toggles the value of this variable.

```
If (LightState=0) Then
      LightState = 1
      Light1.Color = 1 'RED
 Else
      LightState = 0
      Light1.Color = 0 'GREEN
 End If
```

In fact, this is exactly what BetterStates automatic code generator does for you; you draw the diagram, and the code generator automatically generates the code. However, the benefits of such a diagrammatic programming vehicle go much further. Lets consider a few more complex examples.

The first enhancement to the example in Figure 24 consists of the following modifications:

- Two push-button controls, named *NewTruck and NewCar*, exist.
- *Light1* now switches from Red to Green after 3 clicks on the *NewCar* button or, after one or more *NewCar* clicks followed by a *NewTruck* click.

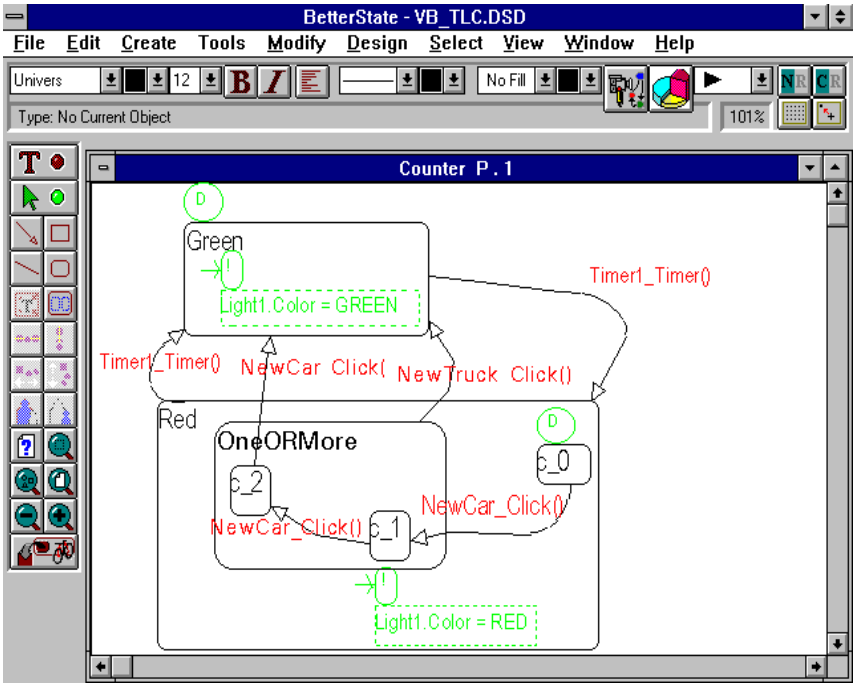


Figure 25 Shows the modified traffic light controller that now includes the new specifications.

Figure 25 illustrates the Extended State Diagram (ESD) that captures this behavior. Note that we have added **hierarchy** to the diagram, where the transition connecting the *OneORMore* state to the *Green* state means that if one or more *NewCar* clicks have occurred (implied by the present state being one of the **substates** of *OneORMore*), and a *NewTruck* click occurs, then the next state will be the *Green* state.

Such hierarchical design capabilities allow any Visual Basic user to practice one of the most fundamental concepts of modern programming, namely, **top-down design**. For example, it allows you to design a high-level diagram that consists of the *Green* and *Red* states alone, and design the contents of the *Red* state later, or assign the design responsibility for the contents of the *OneORMore* state to someone else.

Note how the ESD is actually memorizing **sequences of input events** for you and presenting those sequences in a visual manner. This is one of the primary missions of a state machine of any kind.

ESD's can be beneficial in many other ways as well. One other important service they provide is to visually capture **independent**, especially partially independent, sequences of input events.

Let's enhance the specification for our example one step further, as follows:

- Grid control, named *Grid1*, is added to the Visual Basic front end.

- After any Grid1 key is pressed while *Light1* is Red, then once the *NewTruck* button is clicked, a *Camera* bitmap should appear until Light1 goes Green again.

- Obviously, this behavior describes yet another sequence of input events, which can conveniently be described using a State Diagram. This behavior is **independent** of the activities discussed earlier and could be described by a completely separate ESD. However, suppose we want the *Camera* bitmap to become visible, for a half a second, after two *NewCar* clicks have been detected, while *Light1* is Red. Now, the two ESD designs are not entirely independent; the second ESD design must be able to sense the state of the first ESD.

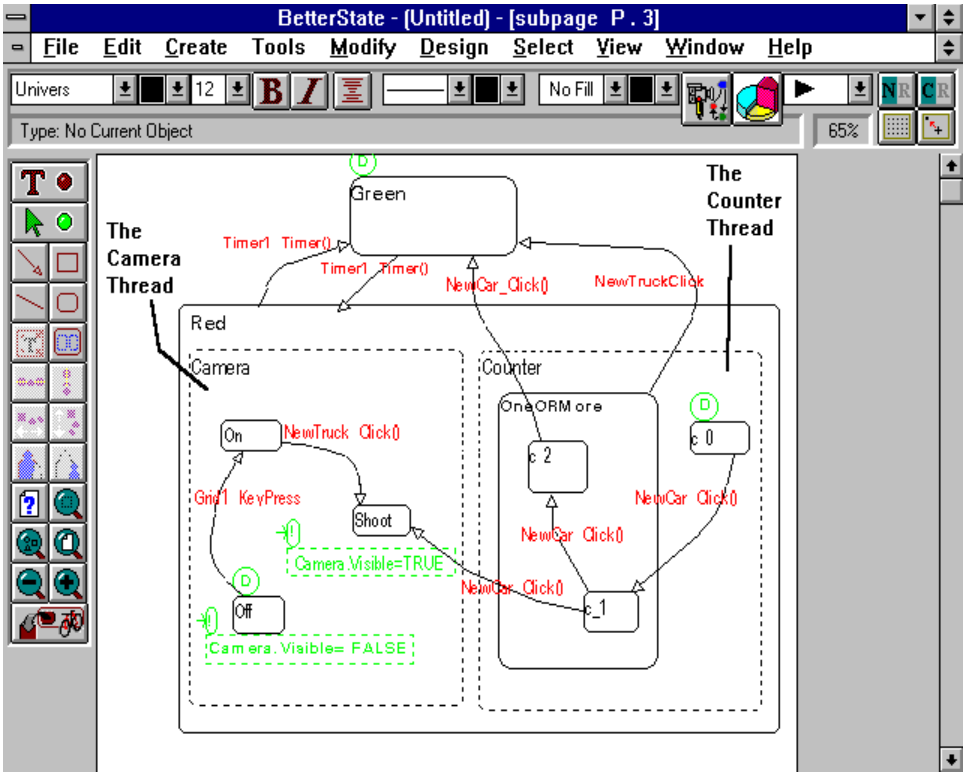


Figure 26 Shows the enhanced design with two independent threads of control.

Figure 26 illustrates an ESD which captures the entire design; the two dashed boxes, labeled *Count* and *Camera*, are called threads, and each encapsulate a “sub”- ESD, one for the Counting activity and the other for the Camera activity. The two threads operate independently of one another. For example, the Counter might move from state *c_0* to state *c_1* when the *NewCar* button is clicked, in which case the Camera does nothing, and therefore remains in it's present state, **or**, when the *NewTruck* button is clicked, they might **both** traverse transitions. The transition from *c_2* to *Shoot* realizes the specification that required some **dependency**, where the Camera must move to the *Shoot* state (thereby making the Camera bitmap visible) when the Counter has counted two *NewCar* clicks.

By now you most probably have figured out that we have been building a game or a prototype of a traffic light and it's controlling software. Such event driven programs can become large, complex, and completely unreadable when described in plain Basic code. Moreover, debugging an equivalent Basic program becomes a real nightmare, with the flow of control jumping between the various event procedures. Debugging plain Basic code for such a design is especially difficult when independent and partially independent activities are involved, because the Basic code, being textual, is by definition **sequential**, whereas independent activities are conceptually **simultaneous**.

Therefore, an important capability supported by leading diagrammatic programming tools is the ability to debug the diagrammatic behavior on the diagram itself, in a **WYSIWYG** (What You See Is What You Get) way. Such debugging capabilities include two main features:

 Animation, where the states of the diagram change colors as input events are accepted; for example, when a *NewCar* event occurs, the *c_1* state should be colored Red (or flicker, or use some other visual highlighting mechanism), to visually highlight the fact that the present state is *c_1*, whereas the *c_0* state should be black again (it is no more the present state). The figure on the next page illustrates how InterActive State Animation operates.

 Break-states, much like Visual Basic breakpoints, you might want the Visual Basic debugging session to stop when a certain state is reached.

Summary

We have shown how state Diagrams, especially Extended State Diagrams, can replace much of the plain textual Basic coding effort that still exists behind the highly visual front-end given by VisualBasic. This allows non experienced programmers to design highly complex behavior for their VB applications, as well as reduce the development time and error count within non-trivial Visual Basic applications. In addition, the same diagrams can become part of the documentation for your design, eliminating the need to draw them somehow just for that purpose.

Part Five, Visual Debugging

One of the most useful and powerful features of BetterState Pro is its Visual Debugging capabilities. With capabilities like ***InterActive State Animation*** you can visually view and animate the execution of your BetterState generated program, state by state. With ***InterActive State Traveler*** you can visually navigate through complex transitions without exiting BetterState to compile your code. ***Animated PlayBack*** give you the powerful capability of recording the actual execution of your program (in the lab or in the real environment) and then later playing back the execution on your PC.

InterActive State Animation

While interpreting or executing your BetterState generated code, your diagram is automatically animated to reflect program execution. The first step in establishing this link is to generate code that supports InterActive State Animation.

The code generation menus for the Visual Basic, C and C++ code generators allow the user to specify if they want to generate code for InterActive State Animation. If this option is enabled the code generator will generate code that can be used with Visual Basic, MS Visual C, C++ and other compiler / debuggers.

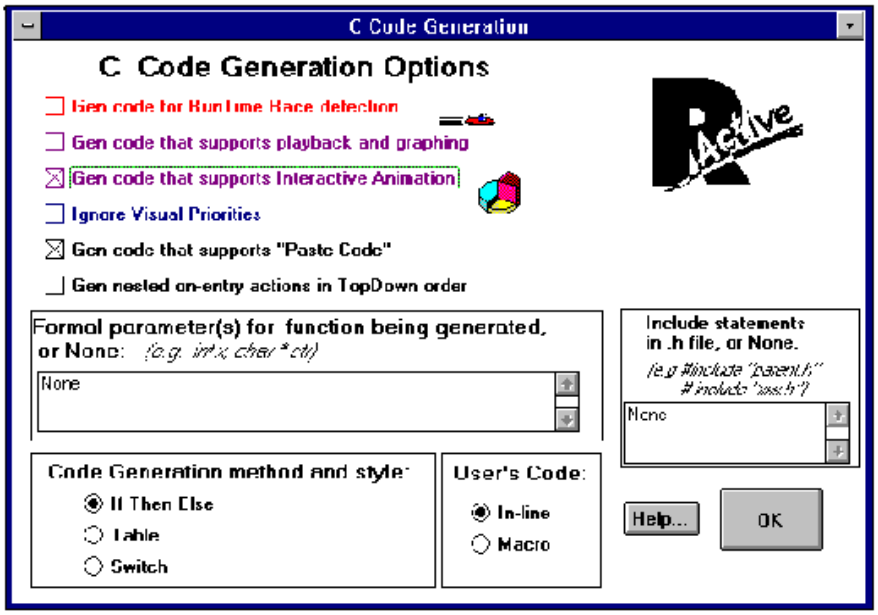


Figure 27 Shows the C Code Generation Menu with various code generation options.

For example, via DDE communications, while you are running Microsoft Visual Basic in one window, you can activate a Visual Basic “control” and

view the actual execution of your Extended State Diagram or Petri Net running in a ***BetterState*** window.

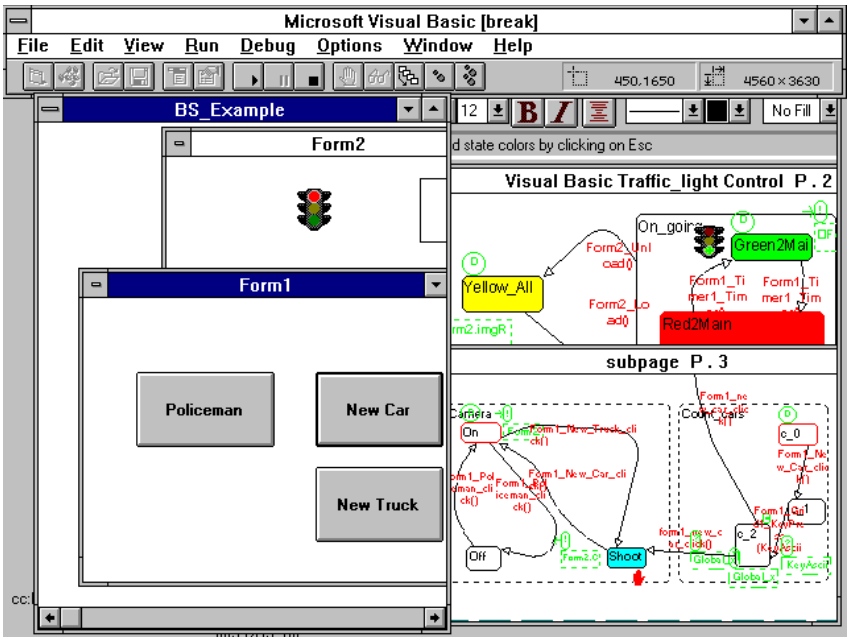


Figure 28 Shows BetterState interacting with Visual Basic.

In Figure 28 everytime you feed it an event, either from the timer or you click a button, a state changes. BetterState will animate the state changes automatically and highlight them in RED. This way you can visually inspect the behavior of your diagrammatic design.

With InterActive State Animation you can set "break states" in the BetterState diagram, so that when program computation reaches such a state the Visual Basic, C or C++ program will stop.

Note: InterActive State Animation is not ***PlayBack.*** It only animates whatever is done in the debugger or during execution (e.g. Visual Basic, Visual C++ or during execution; etc.) however, you can force the debugger to stop, by placing a break state in your diagram.

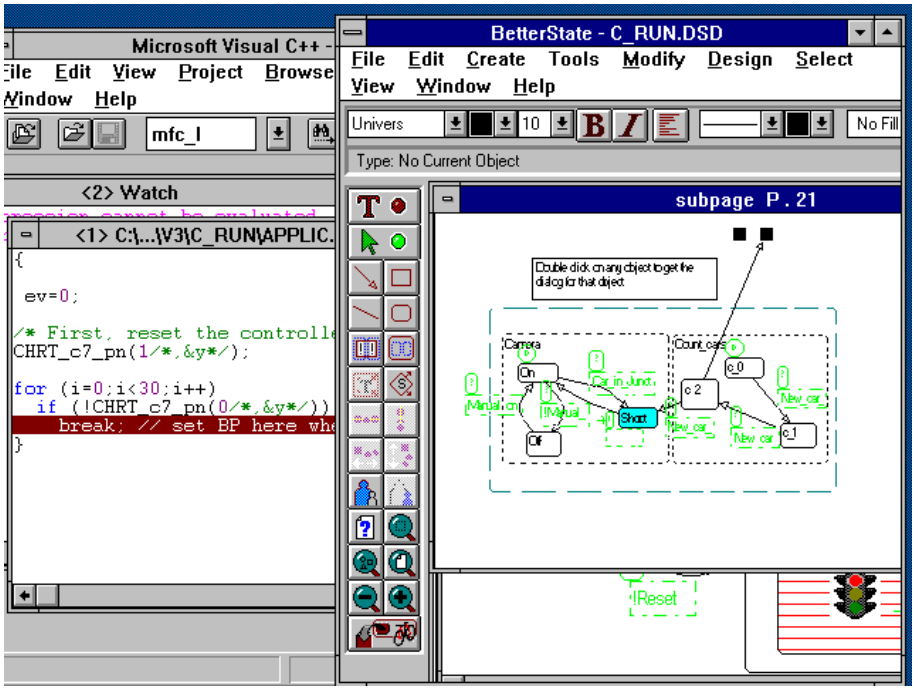


Figure 29 shows another example of InterActive State Animation, this time linking BetterState with Visual C++.

State Traveler

State Traveler provides the designer with a tool to visually see the effect of a transition, or a complex transition, on the compound state of his controller. For example, you might want to inspect the states that are reached if the complex transition fires. To do so, simply select the transition (click on it), and then click on the traveler Icon button. When you click on the Traveler button, the states that will become the present states when that transition fires are highlighted in RED.

Figure 30 shows an example of two threads of control. In this example it might not be clear what the *next state* will be as a result of any of the transitions firing. This situation is easily viewed using State Traveler. Just select the transition your inquiring about, then click the Traveler button, and you'll see the resulting *next state*.

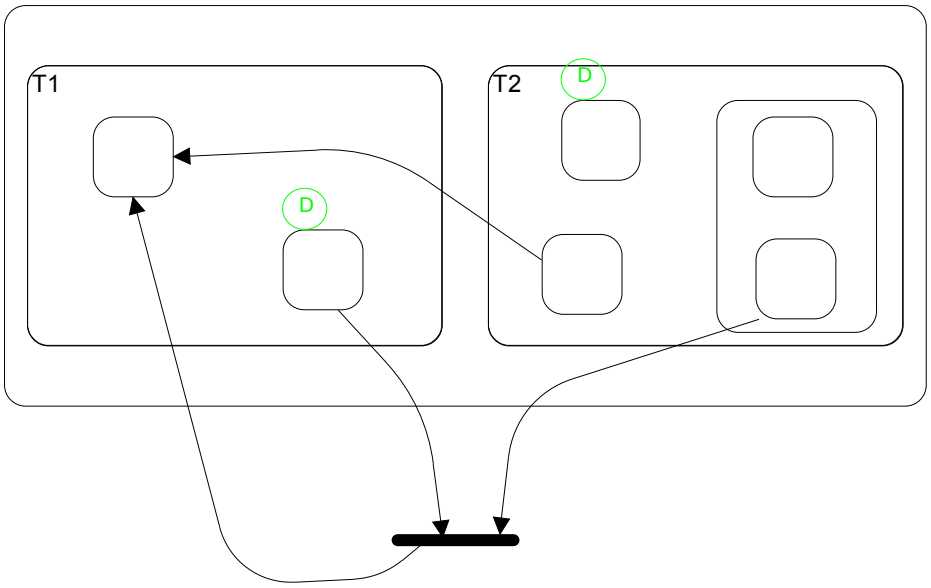


Figure 30 Which state is the *next state*?

Now, given that you know what states are the present states you can choose another transition and see the effect on the present states.

With State Traveler we are not executing code that has been generated by the code generator; everything is internal to BetterState. In fact, the condition code and action code associated with the selected transition are ignored; BetterState is showing you ***"what happens IF the transition fires"***. Therefore, the traveler's behavior does not depend on the stimuli the controller is suppose to receive.

To summarize, the Traveler is an on-line inspection tool, which is very powerful as a learning vehicle (e.g., to understand the meaning of complex and compound transitions).

Note: The Traveler needs a file called Travel.dbf. This file is generated via code generation. Therefore, for the Traveler to work properly, you need to generate code for the diagram (but you do not need to compile, synthesize or execute this code).

Animated PlayBack

BetterState supports a third animation feature called ***Animated PlayBack***. Animated PlayBack serves two purposes:

Œ It provides a visual simulation/debug vehicle for the development phase of your design and it provides a visual analysis tool for the actual field usage phase of your project.

Once again, code that supports Animated PlayBack is produced by the code generators.

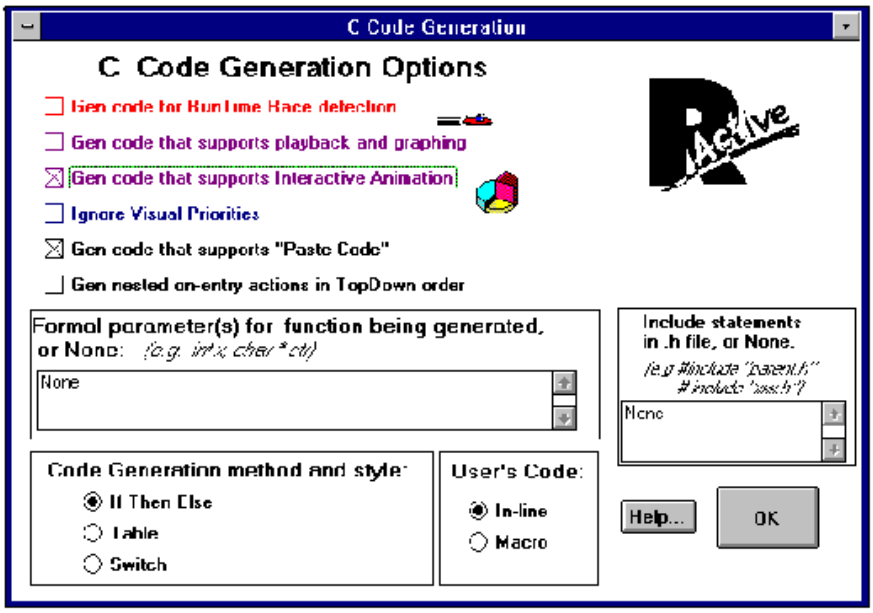


Figure 31 Shows the PlayBack option on the C Code Generator Menu.

Selecting the PlayBack option instructs BetterState to generate code that writes details of the on-going execution (whether running under the debugger or not) into a FoxPro file called Record.dbf, which consists of one record per invocation of the controller, a record that consists of ID's of all ESD states, or PN places, visited during this invocation.

For example, if in Figure 32, at time t states s1 and s3 are visited, and at time t+1, s2 and s3 are visited, and at t+2, s2 and s4 are visited, then the three corresponding records will be <s1,s3>, <s2,s3>, and <s2,s4>.

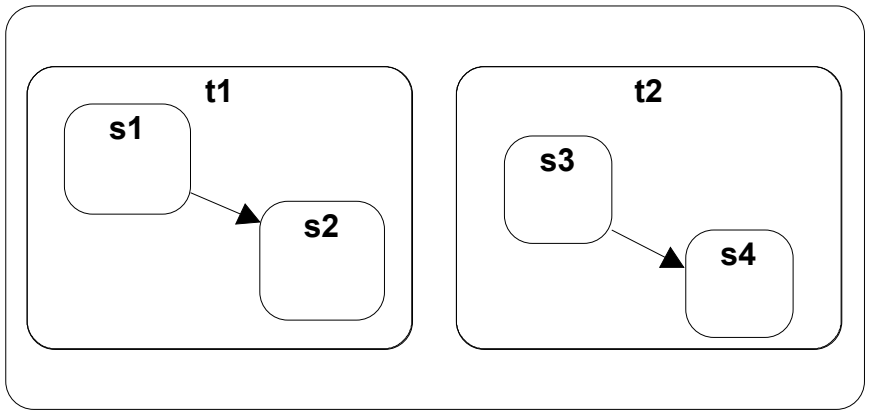


Figure 32

Now run your application, either under a simulator/debugger, or as an executable. Obviously running it under a simulator/debugger allows you to feed in artificial stimuli (**+** *see the Section named "Using your Debugger as a Simulator" in the Advanced Topics Chapter of the BetterState Pro Users Manual*), whereas running it as an executable requires a mechanism for capturing real stimuli and feeding them into the controller, a mechanism that is always part of the final, released application.

In fact, you can run the application in the field using real-life stimuli, provided that the application is an embedded PC application (because the file system is required in order to write Record.dbf). Any one of these ways of execution will write a Record.dbf file.

Finally, in BetterState, select the PlayBack button; this will cause a VCR-like tool bar to replace the conventional toolbar. Figure 33 shows a Play-Back screen shot.

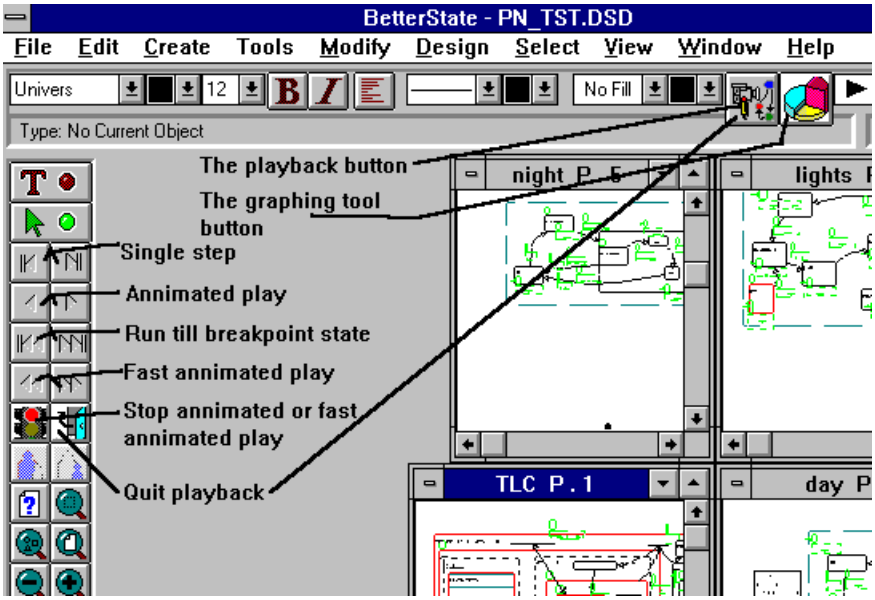


Figure 33 Screen shot of BetterState PlayBack.

With this user-interface you can "play" the execution that has been recorded in Record.dbf. You can do the following:

- Single step forwards and backwards through states; each time you click one of these buttons the next (previous) state(s) will be highlighted.

- Animate forwards and backwards through states; these buttons will go (forwards or backwards) through the records in Record.dbf and highlight the states visited, until stopped by the *stop PlayBack* button, or until the end of Record.dbf is reached.

- Fast animate forwards and backwards through states; these buttons will go (forwards or backwards) through the records in Record.dbf and highlight the states visited, until stopped by the *stop PlayBack* button, or until the end of Record.dbf is reached.

- Run forwards and backwards until a breakpoint state is reached (you can specify any state to be a BP state, by editing it and selecting the breakpoint check box).

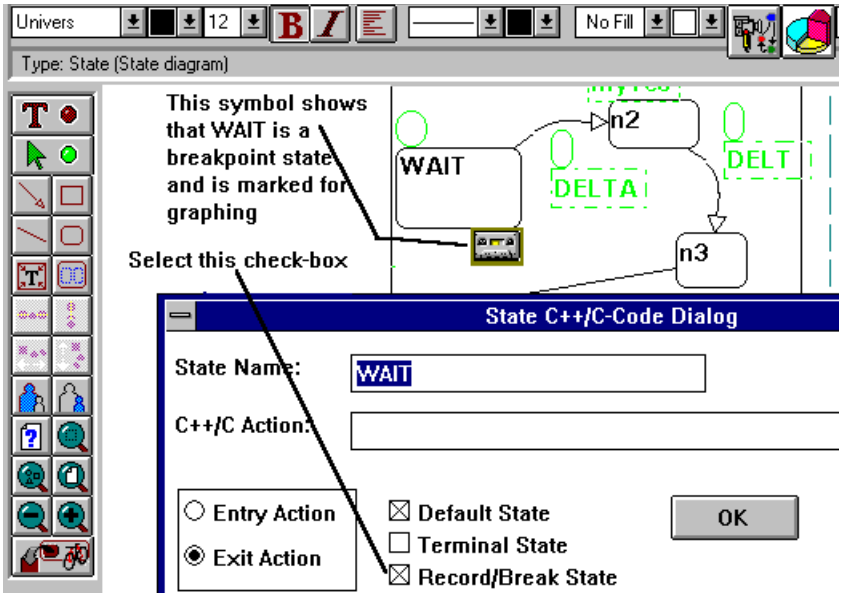


Figure 34 Shows how a Break State is set.

Note: In earlier versions of BetterState the break-state symbol was a "cassette". In version 3.0 it has been changed to a "stop sign" symbol. So if you select a state and define it as a Record/Break state you'll see a "stop-sign" type of symbol next to the state.

Animated PlayBack of the execution can help you to debug your design; rather than looking at state names (or even ID's) in the simulator/debugger environment, you can inspect the state sequences visually. Moreover, you can use this tool as an analysis tool, to analyze real life behavior.

For example, consider an embedded PC application controlling a traffic light, using one of the Traffic Light Controller (TLC) diagrams we've discussed in this manual. If the TLC is using code BetterState generated with the PlayBack and graphing CG option, then the entire daytime behavior of the TLC can be recorded in Record.dbf. PlayBack can be used to trace and play, forwards and backwards, through the sequence of states that had been visited throughout the day, and thereby understand how certain problems occurred and what sequence of states (and corresponding stimuli) caused these problems.

Although PlayBack is done after the fact, you can use the PlayBack tool almost simultaneously with the simulator/debugger. To do so, have two windows open, one with the simulator/debugger (if it's a Windows 3.1 tool) and the other with BetterState, as illustrated using the Visual C++ debugger.

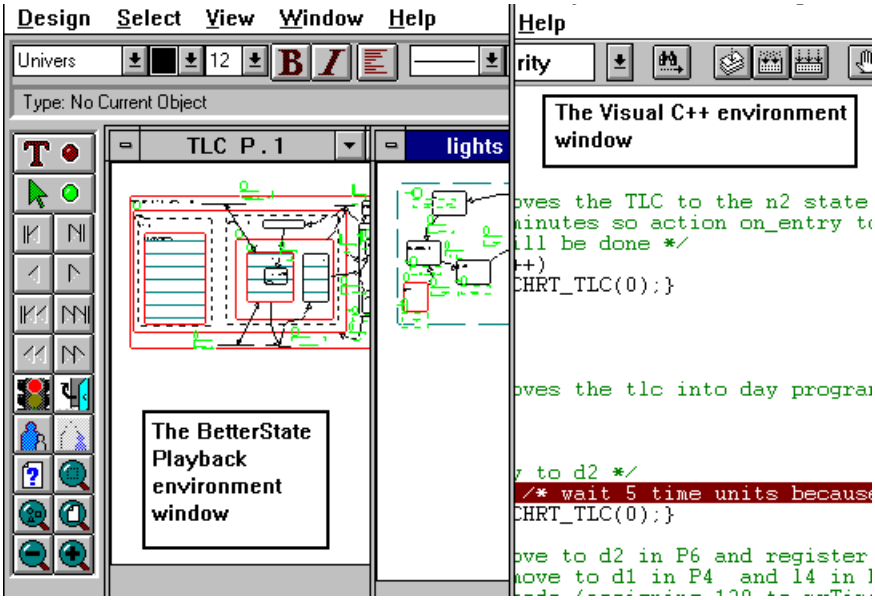


Figure 35 Shows BetterState with Visual C++

Now, you can single step in the debugger, or run the debugger through one or more calls to the controller, and then, in the BetterState window, observe the behavior visually by stepping or running the VCR; and so on.