© Copyright by Paul Jay Lucas, 1993

Software: © Copyright by AT&T, 1993

AN OBJECT-ORIENTED LANGUAGE SYSTEM FOR IMPLEMENTING CONCURRENT, HIERARCHICAL, FINITE STATE MACHINES

BY

PAUL JAY LUCAS

B.S., Polytechnic University, 1989

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

(Certificate of Committee Approval Form goes here.)

To my parents, for their constant love and support and for always being there.

Acknowledgments

I would like to thank my management at AT&T Bell Laboratories for affording me this opportunity.

Contents

Chapter

1	Inti	roduc	tion	1
	1.1	State-	Transition Diagrams	1
	1.2	Statec	harts	2
	1.3	Concu	rrent, Hierarchical, Finite State Machines	3
	1.4	Statec	hart Details	4
		1.4.1	Additional Features	4
			1.4.1.1 Default States and History	4
			1.4.1.2 Conditions, Broadcasting, and Actions	5
			1.4.1.3 Implicit Broadcasting	5
			1.4.1.4 Non-Local Transitions	6
		1.4.2	Other Problems	7
			1.4.2.1 Nondeterminism	7
			1.4.2.2 Cycling	8
		1.4.3	Semantics	9
			1.4.3.1 Statechart Definition	9
			1.4.3.2 Transitions and Ordering	9
			1.4.3.3 Events and Stability	10
		1.4.4	Transition Algorithm	10
		1.4.5	The Cycling Problem Revisited	12
		1.4.6	An Extended Example	12
			1.4.6.1 Overview	12
			1.4.6.2 Statechart Specification	13
2	ΑL	angua	nge System	16
	2.1	_	iew	
	2.2		ying a CHSM	
		2.2.1	The Description File	
		2.2.2	Specifying States	
		2.2.3	Specifying Transitions	
		2.2.4	Specifying Events	
		2.2.5	Specifying State-Names	
		2.2.6	Derived Classes	
	2.3	An Ex	tended Example, Continued	

3	Rui	n-Time	Library Design	29
	3.1		w	
	3.2	Resulta	nt C++ Code	30
	3.3	Class D	esign	30
			Events and Transitions	
			3.3.1.1 Design Considerations	30
			3.3.1.2 The Event Classes	
			3.3.1.3 The Transition Structure	
		3.3.2	States, Clusters, and Sets	35
			3.3.2.1 Design Considerations	35
			3.3.2.2 The State Class	35
			3.3.2.3 The Parent Class	37
			3.3.2.4 The Cluster Class	38
			3.3.2.5 The Set Class	38
			3.3.2.6 Resultant C++ Code	38
		3.3.3	The CHSM Class	40
	3.4	The Inte	eractor	41
4	Cor	_	Design	
	4.1	Overvie	W	43
	4.2		mpiler Class	
	4.3	Lexical	Analysis	46
		4.3.1	Overview	46
		4.3.2	State Overview	47
			Action-Code and Embedded Constructs	
			Condition-Expressions	
		4.3.5	Identifiers and Keywords	48
	4.4	The Yac	cc Grammar	49
		4.4.1	Overview	49
		4.4.2	The Parsing Stack	49
			Improving Error-Messages	
	4.5		-Table Management and Semantic-Checking	
		4.5.1	Overview	53
		4.5.2	The BaseInfo Class	53
			The GlobalInfo Class	
		4.5.4	The ChildInfo Class	55
		4.5.5	The StateInfo, ParentInfo, ClusterInfo, and SetInfo Classes	56
		4.5.6	The DerivedInfo Class	58
			The TransInfo Class	
			The EventInfo Class	
			The UserEventInfo Class	
		4.5.10	The MacroEventInfo Class	60
Ľ	Car	alusis:	me.	Ω1
5		iclusio		
	5.1		ng Expectations	
	5.2		Work	
			Language Extensions	
			J.L. I. LIASSES DELIVEU HUIH LIE EVEHL CIASS	

			5.2.1.2 Code Execution on State Entrances and Exits	
			5.2.1.3 Derived-Type Constructors with Additional Arguments	
			5.2.1.4 Support for All Kinds of Transitions	
			5.2.1.5 Standard Components	
		5.2.2	Compiler Improvements	
			5.2.2.1 Detecting Illegal Transitions	
			5.2.2.2 Transition-Vector Code Optimization	64
ΑĮ	pen	dix		
A	Lan	guag	e Reference Manual	65
			d Conventions	
		A.1.1	CHSM Description File Format	
		A.1.2	Comments	
		A.1.3	Identifiers	
		A.1.4	Keywords	
		A.1.5	Integer Constants	
	A.2	Basic	Concepts	66
		A.2.1	Scopes	
		A.2.2	CHSM Initialization	66
		A.2.3	Types	67
			A.2.3.1 Fundamental Types	67
			A.2.3.2 Derived Types	67
	A.3	Descri	ptions	71
		A.3.1	State Descriptions	71
		A.3.2	Cluster Descriptions	71
		A.3.3	Set Descriptions	71
		A.3.4	Event Descriptions	71
	A.4	State-	Names	72
	A.5	Events	S	73
	A.8		itions	
	A.9	Condit	tions and Actions	74
	A.10	Gramı	mar Summary	74
В	Sup	port (Classes	77
	B.1	Overvi	iew	77
	B.2	Iterato	or Classes	77
	B.3		ist Classes	
	B.5		ector Class	
	B.6		tring Class	
	B.7	The Sy	ymbolTable Class	80
C	Con	npiler	Manual Page	84
Re	efere	nces .		86

List of Tables

4.1	Tokens returned by	y the lexical	alyzer4	47
-----	--------------------	---------------	---------	----

List of Figures

1.1	Replicated transitions in an STD	2
1.2	Exclusive-or grouping.	2
1.3	Exponential increase in the number of states	3
1.4	Default states and History	4
1.5	Conditions, Broadcasting, and Actions	5
1.6	Contrived example for state-transitions	6
1.7	Illegal state-transition	7
1.8	Nondeterministic transitions	8
1.9	Potential cycling problem	9
1.10	Transition algorithm	11
1.11	Microwave oven panel	13
2.1	CHSM description-file template	16
2.2	CHSM description-language example	17
2.3	Use of the event variable in C++ code	19
2.4	Referring to enter and exit events in C++ code	19
2.5	Macro-events	20
2.6	Event numerical-value assignment	20
2.7	Referencing non-local, hidden state-names	21
2.8	Derived-class example	22
2.9	Microwave oven CHSM	24
2.10	Preliminary declarations	25
2.11	Timer class declaration	25
2.12	Clock class declaration	26
2.13	Microwave oven CHSM with C++ code added	26
2.14	Microwave oven CHSM user-code section	27
3.1	Example CHSM to illustrate resultant C++ code	31
3.2	Event/Transition class relationship	31
3.3	Event class declaration	32
3.4	Resultant C++ code for events	32
3.5	UserEvent class declaration	33
3.6	Resultant C++ code for user-events	34
3.7	Transition structure declaration	34
3.8	Resultant C++ code for transitions	34
3.9	State classes hierarchy	35

3.10	State class declaration	36
3.11	Use of macros in a derived-class	36
3.12	Parent class declaration	37
3.13	Cluster class declaration	38
3.14	Set class declaration	39
3.15	Resultant C++ code for states, clusters, and sets	39
3.16	CHSM class declaration	40
3.17	Sample debugging output	40
3.18	C++ code for CHSM class	41
	Example of print-command output	
4.1	CHSM compiler schematic	
4.2	Compiler structure declaration	
4.3	Example header-file for user-events	
4.4	Lexical-analyzer statechart	
4.5	Yacc grammar production forms	49
4.6	Semantic class declaration	50
4.7	Some semantic-value stack macros	
4.8	Sample debugging output	51
4.9	Example use of myerror()	52
4.10	BaseInfo class declaration	53
4.11	GlobalInfo class declaration	55
4.12	ChildInfo class declaration	55
4.13	Child-state declarations and scope	56
4.14	StateInfo class declaration	57
4.15	ParentInfo class declaration	58
4.16	ClusterInfo class declaration	58
4.17	SetInfo class declaration	58
4.18	DerivedInfo class declaration	59
4.19	TransInfo class declaration	59
4.20	EventInfo class declaration	60
4.21	UserEventInfo class declaration	60
4.22	MacroEventInfo class declaration	60
5.1	Classes derived from the Event class	
5.2	Possible syntax for enter/exit code	
5.3	Possible syntax for derived class's constructor arguments	
5.4	Unsupported transitions	63
A.1	State class interface	68
A.2	Parent class interface	
A.3	Use of the Parent::ChildIterator class	
A.4	Cluster class interface	
A.5	Set class interface	
A.6	Derived-class declaration and use	
A.7	Overriding Enter() or Exit() member-functions	
A.8	Accessing non-local state names	
A.9	Referring to state-names in conditions and actions	
	Referring to event-names in conditions and actions	
4 T. I O	TVOTOTTILLE CO C TOTTE THATHES HE CONTAINED HER MCHOTES	

A.11	CHSM grammar	75
B 1	Typical Iterator class	78
	Iterator example	
	List class interface	
B.4	Vector class interface	80
B.5	SubString example	80
B.6	String class interface	81
B.7	SymbolTable/Symbol/Synfo class relationships	82
	SymbolTable class interface	
B.9	Synfo class interface	83
B.10	Symbol class declaration	83

Chapter 1

Introduction

1.1 State-Transition Diagrams

State-transition diagrams (STDs), along with their underlying finite-state machines (FSMs), are used in many computing applications including compilers, operating-systems, and graphical user-interfaces, and electronic applications including digital watches, microwave ovens, and telephone switching systems. With the possible exception of compilers, all of these applications can be classified as *reactive systems*, that is, systems that react to external stimuli.¹

The use of STDs and their associated FSMs, however, does not scale well with system size [1] [2]. The number of states and stimuli in a telephone switching system is daunting, but even microwave ovens are not trivial. Specifically, the problems with STDs and FSMs are that:

- 1. STDs are "flat" and "global" for there is no notion of logical hierarchy nor isolation of components of the STD. This deficiency does not allow systems to be developed in a piecewise manner with other pieces treated as "black-boxes."
- 2. In many cases, transitions must be replicated across a large number of states as is the case with some sort of "high-level interrupt" where the machine goes back to some initial state as shown in figure 1.1.
- 3. As systems grow linearly, the number of states often grows exponentially. This fractures functional commonalty and logical grouping. (An example of this is given when this problem is addressed in §1.2.)

Compilers could be included if each input character is looked upon as a stimulus affecting the current state of recognizing a token. However, compilers are still different in that they eventually terminate (when the source program has been compiled); reactive systems are instead perpetual in nature.

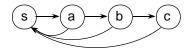


Figure 1.1: Replicated transitions in an STD

4. STDs are sequential in nature and cannot cope easily with concurrent systems, that is, an STD is, at any given moment, in a *particular* state, whereas a concurrent system is usually in a *set* of states simultaneously.

1.2 Statecharts

Statecharts are a graphical notation to represent complex reactive systems more naturally and eliminate the problems mentioned with STDs [1] [2] [3]. Specifically:

- Statecharts are hierarchical in that they can have *child* states grouped together, or "nested," within a *parent* state and thus allow for locality. This allows parent-states to be treated as "black-boxes."
- 2. Replicated transitions are eliminated by the introduction of *logical-exclusive-or* state groups just alluded to and shown in figure 1.2. Henceforth, logical-exclusive-or state groups shall be referred to as *clusters*.

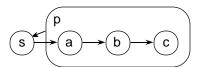


Figure 1.2: Exclusive-or grouping

Here, states a, b, and c are grouped inside cluster p; to be in cluster p is to be in only *one* of its child-states a, b, or c. The transition from cluster p to state s is taken regardless of which child-state the statechart is in; hence, the child-states need no transition to state s.

3. The exponential increase in the number of states when new states are added is eliminated by the introduction of a *logical-and* state groups as shown in figure 1.3a. Henceforth, logical-and state groups shall be referred to as *sets*. In the figure, states a and b are child-states of cluster x and states c, d, and e are child-states of cluster y, clusters x and y, in turn, are child-states of set p; to be in set p is to be in both child-states x and y. Figure 1.3b shows the conventional (rather messy) STD equivalent.

The term "state," however, will be used universally to refer to states, clusters, and sets (since clusters and sets are *still* states); the terms "cluster" and "set" will only be used when it makes a difference. Also, the term "plain-state" will be used to explicitly mean a non-cluster and non-set state.

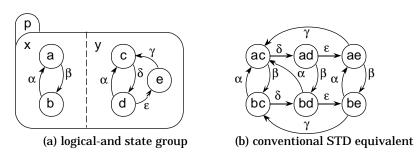


Figure 1.3: Exponential increase in the number of states

4. Sets also allow concurrency. Assuming the statechart of figure 1.3a is in states b and d to start, if event α occurs, the statechart will simultaneously make transitions to states a and c.

1.3 Concurrent, Hierarchical, Finite State Machines

Concurrent, hierarchical, finite state machines (CHSMs)³ are a realization of statecharts just as FSMs are a realization of state-transition diagrams. ⁴ To develop actual working models of reactive systems, software can be written to implement CHSMs. Rather than develop an entirely new language for expressing statecharts as CHSMs, however, a hybrid language system where an existing programming language is augmented with additional constructs can be developed. This approach, also taken by time-honored language systems such as lex [4] and yacc [5], allows all the power of the host language to be used without "reinventing the wheel," requires the user to learn only the additional constructs, and integrates seamlessly with other programs or modules written in the host language.⁵ The host language chosen for CHSMs is C++ [8] [9].⁶

The goal in using a system such as *yacc* is to parse text; the goal in using a system to implement CHSMs would be to simulate the reactive systems they describe (examples of which were given in §1.1). Embedded reactive systems can be prototyped, debugged, and refined before being committed

The abbreviation "CHSM" was chosen because "CHFSM" is too much of a "mouthful" hence a CHSM is a concurrent, hierarchical state-machine.

In the referenced papers, statecharts refer to both the graphical notation and to the underlying state-machines; here, however, they are separated by the introduction of CHSMs to clearly distinguish them from FSMs since, as shown in §1.1 and §1.2, statecharts *can* be implemented using ordinary FSMs.

A completely opposite approach has been taken by the i-Logix corporation with their STATEMATE product [6] [7]. It is a complete statechart "development environment" with its own programming language, editor, compiler, and debugger. This goes against all of the "good" reasons just mentioned. Not only that, but, presumably to recoup their development, maintenance, and documentation costs of such a large product, they have to charge \$50,000 for just the base package.

⁶ C++ was chosen because it is portable, efficient, is becoming the *de-facto* object-oriented programming language, and the object-oriented programming paradigm not only made the development of the CHSM language system easier, but allows it to be easily extended by the user without having to modify any of its source code nor resort to other "hacks." The C++ facilities of class derivation and polymorphism via virtual member-functions (or "methods" for those more familiar with Smalltalk) enable this.

to silicon; software systems can have the CHSM C++ source-code included like any other ".c" file as part of a larger project. 7

The language system could also include a default *interactor*, that is, a simple, command-line interface allowing the user to interact with a compiled CHSM. It would not only allow a user to get events into the system, but could also facilitate debugging by allowing CHSMs to be traced, singled-stepped, and have breakpoints planted [7]. If the user wishes something more exotic, say a mouse-based, graphical user-interface where the mouse can be clicked on an on-screen button to trigger an event and receive visual feedback (such an the icon of a light-bulb illuminating when the *start* button is clicked on a simulated microwave oven), he or she could trivially write a custom interactor.

The development of such a language system is the focus of this thesis. Its design and implementation are covered in the remaining chapters; but first, a more detailed description of statecharts is in order.

1.4 Statechart Details

1.4.1 Additional Features

Statecharts also have a number of additional features that contribute to their usefulness for modeling complex reactive systems [1] [3].

1.4.1.1 Default States and History

Clusters have one child-state as their *default* state, i.e., the child-state that is entered after it itself is entered. Figure 1.3a has been augmented to show child-states of clusters with default arrows in figure 1.4a.

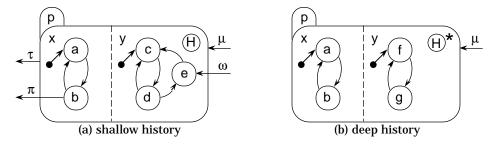


Figure 1.4: Default states and History

Entering set p by a transition on event μ enters both child-states x and y; entering cluster x enters child-state a as indicated by the small arrow; likewise entering cluster y enters child-state c.

Clusters can also have a *history*, that is, entering a cluster enters the child-state that the cluster was in the last time the statechart was in that cluster; a cluster with a history is indicated by the

In many software development environments, files that contain C or C++ source code have names ending with a ".c" to distinguish them.

presence of a circled H. If the statechart was never in a given cluster, then the default child-state is entered instead.

Clusters can alternatively have a *deep history*; this is indicated by the presence of a starred, circled H. In figure 1.4b, assume that states c, d, and e of figure 1.4a have been collapsed into child-states of a new cluster f, and a new state g has also been added. Now when cluster g is entered, not only is child-state f or g entered depending on the history, but if cluster f is entered, either child-state g, g, or g is entered depending on the history also.

There is also the ability for a cluster's history (either shallow or deep) to be forgotten. This allows a statechart to "start-over" from the beginning.

1.4.1.2 Conditions, Broadcasting, and Actions

Transitions can have boolean expressions associated with them as the transition on event α does in figure 1.5. Here, the transition from state b to state a upon the occurrence of event α will be made only if condition x is true at the time.⁸

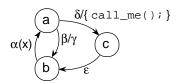


Figure 1.5: Conditions, Broadcasting, and Actions

Broadcasting is the ability to send a global event when a transition is made. For the transition from state a to state b as a result of the occurrence of event β , the event γ is broadcast; this, in turn, will probably cause some other part of the machine to change states due to the occurrence of γ .

Actions can also be associated with transitions. For the transition from state a to state c as a result of the occurrence of event δ , function call_me() is called. A transition can have any, none, or all of a condition, broadcast event, and action associated with it.

1.4.1.3 Implicit Broadcasting

Whenever a state *s* is entered or exited during a transition, an *entered(s)* or *exited(s)* event, respectively, is implicitly broadcast. Transitions elsewhere can occur as a result of the broadcast of either of these events just as with any other. If several transitions are made upon the broadcast of the same event, perhaps with non-trivial associated conditions, then all but one of these, say *s*, can

Attaching conditions implies that the underlying state machine has to have some sort of a memory and/or computational ability to evaluate expressions at "run-time" in order to decide whether to make a given transition. This notion is not new since similar abilities are realized for FSMs using pushdown automata.

Actions can be any arbitrary statement or sequence of statements in some programming language. The semantics of statement sequencing of the given programming language hold; this deviates from [7] where multiple statements in actions are executed in arbitrary order. To integrate a conventional programming language into statecharts, however, the deviation was deemed necessary (and more intuitive).

be replaced by transitions on ex(s).¹⁰ In addition to the direct benefit of not having to replicate the event and its condition across several transitions, use of *enter/exit* events provides another form of isolation: should any of the conditions change, it would only be necessary to modify one transition as opposed to several.

1.4.1.4 Non-Local Transitions

State groups do not make "barriers." In figure 1.4a, the transition on event ω enters set p, hence clusters x and y, and subsequently states a and d overriding the default of cluster y. Likewise for the transitions on events τ and π —the occurrence of event τ causes set p to be unconditionally exited whereas the occurrence of event π causes set p to be exited only if cluster x is in state p at the time. In one respect, this ability to access child-states breaks the concept of having a state hierarchy—the "goto" of statecharts. However, this ability can be used sparingly to get out of "tight spots" in a design that might otherwise require contortions that would serve more to muddle what is going on. As with the goto, the use—or misuse—of this ability is left to the discretion and taste of the user.

In general, any state can make a transition to any other state: from one sibling to another, from parent to child, and from child to parent, or from a state to itself. To illustrate the subtle sequences of events, figure 1.6 has been contrived.

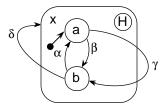


Figure 1.6: Contrived example for state-transitions

Both events β and γ make transitions from state a to state b. The difference is that γ causes its parent cluster x to be exited and re-entered as well. The precise sequence of exit/enter events is: exit(a), exit(x), enter(x), and enter(b). There exists a similar, but more-complicated difference between events α and δ . Event α behaves exactly like β ; δ , like γ , causes both state b and cluster x to be exited and x to be re-entered (in that order), but whether state a or b gets entered depends on cluster x. If x did not have a history (contrary to the figure), then the default arrow would dictate that state a gets entered; the fact that x does have a history, however, means that x will re-enter the last child-state that was active—in this case state b.

The statement regarding transitions was too general since there does exist one restriction: a set's child-states may not have transitions between them. An illegal example, with respect to the transition on event β , is shown in figure 1.7.

The abbreviations en(s) and ex(s) will be used for entered(s) and exited(s), respectively, in statecharts.

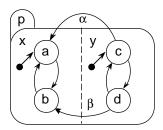


Figure 1.7: Illegal state-transition

The reason that it is illegal is that it would leave cluster y with no active child-state and, recalling the definitions from §1.2, to be in a cluster is to be in one of its child-states. Why not exit cluster y as well? That would violate the definition of set p since to be in a set is to be in all of its child-states.

The transition on event α , however, "gets around" this restriction, is perfectly valid, and thus violates no prior definitions. To see why this is so, the sequence of events can be enumerated: states c, y, and p are all exited (in that order), then set p is re-entered, followed by x then a, and cluster y then c by virtue of the default arrow. The only thing this transition would have accomplished aside from broadcasting all the respective *exitlenter* events—which may have actually caused a flurry of activity somewhere else—is the change from child-state b to a if cluster x were not in child-state a to begin with.

1.4.2 Other Problems

1.4.2.1 Nondeterminism

Just as with STDs, statecharts can have sets of transitions that, when considered together, are nondeterministic. For STDs, there is only "pure" nondeterminism, meaning that there is more than one transition from a state where the triggering events are the same; statecharts, however, can have three forms of nondeterminism: pure (identical to STDs'), potential, and apparent.

In figure 1.8, the transitions on event α , considered together, represent pure nondeterminism. In such cases, a single transition is arbitrarily selected to be taken since there is no reason to prefer one over the other. If each of these transitions had an associated condition, i.e., they instead were $\alpha(x)$ and $\alpha(y)$, then these would only be nondeterministic if both conditions x and y evaluated to true at the same time; this represents *potential* nondeterminism. If it turns out that the set of transitions is actually nondeterministic, then again, a single transition is arbitrarily selected. 11

The transitions on event β represent *apparent* nondeterminism. If nothing more were said, these transitions would, in fact, be nondeterministic; however, such transitions can be accounted for by

If the conditions instead were $\alpha(x)$ and $\alpha(x)$, then the conditions would be mutually exclusive and there would be no nondeterminism.

specifying semantics for this situation. Given at least two such transitions, the "outermost" one is given priority; in the case of figure 1.8, the transition from cluster q to state d would be taken.

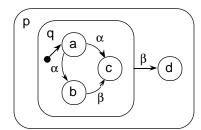


Figure 1.8: Nondeterministic transitions

Why this and not the innermost? Neither [1], [2], [3], [6], nor [7] say why, but at least one argument for this behavior is as follows. Suppose that state b did not have a transition on event β and cluster q were treated as a "black-box" (which is one of the desires for using statecharts in the first place as presented in §1.2). In this case, the transition from cluster q to state d on event β would always (deterministically) be taken. Now suppose that someone *else* changes what goes on within the black-box of cluster q and adds an internal transition on β (returning to match figure 1.8). If the innermost transition were given priority, then the view and behavior from outside the black-box of cluster q would have "mysteriously" changed in that the transition from cluster q to state d on event β would no longer be taken.

However, an argument for the innermost to have priority is that it should override the parent-state's transition. In the case of figure 1.8, the transition from cluster q to state d would be taken except when q is in child-state b. This may be appealing in that it allows child-states to specialize on the more-general behavior of their parent-state, i.e., allows "exceptions to the rule" to be described.

It turns out that the behavior of these latter semantics can be obtained even under the former by associating the simple condition " $not\ in(b)$ " with the transition from cluster q to state d; However, the breaks encapsulation in that child-state b has to be known about outside of cluster q. Reversing the situation, that is being able to have the behavior of the former semantics even under the latter, is not possible (since it doesn't make any sense). Therefore, the former semantics make the most sense as they accommodate both arguments.

1.4.2.2 Cycling

Given the concurrent nature of statecharts, potential cycling problems can arise [1] [3]. Consider figure 1.9: If the statechart is initially in states a and c (as is indicated by the default arrows), what happens if event α is broadcast, that is, what are the next set of states that the statechart will be in?

One possible sequence of events is: event α occurs; this causes a transition from state a to state b and broadcasts event β ; this causes a transition from state c to state d and broadcasts event γ ; this causes a transition from state b to state a and broadcasts event δ ; this causes a transition from state

d to state c and broadcasts event α ; this causes the cycle to repeat forever. Clearly, this chain-reaction/infinite-looping is not desirable. This difficulty can be eliminated by a clear specification of state-chart semantics (§1.4.3) and an appropriate algorithm to accomplish transitions (§1.4.4).

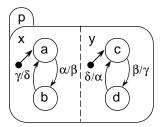


Figure 1.9: Potential cycling problem

1.4.3 Semantics

1.4.3.1 Statechart Definition

A statechart is a fivetuple (*S*, *A*, *T*, *C*, *E*) where:

- 1. *S* is the set of all states.
- 2. *A* is the set of all *active* states: $A \subset S$. An active state is one that the statechart is currently *in* (probably among others). Hence, a state has an associated *status*.
- 3. *T* is the set of all transitions between states.
- 4. C is the set of all transitions where the associated conditions are true: $C \subset T$. A transition that does not otherwise have an associated condition has a default-condition associated with it that always evaluates to be true.
- 5. *E* is the set of all events.

(This definition differs slightly from [1] because it is felt that this definition is clearer. The reader who curious about the intensely gory version is referred to that paper.)

1.4.3.2 Transitions and Ordering

When a state has more than one transition with an associated condition that is true, one transition is selected arbitrarily; when a parent- and child-state both have such a transition, the parent-state's is selected.

Upon a transition, states are always exited from the innermost out, i.e., child-states are always exited before their parent-state; likewise, states are always entered from the outermost in, i.e., a parent-state is entered before any of its child-states. For sets, the relative order that child-states are

In most contexts, such as conventional programming languages, "infinite-loops" are considered to be "bad." In the context of statecharts, it might be tempting to view cycling as a "good." To quell this feeling, one has to keep in mind that the value of statecharts (and STDs) is in their ability to be (and remain) in a given state or set of states waiting for something to happen. If statecharts were allowed to cycle, fueled only by internally-generated events, they would never "settle down" and pay attention to "the outside world" ignoring externally-generated events. This would not be good.

exited and entered is undefined. Additionally, all states to be exited in response to a given event are exited before any state is (re)entered for the transitions on that event.

A transition's broadcast event and action, if any, are carried out for each transition in turn immediately after the source state is exited and before any target state is (re)entered.

1.4.3.3 Events and Stability

Events, both externally- and internally-generated (via broadcasting), when they occur, shall not be queued; if there is no transition to be taken in response to an event when it occurs, it is discarded. A statechart is said to be *stable* after all states involved in a transition in response to an event have been either exited or entered.

1.4.4 Transition Algorithm

The algorithm for performing a set of transitions upon the occurrence of an event is given in figure 1.10 [6]. It is presented in Pascal-like pseudo-code, but with an object-oriented-paradigm slant to it [10]; hence, there are three versions of the functions <code>Enter()</code> and <code>Exit()</code> corresponding to states, clusters, and sets in figures 1.10a through 1.10f.

The -> notation and the keyword *this* are borrowed from C++. The -> applies the function on the right-hand-side to the object on the left-hand-side; the keyword *this* refers to the current object of interest, i.e., *this* state, *this* cluster or *this* set.

Although not explicitly shown (for lack of space in the figures), the <code>Enter()</code> and <code>Exit()</code> functions return a boolean value: true indicates that the state was entered or exited as a result of the call to the function, i.e., it was not already active or inactive, respectively; false indicates that it was already active or inactive. This is used to prevent a state from performing the steps associated with becoming active or inactive more than once.

Arguments, when passed, are omitted in places; where this is the case, they default to nil, represented by \emptyset . The to argument of the <code>Exit()</code> function refers to the target state, i.e., the state that the current state is exiting to; the <code>fromChild</code> argument of the <code>Exit()</code> and <code>Enter()</code> functions is passed when a non-local transition is taking place.

For state entrances, this is checked for by checking the status of state's parent-state. If it is inactive, then this means that the child-state is being entered directly, bypassing its parent-state. In such cases, the child-state must defer its entrance by entering its parent-state instead, passing a pointer to itself in the fromChild argument so that its parent-state, if it is a cluster, will know which child-state to enter overriding its default state and history, if any; the parent-state will enter the child-state in due course. If the parent-state is active, this means that just a local transition is taking place. In such cases, the child-state must notify its parent-state of the change; this only has significance for clusters so that they can update their record of which child-state is active and which is the last one active if they have a history. (The Notify() procedure for clusters is shown in figure

1.10g.) For sets, since all child-states are either active or inactive, the particular child-state is irrelevant.

```
function State::Exit( to, fromChild: State )
                                                    function Cluster::Enter( fromChild: State )
    if not active then
                                                         if not State::Enter() then
        return false
                                                             return false
    unmark as active
                                                         if children \neq \emptyset then
    Broadcast( exited( this ) )
                                                             if from Child \neq \emptyset then
                                                                  lastChild = fromChild
    if parent \neq \emptyset and to \neq \emptyset and
        parent ∉ { ancestors of to } then
                                                             elsif lastChild = Ø or not history then
            parent->Exit( to, this )
                                                                  lastChild = default child
                                                             activeChild = lastChild
    return true
                                                             activeChild->Enter()
end
                     (a)
                                                         return true
                                                     end
function State::Enter(fromChild: State)
                                                                               (d)
begin
    if active then
                                                    function Set::Exit( to, fromChild: State )
        return false
                                                    begin
    if parent \neq \emptyset then
                                                         if active then
        if parent is inactive then
                                                             forall \{ \sigma \mid \sigma \in \{ \text{ children } \} \} do
            parent->Enter( this )
                                                                  \sigma -> Exit()
                                                             return State::Exit( to )
            return true
                                                         return false
        else
            parent->Notify( this )
                                                    end
                                                                               (e)
    mark as active
    Broadcast( entered( this ) )
                                                    function Set::Enter( fromChild: State )
    return true
                                                    begin
end
                                                         if not State::Enter() then
                     (b)
                                                             return false
function Cluster::Exit( to, fromChild: State )
                                                         forall \{ \sigma \mid \sigma \in \{ \text{ children } \} \} do
begin
                                                             \sigma->Enter()
    if active then
                                                         return true
        if from Child = \emptyset then
                                                     end
                                                                               (f)
             activeChild->Exit()
        return State::Exit( to )
                                                    procedure Cluster::Notify( byChild: State )
    return false
                                                     begin
end
                                                         activeChild = lastChild = byChild
                     (c)
                                                     end
                                                                               (g)
```

Figure 1.10: Transition algorithm

Similarly to state entrances, exits caused by non-local transitions must also be accounted for. This is checked for by seeing if the source state's parent-state is an ancestor of the target state. If it is not, then the child-state must force its parent-state to exit also.

The main procedure of the transition-algorithm is shown in figure 1.10h. First, for all active source states for all transitions on the given event whose conditions are true at the time, exit those states and all their child-states, if any, recursively, marking all the transitions involved as having been taken. Then recursively make transitions on the broadcast event associated with the

transitions, if any, and perform the action associated with the transitions, if any. Finally, for all those transitions that were previously marked as having been taken for a given event, enter all of the associated target states.

```
 \begin{array}{l} \textbf{procedure} \ Broadcast(\epsilon : Event\,) \\ \textbf{begin} \\ \textbf{forall} \ \tau, \ \alpha, \ \beta \ | \ \tau \in \{ \ \epsilon' \ s \ transitions \ \cap C \ \}, \ \alpha \in \{ \ \epsilon' \ s \ source-states \ \cap A \ \}, \ \beta \in \{ \ \epsilon' \ s \ target-states \} \\ \textbf{do} \\ \alpha -> Exit(\ \beta) \\ \textbf{if} \ broadcast-event} \ associated \ with \ \tau \neq \emptyset \ \textbf{then} \\ Broadcast(\ broadcast-event \ associated \ with \ \tau) \\ perform \ action \ associated \ with \ \tau, \ if \ any \\ mark \ \tau \ as \ taken \\ \textbf{end} \\ \textbf{forall} \ \tau, \ \beta \ | \ \tau \in \{ \epsilon' \ s \ transitions \ \cap \ taken \ \}, \ \beta \in S \ \textbf{do} \\ \beta -> Enter() \\ unmark \ \tau \\ \textbf{end} \\ \textbf{end} \\ \textbf{end} \\ \textbf{end} \\ \textbf{end} \\ \end{array}
```

Figure 1.10 (continued): Transition algorithm

1.4.5 The Cycling Problem Revisited

Armed with the transition-algorithm just given, the cycling problem raised in §1.4.2.2 can now be dealt with effectively. Starting again with the statechart of figure 1.9 initially in states a and c, the sequence of events is now as follows: event α occurs; state a, having a transition on α , is unmarked as an active state and event β is broadcast; state c, having a transition on β , is unmarked as an active state and event γ is broadcast; no active states have a transition on γ (state b is *not* considered since it has not been marked as an active state yet). Continuing, state d is marked as an active state, and finally, after completing the effects of broadcasting β , state b is marked as an active state. Hence, the final set of states that the statechart is in are states b and d.

1.4.6 An Extended Example

1.4.6.1 Overview

To illustrate how statecharts can be used, an extended example of the states and functionality of a microwave oven is presented. The control panel of the oven is shown in figure 1.11. The oven works as follows:

1. To cook, enter the amount of time digit-by-digit then press start. As digits are entered, they are "shifted" to the left; if more than four digits are entered, the left-most digit is "shifted off" when a new digit is entered. To set a power-level lower than 100%, press power followed by the first digit of the percentage before pressing start. Alternatively, to cook for one minute at full power, or to add one minute to the remaining time, press minute. To stop cooking, press

stop or open the door; to continue cooking, close the door if it was opened, and press start. To cancel cooking once stopped with the door closed, press stop again.

1	2	3	12:	52
4	5	6	Power	Clock
7	8	9	Start	Stop
0	Minute		Open	

Figure 1.11: Microwave oven panel

- 2. To set the time, press clock, then enter the time digit-by-digit, then press clock again. Just as with setting the cooking time, digits are shifted to the left as entered. To cancel the setting of the time and restore the previous time, press stop instead of clock. If clock is pressed and an invalid time is currently on the display, e.g., 00:01 or 17:00, "EE" will be displayed; to restart, press stop.
- 3. If the door is opened at any time during either sequence, the oven is disabled; closing the door resumes the sequence from where it left off.

1.4.6.2 Statechart Specification

The high-level statechart for the microwave oven is shown in figure 1.12a; although there would probably be transition-actions, they have not been shown for clarity. It has three major components: its mode, either disabled or operational, its display, and its light.¹³ The initial state of the microwave oven is that it is operational, displaying the time, and that the light is off.¹⁴

The mode is determined by the status of the door: if open, the oven becomes disabled leaving whichever of the operational states it was in; also, since an exit from the Operational cluster first causes an exit from one of its child-states, if the oven was in Cook, *exited*(Cook) gets broadcast and the light goes off (which is what is supposed to happen). When the door is closed again, the history of the Operational cluster places the oven back in whatever operational mode it was in before; if Cook is reentered, *entered*(Cook) is broadcast and the light goes back on.

The Display normally shows the time; when Program is entered, Not-Time gets entered to display either the counter, the power-level, or "EE" for the error state while setting the time; Not-Time gets exited whenever Idle is entered. Not-Time also has an error state to parallel that of the Set-Time cluster.

¹³ To make the example a little simpler, it is assumed that the microwave oven is plugged in to a live electrical receptacle.

¹⁴ The naming convention adopted here is that event's names are in lower-case letters and state's names are capitalized.

Both the Program, Set-Time, and Cook states are actually clusters; these are shown in figure 1.12b. When the Program cluster is entered, the amount of time is collected digit-by-digit and the power-level is obtained when power is pressed. The Program cluster has a history to remember which of the two child-states to reenter if the door is opened then closed while the oven is being programmed.

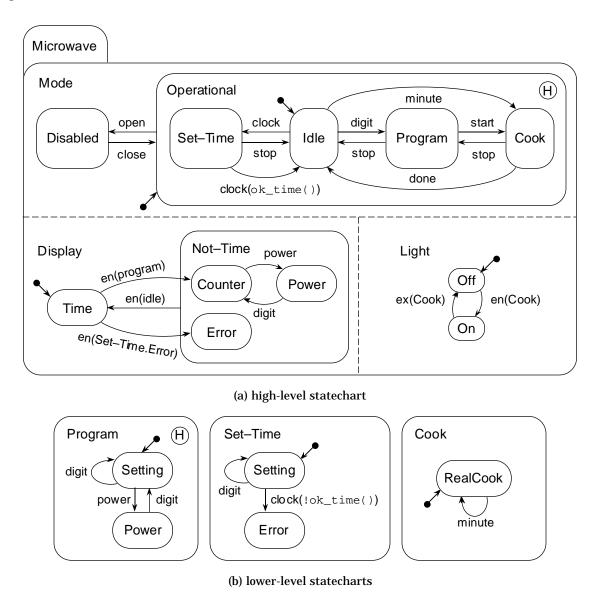


Figure 1.12: Microwave oven statecharts

The Set-Time cluster has a transition upon receipt of clock with the associated condition ok_time(), a user-defined C++ function for checking the validity of the time. The Setting child-state has a transition to the Error state if the time is invalid. This Error state, along with the one in the Display cluster, have no direct transitions to "get out" of being in an error condition; instead, they inherit a transition from their respective parents.

The Cook cluster might seem that it could just be a plain-state, i.e., what is the purpose of the single RealCook child-state? The light is supposed to be on only when the oven is cooking; hence, the transitions on *enter*(Cook) and *exit*(Cook) in the Light cluster. When minute is pressed while cooking, a minute is to be added to the remaining cooking time, i.e., a transition-action would presumably add sixty seconds to some variable storing the remaining time. In this situation, a transition to another state is not wanted, just the side-effect of its action. If Cook were to transition on minute back to itself, the light would flicker off then back on because the Cook state would be exited then reentered. This is the reason for the RealCook child-state: to obtain the side-effect of a transition-action, but without exiting the Cook cluster itself. ¹⁵

-

Initially, this was not realized. It was only as a result of using the language system described in chapter 2 where the statechart for the oven is implemented that the light flicker was noticed; hence, a design-flaw was caught and corrected.

Chapter 2

A Language System

2.1 Overview

As stated in §1.3, software for a language system to implement CHSMs can be developed. A language for specifying CHSMs and the method for preparing the description will now be presented.

2.2 Specifying a CHSM

2.2.1 The Description File

CHSMs can be described by means of a CHSM description file that is an ordinary text file with three sections: declarations, the CHSM description, and an optional user-code section. A template CHSM description file is shown in figure 2.1. Sections are separated by the %% token (mirroring *yacc* grammar descriptions). If the user-code section is omitted, the trailing %% may also be omitted.

declarations
%%
description
%%
user code

Figure 2.1: CHSM description-file template

Since CHSM descriptions are usually integrated with C++ source-code, the declarations section would contain any necessary or useful declarations, such as global variables, constant declarations, etc., and preprocessor directives that the user wants to be used within C++ code fragments in the subsequent sections.

The description section would naturally contain a CHSM description given in some sort of a "CHSM description language." ¹⁶ An example of what the language looks like is given in figure 2.2b where the statechart of figure 2.2a is expressed. (The details of the syntax are covered in the following section as well as in Appendix A: The Language Reference Manual.)

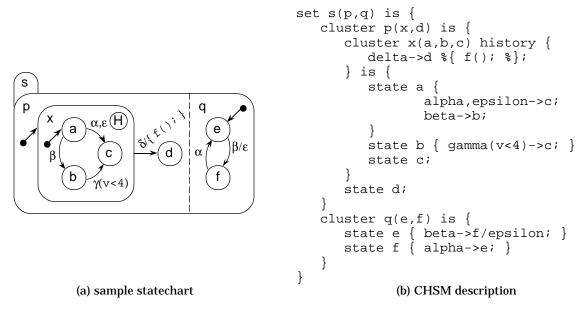


Figure 2.2: CHSM description-language example

The user-code section, if given, would contain any function definitions presumably declared in the declarations-section.

2.2.2 Specifying States

States are specified by means of a *state-description*. A state-description consists of either one of the keywords state, cluster, or set, followed by the state's name and its description.¹⁷ What may or must be specified in the description depends on whether the state being described is a plain-state, cluster, or set.

Plain-state descriptions are the simplest. In figure 2.2b, states a through e are described starting with the keyword state followed by their name and optionally by a list of transitions enclosed in braces. (If a plain-state has no transitions, as state d doesn't, then the transition-list and the braces are replaced by a semicolon.)¹⁸

 $^{^{16}}$ One proposal for a name of the CHSM description language is "CDL."

¹⁷ There may also be a derived-type specification; but coverage of this feature is deferred until §2.2.6.

¹⁸ Such a state can only be escaped from as the result of taking a transition possessed by its parent state. This allows it to be used as a "dead-end" to be transitioned to in the event of some sort of error condition arising.

Cluster and set descriptions are just like plain-state descriptions except that more information must be provided. First, they must declare the names of their child-states, enclosed in parentheses, following their name. Just like plain-states, clusters and sets can have transitions of their own. Cluster x in figure 2.2a has such a transition and it is specified after the child-state declaration. If a cluster has a history, as x does, the keyword history is placed after the child-state declaration and before its transition-list, if any. (If a cluster has a deep history, as none in the figure do, then the keyword deep is placed before history.) Second, cluster and set descriptions must have child-states described in their *bodies*, enclosed in braces, after the keyword is. For clusters, the first child-state defined (not declared) is it's default child-state.

2.2.3 Specifying Transitions

There are many options for transition-specifications. The notation used in the language was chosen to mimic, as closely as possible, the graphical statechart notation; hopefully, much of what follows will be intuitive.

At its simplest, a transition specifies that one state makes a transition to another as the result of the occurrence of some event; such is the case for the transition from state a to state b on event β . This is expressed as: beta->b;, that is, the event's name followed by the arrow token ("->") followed by the target state's name; all transition-specifications must be terminated by a semicolon.

More than one event may be specified to trigger a transition as is the case with the transition from state a to state c on either event α or ϵ ; multiple events are separated by commas as in: alpha,epsilon->c;.

Transitions having events with associated conditions, as the transition on event γ does with its condition being that some global variable v is less than 4, are specified with the condition enclosed in parentheses following the event's name as in: gamma(v<4)->c;

The expression used for a condition can be any arbitrary, valid C++ expression that returns a value.¹⁹ There also exists a built-in CHSM function $\sin()$, which takes a state-name as an argument, that returns true if that state is one of the ones that the CHSM is currently in. For example, to make a transition from some state a to some state b on event α if the CHSM is currently in some other state c, the transition-specification would be: state a { $alpha(\sin(c))->b;$ }.

Transitions may broadcast an event as the transition from state e to state f on event β does with event ϵ by having the name of the event to be broadcast placed after the target state's name and a slash ("/") as in: beta->b/epsilon;

Lastly, transitions may have an action performed as the transition from cluster x to state d on event δ does by having the C++ code enclosed in the *begin*- and *end-code* tokens (" $\{$ " and " $\{\}$ ")

It would be illegal to have the only thing or the last thing in a comma-expression in the condition-expression be a call to a C++ function with a return-type of void.

placed after the broadcast event's name, if any, as in: $delta->d %{f(); %};$. (For this action, the global C++ function f() is called.)

An action for a transition may contain zero or more arbitrary, valid C++ statements. The code is treated exactly as though it were contained in its own unique function having a return-type of $_{\text{void.}}^{20}$

Within C++ code specified for conditions and actions, the variable event is available; it is set to the event that is causing the transition. Figure 2.3 shows how this variable might be used when more than one event can trigger a transition.

Figure 2.3: Use of the event variable in C++ code

2.2.4 Specifying Events

In figure 2.2b, nothing special had to be done to specify event names since they do not have to be pre-declared. Names chosen for events must be unique throughout the CHSM description; it is illegal to have two events or an event and a state with the same name.

In §1.4.1.3, *enter* and *exit* events, which are implicitly broadcast upon entrances to and exits from states, respectively, were introduced; such events can be also be used as the triggering events for transitions. For example, to make a transition from some state a to some state b whenever some other state c is entered, the transition specification would look like: state a { enter(c)->b; }, that is, the state of interest has its name enclosed in parentheses preceded by either of the keywords enter or exit. Conditions may still be associated with such events. Neither *enter* nor *exit* events may be explicitly broadcast. Within C++ code specified for transition-actions, *enter* and *exit* events are referred to by prefixing the keyword enter or exit with a \$ as shown in figure 2.4.

Figure 2.4: Referring to enter and exit events in C++ code

There can also be *macro-events*. A macro-event is a collection of ordinary events grouped under one name. For example, a CHSM describing a vending-machine would probably make a transition

That's because that's how code for actions is actually implemented.

upon the insertion of a coin. Without macro-events, each coin would have to be listed for a transition that takes place when any coin is inserted as shown in figure 2.5a. There may be several transitions in the CHSM description involving coins; repeatedly having to list each coin might be tedious.

Figure 2.5: Macro-events

Instead, a macro-event coin can be declared and used as shown in figure 2.5b. Now, transitions need only have coin listed and the transition will take place regardless of which coin was inserted. Note that coin is not a real event in that the user can not insert a nondescript "coin" object; rather, a *particular* coin, nickel, dime, or quarter, would still be inserted. A macro-event is *just* a shorthand name for any of the events it is composed of.

Events have an implicit numerical value; unless otherwise specified, it is zero. Events listed as belonging to a macro-event are assigned values progressing from zero. In figure 2.5b, the value for nickel, dime, and quarter is 0, 1, and 2, respectively. In C++ code given for event-conditions and transition-actions, the name of an event, when used in a numerical expression, evaluates to its numerical value. Events can alternatively have values explicitly assigned to them as is shown in figure 2.5c. For nickel, dime, and quarter, the obvious values are 5, 10, and 25, respectively; these values then can be added directly to the total amount of money deposited, for example.

```
event nickel = 5;
event dime = 10;
event quarter = 25;

(a)

event { nickel = 5, dime = 10, quarter = 25 }
(b)
```

Figure 2.6: Event numerical-value assignment

If just an alternate numerical value is desired, then events can be listed in either of the forms shown in figure 2.6.

2.2.5 Specifying State-Names

Since state-names are local to a cluster or set, more than one cluster or set may have child-states with the same name. Figure 2.7 has been contrived to show how to refer to non-local state-names with the same name as local state names.

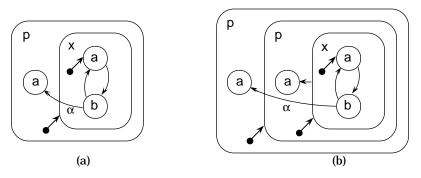


Figure 2.7: Referencing non-local, hidden state-names

In figure 2.7a, the transition from state b on event α is supposed to go to cluster p's child-state a. If the transition were specified as: state b { alpha->a; }, the "a" would be taken to mean cluster x's child-state aand not p 's. The way to indicate a non-local state-name is by specifying the scope of the desired state. There are three ways to do this. The most straight-forward way is by preceding the state name by that of its parent-state's and a period. The transition could now be specified as: state b { alpha->p.a; }.

An even more contrived example is shown in figure 2.7b. Here, the outermost state a is the desired state, but specifying p.a as before would be taken to mean the intermediate state a. The other two ways to specify a state's scope can be used in such cases. The first is to use *relative* specification where a state's name is preceded by one period for every scope to be "backed-up" out of; the state-name could either be specified as ..a or .p.a. The second way is to use *absolute* specification where a state's name is preceded by a double colon and *all* of its parent-state's names separated by periods; the state-name could be specified as ::p.a. The difference between the two methods can be summed-up as: relative specification starts are the current scope and works its way out, and absolute specification starts from the outermost scope and works its way in.

2.2.6 Derived Classes

In the implementation of the run-time library (chapter 3), states, clusters, and sets are actually variables of the C++ classes State, Cluster, or Set, respectively. Such classes have predefined data-members and member-functions; the user can derive classes from these classes to add data-members and member-functions or to augment their behavior when states are either entered or

exited. States, clusters, and sets can be of user-defined derived-classes instead of the ordinary ones. (This section is meant to be just an overview; for complete details on derived-classes, see the Language Reference Manual in appendix A.)

One example of augmenting the behavior of states would be to mimic the behavior of Petri Nets by adding a token-count to states [11]. The example shown in figure 2.8 declares a Token class derived from the State class. The State class has member-functions Enter() and Exit() that are called whenever a state is entered or exited, respectively. Here, every time a token-state is entered, the number of tokens is incremented.

```
class Token : public State {
    int tokens;
public:
    Token( STATE_ARGS ) : State( STATE_INIT ) { tokens = -1; }
    Boolean Enter( Event const & event, State *fromChild = 0 ) {
                  ++tokens;
                  return State::Enter( event, fromChild );
              operator = ( int used ) { return tokens -= used + 1; }
    int
              operator int() const { return tokens; }
};
int water molecules = 0;
응응
set H2O( H2,O2 ) {
    hydrogen( \{H2\} >= 1 \&\& \{O2\} >= 1 \})->H2O %
         ${H2} -= 1, ${O2} -= 1; ++water_molecules;
    왕};
    oxygen( $\{H2\} >= 2 \}-H20 %
         ${H2} -= 2; ++water_molecules;
    응};
} is {
    state<Token> H2 { hydrogen->H2; }
    state<Token> 02 { oxygen->02; }
}
```

Figure 2.8: Derived-class example

The constructor first calls State class's constructor supplying the necessary arguments, as generated by the CHSM compiler, then initializes the number of tokens. (The reason that it is initialized to -1 is that it will be incremented to zero when the state is entered for the first time.) Additionally, the <code>operator==()</code> and <code>operator int()</code> member-operators have been added where <code>operator==()</code> decrements the number of tokens and <code>operator int()</code> converts a token-state into the number of tokens it has accumulated for use in numerical expressions.

The purpose of the CHSM is to make water molecules when the correct number of hydrogen and oxygen molecules are present in accordance with the familiar equation from a freshman chemistry class: $2H_2 + O_2 \rightarrow 2H_2O$. The CHSM is a set composed of the states H2 and O2 that make

transitions to themselves upon the receipt of the events *hydrogen* and *oxygen*, respectively. Because the Enter() member-function has been overridden, whenever this happens, the respective number of hydrogen and oxygen molecules is incremented.

The parent set H2O also has transitions on the events *hydrogen* and *oxygen*, but with conditions: for the transition on *hydrogen*, if the number of hydrogen and oxygen molecules (tokens) are both greater-than or equal-to one, then a water molecule can be created. On the event *hydrogen*, the number of hydrogen and oxygen molecules are both decremented by one and the number of water molecules is incremented. (The reason that the number of hydrogen molecules is only decremented by one instead of two is because the occurrence of the event *hydrogen* means that another hydrogen molecule (token) has just become available, but has not been counted yet because the transition of the parent set supersedes that of the child-state H2.) For the transition on *oxygen* in the parent set, only the number of hydrogen molecules needs to be checked. On the event *oxygen*, the number of hydrogen molecules is decremented by two and again the number of water molecules is incremented. (The reason that the number of oxygen molecules is not decremented is similar to the reason that the number of hydrogen molecules is only decremented by one on the event *hydrogen*.)

The $\{...\}$ notation is used to refer to a state-name within C++ code specified for condition-expressions and transition-actions to distinguish it from ordinary C++ code.²¹

The states H2 and O2 are declared to be states, but they are Token states; this is done by enclosing the derived-class's name within angle-brackets immediately after the keyword state.

2.3 An Extended Example, Continued

The CHSM description and C++ implementation of the statechart for the microwave oven given in §1.4.6 will now be presented. Figure 2.9 first shows just the CHSM; it is a straight-forward, direct transliteration of the statecharts shown in figure 1.12. Subsequent figures in this section add C++ code to make an actual working example.²²

The macro-event digit is declared to comprise all digits since they are treated equally; each digit is assigned its respective numerical value.

Figure 2.10 shows one preliminary, inline function-definition used by subsequent C++ code. Its purpose is to "shift" numbers to the left as new digits are added in the same manner as digits appear when entered on a typical calculator; its precision is two places.

 $^{^{21}}$ This notation is borrowed from the KornShell language; Bolsky et al. [12], p. 171.

While then CHSM itself is complete, the C++ code included is just enough to show how C++ can be used in CHSMs and does not purport to be the "best possible" code for the microwave oven simulation. Better code to draw an oven on the screen graphically, complete with user-clickable buttons, could have been written, but it would only have been "window dressing."

```
응응
event digit is { _0, _1, _2, _3, _4, _5, _6, _7, _8, _9 }
set Microwave( Mode, Display, Light ) is {
    cluster Mode( Operational, Disabled ) is {
         cluster Operational( Idle, Program, Cook, SetTime ) history {
             open->Disabled;
         } is
             state Idle { digit->Program; minute->Cook; clock->SetTime; }
             cluster Program( Setting, Power ) history {
                  start->Cook; stop->Idle;
             } is {
                  state Setting { digit->Setting; power->Power; }
                                { digit->Setting; }
                  state Power
             cluster Cook( RealCook ) { done->Idle; stop->Program; } is {
                  state RealCook { minute->RealCook; }
             cluster SetTime( Setting, Error ) { stop->Idle; } is {
                  state Setting {
                      digit->Setting;
                       clock( ok_time())->Idle;
                       clock(!ok_time())->Error;
                  state Error;
         state Disabled { close->Operational; }
    cluster Display( Time, NotTime ) is {
         state Time {
             enter( Mode.Operational.Program )->NotTime.Counter;
             enter( Mode.Operational.Program.Error )->NotTime.Error;
         cluster NotTime( Counter, Power, Error ) {
             enter( Mode.Operational.Idle )->Time;
             state Counter { power->Power;
             state Power { digit->Counter; }
             state Error;
    cluster Light( Off, On ) is {
         state Off { enter( Mode.Operational.Cook )->On; }
         state On { exit ( Mode.Operational.Cook )->Off; }
    }
}
```

Figure 2.9: Microwave oven CHSM

For the Program cluster, it was decided to make it a derived class. The Timer class, derived from the Cluster class, maintains the cooking time and power-level, overrides the Enter() and Exit() member-functions, and defines a few member-functions of its own. The class declaration is shown in figure 2.11.

The Enter() and Exit() member-functions are defined out-of-line in the user-code section (figure 2.14); the Display() member-function prints the current value of the timer as digits are

entered; Bump() takes a digit and "shifts" it onto the current value of the timer yielding a new value; AddMinute() adds one minute to the remaining cooking time; SetPower() takes an integer argument and sets the power-level to it, printing it in confirmation.

```
inline void ShiftAdd( int &victim, int amount ) {
   if ( (victim = victim * 10 + amount) >= 100 )
     victim %= 100;
}
```

Figure 2.10: Preliminary declarations

```
class Timer : public Cluster {
    int minutes, seconds, power_level;
public:
    Timer( CLUSTER ARGS ) : Cluster( CLUSTER INIT ) { }
    Boolean Enter( Event const&, State* = 0 );
    Boolean Exit ( Event const&, State* = 0, State* = 0 );
             Display() const {
                  cout << "Timer: " << minutes << ':' << seconds << endl;</pre>
    void
             Bump( int digit ) {
                  ShiftAdd( minutes, seconds/10 );
                  ShiftAdd( seconds, digit );
                  Display();
    void
             AddMinute() {
                  if ((seconds += 60) > 99)
                       seconds -= 60, ++minutes;
                  Display();
    void
             SetPower( int level ) {
                  power_level = level * 10;
                  cout << "Power: " << power level << "%\n";</pre>
              }
};
```

Figure 2.11: Timer class declaration

Similarly, the SetTime cluster is also of a derived class. The Clock class maintains the time-of-day, overrides the Enter() and Exit() member-functions, and also defines a few member-functions of its own. The class declaration is shown in figure 2.12.

Again, the <code>Enter()</code> and <code>Exit()</code> member-functions are defined out-of-line in the user-code section (figure 2.14); the <code>Display()</code> and <code>Bump()</code> member-functions are similar to those of the <code>Timer</code> class; <code>ok_time()</code> returns the validity of the newly proposed time-of-day.

Now in figure 2.13, only the parts of the CHSM that now have C++ code in their transition-actions are shown (the line-numbers, of course, are not part of the description).

On line 2, the Program cluster is declared to be a cluster of the derived-class Timer, as opposed to an ordinary cluster; similarly for the SetTime cluster on line 18 of the derived-class Clock.

```
class Clock : public Cluster {
     int hour, minute;
     static int old_hour, old_minute;
public:
     Clock( CLUSTER_ARGS ) : Clock( CLUSTER_INIT ) {
         hour = 12, minute = 0;
     Boolean Enter( Event const&, State* = 0 );
     Boolean
              Exit ( Event const&, State* = 0, State* = 0 );
     void
              Display() const {
                   cout << "Time: " << hour << ':' << minute << endl;</pre>
     void
              Bump( int digit ) {
                   ShiftAdd( hour, minute/10 );
                   ShiftAdd( minute, digit );
                   Display();
              ok_time() const {
     Boolean
                   return hour >= 1 && hour <= 12 && minute <= 59;
};
                         Figure 2.12: Clock class declaration
1
     state Idle { digit->Program; minute->Cook; clock->SetTime; }
2
     cluster<Timer> Program( Setting, Power ) history {
3
         start->Cook; stop->Idle;
4
     } is {
5
         state Setting {
6
              digit->Setting %{ ${Program}.Bump( event ); %};
7
              power->Power;
8
9
         state Power {
10
              digit->Setting %{ ${Program}.SetPower( event ); %};
11
12
13
     cluster Cook( RealCook ) { done->Idle; stop->Program; } is {
14
         state RealCook {
15
              minute->RealCook %{ ${Program}.AddMinute(); %};
16
17
18
     cluster<Clock> SetTime( Setting, Error ) { stop->Idle; } is {
19
         state Setting {
20
              digit->Setting %{ ${SetTime}.Bump( event ); %};
21
              clock( ${SetTime}.ok_time())->Idle;
22
              clock(!${SetTime}.ok_time())->Error %{
23
                   cout << "Time: EE\n";</pre>
24
              응 } ;
25
26
         state Error;
27
```

Figure 2.13: Microwave oven CHSM with C++ code added

// ...

On line 6, for every digit entered while setting the cooking time, the transition-action "bumps it up" by calling the <code>Bump()</code> member-function of the <code>Program</code> cluster; similarly for the setting of the

time-of-day for the SetTime cluster on line 20. In both cases, the pre-declared event variable is passed as the argument evaluating to the numerical value of whichever digit was entered.

On lines 10 and 15, the SetPower() and AddMinute() member-functions are called in the transition-actions to perform the proper actions.

On lines 20 and 21, for the setting of the time-of-day, the validity of the time is checked. If it is valid, the CHSM returns to the Idle state; if not, the Error state is entered until stop is pressed. The transition-action for the transition going to the Error state also prints the "EE" indicating the error to the user.

```
응응
    Boolean
Timer::Enter( Event const &event, State *fromChild ) {
    if ( !Cluster::Enter( event, fromChild ) )
        return false;
    if ( event != close )
        minutes = 0, seconds = event, power level = 100;
    Display();
    return true;
}
    Boolean
Timer::Exit( Event const &event, State *to, State *fromChild ) {
    if ( !Cluster::Exit( event, to, fromChild ) )
         return false;
    if ( event != open )
         Clear();
    return true;
}
    Boolean
Clock::Enter( Event const &event, State *fromChild ) {
    if ( !Cluster::Enter( event, fromChild ) )
        return false;
    if ( event != close ) {
        old_hour = hour, old_minute = minute;
         hour = minute = 0;
    Display();
    return true;
}
    Boolean
Clock::Exit( Event const &event, State *to, State *fromChild ) {
    if ( !Cluster::Exit( event, to, fromChild ) )
         return false;
    if ( event != open ) {
         if ( !ok_time() )
             hour = old_hour, minute = old_minute;
         Display();
    return true;
}
```

Figure 2.14: Microwave oven CHSM user-code section

The user-code section shown in figure 2.14 supplies the code overriding the <code>Enter()</code> and <code>Exit()</code> member-functions for the <code>Timer</code> and <code>Clock</code> derived classes.

Both the Timer::Enter() and Clock::Enter() member-functions call Cluster::Enter(), passing the necessary arguments, to perform the standard actions; if it returns false, then the remaining actions should not be performed. The Timer::Enter() member-function resets the cooking-time and power-level only if the event that caused the Program cluster to be entered is not close; if it was close, it means that the user opened then closed the door, for whatever reason, causing transitions out of the Operational cluster to the Disabled state and back again. In this case, the programming of the cooking-time is supposed to resume from where it left off.

The Timer::Exit() member-function first calls Cluster::Exit(), passing the necessary arguments, to perform the standard actions. The Program cluster should remember its history only if the door was opened then closed again; hence, the check on the event. If it is not open, then its history should be cleared.

The Clock::Enter() member-function saves the current time-of-day in the static data-members old_hour and old_minute to be restored if the user cancels setting the time or if the new time is invalid. It then sets the current hour and minute to zero in preparation for the user to enter a new time.

The Clock::Exit() member-function first calls the Cluster::Exit() member-function, similarly passing the necessary arguments, to perform the standard actions of a cluster. If the time-of-day is not valid when the SetTime cluster is being exited, then the previously-stored time is restored.

Chapter 3

Run-Time Library Design

3.1 Overview

The CHSM run-time library is what "makes it all go" as it supplies the code in which a CHSM operates. In designing the run-time library, the following list of information was taken into consideration since, in a CHSM, it all has to be kept track of by some means:

- The CHSM has to know all the states that comprise it; of those, either it has to know which
 subset of states are active or the individual states themselves have to know what state they
 are in, either active or inactive.
- States have to know which other states are their child-states and which state is their parent. Clusters have to know which child-state is the default one and whether they have a history; if they do, they additionally have to know which child-state was the last one active.
- States have to know which events they make transitions on, whether there is an associated
 condition, which states they make transitions to, whether there is an event to be broadcast,
 and whether an action is to be performed.

Additionally, the following were goals aspired to in the library design; it should:

- Run fast: This can, among other things, mean perform little or no computation nor "table lookup" of information, i.e., everything should be at worst trivially computable and directly accessible.
- Be memory-efficient: Store what is only absolutely necessary unless run-time performance will be seriously degraded.
- Impose no requirements on the end-user to perform actions to initialize the CHSM.
- Be easily tailorable and extensible.
- Be "elegant." Often, elegance must be sacrificed for speed and memory-efficiency.

3.2 Resultant C++ Code

During the class library design, it became necessary to develop at least a tentative design for the resultant C++ code to be generated by the CHSM compiler since the code generated shall dictate a number of things, such as the necessity for certain data-members, what arguments constructors will take, etc. Since this system is being developed in the tradition of *yacc*, the approach it uses was first considered.

The resultant C code generated by *yacc* contains everything necessary to compile it including a complete definition of its main function <code>yyparse()</code>. User-code associated with the grammar productions is folded within a (huge) <code>switch</code> statement in <code>yyparse()</code> where each <code>case</code> is the user-code for a single production. This approach, however, was rejected.

It was desired that the object-code for the library be pre-compiled and stand-alone: pre-compiled to speed compile-times and so as not to make the source-code public; stand-alone so that changes to the library can be made and CHSMs previously compiled need only be relinked with the new library.

Since the library source-code is not being embedded in the resultant C++ code, there would be no place to put a switch statement containing the user-code for condition-expressions and transition-actions; therefore, each code-fragment is placed into its own function.

A nice side-benefit of this is that, in describing the CHSM language, it can be said of such code that it has the semantics as though it were inside a C++ function—because it is. This is a nice, clean, easily-understandable concept and so it doesn't require the user to learn yet another thing.

It also disallows certain dubious actions on the part of the user. For example, in *yacc*, the user can say break in a code-fragment without an enclosing loop or switch statement and yet have the resultant C code compile since the break would exit the *enclosing* switch statement in yyparse().

3.3 Class Design

Being implemented in C++, it should come as no surprise that many things were implemented using classes. As it will become obvious, there are interdependencies between many of the classes, hence there is no way to describe a given class before all of the classes that it depends upon are.

Also, to show the "relevance" of how a class is used, the resultant C++ code generated by the CHSM compiler will be shown for the (tiny) CHSM in figure 3.1.

3.3.1 Events and Transitions

3.3.1.1 Design Considerations

Events—and events alone—make things happen; therefore, the logical thing to be considered first is an event. When a CHSM receives an event, how should it proceed, i.e., how should it go about accomplishing transitions? One approach would be to check each state to see whether it has a transition on that event and, if it does, check to see whether it is active. For a large CHSM, however, this would be wasteful since only a (possibly small) subset of all states make transitions on a given

event. Given a complete CHSM description, it is known exactly which states have a transition on a given event; therefore, for each event, it would make more sense to store a list of only those states that have a transition on it.

```
set p(x,y) is {
    cluster x(a,b) is {
        state a { beta->b; }
        state b { alpha->a; }
    }
    cluster y(c,d) is {
        state c { delta->d; }
        state d { enter(x.a)->c; }
    }
}
```

Figure 3.1: Example CHSM to illustrate resultant C++ code

While this scheme is better, it does not address the issue of where to store the other information associated with a transition, i.e., the target state, the condition, if any, the event to be broadcast, if any, and the action, if any. Clearly, this information should be stored with a transition—transitions are unique, whereas an event can trigger any number of transitions. As long as all of the aforementioned information is now being stored with a transition, the source state might as well be stored too. Now that this is the case, an event, instead of having a list of states that make transitions on it, has a list of the transitions themselves. This relationship is shown in figure 3.2.

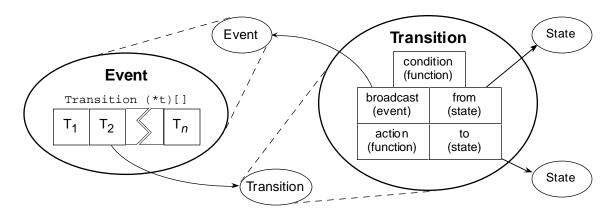


Figure 3.2: Event/Transition class relationship

3.3.1.2 The Event Classes

Now that a data-structure for an event has been shown, the Event class can be presented; its declaration is shown in figure 3.3.

The transitions data-member is implemented as a (constant) vector of (constant) pointers to transitions; the typedef is for convenience. The constructor takes such an argument and an integer to be used for the event's numerical value, defaulting to zero if none is given. The conversion

operator, operator int(), makes an event's name able to be implicitly converted (back) to its numerical value allowing it to be used in arithmetic expressions. The ${\tt Broadcast()}$ memberfunction is the core function for it implements the transition-algorithm; ${\tt operator()()}$ is provided as a synonym allowing (cute) statements like ${\tt alpha()}$ as opposed to ${\tt alpha.Broadcast()}$ when the user wants to broadcast an additional event in a transition-action. The equal and not-equal operators do the obvious things. ${\tt 23}$

```
class Event {
    typedef Transition *const *Transitions;
    Transitions const transitions;
public:
    int const value;

    Event( Transitions t, int n = 0 ) : transitions( t ), value( n ) { }
    operator int() const { return value; }

    void Broadcast() const;
    void operator()() const { Broadcast(); }
};

inline int operator==(Event const &a, Event const &b) { return &a == &b; }
inline int operator!=(Event const &a, Event const &b) { return &a != &b; }
```

Figure 3.3: Event class declaration

In the resultant C++ code, instances of the Event class are only (currently) used to implement *enter/exit* events.²⁴ An example of the code is shown in figure 3.4. So long as it is unique, the name used for an *enter/exit* event is arbitrary: for *enter* events, it is an E followed by the name of the state being entered; for *exit* events, it is an X followed by the state-name.

```
static Transition *const TES1p1x1a[] = {
    &CHSM::transition[3],
    0
};
static Event ES1p1x1a( TES1p1x1a );
```

Figure 3.4: Resultant C++ code for events

The name of the vector of transitions is also arbitrary: it is just a T followed by the event-name. The vector is initialized with the addresses of the transition data-structures stored in the transition vector data-member of the CHSM class (§3.3.3); a nil-pointer marks the end of the

Since there is only one instance of a given event, it is sufficient to compare two event's addresses to see if they are equal or not.

Although the run-time library supports a numerical value for *all* events, the current CHSM language does not support any syntax for assigning one to *enter/exit* events, hence, in the resultant C++ code, none is given in a declaration thus relying on the default argument. The value data-member was placed in the Event class because the language may support this in the future.

vector. Things are declared static since there is no need to make their names available outside of the resultant C++ file.

In the resultant C++ code, state-names also have to be unique because, despite their nesting in the source CHSM description, they are actually all declared at file-scope. To generate a unique name, the fully-qualified name of a state is encoded as an S followed by the name of each of a state's ancestor states preceded by its length. The fully-qualified state-name p.x.a is encoded as S1p1x1a, the state-name Root.Trunk.Branch.Leaf is encoded as S4Root5Trunk6Branch4Leaf. 25

Events generated from "the outside world" are termed user-events; correspondingly, there exists the UserEvent class derived from the Event class; its declaration is shown in figure 3.5. The only addition is the name of the event; a name is needed to support a command-line interactor (§3.4).

Figure 3.5: UserEvent class declaration

The resultant C++ code for user-events is shown in figure 3.6. It adds to that which is generated for ordinary events by just supplying the event-name and numerical value as arguments to the constructor in the user-event declaration.

The user_event vector of the CHSM class (§3.3.3) stores the address of all user-events; a *nil*-pointer marks the end of the vector. This too is needed to support a command-line interactor (§3.4). The condition-compilation preprocessor-directive on CHSM_INTERACT is used to compile the enclosed section only if the interactor is actually being used; if not, it has no use so it is not compiled.

3.3.1.3 The Transition Structure

Now that the data-structure for a transition has been shown (in figure 3.2), the Transition structure can be presented; its declaration is shown in figure 3.7. The condition data-member is a pointer to a function taking a reference to a constant Event and returning a Boolean; this is used for the event-condition; the action data-member is similarly declared and used for the transition-action. The argument is used to pass the event causing the current transition; this can be used to distinguish events when more than one event can cause a given transition. The taken data-member

Why use such an elaborate encoding scheme? Why not just generate names like S1, S2, etc.? Certainly that could have been done, but it wasn't for two reasons. The first is that if the user refers to a state's name in the resultant C++ code in either an event-condition or a transition-action and there is an error in the user's code, the error-message generated by the underlying C++ compiler will have a name in it like S23, which has no meaning to the user, whereas the real state-name can be deduced from the encoded version of S1p1x1a. The second reason is similar to the first in that if the user is using a debugger, state-names, as they are encoded now, can be deduced. This encoding scheme was inspired by the one used by the AT&T C++ compiler. Ellis [8], pp. 122-127.

is used to temporarily mark a transition as having been taken. The resultant C++ code for transitions is shown in figure 3.8. The transition vector of the CHSM class (§3.3.3) stores all of the transitions; it is initialized in the declaration.²⁶

```
static Transition *const Talpha[] = {
    &CHSM::transition[1],
    0
};
UserEvent alpha( Talpha, "alpha", 0 );
static Transition *const Tbeta[] = {
    &CHSM::transition[0],
};
UserEvent beta( Tbeta, "beta", 0 );
static Transition *const Tdelta[] = {
    &CHSM::transition[2],
};
UserEvent delta( Tdelta, "delta", 0 );
UserEvent const *const CHSM::user_event[] = {
#ifdef CHSM_INTERACT
    &alpha,
    &beta,
    &delta,
#endif
};
      Figure 3.6: Resultant C++ code for user-events
struct Transition {
    Boolean
                   (*condition)( Event const& );
    State
                   *from, *to;
    Event const
                   *broadcast;
    void
                   (*action)( Event const& );
    Boolean
                   taken;
};
       Figure 3.7: Transition structure declaration
    Transition CHSM::transition[] = {
         { 0, &Slp1x1a, &Slp1x1b, 0, 0 },
          { 0, &Slp1x1b, &Slp1x1a, 0, 0 },
          \{0, \&Slplylc, \&Slplyld, 0, 0\},
          { 0, &Slplyld, &Slplylc, 0, 0 },
    };
```

Figure 3.8: Resultant C++ code for transitions

Ideally, all the data-members except taken should have been declared const, but C++ does not (currently) allow structures with const data-members to be initialized using initializer-lists.

3.3.2 States, Clusters, and Sets

3.3.2.1 Design Considerations

States, clusters, and sets form a natural class hierarchy with the State class at the base and the Cluster and Set classes derived from it; however, a complication arises when it comes to storing a state's list of child-states. Plain-states do not have child-states, yet both clusters and sets do; ideally, plain-states should not store any data nor implement any code regarding child-states, but the data and (common) code should not be replicated in both the Cluster and Set classes either.

The solution is to introduce a Parent class into the hierarchy, derived from State, to be the common base-class for the Cluster and Set classes. It shall contain the data and provide the common interface for implementing the semantics of child-states, but the actual cluster- or set-specific code shall be deferred to the Cluster and Set classes, respectively. This relationship is shown in figure 3.9.

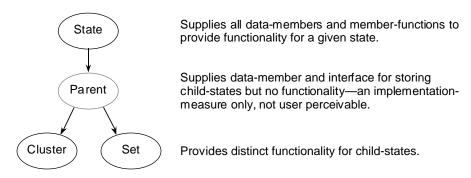


Figure 3.9: State classes hierarchy

3.3.2.2 The State Class

The declaration for the State class is shown in figure 3.10. Every state has an active status, a (fully-qualified) name, and a pointer to its parent state, if any. (For the const data members name and parent, there is no need to make them non-public and provide inspector functions.)²⁷

The Events sub-structure maintains pointers to the events to be broadcast upon entering and exiting a state, if any. Enter and exit events, in theory, are always broadcast; however, if there is no transition on such an event, then it is not actually necessary to broadcast it in practice since it would not do anything anyway. The Events constructor initializes the pointers from the arguments supplied to the main constructor.

The constructor takes the arguments specified by the STATE_ARGS macro to initialize the named data-members. The macro is for the benefit of classes derived from the State class by the user (but the library itself also makes use of them). Since the constructor has a complex argument-list, using

An *inspector* function is (usually) a very simple function, often a "one-liner," that returns the contents of a non-public data-member; alternatively, it could perform a trivial computation to return the value of an "alleged" data-member.

this macro, along with the STATE_INIT macro, a derived class's constructor only has to do what is shown in figure 3.11. This also shields the user in the event that the library evolves and the argument-list changes.

```
#define STATE_ARGS char const *s, Parent *p, Event const *e, Event const *x
#define STATE_INIT s, p, e, x
class State {
   Boolean
                    active;
    struct Events
                         Event const *const enter, *const exit;
                         Events ( Event const *e, Event const *x ) :
                             enter( e ), exit( x ) { }
                     } event;
public:
    String const
                    name;
                    *const parent;
    Parent
    State( STATE_ARGS ) : name( s ), parent( p ), event( e, x ) {
        active = false;
    virtual Boolean    Enter( Event const&, State *fromChild = 0 );
    virtual void DeepClear();
    Boolean
                    Active()
                                const { return active; }
                    Inactive() const { return !active; }
    Boolean
};
                     Figure 3.10: State class declaration
    class MyState : public State {
        // ...
    public:
        MyState( STATE ARGS ) : State( STATE INIT ) { /* ... */ }
        // ...
    };
```

Figure 3.11: Use of macros in a derived-class

The ${\tt Enter()}$ and ${\tt Exit()}$ member-functions were described in the Transition Algorithm section (§1.4.4); here in the implementation, however, an additional Event argument is passed, set to the event causing the state to be entered or exited, respectively. The event, although not used by the library itself, is provided so that it is available to the ${\tt Enter()}$ and ${\tt Exit()}$ member-functions of a derived-class if overridden by the user.

The $\mathtt{DeepClear}()$ member-function is discussed in the following section. The $\mathtt{Active}()$ and $\mathtt{Inactive}()$ member-functions are merely inspector functions that return the truth of the respective inquired status.

An example of the resultant C++ code is deferred to §3.3.2.6 where code for states, clusters, and sets is shown together.

3.3.2.3 The Parent Class

The declaration of the Parent class is shown in figure 3.12.

```
#define PARENT ARGS STATE ARGS, Children c
#define PARENT INIT STATE INIT, c
class Parent : public State {
    friend class State;
protected:
    typedef State *const *const *Children;
    Children const children;
                    Notify( State *byChild );
    virtual void
    Parent( PARENT_ARGS ) : State( STATE_INIT ), children( c ) { }
    class ChildIterator {
        Children pppc;
    public:
                 operator()( Children c ) { pppc = children; }
        State* operator()() { return *pppc ? **pppc++ : 0; }
        ChildIterator( Children c ) { operator()( c ); }
    };
public:
                     DeepClear();
    virtual void
};
```

Figure 3.12: Parent class declaration

The children data-member is a (constant) vector of (constant) pointers to (constant) pointers to all of the parent's child-states; it is initialized by the constructor. (The reason for the seemingly extra level of indirection is discussed in §3.3.2.6).

The DeepClear() member-function is used to clear the history of a given cluster and all clusters descended from it. The Parent::DeepClear() version just recursively calls DeepClear() for all of its child-states; this behavior is overridden by the Cluster class (following section). The reason that the State class also has a DeepClear() member-function, even though plain-states do not have child-states, is so that each child-state will not have to be "asked" if it is a plain-state or not; hence, DeepClear() can be called for every child-state regardless of what type of state it is—plain-states just do not do anything. The Notify() member-function was described in the Transition Algorithm section (§1.4.4).

The Parent class also declares a local ChildIterator class available to derived-classes. It provides a standard method for iterating over a state's child-states. If the implementation were ever to change, code using the iterator would be shielded since, while the method that the iterator uses to

iterate might change, its interface would not. Instances of the iterator are currently used in the library by the Set class and the DeepClear() member-function.

3.3.2.4 The Cluster Class

The Cluster class adds to the Parent class a flag to indicate whether it has a history and pointers to the currently- and last-active child-states. The class declaration is shown in figure 3.13.

```
#define CLUSTER_ARGS PARENT_ARGS, Boolean h = false
#define CLUSTER_INIT PARENT_INIT, h
class Cluster : public Parent {
    Boolean const history;
    State
                      *activeChild, *lastChild;
    virtual void     Notify( State* );
public:
    Cluster( CLUSTER_ARGS ) : Parent( PARENT_INIT ), history( h ) {
         activeChild = lastChild = 0;
    virtual Boolean    Enter( Event const&, State* = 0 );
    virtual Boolean    Exit ( Event const&, State* = 0, State* = 0 );
    virtual void DeepClear();
                      Clear() { lastChild = **children; }
    void
};
```

Figure 3.13: Cluster class declaration

The constructor optionally takes an additional argument—the boolean value to initialize the history data-member. The virtual member-functions <code>Enter()</code>, <code>Exit()</code>, and <code>Notify()</code> are overridden to implement cluster-semantics regarding child-states.

The Clear() member-function only clears the history for a single cluster by setting the lastChild data-member to point to the default (first) child-state. The DeepClear() member-function overrides that of the Parent class; it calls Clear() then calls Parent::DeepClear().

3.3.2.5 The Set Class

The Set class adds nothing to the Parent class; it only overrides the virtual member-functions Enter() and Exit() to implement set-semantics regarding child-states. The class declaration is shown in figure 3.14.

3.3.2.6 Resultant C++ Code

Here now, in figure 3.15, is the resultant C++ code for states, clusters, and sets. Every cluster's and set's declaration is preceded by the declaration of a vector of (constant) pointers to (constant) pointers to states listing the child-states that the given cluster or set has; a *nil*-pointer marks the end of the vector. Similar to the vector of transition-pointers declared for events, the name of this vector is also arbitrary so long as it is unique: it is a C ("C" for "child-states") followed by the encoded name of the state.

```
#define SET_ARGS    PARENT_ARGS
#define SET_INIT    PARENT_INIT

class Set : public Parent {
    public:
        Set( SET_ARGS ) : Parent( PARENT_INIT ) { }

        virtual Boolean        Enter( Event const&, State* = 0 );
        virtual Boolean        Exit ( Event const&, State* = 0 );
};
```

Figure 3.14: Set class declaration

```
static State *const *const CS1p[] = {
    &CHSM::state[1],
    &CHSM::state[4],
};
static Set S1p( "p", 0, 0, 0, CS1p );
static State *const *const CS1p1x[] = {
    &CHSM::state[2],
    &CHSM::state[3],
};
static Cluster Slp1x( "p.x", &S1p, 0, 0, CS1p1x );
static State Slp1xla( "p.x.a", &Slp1x, &ESlp1xla, 0 );
static State Slp1x1b( "p.x.b", &Slp1x, 0, 0 );
static State *const *const CS1p1y[] = {
    &CHSM::state[5],
    &CHSM::state[6],
};
static Cluster Slply( "p.y", &Slp, 0, 0, CSlply );
static State Slplylc( "p.y.c", &Slply, 0, 0 );
static State Slply1d( "p.y.d", &Slply, 0, 0 );
State *const CHSM::state[] = {
    &S1p,
    &S1p1x
    &Slplxla,
    &Slplxlb,
    &Slply
    &Slplylc,
    &Slplyld,
};
```

Figure 3.15: Resultant C++ code for states, clusters, and sets

The pointers to child-states point into the CHSM::state vector where the addresses of all of the states for the entire CHSM are stored. This vector and the vectors of child-states solves a "chicken-and-egg" problem of how to have a state have both the address of its parent-state, if any, available and the addresses of all of its child-states available, i.e., everything cannot be declared before it is referred to. This solution, however, does introduce an otherwise unnecessary level of pointer-indirection on the part of child-states.

Again, things are declared static because there is no need to make their names available outside of the resultant C++ file.

3.3.3 The CHSM Class

The CHSM class contains all the information "external" to the previously-described classes. Since there is only one CHSM, all the members are declared static. The class declaration is shown in figure 3.16.

```
class CHSM {
    static Event
                                  prime;
    static UserEvent const
                                  *const user_event[];
    friend class
                                  Interactor;
public:
    static State
                                  *const state[];
    static Transition
                                  transition[];
    enum {
         D_none = 0x00, // no debugging D_enex = 0x01, // show entrance
                                 // show entrances & exits
     };
    static u_int
                                  debuq;
    CHSM() { state[0]->Enter( prime ); }
};
```

Figure 3.16: CHSM class declaration

The vectors state, transition, and user_event store the lists of all the states, transitions, and user-events, respectively, for the entire CHSM. None are defined in the library; instead, they are defined in the resultant C++ code. The state vector was discussed in §3.3.2.6; the transition vector, §3.3.1.3. The user_event vector contains the address of all (just) the user-events; this is needed by the command-line interactor (§3.4); the end of the vector is marked by a *nil*-pointer.

The debug data-member is set to a non-zero value when debugging information is to be produced to standard-error. Currently, there is only one non-zero value defined, that of D_enex; when debug is set to that value, enter and exit messages are printed. An example of the output generated is shown in figure 3.17 where one state is being exited and another is being entered. The leading | characters are present to allow debugging output lines to be distinguished easily from other output that the CHSM may generate.

```
|exiting : p.x.a
|entering: p.x.b
```

Figure 3.17: Sample debugging output

The constructor initializes the CHSM by entering its first state; that state, in turn, shall enter all of its child-states using the appropriate child-state semantics. The prime data-member is a dummy-event needed just to pass something to the Enter() member-function for the initialization.

The resultant C++ code is shown in figure 3.18. The name used for the instance-variable of the CHSM is immaterial since it is not used anyway (since all the members are declared static); the only reason an instance is declared is to have the constructor called once to initialize the CHSM. The conditional-compilation preprocessor-directives are used to determine the initial status of debugging.

Figure 3.18: C++ code for CHSM class

3.4 The Interactor

The run-time library includes a simple, command-line interface, default *interactor* as means of interacting with a compiled CHSM.²⁸ All output produced by the interactor is preceded by a | just like the debugging output is and for the same reason.

Events are entered into the system by typing the event's name at the interactor's prompt. This is why the UserEvent class needs a name data-member and why the CHSM::user_event vector exists—when an event's name is entered, the user_event vector is searched looking for an event with a matching name. If a matching name is found, that event is broadcast; if not, the interactor issues an error-message stating that the specified event does not exist.

The interactor has only three commands: print, debug, and quit. The first letter of the command is used to issue it preceded by a / to distinguish it from an event-name. The print command prints all the states' names; if a given state is active, its name is preceded by a *; an example is shown in figure 3.19.

The debug command toggles debugging output overriding and option specified at compile-time. The quit command does what is expected.

The run-time library also provides a default main() function that just starts the interactor running. This main() can be overridden by the user by providing his or her own main() function thereby allowing the user to perform any other initialization before and any clean-up after the

The details of the Interactor class are not discussed because it's all fairly mundane and simple code most of which deals with getting and parsing user-input. Also, as mentioned in chapter 2, the user can write his or her own fancier interactor completely ignoring the default one. The fact that an Interactor class is made a friend of the necessary other classes in the run-time library allows this, i.e., any class named Interactor will work.

interactor runs. In that case, the user can call the static member-function Interactor::Go() in his or her own main(); it takes no arguments and returns no value.

|*p |*p.x |*p.x.a | p.x.b |*p.y |*p.y.c

Figure 3.19: Example of print-command output

Chapter 4

Compiler Design

4.1 Overview

From the user's point of view, there is only the CHSM compiler with the command-name chsm; in actuality, the CHSM compiler is composed of two files: a KornShell [12] driver-script that serves as the compiler's user-interface, named chsm, and the C++ object-code for the compiler itself, named chsm2c++.29

The reason for having a separate driver-script is to have the compiler, in terms of a user-interface and input/output, as simple as possible. The driver-script concerns itself with the local environment and operating-system, i.e., file-name extensions, the name of the C++ compiler, etc. Being a script, such dependencies are easily changed when the CHSM compiler as a whole is ported.

The script collects command-line options specified by the user, runs the compiler on the source-file, and (most often) takes the resultant C++ code and feeds it to the C++ compiler.

The compiler has to concern itself only with performing the actual compiling. A schematic diagram of the compiler's actions is shown in figure 4.1. A source-file, presumably containing a CHSM description, is read, lexically-analyzed, parsed, semantically-checked, and has the user-code given in transition-actions and condition-expressions spooled off into a temporary-file; then the resultant C++ code is emitted joined by the user-code to standard-output. The reason that the user-code is spooled into a file, as opposed to just be copied straight through, is that it is defined to be in the scope of all state- and event-names; hence, it is emitted only after the code for states and events has been emitted.

The term "script" shall henceforth refer to the user-interface driver script and the term "compiler" shall henceforth refer to the actual compiler.

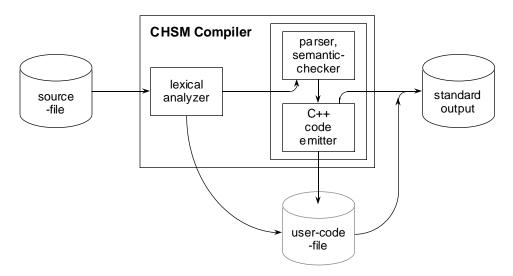


Figure 4.1: CHSM compiler schematic

As with any non-trivial program, the source-code for the compiler is in several modules; these are all described in the following sections.

4.2 The Compiler Class

For a compiler, there would tend to be a lot of global variables: which command-line options were given, the name of the source-file, the line-number being compiled, the number of errors and warnings issued, etc. Rather than having many unrelated variables polluting global name-space, they were all placed inside a Compiler structure as static data-members. The declaration of the structure is shown in figure 4.2.

The me data-member is the name of the compiler as passed to it by the host operating-system;³⁰ ident is the name used in the "banner" line emitted as the first line of the resultant C++ code, e.g., AT&T CHSM Compiler; version is obvious; the explanation of newlined is deferred to §4.4.3.

The source, target, options, and userCode data-members are sub-structures dealing with the respective things; the purpose of the data-members of those sub-structures should be obvious.

The Source structure's destructor closes the source-file and prints the total number of errors and warnings produced during the compilation. The UserCode structure's constructor names the temporary spool-file for user-code with a unique name containing the compiler's process-id; its destructor deletes it.

The Initialize() member-function processes command-line options and arguments; if there is an error in these, it issues an error message by calling Usage(). It also sets the compiler to catch operating-system signals and opens the source and user-code files.

It is assumed that the host operating-system behaves similarly to UNIX where a program's name is passed via <code>argv[0]</code>.

The IdentStamp() member-function emits the "banner" line containing the ident string and the version of the compiler. The Error(), Fatal(), and Warning() member-functions provide a means of consistently reporting errors and warnings, i.e., all having the same "look."

```
struct Compiler {
    static char const
                           *me;
                                     // program name
                           *ident; // what _we_ like to call ourselves
    static char const
    static char const
                           *version;
    static Boolean
                           newlined;
    struct Source {
                                     // source-file info.
         static char const *name;
         static ifstream file;
         static u_int
                           line_no;
         static u_int
                           errors, warnings;
         static ostream&
                           Error ( u_int alt_no = 0 );
         static ostream& Warning( u_int alt_no = 0 );
         ~Source();
    }
                           source;
    struct Target {
                                     // target-file info.
         static u_int
                           line_no;
                           target;
    struct Options {
                                     // compiler options
         static Boolean
                           no_line;
                                         // -d
                                         // -h
         static Boolean
                           gen_dot_h;
                           stack_debug; // -S
         static Boolean
                                         // -Y
         //
                           yydebug;
    }
                           options;
                                     // user-code file info.
    struct UserCode {
         static char
                           name[];
         static fstream
                           file;
         UserCode();
                                     // names temporary file
                           { ::unlink( name ); }
         ~UserCode()
    }
                           userCode;
    struct Header {
                                     // header-file info.
         static char const *name;
         static ofstream
                           file;
                           header;
                       Initialize( int argc, char const *argv[] );
    static void
    static void
                       Usage();
    static ostream&
                       IdentStamp( ostream& );
    static ostream&
                       Error() { return cerr << me << ": error: "; }</pre>
                       Fatal() { return cerr << me << ": fatal error: "; }</pre>
    static ostream&
    static ostream& Warning() { return cerr << me << ": warning: "; }</pre>
private:
    static void
                       CatchSignal( int sig_id );
};
```

Figure 4.2: Compiler structure declaration

The Options sub-structure stores the status of the command-line options: the -d option suppresses #line preprocessor directives in the resultant C++ code; -h generates a separate header-file chsm_incl.h containing C++ extern declarations for user-events so that other .c files, if any, can have access to them (an example is shown in figure 4.3); -s turns on parsing-stack debugging; -y turns on yacc debugging (§4.4.2). The latter two options are of use only to the compiler's implementor.

```
#ifndef _CHSM_INCL_
#define _CHSM_INCL_
#include <chsm.h>
extern UserEvent alpha;
extern UserEvent beta;
// ...
#endif
```

Figure 4.3: Example header-file for user-events

The problem with having all the members declared static is that referring to them becomes tedious by having to use full qualification all of the time, e.g., Compiler::me, Compiler::Error(), etc. Instead, a single instance variable g ("g" for "global") is defined; now, members can be referred through g as g.me, g.Error(), etc.

4.3 Lexical Analysis

4.3.1 Overview

Lexical analysis is done by using the code produced by a lexical-analyzer generator such as lex or flex [13]. In either case, the input/output routines were altered so that the C++ iostream library is used rather than the C stdio library. For reference, table 4.1 lists all of the tokens returned by the lexical-analyzer (in no particular order).

Extensive use is made of the "start-condition" feature of *lex* to have several "mini-scanners" for processing the declarations-section, the description-section, comments, strings, C++ code and expressions and embedded state-names therein, and the user-code section. The states that the lexical-analyzer can be in may be represented by the statechart shown in figure 4.4.

The statechart shown is mere graphical convenience—a CHSM was not created and is therefore not the controlling mechanism for the lexical-analyzer. Since the only thing that makes figure 4.4 a statechart, as opposed to an ordinary STD, is the lone cluster with a history, it was not deemed

The *lex* source-file is such that it can be compiled using either *lex* or *flex*. The *flex* generator defines a symbol FLEX_SCANNER that is tested for using condition-compilation directives. Also, henceforth, the term *lex* will be used to mean either *lex* or *flex*.

worthwhile to use an actual CHSM in this case. Additionally, extra work would have been involved either to *bootstrap* the CHSM compiler to use it to compile a CHSM for its own lexical-analyzer or to hand-code the CHSM directly.³² Instead, a simple vector was used to implement a stack of start-conditions to keep track of the previous ones; fortunately, *lex* maintains its current start-condition in a global variable.³³

Table 4.1: Tokens returned by the lexical analyzer

cluster	T_CLUSTER	%%	T_PERCENT)	T_RPAREN
deep	T_DEEP	->	T_ARC	{	T_LBRACE
enter	T_ENTER	/	T_SLASH	}	T_RBRACE
event	T_EVENT	,	T_COMMA	<	T_LANGLE
exit	T_EXIT	=	T_EQUAL	>	T_RANGLE
history	T_HISTORY	;	T_SEMI	% {	T_LCODE
in	T_IN	\$	T_DOLLAR	% }	T_RCODE
is	T_IS	::	T_COLONS	identifier	T_IDENT
set	T_SET	sequence of .	T_DOTS	. identifier	T_SUBSTATES
state	T_STATE	(T_LPAREN	number	T_NUMBER

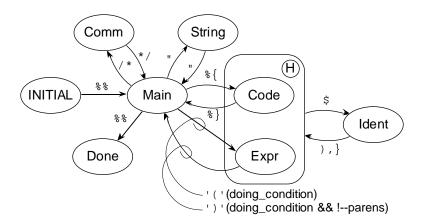


Figure 4.4: Lexical-analyzer statechart

4.3.2 State Overview

The lexical-analyzer starts out in the INITIAL state processing the declarations section. Text is copied verbatim from the source-file to standard-output until the %% token is encountered in column one; when that happens, the lexical-analyzer makes a transition to the Main state.

The Main state handles and returns tokens for everything except that which is specially handled by the other states in figure 4.4 until either another %% token is encountered in column one or the end of the source-file is reached; when that happens, the lexical-analyzer makes a transition to the Done state.

³² Aho et al. [14], p. 725.

³³ In lex, it's yybgin (no "e"), in flex, it's yy_start.

The Done state functions similarly to the INITIAL state, but for the user-code section. Text is copied verbatim from the source-file to standard-output until the end-of-file is reached; when that happens, lexical-analysis is complete.

The Comm state is for processing /* ... */ comments; these, as well as // comments are stripped from the resultant C++ code. The String state is for processing strings; there is nothing special about it.

4.3.3 Action-Code and Embedded Constructs

The Code state is used to copy any C++ code given for transition-actions; it is entered when the f token is encountered and exited when f is. The C++ code is, for the most part, copied verbatim from the source-file to the user-code file. When a f token is encountered, however, the lexical-analyzer makes a transition to the Ident state so that it will return the tokens used for the special constructs of f in (), f exit(), and f is encountered.

4.3.4 Condition-Expressions

The Expr state is used to copy any C++ code given for condition-expressions; it is entered when a (is encountered only if the global variable $doing_condition$ is true and exited when a) is encountered. The $doing_condition$ variable is set to true by the parser when a condition-expression is about to be parsed and cleared afterwards. The reason the variable is needed is that, unlike the % and % token pairs, parentheses have other uses in the CHSM language, namely child-state declarations. The processing described here should only take place for condition-expressions.

The C++ code is, for the most part, copied verbatim from the source-file to the user-code file. Embedded constructs are handled just as in action-code processing. Condition-expression processing additionally requires that parentheses be counted (balanced) so that the \mathtt{Expr} state is exited after having encountered the correct).

4.3.5 Identifiers and Keywords

Lexically speaking, identifiers and keywords can be the same. One approach for distinguishing keywords is to have a separate regular-expression for each keyword; however, this makes lexical-analysis slower and the tables produced by *lex* bigger. A better approach is commonly known, that being to have *lex* only scan for identifiers and, before a token is returned, check to see whether the identifier is instead a keyword by doing a (quick) lookup in a keyword-table, usually using a binary-search; if it is a keyword, return the keyword token, otherwise, just return an identifier token.

4.4 The Yacc Grammar

4.4.1 Overview

The parser for the CHSM compiler was created using *yacc*. (The file wherein the grammar for the CHSM description language is specified shall henceforth be referred to as the *grammar file*.) In addition to the grammar, the grammar file contains the code that orchestrates the emission of C++ code and performs semantic-checks.

4.4.2 The Parsing Stack

Productions in *yacc* ordinarily have semantic-values associated with them; for any non-trivial grammar, productions can have any one of several semantic-value types. When that is the case, it is necessary to use the *yacc* %union construct to allow this. For example, assume that one of the semantic-value types for a production is an integer. Code that might be typically seen is shown in figure 4.5a.

Figure 4.5: Yacc grammar production forms

However, this traditional method for building-up semantic-values for productions was not used for two reasons. The first is that using *yacc's* positional semantic-values becomes tricky when referring to those on the stack before the ones of the current production; also, *yacc* does not allow positions to be computed. During development, the grammar often changes; this can affect the position of a given semantic-value on the stack necessitating one to edit all the occurrences of a semantic-value's position; additionally, always having to know the exact position of a semantic-value, as opposed to being able to compute it, complicates the grammar and its development. The second reason is that *yacc's* standard debugging output is unwieldy and verbose (although it can still be used to some extent).

Instead, a separate stack of semantic-values is maintained replacing *yacc*'s. This stack addresses both complaints about *yacc*. First, it is a *true* stack, i.e., elements are explicitly pushed and popped; therefore, accessing semantic-values is done *relative* to the top of the stack (as opposed to their absolute position) and allows the depth of a semantic-value in the stack to be computed. Second, its debugging output is much less verbose. The semantic-stack contains elements of the Semantic class whose declaration is shown in figure 4.6.

Elements store either an integer or a pointer to "something"; each instance is also tagged to discriminate which. The constructors take an argument of either type and set the tag accordingly;

the conversion-operators convert an instance back either to an integer or pointer. The reason for the tag is to check when an instance is being converted back to a type, that it really is of that type.

```
class Semantic {
    union { int i; void *p; }
    enum Tag { int_t, ptr_t } tag;
public:
    Semantic( int x ) { i = x, tag = int_t; }
    Semantic( void *x = 0 ) { p = x, tag = ptr_t; }

    operator int() const { check( int_t, (void*)i ); return i; }
    operator void*() const { check( ptr_t, p ); return p; }
private:
    void check( Tag, void* ) const;
};
```

Figure 4.6: Semantic class declaration

For use in the production-actions, there exists a set of preprocessor macros for pushing and popping elements; a sampling of these is shown in figure 4.7.

Figure 4.7: Some semantic-value stack macros

Every other macro is defined in terms of either of the first two: PUSH() and POP(); these are just front-ends to the functions PushLine() and PopLine() that are overloaded to take either an integer or void-pointer argument. In addition to taking a value to be pushed or a reference to a variable to have a value popped into, respectively, they also take the line-number in the grammar file on which the pushing or popping is being done; the PUSH() and POP() macros supply the __LINE__ preprocessor variable to pass the current line-number. If stack-debugging has been turned on via the -S option, then a trace through the grammar by line-number is emitted to standard-output. This allows one to follow yacc easily through the grammar as it parses input. In addition, when a symbol-name is pushed or popped, its name and scope are also emitted. The debugging lines are preceded by the C++ // token so that, even though the debugging information is begin emitted to standard-output right along with the resultant C++ code, the output will be ignored by the C++ compiler. A sample of the debugging output is shown in figure 4.8.

The other popping macros in figure 4.7 are used as conveniences that simultaneously declare the variable to receive the popped semantic-value and also to pop it. The local scopes and variables are used to prevent temporary-generation on the part of the C++ compiler. In addition, there are also

other macros (not shown) that allow semantic-values at arbitrary depths in the stack to be "peeked" at.

```
// **** line: 784: Push: 0x87660
                                      // **** line: 462: Peek: 0x87888
                                      // **** name: S1p1x <-2>
// ***** name: x <1>
                                      // ***** line: 253: Push: 0x0
// ***** line: 365: Pop : 0x87660
// ***** name: x <1>
                                      // **** line: 254: Push: 0x0
// **** line:
              408: Push: 0x87888
                                      // **** line: 225: Pop : 0x0
                                      // ***** line: 226: Pop : 0x0
// **** name: S1p1x <-2>
// **** line: 784: Push: 0x87928
// **** name: a <-2>
                                      // ***** line: 227: Pop : 0x87888
// **** name: a <-2>
                                      // **** name: S1p1x <-2>
// ***** line: 461: Pop : 0x87928
                                      // ***** line: 228: Pop : 0x0
// **** name: a <-2>
                                      // **** name: (NIL)
// ***** line: 462: Peek: 0x87888
                                      // **** line: 244: Push: 0x87888
// **** name: S1p1x <-2>
                                      // **** name: S1p1x <-1>
// **** line: 784: Push: 0x879e8
                                      // **** line: 792: Pop : 0x87888
// **** name: b <-2>
                                      // **** name: S1p1x <-1>
// **** line: 461: Pop : 0x879e8
                                      // **** line: 793: Push: 0x875c0
// **** name: b <-2>
```

Figure 4.8: Sample debugging output

Now that all of this has been explained, figure 4.5b can be mentioned that shows how the traditional production-action form of figure 4.5a would be written using the scheme described.

The check() member-function of the Semantic class is used to check that the value being popped is of the type requested by a given popping macro; if not, the compiler terminates with a fatal error. In a working compiler, of course, this should never happen.

In a production-version of the compiler, the code to implement the semantic-value type-checking can be compiled-out via conditional-compilation making the -S option much like the -Y option in that the compiler must be specially compiled for it to be used.

4.4.3 Improving Error-Messages

Ordinarily, one using yacc writes an error-message printing routine called yyerror() that is called by yacc when a syntax-error is detected in the source-code. From yyerror(), however, the compiler-writer has no idea as to which production the error occurred in; the only argument passed to yyerror() is the text of the error-message to be printed, which is almost always "syntax error." A compiler that just prints this is not very helpful to the compiler-user.

To improve error-messages, an additional error-message printing routine called <code>myerror()</code> was written. Figure 4.9a shows a code fragment containing a syntax-error where the boxed comma is the character about to be parsed. Figure 4.9b shows a portion of the actual *yacc* grammar used to parse child-state declarations; it says: where a child-state's name is expected, an identifier must be present, otherwise it is an error.

With a comma being present, it is indeed an error, so *yacc* matches the error token and calls yyerror(); it calls Compiler::Source::Error() to print the file-name, line-number, and general

statement of an error, then prints the message supplied by *yacc* that is, as mentioned, almost always "syntax error." Finally, in the production-action, <code>myerror()</code> is called to print additional information about the error, in this case, what is expected and what a possible cause of the error might be; this is shown in figure 4.9c.

Figure 4.9: Example use of myerror()

There is a complication due to a feature of yacc. Sometimes during parsing, one syntax-error starts a "cascade" of error-messages because the parser is out of "sync" with the source-text; this stops when it encounters a synchronizing token. To reduce the number of error-messages printed, yacc only calls yyerror() once then ignores source-text until a such a token is encountered. The complication is that, in the error production-actions, myerror() is always called; myerror() should print its message only if yyerror() was just called.

A simple solution is just to maintain a flag that <code>yyerror()</code> and <code>myerror()</code> share. As mentioned, <code>yyerror()</code> calls <code>Compiler::Source::Error()</code> then prints its message; it does so <code>without</code> printing a new-line character and clears the flag <code>Compiler::newlined</code> (§4.2). When <code>myerror()</code> is called, it checks the flag: if it is clear, it means that <code>yyerror()</code> was just called so it should print its message <code>with</code> a new-line and set the flag; if it is set, it means that <code>myerror()</code> was called again without an intervening call to <code>yyerror()</code>, i.e., <code>yacc</code> is suppressing error-messages, so it should do nothing.

The newlined flag also serves to insure that new-line characters are printed properly, i.e., error or other messages always start at the beginning of a line and that two new-line characters are not printed in succession.

4.5 Symbol-Table Management and Semantic-Checking

4.5.1 Overview

The symbol-table is one of the most important data-structures used in any compiler for it is where all identifiers along with their type information are stored. The CHSM compiler doesn't do any symbol-table management itself; instead, it just uses an instance of the SymbolTable class named, appropriately enough, symbolTable.³⁴ Since the SymbolTable class is not part of the compiler's source-code proper, it is not discussed here (it is instead discussed in §B.7); information stored *with* the various kinds of symbols, however, is part of the compiler's source-code and so is discussed here.

The symbol-table is used to perform some of the "obvious" semantic-checks such as multiply-declared events or multiply-defined states, type-checking a symbol for an identifier against the type expected in a given context. Other semantic-checks are mentioned in the following sections.

4.5.2 The BaseInfo Class

The BaseInfo class, as its name suggests, serves as the base-class for all other classes of symbol information. Its declaration is shown in figure 4.10. (The Synfo base-class is defined as part of the SymbolTable library.)

```
struct BaseInfo : Synfo {
    Boolean
                 used;
    u_int const first_ref;
    enum {
                      = 0 \times 00000,
        unknown_t
        undeclared_t
                         = 0 \times 0001,
        global_t
                          = 0 \times 0002
                          = 0x0004,
        child_t
                          = 0x0008,
        state_t
        cluster_t
                          = 0x0010,
                          = 0x0020,
        set_t
        derived_t
                          = 0x0040,
        event_t
                          = 0x0080,
        userEvent_t
                          = 0x0100,
        macroEvent_t
                          = 0x0200,
    };
    virtual Type What() const { return unknown_t; }
    static String
                     TypeString( u_int );
    BaseInfo( int s = SymbolTable::Scope() ) :
        Synfo( s ), first_ref( g.source.line_no ) { used = false; }
};
```

Figure 4.10: BaseInfo class declaration

[&]quot;Management" refers to the creation, storage, retrieval, and destruction of symbols. The CHSM compiler does none of these things; instead, it just calls member-functions of the SymbolTable class.

The used data-member is used to keep track of whether that has been defined in a CHSM description has actually been used (this is done with explicit event and event-macro declarations); if not, the compiler issues a warning at the end of the compilation. This helps prevent user "typos" from slipping by as in:

```
event widget = 42;
// ...
state a { wiget->b; }
```

Of course, the CHSM is still valid because wiget is taken to be a distinct (undeclared) event; it's just that if event widget occurs, the above transition will not take place. This still is not fool-proof because the warning will not be generated if widget is spelled correctly at least once elsewhere.³⁵

The first_ref data-member is used to store the first line-number that a symbol was referenced on. This too use used in error- and warning-messages as in:

```
"foo.m", line 666: warning: event "hell_freeze_over" not used
```

The enum declaration serves to define unique type-values for information classes derived from the BaseInfo class. The reason for the specific bit-values is so that types can be "ored" together when performing type-checking and more than one type is allowed in a given context, e.g., event-macros are usable in some places that events are. The What() virtual member-function returns the appropriate type for a given class object. The TypeString() member-function takes a set of "ored" types and returns a textual representation to be printed in error- and warning-messages.

The constructor takes the scope of the symbol information to be used, defaulting to the current scope, and initializes first_ref and used.

4.5.3 The GlobalInfo Class

As it happens, most things defined in a CHSM description are global in nature, i.e., their names have to be known everywhere. For example, event- and (fully-qualified) state-names must be global because any state can make a transition to (almost) any other and on any event regardless of its nesting level. The only things not global are child-state names (discussed in the next section). The GlobalInfo class declaration is shown in figure 4.11.

The constructor just passes the pre-defined constant Synfo::Global to BaseInfo to indicate that this symbol information, or anything derived from it, is to have global scope.

A concordance listing might help this; perhaps a future version of the compiler will include the option of producing one.

This is one simple form of what is known as "run-time type identification." There are many approaches to this since it is not supported by C++, although it may be in the future.

```
class GlobalInfo : public BaseInfo {
public:
    GlobalInfo() : BaseInfo( Synfo::Global ) { }
    virtual Type What() const { return global_t; }
};
```

Figure 4.11: GlobalInfo class declaration

The GlobalInfo class is also used to assign "some" type to a forward-referenced state-name as in:

```
// ...
    state a { alpha->y; } // forward-reference
    // ...
state y { /* ... */ }
```

Here, a symbol for state y is created, but it is not known whether y is a plain-state, a cluster, or a set until its description is encountered; hence, in the meantime, the type of the symbol is just set to global_t. When the description is encountered, its symbol's type is changed to the actual type; if the description is *never* encountered, the compiler checks to see if there are any symbols of the type global_t left and, if so, it issues error-messages like:

```
"foo.m", line 7: error: state "bliss" not defined
```

4.5.4 The ChildInfo Class

The ChildInfo class is used by symbols for child-state identifiers. As previously mentioned, these are the only types of symbols that are not global, i.e., they are destroyed by the SymbolTable class when a cluster's or set's scope is exited. The class declaration is shown in figure 4.12.

```
class ChildInfo : public BaseInfo {
public:
    Symbol const *const parent;

ChildInfo( Symbol const *p = 0 ) :
    BaseInfo( SymbolTable::Scope() + 1 ), parent( p ) { }

virtual Type What() const { return child_t; }
};
```

Figure 4.12: ChildInfo class declaration

Child-states, of course, have a parent-state; hence, the constructor takes a pointer to the symbol that represents its parent-state and initializes the parent data-member. When a child-state is declared, the parent data-member is checked to see if it already has a parent-state equal to the

current parent-state;³⁷ if it does, it means that the current child-state has been previously declared and the compiler issues an error-message like:

```
"foo.m", line 2: error: child "deja_vu" previously declared.
```

The constructor also passes the current scope plus one to the BaseInfo constructor because, as mentioned, child-state symbols are not global. Why "plus one?" When the scope of a cluster or set is exited, the child-states' symbols should be destroyed. A cluster's or set's scope is exited upon encountering the closing } describing its body, which is in the next *higher* scope (see figure 4.13); hence, to get the child-states' symbols destroyed, it is necessary to "push" them into the next scope, hence the "plus one."

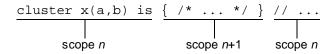


Figure 4.13: Child-state declarations and scope

4.5.5 The StateInfo, ParentInfo, ClusterInfo, and SetInfo Classes

The class-derivation hierarchy for the StateInfo, ParentInfo, ClusterInfo, and SetInfo classes parallel that of the State, Parent, Cluster, and Set classes in the CHSM run-time library for the same reasons. The StateInfo class, in addition to storing the relevant information regarding states, serves as the base-class for the other classes mentioned. The class declaration is shown in figure 4.14.

These classes inherit a virtual member-function <code>Emit()</code> from the <code>Synfo</code> class; its purpose is to emit the object-code necessary for a given symbol. Since the code emitted for states, clusters, and sets is similar, the common code has been "factored out" and placed into the <code>CommonEmit()</code> member-function. An additional <code>type</code> argument has passed in it the type of the state, i.e., either the string <code>state</code>, <code>cluster</code>, or <code>set</code>.

States, of course, have a parent-state, hence the parent data-member. It is used when a state is described to see if was declared to be a child-state of its parent-state. The compiler maintains a global variable parent that stores a pointer to the symbol for the current parent-state. If the two parent-state pointers do not match, then the compiler issues an error-message like:

"foo.m", line 7: error: state "bam_bam" is not a child of "wilma"

³⁷ It is legal to have a parent-state not equal to the current parent-state; in that case, it means that the user has named a child-state with the same name as some other child-state of some ancestor state. The current declaration, of course, hides the previous one.

```
class StateInfo : public GlobalInfo {
    virtual ostream& Emit( ostream&, char const *name ) const;
protected:
    ostream& CommonEmit(
                  ostream&, char const *type, char const *name ) const;
public:
                       *const parent, *const derived;
    Symbol const
    struct Events
                           Boolean enter, exit;
                           Events() { enter = exit = false; }
                       } event;
    StateInfo( Symbol const *p = 0, Symbol const *d = 0 ) :
         parent( p ), derived( d ) { }
    virtual Type
                      What() const { return state_t; }
};
```

Figure 4.14: StateInfo class declaration

The derived data-member is used to store a pointer to the symbol of the derived-type if a given state is to be derived from a class in the resultant C++ code. No check can be (easily) made to see whether the named type has actually been declared to be a derived-class; instead, this check is left to the underlying C++ compiler.³⁸

The event data-member contains two flags to indicate whether a state actually needs to broadcast an *enter/exit* event when it is entered or exited, respectively. If it does, then the C++ code taking the address of the *enter/exit* event will be emitted; otherwise a 0 (representing the *nil*-pointer) will be (refer to figure 3.15). These flags, if set, are set at the end of the compilation after seeing if either of the *enter/exit* event was actually used in any transition (this could be considered "back-patching"); that is why the data-members are not declared const.

The ParentInfo class adds to StateInfo a Vector of unsigned integers, starting at zero, where each is the sequence-number of a state, i.e., the first state described is zero, the next is one, etc.³⁹ It also supplies its own CommonEmit() that also factors out common emission code for clusters and sets. The class declaration is shown in figure 4.15.

The ClusterInfo class only adds to ParentInfo a flag to indicate whether a given cluster has a history. The optional deep-history property of clusters is implemented by maintaining a pointer to the outermost cluster that has the deep-history option specified for it named outerDeep. All lexically-enclosed clusters then have the history flag set regardless of whether any history option is

To have the CHSM compiler check for this, the declarations section would have to be parsed looking for class declarations derived from either the State, Parent, Cluster, or Set classes, i.e., the parser would also have to be able to parse all valid C++ declarations. This was deemed to be redundant (not to mention too much work) and it would have doubled the compile-time for the declarations section, so letting the underlying C++ compiler catch the error seems like a reasonable alternative.

The parameterized-type Vector class is discussed in §B.5; for now, it is sufficient to know that a Vector emulates all the behavior of a built-in C++ vector.

specified for them or not. Once the scope of that outermost cluster is exited, outerDeep is reset to *nil*. The class declaration is shown in figure 4.16.

```
class ParentInfo : public StateInfo {
protected:
    ostream& CommonEmit(
                  ostream&, char const *type, char const *name ) const;
public:
    Vector<u_int> children;
    ParentInfo( Symbol const *p = 0, Symbol const *d = 0):
         StateInfo( p, d ) { }
};
                      Figure 4.15: ParentInfo class declaration
class ClusterInfo : public ParentInfo {
    virtual ostream& Emit( ostream&, char const *name ) const;
public:
    Boolean const
                       history;
    ClusterInfo(
         Symbol const *p = 0, Symbol const *d = 0, Boolean h = false
     ) :
         ParentInfo( p, d ), history( h ) { }
    virtual Type
                       What() const { return cluster_t; }
};
```

Figure 4.16: ClusterInfo class declaration

The SetInfo class adds nothing to the ParentInfo class; it just overrides the Emit() virtual member-function. Its class declaration is shown in figure 4.17.

```
class SetInfo : public ParentInfo {
    virtual ostream& Emit( ostream&, char const *name ) const;
public:
    SetInfo( Symbol const *p = 0, Symbol const *d = 0 ) :
        ParentInfo( p, d ) { }

    virtual Type    What() const { return set_t; }
};
```

Figure 4.17: SetInfo class declaration

4.5.6 The DerivedInfo Class

The DerivedInfo class does not do much (it doesn't need to); it only overrides the What() virtual member-function to return its type. Its class declaration is shown in figure 4.18.

4.5.7 The TransInfo Class

The TransInfo class declaration is shown in figure 4.19.

```
class DerivedInfo : public GlobalInfo {
   public:
        virtual Type What() const { return derived_t; }
};

        Figure 4.18: DerivedInfo class declaration

class TransInfo : public GlobalInfo {
```

```
Symbol const *const from, *const to, *const broadcast;
u_short const condition_id, action_id;

virtual ostream& Emit( ostream&, char const *name ) const;
public:
    TransInfo(
        u_short c_id, Symbol const *f, Symbol const *t,
        Symbol const *b, u_short a_id
    ):
        condition_id( c_id ), from( f ), to( t ),
        broadcast( b ), action_id( a_id ) { }
};
```

Figure 4.19: TransInfo class declaration

The C++ code for event-conditions and transition-actions is placed inside functions, each with a unique name; the function-names for event-conditions start with CHSM_C and those for transition-actions start with CHSM_A and they both end with a unique integer. Those integers are stored in the condition_id and action_id data-members, respectively.

Since transitions do not have names, symbols for transitions do not need to be (and aren't) stored in the symbol-table; instead, a separate queue of their symbol-pointers is maintained. 40

4.5.8 The EventInfo Class

The EventInfo class declaration is shown in figure 4.20. (The CommonEmit() member-function should be familiar by now.) The transition_id data-member is used to store the indices of the transitions that this event has. The indices are relative to the CHSM::transition vector (refer to figure 3.6).

The constructor takes a pointer to a Symbol-pointer member, either the enter or exit data-member, or neither. In the former case, this event represents and *enter/exit* event, so the pointer pointed-to is set to point to the symbol representing the state being either entered or exited. At the end of the compilation, EventInfo instances that have a non-nil enter or exit data-member are used to "back-patch" the state symbols pointed to in order to indicate that their *enter/exit* event should really be broadcast and therefore have the necessary C++ code emitted for doing so.

If transitions are not stored in the symbol-table, why are they stored using symbols? Because the presence of the <code>Emit()</code> virtual member-function made emission of transitions convenient, i.e., the mechanism was already there.

```
class EventInfo : public GlobalInfo {
   virtual ostream& Emit( ostream&, char const *name ) const;
protected:
   ostream& CommonEmit(
                ostream&, char const *storage, char const *type,
                char const *name ) const;
public:
    Symbol const
                    *enter, *exit;
                    Vector<u_int>
    EventInfo( Symbol const *EventInfo::*e = 0, Symbol const *s = 0 ) {
        enter = exit = 0;
        if (e) this->*e = s;
    virtual Type
                    What() const { return event_t; }
};
```

Figure 4.20: EventInfo class declaration

4.5.9 The UserEventInfo Class

The ${\tt UserEventInfo}$ class adds the numerical value that can be defined for user-events and overrides the ${\tt Emit}()$ virtual member-function. The class declaration is shown in figure 4.21.

```
class UserEventInfo : public EventInfo {
    virtual ostream& Emit( ostream&, char const *name ) const;
public:
    int value;

    UserEventInfo( int n = 0 ) : value( n ) { }

    virtual Type     What() const { return userEvent_t; }
};
```

Figure 4.21: UserEventInfo class declaration

4.5.10 The MacroEventInfo Class

The MacroEventInfo class maintains a vector of pointers to all the symbols for those events that comprise a given macro-event. The class declaration is shown in figure 4.22.

Figure 4.22: MacroEventInfo class declaration

After a new transition is added to the transition-queue, the triggering event is checked to see whether it is a macro-event; if it is, then that transition's id. (that being the current size of the transition-queue minus one) is added to all the comprised events' vector of transition-id's, otherwise it is just added to the single event's vector of transition-id's.

Chapter 5

Conclusions

5.1 Fulfilling Expectations

It was the goal to develop an object-oriented language system for implementing CHSMs as a realization of statecharts to model reactive systems. With the development of a CHSM description language, a compiler, and an object-oriented, user-extensible, run-time library, the author believes that this goal has been reached.

5.2 Future Work

No system or project is every really "done"; this one is no exception. In the following sections, extensions and improvements for the CHSM language and its compiler are mentioned.

5.2.1 Language Extensions

In terms of the CHSM description language, there are additional features that could have language support; experience will indicate whether there is sufficient need for them to be actually implemented as some of the proposed extensions will exact time or space penalties.

5.2.1.1 Classes Derived from the Event Class

This seems straightforward enough since the syntax for doing it would be the same as that used for specifying derived-classes for the classes that can be derived from now as shown in figure 5.1.

```
class Major : public Event {
    // ...
}
%%
event<Major> marriage;
```

Figure 5.1: Classes derived from the Event class

The potential benefit would be to allow the user to augment the behavior of events when they are broadcast with arbitrary C++ code just as the behavior of state entrances and exits can be augmented now.

5.2.1.2 Code Execution on State Entrances and Exits

This can be implemented in the run-time library by the addition of two pointers-to-function to the State class. There are three reasons why it was not implemented.

The first is that it is not known if the advantage of being able to do this directly would outweigh the cost of storing two additional pointers since *nil*-pointers would have to be stored for those states that do not make use of this feature.

The second reason is that this functionality, if really desired, can be achieved now via derived classes. In a derived class, the user would override the <code>Enter()</code> or <code>Exit()</code> member-functions to perform additional actions either before or after the state is entered or exited (or both). If the pointers-to-function were used, it would have to be decided, perhaps arbitrarily, whether the additional actions would be performed before or after. The user could be allowed to decide which, but he or she could not specify actions for both—permitting this would require an *additional* two pointers-to-function.

The third reason is that a "nice" syntax for doing this has not been thought of yet. One idea is something like that shown in figure 5.2; however, the author does not care too much for how transitions are specified with this extra syntax added.

Figure 5.2: Possible syntax for enter/exit code

5.2.1.3 Derived-Type Constructors with Additional Arguments

If derived classes are allowed in general, it would seem reasonable to allow derived class's constructors to take additional arguments. Figure 5.3 shows two possible versions of a syntax. The

second syntax shown would obviously require a change in the language for how derived classes in general are specified, i.e., for constructors that do not take additional arguments; the second syntax, however, does seem more natural since it parallels the way base-class constructors are called in C++.

Figure 5.3: Possible syntax for derived class's constructor arguments

5.2.1.4 Support for All Kinds of Transitions

The language currently does not have a syntax to describe a transition like the one on event γ first shown in figure 1.6 (repeated in figure 5.4a from §1.4.1.4 for convenience) and the one on event β first shown in figure 1.7 (repeated in figure 5.4b), i.e., to differentiate them from transitions like the ones on event β in both figures. In order to support this, there would have to be a syntax for annotating a transition with the scope that it reaches on its outermost point and the transition algorithm (§1.4.4) would have to be modified to handle this new scope information .

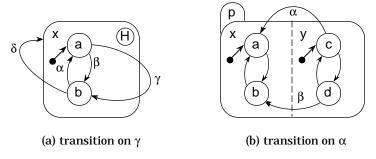


Figure 5.4: Unsupported transitions

5.2.1.5 Standard Components

Another thing that could be included with the CHSM run-time package is a set of standard derived-classes much like the Token class presented in chapter 2. Users will surely come up with many good ideas.

5.2.2 Compiler Improvements

5.2.2.1 Detecting Illegal Transitions

Illegal transitions between child-states of sets, like the one on event β in figure 5.4b, are currently not detected and reported as errors. For each transition, this would involve checking the first common parent-state of the source and target states to see if it is a set; if it is, then the transition is illegal.

5.2.2.2 Transition-Vector Code Optimization

The resultant C++ code emitted for events' transition vectors (§3.3.1.2) can be optimized in terms of storage: If two or more events have the same set of transitions, then they can share the same transition vector. While this storage optimization can be performed for any set of events in general, it ought to be performed first for macro-events. For each event of a macro-event, if it has no transitions in addition to those of the macro-event, then it can share a common transition vector. In the microwave oven example (§2.3), all the individual digit events $(_0, _1,$ etc.) comprising the macro-event digit have the same (repeated) transition vector in the resultant C++ code; obviously, all these can share the same vector. If a given digit event were to have additional transitions, only then must it have its own vector.

Appendix A

Language Reference Manual

A.1 Lexical Conventions

A.1.1 CHSM Description File Format

A CHSM description file is an ordinary text file with three sections: declarations, description, and user-code. The sections are separated by the %% token, which must be at the beginning of a line. Any section can be empty; if the user-code section is empty, then the trailing %% may also be omitted. If present, the declarations and user-code sections are passed through, untouched, to the underlying C++ compiler. The description file is otherwise free-format.

The declarations section is meant to contain any C++ declarations or definitions, or preprocessor directives needed to compile the resultant C++ code.

A.1.2 Comments

Both $/*\ \dots\ */$ and // comments may be used anywhere in the CHSM description file where whitespace is legal.

A.1.3 Identifiers

Identifiers follow the same rules for what constitutes a valid identifier in C++.

A.1.4 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

cluster	deep	enter	event	exit
history	in	is	set	state

⁴¹ Although it is not very useful to have the description-section empty.

In addition, identifiers containing a double underscore (__) are reserved for use by C++ and should be avoided.

The following characters and character-combinations are used as punctuation or operators:

A.1.5 Integer Constants

Integer constants can be expressed in all forms accepted by C++, i.e., decimal, octal, and hexadecimal except that the long and unsigned suffixes (1, μ , ν) are not supported.

A.2 Basic Concepts

A.2.1 Scopes

There are three kinds of scope: *global*, *local*, and *C++-code-block*.

- Global scope is that which is lexically at the outer-most level, i.e., the level at which the first state is declared.⁴³
- *Local* scope exists only within clusters and sets and only for child-state identifiers.⁴⁴ One identifier may be hidden by another due to a child-state declaration with an equal name in a lexically-enclosed scope. Such an identifier can still be referenced by using one of the scoperesolution operators (§A.4).
- C++-code-block scope exists only within () delimited event-conditions and % { % } delimited transition-actions. C++ variables declared within such blocks are known only therein and, with the exception of variables declared static, are destroyed upon exit from those blocks. 45 All state-names and all event-names not explicitly declared are in the scope of all blocks; 46 event-names explicitly declared must be declared before their first use.

A.2.2 CHSM Initialization

A CHSM is fully initialized before the start of execution of the C++ function main(). The first state declared at global scope in the description is the only one entered; from there, the individual

⁴² Since integer constants are used only to assign a numerical value to an event, this limitation is not deemed to be serious.

The term "state" will be used to refer to states, clusters, and sets indiscriminately; the terms "cluster" and "set" will be explicitly used only when it makes a difference.

Each new local scope is said to be *lexically-enclosed* or *lexically greater-than* those scopes that enclose it; *those* scopes are said to *lexically-enclose* or be *lexically less-than* those scopes that they enclose.

 $^{^{45}}$ That is, they are treated exactly like C++ functions (because that is how they were implemented).

This implies that, within a given C++-code-block, states that have not been described yet can be referenced anyway.

semantics for state, cluster, or set child-state entrances take over (§A.2.3.1). Thus global scope is treated semantically as though it were an unnamed cluster.

A.2.3 Types

A.2.3.1 Fundamental Types

There are four fundamental types: state, cluster, set, and event; each is an instance of either the State, Cluster, Set, or Event class, respectively.

A state is the simplest since it is an atomic entity; clusters and sets are usually composed of child-states. A cluster represents a *logical-exclusive-or* grouping of its child-states where exactly *one* child-state is active at any given time; a set represents a *logical-and* grouping where all of its child-states are active concurrently. All child-states are entered after and exited before their parent-state.

A cluster may have a history. If a cluster does not have a history, or it does have a history but has not previously been active, then the child-state entered after it itself is entered is the *default* child-state; if a cluster does have a history, then the child-state entered after it itself is entered is the one that was last active. A cluster may alternatively have a deep history. Such a cluster behaves exactly as one with an ordinary history; the difference is that all clusters lexically-enclosed by it also have a history.

An event is an external stimulus that is globally broadcasted across a CHSM; it can cause CHSMs to make transitions.

A.2.3.2 Derived Types

A derived-type is a class derived from either the State, Parent, Cluster, or Set class.⁴⁷ Derived-classes, of course, have access to the protected and public parts of their base classes.⁴⁸ In order that the user can create derived-classes, the interface for each class is shown in figures A.1, A.2, A.4, and A.5; the declaration of the State class is shown in figure A.1.

The STATE_ARGS and STATE_INIT macros are defined to insulate the user from the complex argument-list required of the constructor. The name of a state is its fully-qualified name, e.g., p.x.a; the parent pointer should be obvious.

The Event argument to the ${\tt Enter()}$ and ${\tt Exit()}$ member-functions is a reference to the event causing the current transition. 49 The to argument is the state that the transition is going to; although it has a default, its value must be passed along to ${\tt State::Exit()}$ if overridden in a

 $^{^{47}}$ There is currently no way to derive a class from the Event class.

The reason for having a derived-type is to extend (or alter) the behavior of the aforementioned classes, in particular what happens upon entering or exiting a state by virtue of the virtual member-functions Enter() and Exit().

Although this argument is not used by the run-time library itself, it may be useful in <code>Enter()</code> and <code>Exit()</code> member-functions overridden in derived classes.

derived class. The fromChild argument is non-nil only if a state is being exited as a result of one of its child-states being exited directly.

Figure A.1: State class interface

The DeepClear() member-function clears the history of a cluster, if any, and all descendant child-states that are clusters, if any; the reason it is declared in the State class is so that all derived classes inherit it. The Active() and Inactive() member-functions return the state's active status.

```
#define PARENT_ARGS /* ... */
#define PARENT INIT /* ... */
class Parent : public State {
   // ...
protected:
    typedef State *const *const *Children;
    Children const children;
    virtual void
                    Notify( State *byChild );
    Parent( PARENT_ARGS );
    class ChildIterator {
        // ...
    public:
        void operator()( Children );
        State* operator()();
        ChildIterator( Children );
    };
public:
    virtual void DeepClear();
};
```

Figure A.2: Parent class interface

The declaration of the Parent class, from which the Cluster and Set classes are derived, is shown in figure A.2.

The children data-member is a constant vector of (pointer to) constant pointers to constant pointers to states. This complexity is hidden by the declaration of a local ChildIterator class that is available to derived-classes. It provides a standard method for iterating over a state's child-states; the canonical form for using a ChildIterator is shown in figure A.3 where the example code-fragment is within a derived-class's member-function. The <code>operator()()</code>, when applied to an iterator instance, returns a pointer to the next child-state or <code>nil</code> if there are no more child-states; the <code>operator()(</code> Children) can be used to reset the iterator.

```
ChildIterator next( children );
for ( register State *child; child = next(); )
    // do something with child
```

Figure A.3: Use of the Parent::ChildIterator class

The Notify() member-function is called by a child-state to notify its parent-state that it has become active as the result of a local transition; when called, the state passes a pointer to itself.

The declaration of the Cluster class is shown in figure A.4. The only addition is that of the Clear() member-function that clears the given cluster's history, if any. The declaration of the Set class is shown in figure A.5; there is nothing special about it.

Figure A.4: Cluster class interface

Figure A.5: Set class interface

A derived-type class-declaration would appear in the declarations-section. A sample declaration is shown in figure A.6 (only a class derived from the State class is shown, but deriving from the Cluster and Set classes is the same).

```
class MyState : public State {
    // additional data-members, if any
public:
    MyState( STATE_ARGS ) : State( STATE_INIT ) { /* ... */ }
    // additional member-functions, if any
};

%%
// ...
    state<MyState> spiffy { /* ... */ }
```

Figure A.6: Derived-class declaration and use

The derived-class must at least have one constructor exactly of the form shown in figure A.6 (except that it may have additional default arguments). 50

To augment the behavior of states when entered or exited, the <code>Enter()</code> and <code>Exit()</code> memberfunction can be overridden in derived classes as shown in figure A.7a.

```
class MyState : public State {
    // ...
public:
    MyState( STATE ARGS ) : State( STATE INIT ) { /* ... */ }
    Boolean Enter( Event const&, State* = 0 );
    Boolean Exit (Event const&, State* = 0, State* = 0);
    // ...
};
                               (a)
Boolean MyState::Enter( Event const &event, State *fromChild ) {
    if ( !State::Enter( event, fromChild ) )
         return false;
    // do something above and beyond what ordinary states do
    return true;
}
                               (b)
```

Figure A.7: Overriding Enter() or Exit() member-functions

Figure A.7b shows the canonical form for writing derived-class <code>Enter()</code> or <code>Exit()</code> memberfunctions (only <code>Enter()</code> is shown) where the base class's version is called first, passing the necessary arguments. It is important that the code be written in the manner shown; if either the <code>Enter()</code> or <code>Exit()</code> functions return false, then no additional action should be taken.

Other constructors and additional default arguments are permitted only because they cannot be forbidden. The CHSM compiler cannot emit C++ code to use them, hence this is useless.

A.3 Descriptions

In expressing the syntax of descriptions, a dialect of BNF (Backus-Naur Form) is used. Items enclosed [likethis] indicate optional items; items stacked vertically this indicate a choice of items; items enclosed {likethis} are used to delimit a required choice of items; an item followed by an ellipsis indicates that it may be repeated.

A.3.1 State Descriptions

Plain-state descriptions have the form:

```
state [<derived-type>] identifier \left\{ \left\{ \begin{array}{c} transition-list \\ \end{array} \right\} \right\}
```

A state without a *transition-list* functions as a "sink"; such a state can only be "escaped" from by means of a transition from its parent-state, if any.

A.3.2 Cluster Descriptions

Cluster descriptions have the form:

```
cluster [<derived-type>] identifier (child-list) [[deep] history] [{ transition-list }]
is { description-list }
```

The first child-state defined in the *description-list* is the default child-state.

A.3.3 Set Descriptions

Set descriptions have the form:

```
set [<derived-type>] identifier (child-list) [{ transition-list }] is { description-list }
```

A.3.4 Event Descriptions

Event descriptions have two forms:

```
event identifier [= integer];
event [macro-identifier is] { identifier [= integer] [, identifier[= integer] ]... }
```

The first form is used to declare a single event having the specified numerical value; if none is given, it defaults to zero. 51

The second form (without the *macro-identifier*) is used to declare sequences of events comprising the subsequent event-names with their associated numerical values, if specified; if not, the values assigned progress from zero. If a numerical value is specified for an event, subsequent events up to

Since events need not be declared, declaring one without assigning it a numerical value is redundant; hence, the ability to omit a numerical value in this case does not purport to be a useful feature; it is just easier to allow it than to disallow it.

but not including the next event having an explicitly-specified numerical value, if any, have their numerical values progress from that value.⁵²

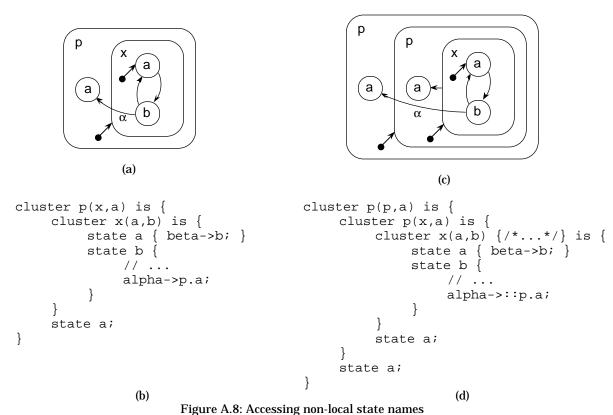
Event sequences can be grouped under a macro-event name. A macro-event is not a new event; it is just a short-hand for the events listed.

An event may not be declared more than once; two or more events may have the same numerical value, however. 53 Although useless, it is legal to declare, but never use an event.

A.4 State-Names

When referring to state-names, they have the form:

That is, either an optional double-colon or string of periods, followed by an identifier, optionally followed by a list of identifiers, each preceded by a period. The double-colon and the string of periods constitute the *scope-resolution* operators (the first is borrowed from C++).



This behavior exactly parallels that of enumeration declarations in C++.

There is currently no way to assign an *enter/exit* event a numerical value.

In the statechart of figure A.8a, described in figure A.8b, the transition on event α is supposed to go to state p's child-state a, but here, "a" by itself is taken to mean cluster x's child-state a. The way to indicate a non-local state-name is by specifying the scope of the desired state. There are three ways to do this. The most straight-forward way is by preceding the state name by that of its parent-state's name and a period. Thus, the transition could be specified as is shown in figure A.8b: p.a. In many cases when scope-resolution is required, this method will be sufficient. 54

A case where this method does not suffice is shown in figure A.8c. Here, the statechart of figure A.8a has been nested one level deeper within another cluster p. Now the outermost state a is the desired state, but specifying p. a as before would be taken to mean the intermediate state a. The other two ways to specify a state's scope can be used in such cases.

The first is to use *relative* specification where a state's name is preceded by one period for every scope to be "backed-up" out of; hence, the state-name could either be specified as . . a or .p.a. The second way is to use *absolute* specification where a state's name is preceded by a double-colon and *all* of its parent-state's names separated by periods; the state-name could be specified as shown in figure A.8d, ::p.a.⁵⁵

A.5 Events

Events have two forms:

The first is for a simple event or macro-event name; the second is for *enter/exit* events. *Enter/exit* events are implicitly broadcast upon the entering/exiting of states; other states can make transitions upon these events just like ordinary events. *Enter/exit* events may not be explicitly broadcast.

A.8 Transitions

Transitions have the form:

When a state has more than one transition with an associated condition that is true, one transition is selected arbitrarily; when a parent- and child-state both have such a transition, the parent-state's is selected.

⁵⁴ It could also be hoped that, in practice, state-names will be different and there will be few non-local transitions.

The difference between the two methods can be summed-up as: relative specification starts at the current scope and works its way out, and absolute specification starts at the outermost scope and works its way in.

Upon a transition, states are always exited from the innermost out, i.e., child-states are always exited before their parent-state; likewise, states are always entered from the outermost in, i.e., a parent-state is entered before any of its child-states. For sets, the relative order that child-states are exited and entered is undefined. Additionally, all states to be exited in response to a given event are exited before any state is (re)entered for the transitions on that event.

A transition's broadcast event and action, if any, are carried out for each transition in turn immediately after the source state is exited and before any target state is (re)entered.

A.9 Conditions and Actions

The code used for an event-condition is that which constitutes a valid C++ expression that yields a non-void value; the code used for a transition-action is a sequence of zero or more valid C++ statements. Such code is passed through, untouched, to the underlying C++ compiler.

Within such code, the function \$in() is available; it takes a state-name as an argument and returns true only if the CHSM is in that state at the time of the call.

Within such code, a state-name may be referred to by enclosing it within \S { }. This is useful with derived-classes in order to access data-members or member-functions. An example is shown in figure A.9.

```
class MyState : public State {
    enum { no, tryLater, anyMinute, yes } permission;
public:
    MyState( STATE_ARGS );
    // ...
    Boolean Okay() const { return permission == yes; }
};

%%
state MayI { ask( ${Receptionist}.Okay() )->GoOnIn; }
state<MyState> Receptionist { /* ... */ }
// ...
```

Figure A.9: Referring to state-names in conditions and actions

Within such code, the variable event is available; it is a constant reference to the event causing the current transition. Additionally, the name of an event evaluates to its numerical value. To refer to *enter/exit* event-names within C++ code, the event-names are preceded by a \$ as in enter(s.x) and exit(s.y). An example is shown in figure A.10.

A.10 Grammar Summary

The grammar summary of CHSM description language syntax shown in figure A.11 is intended as an aid to comprehension, not as an exact specification of the language. This grammar is not

exactly the same one used with *yacc* in the development of the CHSM compiler; it is instead a more human-readable form of it.

Figure A.10: Referring to event-names in conditions and actions

```
CHSM
                                     anything<sub>opt</sub> %% description-list<sub>opt</sub> the-end<sub>opt</sub>
description-list
                                     description-list | description
                                     state-description | cluster-description | set-description |
description
                                          event-description
the-end
                                     %% anything<sub>opt</sub>
state-description
                                     state state-declaration state-transition-spec
                                     cluster state-declaration child-declaration history-option<sub>opt</sub>
cluster-description
                                          transition-spec_{opt} body
history-option
                                     deep<sub>opt</sub> history
set-description
                                     set state-declaration child-declaration transition-specopt body
state-declaration
                                     derived-type<sub>opt</sub> identifier
                                     < identifier >
derived-type
child-declaration
                                     ( child-list )
child-list
                                     identifier | child-list , identifier
event-description
                                     event event-item
                                     event-declaration ; | event-macro<sub>opt</sub> { event-list }
event-item
event-declaration
                                     identifier event-value<sub>opt</sub>
event-value
                                     = integer
event-macro
                                     identifier is
event-list
                                     event-declaration | event-list , event-declaration
state-transition-spec
                                     transition-spec | ;
transition-spec
                                     { transition-list }
transition-list
                                     transition | transition-list transition
                                     event\text{-}condition\text{-}list -> state\text{-}name\ broadcast\text{-}event_{opt}\ action_{opt}\ ;
transition
event-condition-list
                                     event-condition | event-condition-list , event-condition
event-condition
                                     event condition<sub>opt</sub>
event
                                     identifier | enter-exit ( state-name )
```

Figure A.11: CHSM grammar

```
enter | exit
enter-exit
condition
                             ( anything_{opt} embedded-state-name-list anything_{opt})
broadcast-event
                             / identifier
                             scope-resolver_{opt} identifier substates_{opt}
state-name
scope-resolver
                             dot-list | ::
dot-list
                             . | dot-list .
substates
                             substate | substates substate
substate
                             . identifier
body
                             is { description-list }
                             action
embedded-state-name-list
                             embedded-state-name | embedded-state-name-list
                                embedded-state-name
embedded-state-name
                             ${ state-name } | $embedded-keyword( state-name )
embedded-keyword
                             in | enter | exit
```

Figure A.11 (continued): CHSM grammar

Appendix B

Support Classes

B.1 Overview

The development of both the CHSM run-time library and the compiler were greatly eased and the size of the source code greatly reduced by the presence of pre-written classes implementing many popular data-structures: lists (that can also be used as stacks and queues), dynamic vectors, (proper) strings, and symbol-tables. They are presented here for the sake of completeness as they were referenced in the preceding chapters. The coverage is limited since these classes are not the focus of this thesis; the only exception is the coverage of the SymbolTable and its related classes since a symbol-table is an essential part of any compiler.

All of these classes were developed by this author.

B.2 Iterator Classes

All of the classes listed in this appendix also have an associated *iterator* class.⁵⁷ All of the template classes have one of the form shown in figure B.1 where *class* is a given class's name, e.g., ListIterator, VectorIterator, etc.

The constructor takes an instance of a container class to be iterated over. The () operator has been overloaded to return a pointer to the next element; when there are no more elements, it returns a *nil* pointer. The () operator that takes an argument is used to attach an iterator to another (or to the original) container instance to iterate from the beginning (again).

With the exception of the String class, these are all referred to as *container* classes.

⁵⁷ Stroustrup [9], pp. 242-244.

Figure B.1: Typical Iterator class

Iterator instances are typically named next; this allows the iterator idiom shown in figure B.2 to be used, e.g., using a ListIterator to iterate through a List of Symbols.

```
ListIterator<Symbol> next( myList );
for ( register Symbol *e; e = next(); )
    // ...do something with *e...
```

Figure B.2: Iterator example

B.3 The List Classes

The list classes implement doubly-linked lists. The List class is meant to store elements of the same type; a List can be copied. The MixedList class is able to store elements of a type and any type derived from that type; a MixedList can not be copied since there is currently no direct language support in C++ for run-time type-identification.

An element added to either type of list by reference is the responsibility of the user, i.e., it will not be destroyed when the list is; an element added by pointer becomes the responsibility of the list, i.e., it *will* be destroyed when the list is. For elements that have been statically-allocated or allocated on the stack, the former case would be used; for elements that have been dynamically-allocated, the latter case would be used. The former could also be used to allow an element that has been dynamically-allocated to be on more than one list simultaneously.

A list has the notion of a *current* element; the list operations Cut(), CutAll(), Copy(), CopyAll(), Delete(), DeleteAll(), AtAppend(), AtInsert(), AtPop(), Prev(), and Next() are performed relative to it. The current element of a list is maintained independently of the position of any iterator iterating over it.

A list can be used as a stack or a queue by just using the right member-functions: Insert() or Append() to add elements to a list at the head or tail, respectively, and Pop() to remove them.

Many functions are triply overloaded taking a T const&, a T*, and a List<T>& (or a MixedList<T>& for a MixedList). To save space, only the first of the three shall be listed followed by a \dagger to indicate the omission. The public interface of the List class (MixedList is just a subset not having Copy() and CopyAll()) is shown in figure B.3.

```
template<class T> class List {
    // ...
public:
    List();
    List( T const& );<sup>†</sup>
    ~List();
    List<T>& operator=( T const& );<sup>†</sup>
              operator Boolean() const; // anything in list?
    List<T>& Append( T const& );
                                          // append after tail of list
    List<T>& Insert( T const& );<sup>†</sup>
                                          // insert before head of list
    List<T>& operator+=( T const& );<sup>†</sup>
                                          // append: list += elt
    List<T>& operator, ( T const& );
                                          // append: list += elt, elt;
    List<T>& operator^=( T const& );<sup>†</sup>
                                          // insert: list ^= elt
    List<T>& operator[]( int index );
                                         // set current element to index
             operator()() const;
                                          // return current element
    T&
             operator()( long offset ) const;
    List<T> Cut( u_int count = 1 );
                                          // cut from current element on
    List<T> CutAll();
                                          // cut to end of list
    List<T> Copy( u_int count = 1 );
                                          // copy from current element on
    List<T> CopyAll();
                                          // copy to end of list
    List<T>& Delete( u_int count = 1 ); // delete from current element on
    List<T>& DeleteAll();
                                          // delete to end of list
    List<T>& AtAppend( T const& );
                                         // append after current element
    List<T>& AtInsert( T const& );<sup>†</sup>
                                         // insert before current element
            AtPop( u_int count = 1 ); // pop and return current element
             Pop( u_int count = 1 );
    T&
                                         // pop head element
    Т
              *Head(), *Tail();
                                          // position at head/tail
              *Prev(), *Next();
                                          // move to prev/next element
    List<T>& Reset();
                                          // set current element to first
    u_int Size() const;
                                          // number of elements
    int
             At() const;
                                          // index of current element
};
```

Figure B.3: List class interface

B.5 The Vector Class

The Vector class implements dynamic vectors, i.e., vectors that automatically "grow" when an element past the last element is accessed [15]; aside from this, Vectors are used exactly as ordinary vectors. Like the List class, vector elements must be homogeneous. The public interface of the Vector class is shown in figure B.4.

B.6 The String Class

The String class implements true character-strings (as opposed to traditional char*'s). Like Vectors, Strings automatically "grow" as needed. Many functions can be performed on Strings: appending and inserting characters, substring matching and replacement, token extraction using

user-specified delimiters, concatenation, lexical comparisons, etc. There also exists a SubString class that is used to refer to a part of a String. SubStrings are *l-values* thus allowing replacement; an example is given in figure B.5.

```
template<class T> class Vector {
    // ...
public:
    Vector( Vector<T> const& );
    ~Vector();
    Vector<T>&
               operator=( Vector<T> const& );
                operator Boolean() const; // anything in vector?
    T&
                operator[]( u_short i );
                                        // access i'th element
               operator+=( T const& );
operator, ( T const& );
    Vector<T>&
                                        // append: v += elt
                                        // append: v += elt, elt
    Vector<T>&
             Size() const;
    u_short
                                        // number of elements
    Vector<T>&
               Size( u_short new_size ); // set size
};
```

Figure B.4: Vector class interface

Figure B.5: SubString example

Many functions are triply overloaded taking a String const&, a char*, and a char; also, many binary operators are overloaded to take combinations of the aforementioned argument types (to avoid temporary variable generation). To save space, only a single representative function or operator shall be listed followed by a † to indicate the omission. The public interface of the String class is shown in figure B.6.

B.7 The SymbolTable Class

The SymbolTable class provides full symbol-table management and was designed to be trivially user-extensible for various user-defined symbol-types. It uses a hash-table of the symbol's names; each entry is (of course) a list of all those symbols in collision for a given hash-code. The Symbol class contains a name and a list of symbol information, i.e., instances of the Symfo class. It is the Symfos that actually contain the interesting information about a symbol at a given scope. Typically, if a symbol has a name that matches a preexisting symbol in a lexically-enclosing scope, a new

The hash function used is *hashpjw*, used in Peter J. Weinberger's C compiler. Aho et al. [14], pp. 435-436.

Synfo is pushed at the head of the list of Synfos, hence, when subsequently accessed, the current information is returned. These class relationships are shown in figure B.7.

```
class String {
    // ...
public:
    String( int reserve = 0 );
    String( String const&, int at_most = INT_MAX );<sup>†</sup>
    ~String();
    String& operator=( String const& ); †
              operator char const*() const;
    String& Append(String const&); †
    String& operator+=( String const& ); †
    SubString operator[]( int index );
    SubString operator[]( String const& ); †
    SubString operator()( int at_most, int at );
    SubString operator()( char const *delim, int at );
    friend Boolean operator == ( String const&, String const& ); †
    // ...remaining relational operators elided...
    String Cut ( int at_most, int at = 0 );
    String Copy( int at_most, int at = 0 ) const;
    String Cut ( char const *delim = White_c, int at = 0 );
    String Copy( char const *delim = White_c, int at = 0 ) const;
    String& Delete( int at_most, int at = 0 );
    String& Delete( char const *delim, int at = 0 );
String& Trunc( int new_len );
    String
            Trunc( char const *delim, int at = 0 );
    int
              Length();
    friend ostream& operator<<( ostream&, String const& );</pre>
};
```

Figure B.6: String class interface

The SymbolTable class is actually very simple and contains only a few member-functions. The public interface of the SymbolTable class is shown in figure B.8. When entering a new scope, one calls OpenScope(); when exiting the current scope, one calls CloseScope() to destroy all symbols local to that scope.59

The Synfo class contains the actual information for a symbol, i.e., its scope and its type. Its complete class declaration is shown in figure B.9. The Synfo class as-is is virtually useless; it is the intention that various classes be derived from Synfo, one for each type of symbol used in a given compiler.

It is the Synfos that are actually destroyed; a Symbol is destroyed only when its list of Synfos is empty.

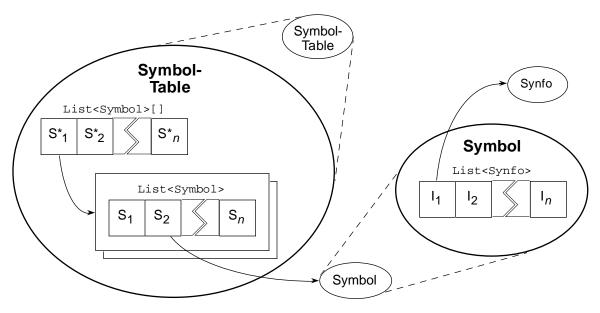


Figure B.7: SymbolTable/Symbol/Symfo class relationships

```
class SymbolTable {
    // ...
public:
    SymbolTable();
             operator[]( char const* ); // look-up name
    Symbol*
             operator+=( char const* ); // add to table
    Symbol*
    Symbol*
             operator+=( Symbol* );
                                         // return current scope
    static int
                  Scope();
    SymbolTable& OpenScope();
                                         // create a new scope
    SymbolTable& CloseScope();
                                         // destroy all local symbols
};
```

Figure B.8: SymbolTable class interface

The Synfo class maintains the current scope in addition to the scope for itself. The constructor optionally takes a scope, defaulting to the current scope if none is given; the destructor is virtual to handle destruction of objects of classes derived from the Synfo class properly. The What() member-function is meant to return the type of the symbol information that a Synfo instance is; the member-function is overridden in derived classes to return different types. The Emit() member-function is used to have a symbol emit its representation, whatever it might be, into the object-code; this too is overridden in derived classes.

The complete class declaration of the Symbol class is shown in figure B.10. Its constructor takes the symbol's name-to-be and an optional pointer to a Symfo.60 The Info() member-function that

In many cases, the Synfo argument is not given since, in a typical compiler that employs *lex* and *yacc*, an identifier returned by *lex* is added to the symbol-table before the code executing that figures out what type of symbol it is supposed to be.

takes a Synfo* inserts a new Synfo into the symbol's list in scope order with the highest scope being at the head of the list. The Info() member-function that takes a u_int returns a pointer to the Synfo, hence the information for, the symbol (offset in scope by the argument, if given); if there is no Synfo at the current (or specified) scope, zero is returned. The What() member-function behaves similarly to the latter Info(), but instead returns the type of the symbol.

```
class Synfo {
    static int currentScope;
protected:
    virtual ostream& Emit( ostream&, char const *name ) const;
public:
                      { NoScope = -2, Global };
    enum
    typedef u_int
                      Type;
    int const scope;
    Synfo( int s = currentScope ) : scope( s ) { }
                                   // we expect to be derived from
    virtual ~Synfo();
    virtual Type What() const; // type of symbol
                  IsLocal() const { return scope == currentScope; }
    friend ostream& operator<<( ostream&, Symbol const& );</pre>
    friend class SymbolTable; // so it can get at currentScope
};
                        Figure B.9: Synfo class interface
class Symbol {
    MixedList<Synfo> info;
public:
    String const
                    name;
    Symbol( char const *s, Symfo *i = 0 ) : name( s ), info( i ) { }
                  Info( u_int in = 0 ) const; // return info.
    Synfo*
                  Info( Synfo* );
                                                 // insert new info.
    Synfo*
                                                 // delete current info.
    Symbol&
                 DeleteInfo();
    Synfo::Type What( u_int in = 0 ) const; // type of symbol
                 Scope() const;
                                                // scope of symbol
    int
    // ...relational operators elided...
```

Figure B.10: Symbol class declaration

};

The Symbol class has the << operator overloaded for ostreams in order that symbols can be emitted easily; the actual emission is deferred to the Symfo class, passing the symbol's name.

Appendix C

Compiler Manual Page

NAME

chsm - Concurrent, Hierarchical, Finite State Machine compiler

SYNOPSIS

chsm [options] source-file

DESCRIPTION

chsm is the AT&T Concurrent, Hierarchical, Finite State Machine (CHSM) compiler. The command uses **chsm2c++** for syntax-checking, type-checking, intermediate code generation, and **CC**(1) for code generation.

chsm requires CHSM source files to end with a **.m** suffix. Unless either the **-c** or the **-o** option are used, **chsm** produces an output file names **a.out**.

For the CHSM source file, **chsm** creates a temporary file, *file*.**C**, containing the generated C++ code in the current directory for subsequent compilation with CC(1). The +i option preserves the file.

In addition to the options described below, **chsm** accepts other options via the $-\mathbf{O}$ option and passes them on to the C++ compilation system tools. See **cpp**(1) for preprocessor options, **CC**(1) for C++ compiler options, **cc**(1) for C compiler options, **ld**(1) for link-editor options, and **as**(1) for assembler options.

Unlike some other compilation tools, options supplied to **chsm** are *position-independent*. The following options apply to the chsm-specific parts of the compilation process:

-c	Passed to CC (1) to suppress linking with Id (1) and just produce a .o file.
	This option renders the -l and -L options useless.

- -d Passed to chsm2c++ to suppress #line directives in the intermediate C++ file.
- -**D**name[=def] Passed to **CC**(1) to define a symbol name for **cpp**(1). To turn on CHSM debugging by default, define the symbol CHSM_DEBUG.
- -E Run only **chsm2c++** on the CHSM course file and send the result to standard-output. This option overrides all other options except -h.
- -g Passed to **CC**(1) to produce additional symbol-table information used for C++ source debuggers.

-h Passed to **chsm2c++** to generate the file **chsm_incl.h** with C++ **extern** statements so that other C++ source files can access event names.

+i Leaves the intermediate .C file in the current directory after the compila-

tion process.

-**I**directory Passed to **CC**(1) to add directory to the list of directories in which to search

for **#include** files for **cpp**(1).

-l*library* Passed to **CC**(1) to link the resultant C++ object code file with the library

library.

-Ldirectory Passed to CC(1) to add directory to the list of directories in which to search

for libraries for linking using ld(1).

-O", other" Pass the comma-separated argument string "other" to CC(1) to allow

implementation-dependent options to be specified. Note the requirement

of the initial comma.

FILES

file.m chsm source file

*file.***C** intermediate C++ source file

file.o object file

a.out default executable output file name

chsm CHSM driver shell script

chsm2c++ CHSM compiler

libchsm.a CHSM run-time library

SEE ALSO

 $\mathbf{cpp}(1)$, $\mathbf{CC}(1)$, $\mathbf{cc}(1)$, $\mathbf{as}(1)$, $\mathbf{ld}(1)$, and "An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines," by Paul J. Lucas, M.S. thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1993.

DIAGNOSTICS

The diagnostics produced by **chsm** itself are intended to be self-explanatory.

References

- [1] David Harel, et al. "On the Formal Semantics of Statecharts." *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, NY, 1987. pp. 54–64.
- [2] David Harel. "On Visual Formalisms." *Communications of the ACM*, vol. 31 no. 5, May 1988. pp. 514–530.
- [3] —. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, vol. 8, 1987. pp. 231–274.
- [4] M. E. Lesk. "Lex—A Lexical Analyzer Generator" *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [5] S. C. Johnson. "Yacc—Yet Another Compiler Compiler." *Computing Science Technical Report* 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [6] i-Logix, Inc. The Semantics of Statecharts. i-Logix Inc., Burlington, MA, 1991.
- [7] David Harel, et al. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Transactions on Software Engineering*, vol. 16 no. 4, April 1990. pp. 403–414.
- [8] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.
- [9] Bjarne Stroustrup. *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, MA. 1991.
- [10] —. "What is Object-Oriented Programming?" *IEEE Software*, vol. 5, May 1988. pp. 10–20.
- [11] Tadao Murata. "Petri Nets: Properties, Analysis and Applications." *Proceedings of the IEEE*, vol. 99, issue 4, April 1989. pp. 541–580.
- [12] Morris I. Bolsky and David G. Korn. *The KornShell Command and Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [13] Vern Paxson, et al. Flex—A Fast Lexical Analyzer generator. No published paper. Flex is available via anonymous ftp from ftp.ee.lbl.gov.
- [14] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
- [15] Thomas Cargil. "A Dynamic Vector is Harder Than It Looks." *C++ Report*, vol. 4 no. 5, June 1992. pp. 47–50.