4-22-2014 12:00 AM

State Controlled Object Oriented Programming

Jamil Ahmed
The University of Western Ontario

Supervisor
Dr. Stephen M. Watt
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy
© Jamil Ahmed 2014

STATE CONTROLLED OBJECT ORIENTED PROGRAMMING

(Thesis format: Monograph)

by

Jamil Ahmed

Graduate Program in Computer Science

A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

The School of Graduate and Postdoctoral Studies

The University of Western Ontario

London, Ontario, Canada

April, 2014

Abstract

In this thesis, we examine an extension to the idea of object oriented programming to make programs easier for people and compilers to understand. Often objects behave differently depending on the history of past operations as well as their input that is their behavior depends on state. We may think of the fields of an object as encoding two kinds of information: data that makes up the useful information in the object and state that controls its behavior. Object oriented languages do not distinguish these two. We propose that by specifying these two, programs become clearer for people to write and understand and easier for machines to transform and optimize.

We introduce the notion of state controlled object oriented programming, abbreviated as "SCOOP", which encompasses explicit support of state in objects. While introducing an extension to object oriented programming, our objective is to minimize any burden on the programmer while programming with SCOOP. Static detection of the current state of an object by programming languages has been a challenge. To overcome this challenge without compromising our objective, a technique is presented that advances contemporary work.

We propose an implementation scheme for a SCOOP compiler that effectively synchronizes the external and internal representation of state of objects. As an implication of this scheme, SCOOP would provide the memento design pattern by default.

We also show how a portion of an object particular to its state can be replaced dynamically, allowing state dependent polymorphism. Further, we discuss how programs coded in SCOOP can be model checked.

Keyword: State Oriented Programming, Object Oriented Programming, Typestate, Finite State Automata, Dynamic Compositional Adaptation, Specification and Verification.

Acknowledgments

I would like to express my heartily gratitude for being supervised by Prof. Dr. Stephen. M. Watt. Without his guidance, mentorship, vision and shrewd insight, this work would have not been accomplished. I am highly impressed how his extraordinary determination, humbleness, patience and intellect overcome difficult situations not only in computing research but also in normal day to day life. His immense support, consistently motivating attitude and willingness to elucidate significantly contributed to this research. I have highly benefited by his skill of not just preparing his students for thesis defense but also preparing them to research independently.

I will always be deeply thankful to my original supervisor Prof. Dr. Sheng Yu. Although his precious advice lasted temporally for only two year due to his sudden passing in Jan 2012, it will enlighten my career forever. His initial work related to this thesis sparked tremendous ideas to integrate in this research. It was a difficult time when he passed, half way into my PhD program, and then Dr. Stephen M. Watt kindly accepted supervising me and gave me freedom of thought.

I am thankful for the funding for my PhD program from Higher Education Commission Pakistan, University of Karachi and Western Graduate Research Scholarship of Department of Computer Science, Western University.

I am also thankful to my brother Sherjil Ahmed for many fruitful technical discussions. The selfless love and care from my parents gave me the confidence to go through the entire horizons of my PhD program and life as well.

Contents

Typographic Conventions

We use the following typographic conventions in the thesis.

Italic

State names, Typestate names.

E.g. *traversal*

Courier font

Programs, Program constructs.

E.g. `class`

In single quotes

Program name, Class names, Object names, Names of object types, Attribute names, Method names and Event names excluding the programs and program constructs.

E.g. ‘Progfile’

Abbreviations

FSA	Finite State Automata
LALR	Look Ahead Left to Right
MAPE-K	Manage Analyze Plan Execute Knowledge
OOP	Object Oriented Programming
SCOOP	State Controlled Object Oriented Programming
TSOP	Typestate Oriented Programming

List of Figures

Chapter 1

1 Introduction

1.1 State Controlled Object Oriented Programming

In this thesis, we propose a “state controlled” object oriented programming language. We also propose several related techniques aimed at making the proposed language easier for the programmer to use in comparison with similar approaches.

One main objective of software engineering is to provide effective methods and techniques for the design, development and maintenance of software. On its way to achieve this objective, software engineering has advanced significantly and influenced many software technologies, including programming languages. Programming language paradigms have evolved from low-level machine languages to current state-of-the-art higher-level languages, achieving better abstraction, increased readability and more flexible reusability. Object oriented programming (OOP) is one of the main technologies used by software developers today for developing robust software [1]. However, OOP does not allow an object to be specified as a state machine in a convenient, readable way. Moreover, there is a significant gap between OOP and the input languages for formal verification tools [53]. For these reasons, OOP is still not robust enough to address many of the needs of the formal language community. In order to overcome these limitations, many extensions to the idea of OOP have been studied and proposed over the years.

In conventional object oriented programming languages, a software object encapsulates attributes (data) and methods (behavior) [2], similar to a real-world object. The attributes and methods of a software object are defined with explicit syntax. Besides attributes and methods, real-world objects maintain “state” as an inherent characteristic. Therefore, in object oriented programming, a software object modeled on the real world should also support “state” as a built-in characteristic.

It has been argued that the inclusion of “state” in software objects makes OOP more intuitive for the programmer. For instance, a software application may have a ‘Car’ object existing either in an assembled state or a disassembled state. It can be observed in this examplethat a state is not an independent entity by itself, but rather an inherent characteristic of an object. The assembled and disassembled states of the ‘Car’ object do not exist as independent entities; but, it makes sense that the ‘Car’ object is an independent entity that can be in the disassembled or the assembled state. We argue that an object encapsulates its states in addition to its attributes and methods. Therefore, an object definition should also support an explicit syntax in order to declare its states and those of its related counterparts. This is our main theme.

We refer to an object with explicit support of state as a “stated object” and programming with stated objects as “State Controlled Object Oriented Programming”, which we abbreviate “SCOOP” (we use this name both for the concept of programming in this way and for the specific language we propose). The class to which a stated object belongs is referred to as its “stated class”.

In standard object oriented programming, objects have attributes (data) that can be modified by methods, so the usual objects already have “state” in this sense. But in our sense of stated object, State Controlled Object Oriented Programming provides an explicit syntax to define the states of an object such that the defined states are encapsulated within the object in the same way as attributes and methods. Several definitions related to the notion of a “stated object” are given in Section 2.1. There are other notions of state used in automata theory, some including time, and it would be possible to explore these, but that is not the focus of the present work.

Engineers, scientists, designers and the formal language community use finite state automata to model systems and the objects of systems. However, there is not yet a standard to directly transform a finite state model of an object to an object oriented application. This is due to the lack of explicit formal support for states (in our sense) in standard OOP languages. For instance, a DVD player can be viewed as a stated object that can be represented by a finite state model. The DVD player has states *off*, *play*, *pause*, *fast forward* and *reverse*, and transitions among these states. We intend to formalize the process of modeling a software object to include a finite state automaton and to use it to increase the clarity of programs and their representation.

The terms “state” and “typestate” are sometimes used interchangeably. In this thesis, we use the term “state” in a general context or when discussing the usual notions of finite state machines. We use the term “typestate” in discussions on both the theoretical and practical aspects of software using state control [23].

In this chapter, we lay out the motivation for state-controlled object oriented programming and present the thesis contribution as well as its orientation.

1.2 State Abstraction

Abstraction is the process in programming that allows the complexity of certain aspects of a program to be limited to parts of the program and provides a clean interface for using those features. For instance, object oriented features such as polymorphism and encapsulation can be implemented in a non-object oriented programming language (such as C) by exploiting related design patterns [3], but at the cost of complex logic to be coded by the programmer. Alternatively, object oriented languages provide declarative syntax that abstracts the object oriented features and reduces the complexity of code to be written by the programmer. Similarly, in conventional object oriented programming, a programmer can write code to maintain the state of an object, but at the cost of writing the state maintenance code himself.

Usually state maintenance code is in the form of conditional checks, state tables or in the form of state design patterns that potentially make the object oriented code more complex.

Alternatively, state controlled object oriented programming provides an abstract "state" feature which hides the complexity of the state maintenance code inside the language. The abstract "state" of an object encapsulates the nonfunctional aspects (i.e. the state-related data members) and the functional aspects (i.e. the behavior or methods specific to state) inside the object.

1.3 Motivation

Stated objects can be simulated as finite state automata due to built-in abstraction of state and state transitions. Finite state automata (FSA) deterministically allow valid transitions (actions) and prohibit invalid transitions (actions) contingent on the current state of a system. Therefore, finite state automata inherently support validation in that, depending on the current state, transitions are either valid or invalid. Similarly, an object with state that can simulate an FSA can allow invocation of valid actions (methods) and prohibit invocation of invalid actions (methods) contingent on its current state. The ability of an object to allow or prohibit invocation of its methods provides better support for software validation. Therefore, by way of finite state automata functionality, SCOOP ensures that the programmer can write safe code that prohibits method invocation and field references that are invalid for the reason of state.

In object oriented programming, the object protocol [4] defines the set of valid methods that can be invoked and valid fields that can be referenced. In state controlled object oriented programming, the current typestate of a software object defines its protocol, since the software object can assume different states during its lifetime. The SCOOP language performs static (compile time) analysis to detect an invalid field invocation of a software object according to the object's current typestate. In this way, typestate is a useful tool for enforcing the checking of the invocation of methods of software objects, which minimizes the run time exceptions caused by invalid method invocation.

An important implication of state abstraction is that it allows the definition of a stated object composed of stated objects. Defining a stated object composed of stated objects allows better and more precise simulation of real-world scenarios. The encapsulating stated object can have states that can be a combination of the states of encapsulated stated objects.

In conventional OOP, the state of an object is represented by the instance data members. SCOOP, however, allows explicit declaration of the abstract states of an object. An abstract state is accessible from outside of the object, and is bound with state-related instance data members inside the object definition. The internal state of the object is represented by data members that are referred to as typestate invariant. Therefore, in order to access the typestate of an object, the programmer can access the abstract external typestate, i.e. a single piece of information, of an object from outside the object rather than accessing many internal typestate invariants. This mechanism does not violate the information hiding principle and maintains a typestate of the object that is accessible from outside the object.

In conventional OOP, polymorphism is implemented either through method overloading within a class or both method overriding and overloading in a subclass. Stated objects introduce another dimension of polymorphism, i.e. the same method can have different implementations in different states of the stated object.

In SCOOP, the state of an object is also a first class language entity. This implies that an object of type “typestate” can be instantiated. As the typestate of an object encapsulates the attributes and methods specific to it, the state of an object can be returned from a function, can be saved in an object of type “typestate” or can be passed on as an argument to a function.

We use SCOOP to introduce a language-based approach for dynamic adaptation. A stated object can dynamically adapt to an entirely new typestate behavior that was not known at the time of compiling that stated object.

1.4 Contribution of Thesis

This thesis advances the features and architecture for encapsulating explicit typestate in objects for better clarity of programs, as discussed in Chapter 5. The proposed SCOOP language, described in Section 5.3, has not been implemented. However, we present related components and techniques to achieve the overall desired objectives of the proposed language. We integrate many aspects of typestate and their solutions into one programming model, SCOOP. Both an object and its typestate as a first class language concept are introduced for the first time, to the best of our knowledge, which we argue allows for increased flexibility of stated objects.

SCOOP provides a technique for typestate checking under aliasing. A novel "automated alias analysis" is performed that exploits the proposed alias table, as discussed in Section 5.14 and 5.15. We propose a flow-sensitive stated type system in Section 5.4. The proposed stated type system performs typestate analysis by using a proposed implementation technique for stated objects, the "proxy and state class" architecture, in Section 5.10. The proposed "proxy and state class" architecture allows static detection of invalid invocation of methods by using the instances of state class.

Our proposed approach to static and dynamic typestate checking is without any additional program annotation. Typestate checking is performed along with static analysis; therefore, a separate typestate analysis phase is not required. SCOOP proposes a built-in transition function that can be used to directly transition an object to another typestate. Further, SCOOP proposes the translation of programmer-defined code to intermediate code that is amenable to compiler interpretation of object typestates.

Our proposed typestate checking technique, presented in Section 5.17, generates a compile time error or a warning for any invalid method invocation, allowing the programmer to write safe code. If neither an error nor a warning is generated at compile time then it is assured that the code does not contain any method invocation that is invalid for reasons of state.

A novel “proxy and state class” architecture that allows static typestate checking in the presence of aliasing is proposed in Section 5.10. The proposed architecture does not impose any limitation on aliasing and it is generalized to allow typestate extension and subclassing of stated objects simultaneously.

A novel typestate checking technique is proposed that statically computes the set of current possible typestates of a stated object and updates each alias of that object to be aware of that set of typestates.

Implementation techniques for typestate invariants and invariant binding are proposed in Sections 6.3 and 6.5 respectively. The implementation technique for typestate invariants introduces a novel invariant table concept. The notion of invariant table is generalized enough to allow typestate extension and sub classing simultaneously. The stated type system proposes to synchronize the external typestate and its internal typestate invariant. Further the proposed invariant table ensures that each typestate is aware of its typestate invariant data. An implication of our proposed implementation technique for typestate invariants is that the “memento” design pattern is supported by SCOOP as a built-in feature.

We propose a SCOOP language-based dynamic adaptation technique for autonomic computing. Our proposed novel dynamic adaptation technique, based on replacing typestate-related behavior of stated objects, allows a specific typestate-related part of an object to be replaced dynamically. This typestate-based dynamic adaptation is also achieved due to the generality of the proposed novel ‘proxy and state class’ architecture.

In Section 7.4, several algorithms are introduced that can implement the dynamic swapping of typestate behavior at run time, even in the case of sub classing and extended typestate.

We also present the ability of SCOOP programs to be directly model-checked by model checking software.

1.5 Thesis Orientation

Chapter 2:

In this chapter, we present the main concepts and the features supported by our proposed SCOOP.

Chapter 3:

In this chapter, we present several case studies of business and control-related or embedded applications that can benefit from SCOOP. The purpose of this chapter is to demonstrate that many programming scenarios can be thought of as finite state machines.

Chapter 4:

In this chapter, we present background and related work. Comparison is made with earlier state oriented approaches and the distinguishing features of the proposed SCOOP are also mentioned.

Chapter 5:

In this chapter, the basic theoretical concepts of SCOOP are discussed. The syntax of a sample program written in the proposed SCOOP language is given in Figure 5.1. Figure 5.2 illustrates the translation of programmer-defined code by SCOOP. The architecture to organize the states in a stated object is given in Section 5.10. In Section 5.3, a context free grammar of the proposed SCOOP language is discussed. Later in the chapter, several related components and techniques for static analysis of SCOOP programs are presented. Overall, these components and techniques allow static typestate checking in the presence of aliasing.

Chapter 6:

In this chapter, we present a typestate invariant implementation technique. The use of an invariant table is also presented. The idea of default "state preservation" by SCOOP is introduced.

Chapter 7:

In this chapter, we present a dynamic adaptation technique based on replacing the "typestate structure" of a stated object.

Chapter 8:

In this chapter, we present how SCOOP can be exploited for model checking.

Chapter 9:

We conclude the thesis and mention the future directions of work in this chapter.

Chapter 2

2 Main Concepts

In order to achieve the best implementation, SCOOP is based on precise definitions and well-defined fundamental concepts. We emphasize that research in this area should differentiate between the stated object type and typestate. Therefore, we first present the definitions of related concepts and then we propose an architecture for SCOOP that aligns with the well-defined fundamental concepts. At the conceptual level, our definitions coincide with the originally proposed typestate concepts in [23], but our revision of these underlying concepts, specific to object oriented programming, is significant.

2.1 Definitions

In OOP an object type is a description of a set of fields (data members and methods) of the object without giving any implementation [25]. Similarly, in SCOOP a stated object type is a description of a set of fields (data members and methods) and a set of typestates of the stated object without giving any implementation.

Definition of Typestate

In this thesis we consider each object to have a property called its "typestate". The typestate of an object has a value belonging to a finite set, and is used as a label. As we shall see later, the set of methods or fields available in the object will depend on the value of the object's typestate, and executing methods of the object may change its typestate. The only operations available on typestates are tests for equality or inequality of typestates. We write T_O for the set of typestates available to the object *O*.

Formally, we may identify these typesates with the set of methods and fields that are available when the object is in that typestate.

Definition of Stated Object Class

A "stated object class" is tuple *$(C, T, B, M, \beta, \mu, v)$* consisting of an object class, C, together with a set of typestates, T, a set of field labels, B, and a set of method labels, M, together with three mappings β, μ and v.

The mapping β: $T \rightarrow 2^B$ specifies the subset of fields available to an object in a given typestate.

The mapping μ: $T \rightarrow 2^M$, specifies the subset of methods that are available an objects in a given typestate.

The mapping v: $T \times M \rightarrow 2^T$ specifies the set of possible resulting typestates from invoking a given method from an object in a given typestate.

Definition of Stated Object

A "stated object" is an object belonging to stated object class. It may be viewed as an ordered pair $s = (o, t) \in C \times T$, where the fields and methods available are given by $\beta\ (t)$ and $\mu\ (t)$, respectively.

2.2 Typestate

In conventional OOP, an object encapsulates its fields (data) and methods (behavior) [2]. We propose that a typestate of an object can be thought of as a new kind of property, in addition to the attributes and methods of the object. The explicitly defined typestates within a class form part of the object interface. We refer to such a "typestate" as an external typestate of the object that is always publicly accessible.

The type of an object defines some conditions, or rules, for the object. For instance, it defines the selected kinds of instances that can only be assigned to the object referent. Further, the type defines the description of the interface of an object without implementation [25]. The interface includes the selected fields (data members and methods) that can only be invoked by the object. The conditions defined by the object type are validated by the type system. The type system detects, either statically or dynamically, any violation of those conditions. Type theory for conventional object oriented programming has been studied and has a sound foundation. However, type theory for OOP does not include any specification for the explicitly defined typestates of an object.

In order to address the issues for typestate, we need to have a type system with support for typestates [23] rather than a conventional type system as for conventional object oriented programming. We refer to such a type system with typestates as a "stated type system". Typestate is the theoretical basis for the state abstraction feature. Considering typestate as a formal representation allows a stated type system to be subject to formal reasoning. For example, a stated type system can perform global analysis for typestate tracking of stated objects.

A typestate defines the conditions or rules of an object for a given state of that object, and it describes the state of the object without implementation. For example, it defines the fields (attributes and methods) particular to a state of the object. Effectively, a stated type system enforces the conditions defined by typestates.

A stated object during its life-time holds one type but possibly transitions among its many typestates. A fully qualified typestate of a stated object is given by (type, typestate). E.g. the *openfile* typestate of an object of type 'File' can also be represented as (File, *openfile*) and it can transition to (File, *closefile*). It is important to highlight that when a stated object switches from one typestate to another, its corresponding typestate changes but its type remains the same.

2.3 Software Verification

Software verification can be viewed in one way as a process performed directly by the programming language. In this case, verification includes static analysis, i.e. at compile time, to detect invalid field invocations of a software object according to the current typestate. Therefore, static typestate checking allows for writing safer code by finding at compile time whether a programmer has attempted to use a method or data member of a stated object that should not be accessible in its current state. This view of software verification captures the notion of object protocol.

2.4 The Object Protocol

The current typestate of an object determines the set of accessible attributes and methods that are available or make sense for the object. We call this the object's "protocol" [4]. The set of accessible methods varies with the change of an object's typestate. Therefore, the current typestate of an object controls the available behavior of the object. For instance, a method of an object that is available in one typestate may not be available in another typestate.

In conventional object oriented programming, the typestates of an object are not explicitly exhibited by the object; therefore, the current typestate cannot be detected. In SCOOP we can directly detect the set of possible current typestates of an object because they are explicitly defined.

The method invocations that should not be accessible in a specific typestate of an object are referred to as "invalid" methods. Invalid method invocations cause run time exceptions or unexpected results. SCOOP uses compile-time analysis to find invalid method invocations, thus helping programmers to write safe code. Detecting an invalid method invocation according to the current typestate, i.e. violation of object protocol, is contingent on whether the current typestate of an object can be detected or not.

2.5 Typestate Based Polymorphism

Typestates allow a new kind of polymorphism. Different implementations of functions having the same signature may coexist in different typestates, leading to typestate based polymorphism. This feature is studied in [17, 18, 20, 21, 22, 24] and supported by SCOOP as well.

2.6 The Typestate Invariant

In conventional object oriented programming, the programmer uses attributes to represent the state of an object. In state controlled object oriented programming, the language processor recognizes the attributes (data) of an object as the typestate invariants, as discussed in Chapter 6. The external typestates of an object are tied up with typestate invariants. These attributes represent the internal state of the object. Our proposed stated type system also tracks the external typestate by monitoring the typestate invariants of a typestate.

For instance, the *openfile* typestate of a 'File' object can be associated with the 'filepointer' attribute of the 'File' object. Such an association can be controlled by a rule. In this case, a rule can be defined such that if 'File' is in the *openfile* state then the 'filepointer' attribute must hold a valid address of a file control block in memory and cannot be null.

2.7 Typestate Extension

SCOOP allows for the extension of a typestate. The typestates of an object, defined at the time of object definition, are not necessarily fixed but can also be extended through subclasses. A subclass can override the typestate of its parent class and may override all methods of the typestate of the parent class. Moreover, in the subclass the overridden typestate can be extended. Typestate extension is illustrated in Section 6.6 and Section 6.7.

2.8 Typestate Mapping

In software systems, interacting objects are likely to have interdependent typestates. In such circumstances, we need to have a mechanism for mapping similar or interdependent typestates of the objects. For instance, a printer and a cartridge are two objects arranged in such a way that the printer includes a cartridge. Since the printer includes the cartridge, we refer to the printer as a composite object. Their behavior and typestates are also similar to each other. If the cartridge is in the *fullcharged* typestate, so is the printer. If the cartridge is in the *halfcharged* typestate, so is the printer. The mapping of similar typestates of these objects can enforce that whenever the printer transitions to the *fullcharged* typestate then the cartridge must also transition to the corresponding typestate.

An obvious benefit of such mappings is that the programmer need not take care of transitioning the typestate of corresponding objects. The transition of a typestate in one object can automatically transition the mapped typestate of the corresponding object. This is illustrated in Case Study 7.5.1.

2.9 State Preservation

In conventional OOP, as previously mentioned, attributes (data members) represent the states of an object. Therefore, in order to preserve state, each and every state data member of an object has to be extracted by the programmer and passed on to another function one by one. Otherwise the programmer needs to write a separate function to package all its state data members in the form of a data structure, e.g. an ‘Array’. This is how state can be preserved. Alternatively the “memento” design pattern has to be implemented and used by the programmer to preserve the state.

One of the contributions of our proposed state controlled object oriented programming over conventional object oriented programming is that the state controlled object oriented language maintains and is aware of all typestate data of stated objects, which can be extracted just by using the external typestate name, i.e. a single piece of information. The external typestate data can be passed on to another stated object. This is an implication of the fact that state is a first class language concept in SCOOP. This “state preservation” capability allows a programmer to write simpler code. It is described in Section 6.8.

2.10 Dynamic Adaptation

Dynamic adaptation is one of the desired requirements of autonomic computing [27, 34 - 40, 42]. SCOOP proposes that partial behavior particular to a typestate of a stated object can be modified dynamically. The rest of the typestates, data and behavior associated with each typestate remain intact. This is unlike conventional object oriented programming. In conventional object oriented programming, if we replace an object with another object instance then the entire object instance is overwritten, such that all of its typestate data and the behavior associated with any typestate is replaced.

Such dynamic adaptation is desirable in many software development cases [41, 42]. This is illustrated in the ‘Queue’ stated object Case Study 3.11 and ‘File’ stated object Case Study 3.14. We also consider the dynamic adaptation behavior of an encrypted message over a network in Section 7.2.

2.11 Feasible for Model Checking

Model checking is usually applied at the software design level rather than at the software source code level because source code does not exhibit any representation suitable for model checking. The inability to model check source code results in the inability to catch software bugs introduced in the source code. In order to catch errors in source code through model checking, we require that the source code have an abstraction that exhibits a suitable representation for model checking, such that a model of source code can be built. We demonstrate in Chapter 9 that SCOOP can be used to model check its programs.

Chapter 3

3 Case Studies of SCOOP

Many software applications and software objects adhere to the behavior of finite state machines. These software applications span from system level software to higher level applications, control-related embedded software to business applications, basic data structures to programming languages, robotics and automation to communication systems, word processing to scientific and mathematical tools etc. We argue that SCOOP is useful for writing these kinds of software applications because SCOOP allows object states to be directly encoded, leading to increased readability. Implementing these applications using stated objects also helps the programmer to avoid writing state maintenance code.

In this chapter, we illustrate these software applications and a set of objects that behave similarly to FSA. In Appendix B, we illustrate a stated object coded in SCOOP. Its equivalent code in OOP is also presented using conditional constructs and a state design pattern, demonstrating that SCOOP code is more readable.

3.1 Screen Redraw Thread

An operating system is a system-level program that creates many threads to perform different tasks at the same time. In such a multitasking system, an application (e.g. a 'digital circuit simulation' application) running on the operating system, can initiate more than one thread of execution to perform different tasks simultaneously. When the digital circuit simulation application is run, the operating system creates a process and the application is loaded into that process. Each thread created by the 'digital circuit simulation' application exploits the corresponding operating system thread feature. The operating system, e.g. Windows, provides a thread object that exists in different states during its life time [8]. The state diagram in Figure 3.1, from [8], shows the typical states of a thread. We argue that the thread object can better be represented by a stated thread object because a stated thread object has a default feature to maintain its states. In the case of conventional (non-stated) thread objects, the operating system and conventional object oriented programming language need to keep track of thread states by themselves. Typically a 'Windows' thread is in one of six states, as in Figure 3.1. The states of a thread object are described in Table 3.1.

State	Description
ready	May be scheduled for execution. The microkernel dispatcher keeps track of all ready threads and schedules them in priority order.
standby	A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.
running	Once the microkernel performs a thread or process switch, the standby thread enters the running state and begins execution and continues execution until it is preempted, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the ready state.
waiting	A thread enters the waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.
transition	A thread enters this state after waiting if it is ready to run but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state
terminated	A thread can be terminated by itself, by another thread, or when its parent process terminates. Once housekeeping chores are completed, the thread is removed from the system, or it may be retained by the executive for future re-initialization.

Table 3.1 'Windows NT Thread' Stated Object States, from [8]

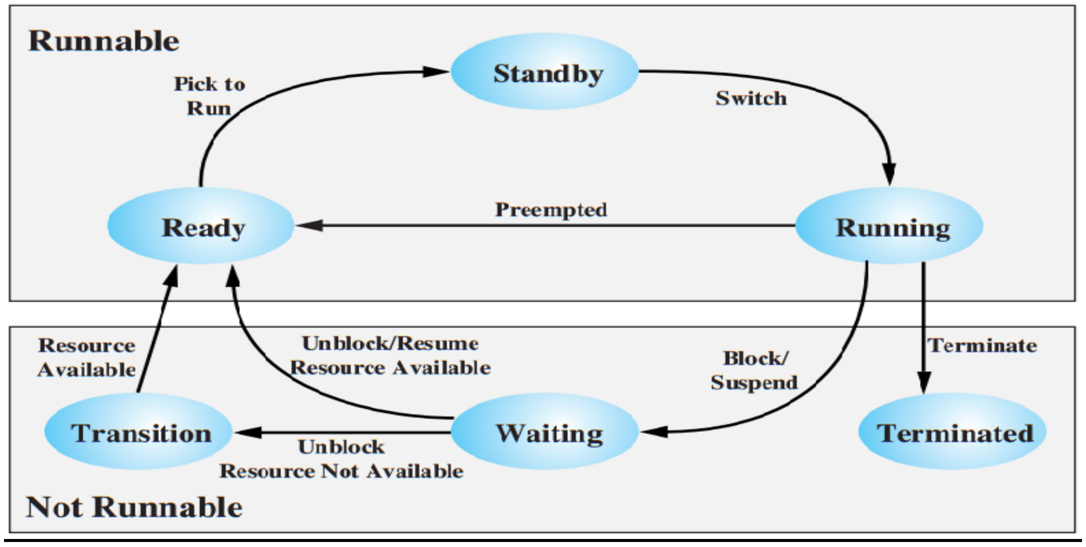


Figure 3.1 ‘Windows NT Thread’ Stated Object Model, from [8]

In this case study, we illustrate how a ‘digital circuit simulation’ application has a distinct functional component that redraws the screen because of scroll bar positioning or a screen resize request from the user. This functional component has a ‘redraw’ function that keeps a record of the screen data i.e. the logic symbols and gates currently drawn by the user. The ‘redraw’ function gets the current screen size and scroll bar position and redraws the screen data accordingly. If for some reason drawing the screen data is time consuming, then the redraw operation calls the ‘sleep’ method to temporarily suspend the drawing operation so that other threads can continue their task.

The ‘digital circuit simulation’ application creates a separate thread, i.e. a redraw thread, and invokes the redraw method through the redraw thread. The operating system starts the redraw thread from its *ready* state and if no other thread is available, then the redraw thread is picked up and transitioned to the *standby* state. If the processor is available then the redraw thread is transitioned from the *standby* state to the *running* state right away and starts drawing the screen. While drawing the current screen of the simulation, the redraw function suspends the thread for one hundred milliseconds. After the suspension has timed out, the redraw thread is re-transitioned to the *ready* state by the operating system. If no other thread is in the queue, then this thread is transitioned to the *standby* state by the operating system and is then brought to the *running* state. As soon as the redraw thread is transitioned to the *running* state, the drawing operation continues from where it was suspended.

3.2 Electronic Workbench (EWB) Stated Process

Usually an operating system process switches states when the operating system transfers the control of execution from one process to another process. Switching of process states is required so that a process cannot monopolize the processor. Due to this requirement to switch among states, the usual process object can be better represented by a stated process object because the stated object inherently maintains the states and transitions of the object. Representing a process by a stated process object also helps to reduce the responsibility of the operating system to track the data structures used for keeping the state information of the process. This is because the stated process object keeps track of its state information by default. According to one of the adopted standards [8], a process typically realizes seven states. These well-defined seven process states, given in the table below, can be used as the seven states of the stated process object.

States	Description
New	A process that has just been created but has not yet been admitted to the pool of executable processes by the operating system.
ready	A process is in main memory and is prepared to execute when given the opportunity.
ready/suspend	A process is in secondary memory but is available for execution as soon as it is loaded into main memory.
running	A process that is currently being executed.
blocked/suspend	A process is in secondary memory and awaiting an event.
blocked	A process is in main memory and awaiting an event to occur.
Exit	A process that has been released from the pool of executable processes, either because it halted or because it aborted for some reason.

Table 3.2 ‘Process’ Stated Object States, from [8]

In the process state diagram, Figure 3.2, from [8], a transition arc from each state to the *exit* state is implicit but not shown for clarity. However, it is possible that from any state the process can be transitioned to the *exit* state either because the user decides to close the process or the operating system forces the process to exit.

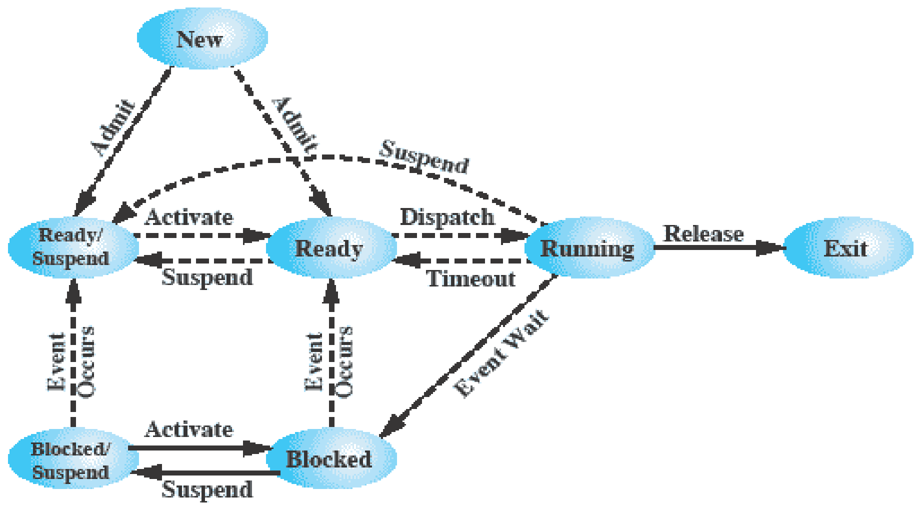


Figure 3.2 ‘Process’ Stated Object Model, from [8]

Let us assume that the operating system receives a user request to initiate an Electronic Workbench (EWB) software application. The operating system creates the EWB stated process object, in the *new* state, so that it can load the EWB application in the EWB process space. The EWB application is supposed to be loaded in a single thread. Immediately after creating the stated process object, the operating system invokes the ‘admit’ function that transitions the EWB process object to the *ready* state and the EWB application is loaded in the EWB process space in main memory. If the operating system is already running enough processes, the EWB stated process is kept in the *ready* state for a period of time and the user has to wait. After waiting, the operating system invokes the ‘dispatch’ function to transition the EWB process to the *running* state and the EWB application starts execution. The EWB application is now running and the user can interact and use the EWB application as long as its EWB process is in the *running* state.

When the user requests the EWB application to open an existing workbench file, the EWB switches to ‘file_open’ mode to wait for the user to select and issue the ‘open’ command for the selected file. Now the operating system realizes that the EWB application has to wait for an event to occur. Therefore, the operating system transitions the EWB process to its *blocked* state. When the user has selected the file to open and the ‘open’ method is invoked then the operating system catches the ‘file_open’ event along with the selected file name. Furthermore, the operating system issues the ‘event_occurred’ command to transition the EWB process object from its *blocked* state to the *ready* state. If there is no other ready process in the *ready* state queue then operating system can transition the EWB process object to the *running* state immediately and the EWB application can proceed to load the selected file to open in the EWB application.

If the EWB process is in the *running* state then the user can use the EWB application. If the user selects the ‘close’ command for the EWB application, then the operating system transitions the EWB process to the *exit* state and the EWB application is unloaded from main memory, i.e. the EWB application is unloaded from the EWB process space. Since the screen handle is not readily available, the redraw thread is transitioned to its *transition* state and the EWB process stays in its *blocked* state. As soon as the screen handle is released by the other process, the operating system triggers the ‘screen_available’ event to the EWB process while passing along the screen handle. Now the EWB window transitions to its *ready* state, triggers the ‘resource_ available’ event to the redraw thread that transitions the redraw thread to its *ready* state. According to the scheduling of the operating system, the redraw thread is transitioned to its *standby* state where it is eventually switched back to its *running* state and continues the screen redraw operation from where it was suspended.

3.3 Electronic Workbench Stated Process With a Stated Screen Redraw Thread

In this case study we illustrate that the software objects of Case Study 3.1 and 3.2 can be collectively used as stated objects. The electronic workbench (EWB) application creates a new thread and runs the screen redraw component on that ‘redraw’ thread. If a process is in the *running* state, it may spawn a new thread to run within the same process. In this case, the stated ‘process’ object is composed of a stated ‘thread’ object or a stated ‘thread’ object is a component of the stated ‘process’ object. The ‘process’ is referred to as the parent process and the spawned ‘thread’ is referred to as the child thread. The parent ‘process’ and child ‘thread’ need to communicate and cooperate with each other. Both the ‘process’ and the ‘thread’ object will exist in their respective states as mentioned in the state diagrams in Figure 3.1 and Figure 3.2 respectively. Note that the possible state transitions of a parent ‘process’ and child ‘thread’ can be dependent on each other. For instance, if a parent ‘process’ goes to the *exit* state then the operating system can force the child ‘thread’ to transition to the *exit* state. It is also possible that the transition of the child ‘thread’ to its *blocked* state leads the operating system to transition the parent ‘process’ to its *blocked* state. The operating system allows a child ‘thread’ to be instantiated in its *new* state only when the parent ‘process’ is in the *running* state. In some cases, it may also be mandatory that a child thread can be in its *running* state only if the parent ‘process’ is in its *running* state. Such an interdependence of states can be defined by the state mapping feature of the stated object, in Section 2.8. Once the programmer has defined the interdependence between the states of ‘process’ and ‘thread’ objects, the state controlled object oriented language will track any violation of the state interdependence.

In this example, we illustrate the interdependence between the EWB process and its ‘screen redraw’ thread. The EWB application creates a separate thread and runs the screen redraw component on that redraw thread. The EWB process is created in its *new* state by the operating system due to the user’s request to initiate the EWB application. The operating system transitions the process to the *ready* state from its *new* state and then transitions the EWB process to the *running* state. When in its *running* state, the EWB process initiates a screen redraw thread in its *ready* typestate. The redraw thread will be immediately transitioned to the *standby* state by the operating system if there are no other threads to run. As the EWB process is still in its *running* state and the processor is not busy executing any other thread, the screen redraw thread is transitioned to the *running* state because the user has resized the EWB window. While redrawing the EWB screen with the redraw thread, the operating system can take the ‘screen handle’ from the redraw thread and assign it to another process. Since the screen handle, i.e. an I/O resource, is not available for the redraw thread, the operating system transitions the redraw thread to its *waiting* state. Transitioning the redraw thread to the *waiting* state also leads the operating system to transition the EWB process to its *blocked* state. At this stage, the EWB application window is disabled and the mouse cursor turns to its *busy* state as long as it is over the EWB application window.

3.4 Lexical Analyzer

Lexical analysis is a well-known application of finite state automata. A lexical analyzer reads source code character by character, separates the lexemes, and generates tokens. The lexical analyzer recognizes a certain number of classes (categories) and generates the tokens corresponding to each of these classes. Whenever the lexical analyzer finds a lexeme, it transitions to a specific state corresponding to that lexeme so that a corresponding token can be generated from that specific state. Most of the states given in the table below generate a corresponding token, as is obvious from the state name. A state diagram for a lexical analyzer is given in Figure 3.3. While scanning the characters, the lexical analyzer may encounter separator characters, e.g. space, tab, new line, punctuation, which determine that the characters read so far make up a lexeme. The lexical analyzer then begins scanning from its *initial* state. While scanning source code characters from any state of the lexical analyzer, an invalid or erroneous character may also be found that also serves as a separator. In the case of an invalid character, an ‘error’ token is generated for that lexeme and control is transitioned back to the *initial* state to restart scanning from the next character to find the next lexeme. After scanning a single character from the *initial* state, the lexical analyzer either knows to exactly what class the lexeme belongs or to what potential class that lexeme may belong. In both cases the lexical analyzer transitions its control to the corresponding state. Below, we describe below a few possible transitions of the lexical analyzer from its *initial* state according to a single character scanned.

- If the character read is a punctuation symbol, e.g. a semicolon, then the lexical analyzer will transition to its *punctuator* state. From the *punctuator* state, a token for punctuation is generated and control is transitioned back to the *initial* state to restart scanning from the next character to determine the next lexeme.

- If the character read is a letter, e.g. ‘a’, then the lexical analyzer will transition to its *identifier* state because a letter means that the lexeme is potentially an identifier or a keyword. In the *identifier* state, subsequent characters are being read until a separator is detected so that a complete lexeme is found. Upon detection of a complete lexeme in the *identifier* state, the *identifier* state will decide whether or not the lexeme belongs to a keyword. If the lexeme is a keyword then a 'token' for the keyword will be generated from the *identifier* state. Otherwise a token for the identifier will be generated and control will be transferred back to the *initial* state to restart scanning to identify the next lexeme.

- If the character read is a numeric literal symbol, e.g. a digit six "6", then the lexical analyzer will transition to its *literal* state because the complete lexeme is potentially a numeric literal. From the *literal* state, scanning continues to read the later digits of the number. Once a complete numeric lexeme is found, a literal token is generated from the *literal* state and control is transitioned back to the *initial* state to restart scanning to identify the next lexeme.

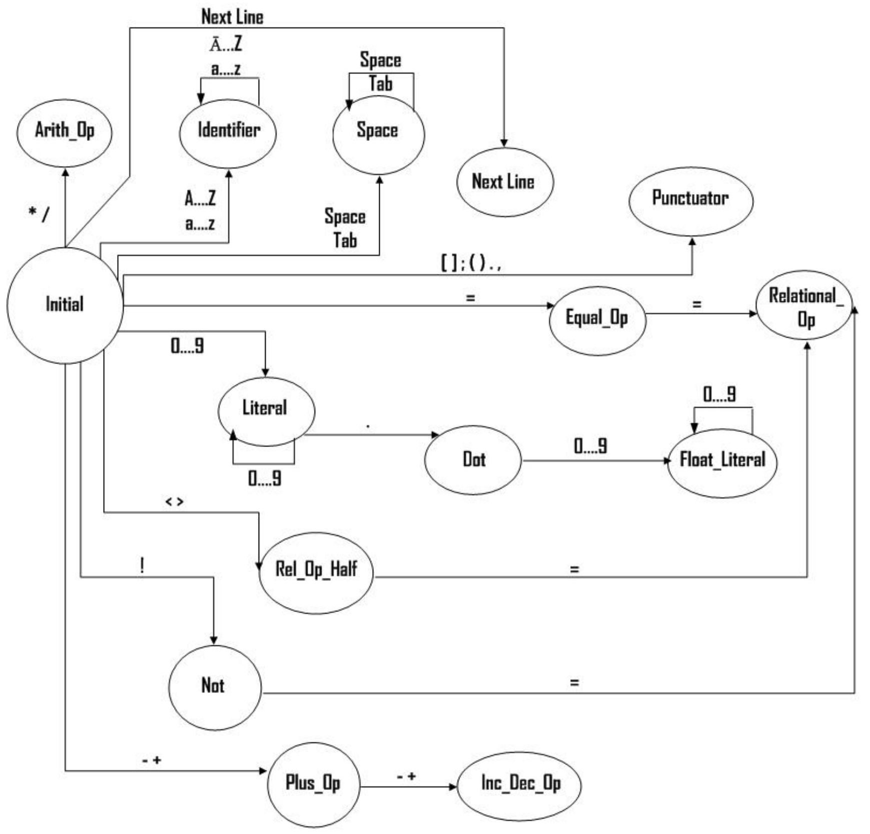


Figure 3.3. 'Lexical Analyzer' Stated Object Model

The states given in the following table are named after the typical token class names. A corresponding token is generated as soon as a complete lexeme is found in a particular state.

Therefore, we argue that we can better implement the lexical analyzer as a stated object that exists among any of its states.

State	Description
initial	Upon reading a single character, control of lexical analyzer is transitioned to either of the following states.
identifier	This state determines the lexeme by continuing to read the characters. When a lexeme is found, then this state searches for whether the lexeme belongs to a ‘keyword’, then a keyword token is generated, otherwise an ‘identifier’ token is generated.
whiteSpace	This state continues reading the space or tab characters unless another character is found. As long as any other character is found, control is switched back to the *initial* state. No token for space or tab etc. is generated.
punctuators	This state generates a punctuator token and pass control back to the *initial* state.
eqOp	The *initial* state transitions to this state if an equals symbol (=) is found. This state scans the next character. If the next character is not a valid character then the ‘equal operator’ token is generated.
relOp	Control is switched to this state if a relation operator is found. This state generates a token for the relational operator.
relOp_half	Control is switched to this state if a half part of the relational operator is read. This state reads the next half part of the relational operator and passes control to the *relOp* state.
not	This state recognizes the not (!) operator as the first part of not equal to (!=) operator and looks for the equal operator.
plusOp	This state generates a token for the ‘plus’ operator (+).
inc_dec_Op	This state generates a token for the ‘increment decrement’ operator. E.g. ++ or -- .
aritOp	This state generates a token for the ‘arithmetic’ operator.
Lit	This state generates a token for the ‘numeric’ literal.
Dot	This state reads the character and checks if a dot symbol is found.
lit_float	This state recognizes the decimal part of a floating literal and generates a token for the floating numeric literal.
nxt_Line	This state generates no token and simply decides that a next line character is found and transitions back to the *initial* state upon occurrence of the next character.

Table 3.3 ‘Lexical Analyzer’ Stated Object States

3.5 A Simple Two Tank Pumping System

In this case study, we illustrate that a two tank pumping system is a finite state model and therefore a software application for such a system can be directly written and better represented by stated objects. This example was previously used to illustrate the application of model checking by representing this system in a model checking language [7]. We argue, in Chapter 8, that if such applications are already coded as stated objects then the application could be transformed to a model checking language directly and conveniently. This is because the stated object oriented code itself exhibits the possible states of the system that can be used directly to make a model. Thus, stated objects increase readability for model checking as well.

This is a case study of a simple pumping control system. The water is transferred by Pump P from a source Tank A to another sink Tank B. Each tank has two level meters, one to detect whether its level is empty and the other to detect whether its level is full. The tank level is at the *ok* state if the water level is above the empty meter but below the full meter. Initially, both tanks are in their *empty* state but water can be supplied to the source Tank A from an external pipe. The pump is to be transitioned to its *on* state as soon as the water level in Tank A reaches its *ok* state from the *empty* state, provided that Tank B is not in its *full* state. The pump remains in its *on* state as long as Tank A is not in its *empty* state and as long as Tank B is not in its *full state*. The pump is to be transitioned to its *off* state as soon as either Tank A transitions to its *empty* state or Tank B transitions to its *full* state. The system should not attempt to transition the pump to its *off* (*on*) state if it is already in its *off* (*on*) state. While this example may appear trivial, it may easily extend to a controller for a complex network of pumps and pipes to control multiple sources and sink tanks, such as those in water treatment facilities or chemical production plants. The possible states of the source tank A as a stated object are described below.

State	Description
empty	If water level is equal to or less than Empty marker.
Ok	If water level is greater than Empty marker and less than Full marker.
Full	If water level is less than or equal to Full marker.

Table 3.4 ‘Tank A’ Stated Object States

The possible states of sink Tank B as a stated object are as below.

State	Description
empty	If water level is equal to or less than Empty marker.
Ok	If water level is greater than Empty marker and less than Full marker.
Full	If water level is less than or equal to Full marker.

Table 3.5 ‘Tank B’ Stated Object States

The possible states of the pump as a stated object are as below.

State	Description
On	If pump is switched on.
Off	If pump is switched off.

Table 3.6 ‘Pump’ Stated Object States

3.6 CIP System

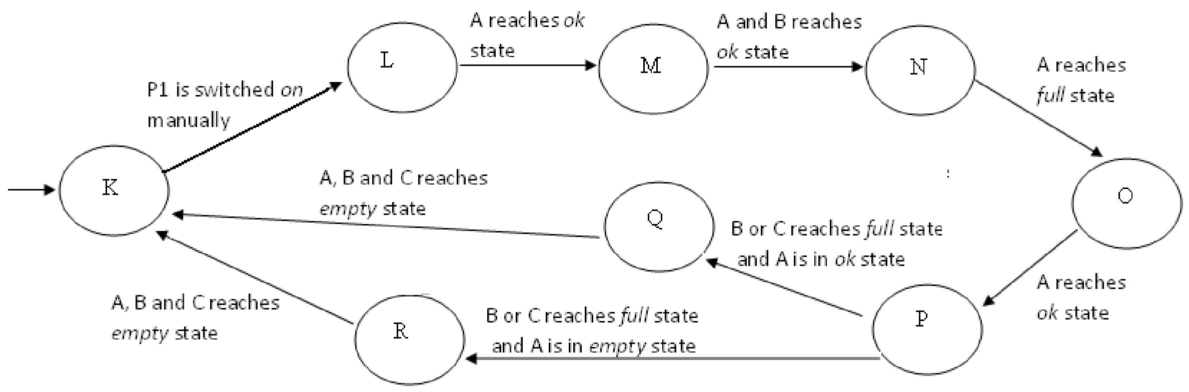
A cleaning in place (CIP) pumping system can be modeled by a finite state automaton (Figure 3.4);therefore, a software application for such a system can be directly written and better represented by stated objects. The transitions in Figure 3.4 represent the events of the CIP system. Upon any occurrence of an event, the system triggers the corresponding transition. We argue, in Chapter 8, that if such applications are already coded as stated objects then the application could be transformed to a model checking language directly and conveniently.

This case study is a customized case of a typical CIP system that is often used in the chemical industry. The system has three pumps (P1, P2 and P3) and three tanks (A, B and C). Each tank has two level meters, one to detect whether its level is empty and the other to detect whether its level is full. The tank level is at the *ok* state if the solution level is above the empty meter but below the full meter. Each pump can be transitioned to its *on* state or *off* state. Initially, all three tanks are in their *empty* state and all pumps are in their *off* state. Pump P1 pumps the solution to Tank A, P2 pumps the solution from Tank A to B and C. Finally P3 pumps the solution out of the three Tanks A, B and C to vacate the tanks and output the CIP return. A combination of different states of these Tanks and Pumps will make up the states of the whole system.

Pump P1 is transitioned to its *on* state manually. As soon as the solution level in Tank A reaches its *ok* state from the *empty* state, then pump P2 transitions to its *on* state and pumps the solution from Tank A to Tanks B and C, provided that Tank B and C are not in their *full* state. Pump P2 remains in its *on* state as long as Tank A is not in its *empty* state and Tank B or C are not in their *full* state. As soon as Tank A reaches its *full* state, pump P1 transitions to its *off* state. Pump P2 transitions to its *off* state as soon as Tank B or C transitions to its *full* state. As soon as Tank B or C transitions to its *full* state then pump P3 transistions to its *on* state. The system should not attempt to transition a pump to its *off* (*on*) state if it is already in its *off* (*on*) state. We define the overall states of the system in terms of combinations of different states of the Pumps and Tanks as below.

State	Description
K	P1=*off*, P2=*off*, P3=*off*, A=*empty*, B=*empty*, C=*empty*
L	P1=*on*, P2=*off*, P3=*off*, A=*empty*, B=*empty*, C=*empty*
M	P1=*on*, P2=*on*, P3=*off*, A=*ok*, B=*empty*, C=*empty*
N	P1=*on*, P2=*on*, P3=*off*, A=*ok*, B= *ok*, C= *ok*
O	P1=*off*, P2=*on*, P3=*off*, A=*full*, B=*ok*, C=*ok*
P	P1=*off*, P2=*on*, P3=*off*, A=*ok*, B=*ok*, C=*ok*
Q	P1=*off*, P2=*off*, P3=*off*, A=*ok*, B=*full*, C=*full*
R	P1=*off*, P2=*off*, P3=*off*, A=*empty*, B=*full*, C=*full*

Table 3.7 ‘CIP System’ Stated Object States



K
L
M
N
O
P
Q
R
P1 is switched *on* manually
A reaches *ok* state
A and B reaches *ok* state
A reaches *full* state
A reaches *ok* state
B or C reaches *full* state and A is in *ok* state
B or C reaches *full* state and A is in *empty* state
A, B and C reaches *empty* state
A, B and C reaches *empty* state

Figure 3.4 'CIP System' Stated Object Model

3.7 Digital Counter Composed of Flip-Flops

In this case study, we show that a digital counter can be made up of flip flops [9]. Each flip-flop has two states: enabled (ebl) or disabled (dbl). Therefore, flip flops can be viewed as stated objects. Two flip-flops can exhibit a combination of states as ebl-ebl, dbl-dbl, ebl-dbl and dbl-ebl. These combination states can be viewed as the states of a counter stated object that is composed of these two flip-flops. Therefore, a digital counter with *n* flip-flops is a stated object having 2^n states as it is composed of a number of flip-flop stated objects.

3.8 Master Detail Data Entry and Navigation Form

In this case study, we show that a typical 'data entry and navigation form' for a database application can have many display states and can be coded with a stated 'Form' object. Both the GUI and the underlying mechanism of this 'Form' object demonstrate the FSA functionality. To display data from a one-to-many underlying table of a database, we often use a typical 'Form' which has two parts. The first part displays data from one row of the master table and the second part of the 'Form' display all the detail data rows corresponding to the row shown in the first part. In other words, the first part displays one row from the master table and the second part could possibly display many rows from the detail table corresponding to the one row displayed in the first part of the 'Form'. Initially when the

‘Form’ is loaded then it is possible that there is no row in the table to display, so the ‘Form’ will be displayed without any data, i.e. in the *empty_form* state. If the ‘Form’ has at least one row in the master table then the ‘Form’ will display the first row and will be in the *display_first_row* state.

The ‘Form’ has two kinds of buttons: navigation and data buttons. Using the navigation buttons, the user can move to other records. Using the data buttons, the user can modify the data of rows. The buttons can be enabled or disabled depending on the current state of the ‘Form’. Using the NEXT navigation buttons on the ‘Form’, the user can invoke the ‘next’ method of the ‘Form’ to display the next row and the ‘Form’ will transition to the *display_intermediate_rows* state from the *display_first_row* state. The ‘next’ method of the *display_first_row* state will transition the ‘Form’ to the *display_intermediate_rows* state if there are more than two rows in the underlying master table. The ‘next’ method of the *display_first_row* state will transition the ‘Form’ to the *display_last_row* state, if there are only two rows in the underlying master table. Similarly, the user can also invoke the ‘edit’ or ‘addnew’ methods and the ‘Form’ will transition to the corresponding display state. From the *edit* or *addnew* state, the user may attempt to invoke either the ‘save’ method to save the data or the ‘cancel’ method to exit the ‘edit’ or ‘addnew’ method. The ‘cancel’ method transitions the ‘Form’ to the same previous state from where the ‘edit’ or ‘addnew’ method was invoked. This shows that the stated object may also act as a mechanism to track its last state or a history of its previous state transitions. The table below shows all the states of the ‘Master Detail Data Entry and Navigation Form’ as a stated object.

State	Description
empty_form	There is no row in the table to display therefore the 'Form' displays all empty fields and all navigation and data buttons are disabled except the 'AddNew' button. All fields are disabled.
display_first_row	The first row from the table is shown and the navigation and data buttons are enabled. All fields are disabled.
display_intermediate_rows	The intermediate rows from the table are shown. The navigation and data buttons are disabled. All fields are disabled.
display_last_row	The last row is shown. All buttons are enabled except the Next button. All fields are disabled.
Edit	All fields are enabled. All buttons are disabled except the Cancel and Save buttons.
Addnew	All fields are enabled. All buttons are disabled except the Cancel and Save buttons.

Table 3.8 'Master Detail Data Entry and Navigation Form' Stated Object States

3.9 Elevator

We illustrate from [11] a state-machine model of an elevator. A software application may need to simulate an elevator as a stated object. For simplicity in this case study, the elevator does not actually change floors. The only possible input to the elevator is to open or close its doors, or do nothing. So, the possible inputs to this machine could be to invoke the methods 'command_open', 'command_close' and 'no_command'.

The doors of an elevator do not open or close instantaneously, so we model the elevator to assume four possible states as *opened, closing, closed,* and *opening* states. These states correspond to the doors being fully open, starting to close, being fully closed, and starting to open, respectively.

In the *closed* state, if the elevator is commanded to open, it goes into the *opening* state. In the *opening* state, the 'command_open' or 'no_command' inputs cause a transition to the *opened* state. In the *opening* state, the 'command_close' input causes a transition to the *closing* state, which is displayed in the table below.

State	Description
opened	The door has been completely opened.
closing	The door is closing but not completely closed.
closed	The door has been completely closed.
opening	The door is opening but not completely opened.

Table 3.9 'Elevator' Stated Object states, from [11]

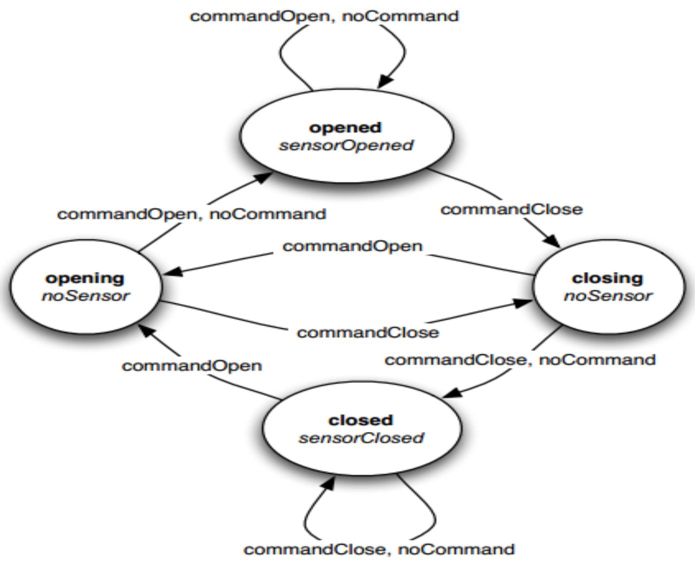


Figure 3.5 'Elevator' Stated Object Model, from [11]

3.10 Bank Account System

We now study an online banking application that performs the job of printing an account statement. We suppose that the application is comprised of two basic components (1) A 'print manager web service' and (2) A 'managed printing web service'. The 'print manager web service' monitors relevant system parameters to determine the mode of printing. The 'managed printing web service' actually sends the printing job to the printer.

The banking application provides an online interface to clients so that they can request to print their account statement. Clients of the application will connect to the managed printing service through a web interface. The 'managed printing service' as a stated object either streams all print requests only to a main printer, or distributes the print requests between a main printer and a support printer. Therefore, printing is performed either in the *streaming* state or in the *distribution* state. The ‘print manager web service’ periodically analyzes the relevant information, e.g. the number of clients connected, and accordingly triggers any change of state to the ‘managed printing web service’.

The state to be assumed by the ‘managed printing service’ is determined based on the number of connected clients. The ‘print manager service’ keeps track of the total number of connected clients. If the number of clients increases beyond a set threshold, the print manager invokes an event to the managed printer to transition it to the *distribution* state. Otherwise it transitions or keeps it in the *streaming* state. Therefore, the ‘manager service’ controls the state transition of the ‘managed service’ and the ‘managed service’ continues the print job accordingly.

State	Description
streaming	All print requests are sent to a single printer.
distribution	All print requests are distributed among the available printers.

Table 3.10 ‘Managed Printing Web Service’ Stated Object States

3.11 Queue

In this case study, dynamic adaptation features of the functional and non-functional aspects of state controlled object oriented language are illustrated. When we use the term “client-side code”, we mean the code fragment that creates an object instance of the stated object. In this example, we use the notion of “abstract class” from OOP. The objects of an “abstract class” cannot be instantiated but can be declared. Object instances of its compatible classes can be assigned to the declared reference of the “abstract class”. Let us consider a `Queue` abstract class. The client-side code can declare an abstract stated object `Q` of `Queue` as below.

```
Queue Q;
```

An `ArrQueue` is an array-based stated queue object with *empty* and *nonempty* states. An array in the *nonempty* state keeps state data, i.e. the non-functional aspect, of `ArrQueue`, while `put`, `get` and `isfull` are the methods, i.e. the functional interface, of `ArrQueue`.

A `HashQueue` is a hash table-based stated queue object with *empty* and *nonempty* states. A hash table in the *nonempty* state keeps the state data for the `HashQueue` while `put`, `get` and `isfull` form the functional interface for `HashQueue`.

In the client-side code, an object of `ArrQueue` can be instantiated and assigned to the `Q` object as below:

```
Q=new ArrQueue();
```

An object of `HashQueue` can be instantiated as below:

```
HQ=new HashQueue();
```

Suppose `Q` is currently assigned an instance of `ArrQueue`. On the client side, this `Q` is populated. When the client realizes that the array-based queue object `Q` is not sufficient to allocate more data then, at first, the data in the 'array' of `ArrQueue` can be passed on to the `HQ` using the following method:

```
HQ.updatestate(Q.nonempty);
```

The above `updatestate` custom method has to be written by the programmer in the `HashQueue` stated class to transfer the data of 'array' to the 'hash table'. Secondly the client could dynamically replace the *nonempty* state of the `ArrQueue` object with the *nonempty* state of the `HashQueue` object. In SCOOP, this replacement is as simple as the following statement:

```
Q.nonempty=HQ.nonempty;
```

The statement above transforms the *nonempty* state of the array-based `Q` stated object to the *nonempty* state of the hash table-based `HQ` stated object. Note that the *nonempty* state of `ArrQueue` has an array-based implementation while the *nonempty* state of `HashQueue` has a hash table-based implementation. We elaborate in Chapter 7 on how SCOOP implements this dynamic typestate replacement.

The advantage is that only the partial behavior of the `Q` object is adapted at run time and the rest of the states, data and behavior associated with each state remain intact. In conventional object oriented programming, if we replace an array-based queue with another hash table-based queue then the entire array-based object instance is overwritten. All of its state data and behavior associated with any state is replaced. In the case of a stated object, after dynamically adapting the behavior of one state, the behavior particular to the rest of the states of the stated object remains as it was and may still be used as before. In the table below, we show the possible states of the `Queue` stated object.

State	Description
empty	The Queue is empty.
nonempty	The Queue is not empty i.e. has at least one data element.

Table 3.11 ‘`Queue`’ Stated Object States

3.12 Iterator

In this case study, we show the binding of the external states of a stated object with its internal state invariant data. The complete code sample of a stated iterator object is given in Figure 6.1. The iterator is a programming construct that performs custom iteration over a data structure. The stated iterator object of Figure 6.1 is taken from standard OOP ‘iterator’ code from [6] and transformed into SCOOP code. The iterator definition encapsulates the logic to iterate the data structure which may be simple or complex depending on the data structure. The iterator can also be considered to maintain its abstract states during its traversal. For instance, the iterator may be in the *initial* state when it has not started iteration. The iterator may be in the *traversal* state during its iteration and in the *end* state when iteration is completed. The iterator can have many internal states, for the following reasons:

- The iterator may encapsulate a complex data structure, e.g. a two dimensional array which has a number of states during its traversal.
- The iterator may encapsulate another iterator which has a number of states.

We introduce three states for the iterator as mentioned above, i.e. *initial*, *traversal* and *end*. The `step()` function in the sample code of Figure 6.1 iterates the iterator, `HIter`, a step forward. When the current state of the iterator `HIter` is set to *traversal*, then this state serves as an external and formal representation of its internal state. Internally, this state is represented by its state invariant data members. For the *traversal* state, the state invariant data members are `i, j` and `ht→buckc`. The *traversal* state holds as long as the following rule is true:

```
if ( j==0 AND i < ht→buckc)
```

Similarly, we have another rule for the *end* state of iterator `HIter`:

```
if ( I == ht→buckc)
```

And for the *initial* state, the rule is:

```
if   ( j== -1 AND i < ht→buckc)
```

Since the `step()` function iterates an iterator a step onward, it makes sense to allow the `step()` function in both the *initial* and *traversal* states because the iterator can iterate next in each of these states. Making the `step()` function available in both states illustrates state-based polymorphism, as mentioned in Section 2.5. As soon as the iterator transitions to the *end* state, forward iteration becomes illegal.

State	Description
initial	The iterator is at the first record of data.
traversal	The iterator is at any part of the record of data except first or last record.
End	The iterator is at the last record of data.

Table 3.12 ‘Iterator’ Stated Object States

3.13 Printer

In this case study, we illustrate a stated printer object. It is assumed that the printer can be in either the *fullcharged* or in the *halfcharged* state. Printing can be performed in either of the states, but with different functionality. The *fullcharged* state of the printer may abstract over its internal invariant data, such as ‘cartridge ink level’ and ‘cartridge last replaced’. Values for each of these data members collectively or individually may cause the state of the printer to transition from the *fullcharged* to the *halfcharged* state, if either ‘cartridge ink level’ goes below 50% or ‘cartridge last replaced’ exceeds 20 days. The states of the stated printer object are presented in the table below.

State	Description
fullcharged	In this state the printer will perform dark printing.
halfcharged	In this state the printer will perform dim printing.

Table 3.13 ‘Printer’ Stated Object States

Moreover, the stated object printer can be composed of a stated object cartridge. In this case, the states of the printer object can be mapped to the corresponding states of the cartridge object. Intuitively, a transition in the cartridge state may trigger the corresponding transition in the state of the printer object and vice versa. Such a mapping of a stated object with its composed stated object has not yet been studied.

3.14 File

In this case study of stated objects from [18], we argue that only the partial behavior particular to a state of a stated object is replaceable, rather than replacing a complete object at run time. For instance, an ‘ImageFile’ object may need to adapt a new ‘read’ behavior at run time so that it can read an image from a newly connected device that was not known at compile time, because its mechanism to read an image from a newly connected device is different. Since ‘read’ is a behavior particular to the *openfile* state of the ‘ImageFile’ object, the behavior associated only with the *openfile* state of the ImageFile needs to be replaced at run time without compiling the original ‘ImageFile’ object again.

Let us suppose there is a ‘File’ object, written by a programmer. The code of the ‘File’ object is compiled in a program, ‘ProgFile’. The programmer intends to enable ‘ProgFile’ such that if ‘ProgFile’ has compiled and run then at run time the ‘ProgFile’ can input a new behavior for the ‘read’ method of the *openfile* state of the ‘File’ object. Therefore, the programmer also writes the client code for a ‘File’ object in the ‘ProgFile’ program so that ‘ProgFile’ can input a new ‘read()’ method for the *openfile* state of the ‘File’ objects as desired. ‘ProgFile’ is then compiled with the code of ‘File’ object and its client. The same programmer, or another programmer, writes a separate program ‘ProgNewFile’ which has either a new ‘File’ object or a new *openfile* state that encapsulates a different ‘read()’ behavior for the *openfile* state of the ‘File’ object. The code of the new ‘File’ object or a new *openfile* state is compiled and placed into a separate library.

When ‘ProgFile’ is run, it can input a new *openfile* behavior from ‘ProgNewFile’ and replace it with the original *openfile* state. Therefore, the new ‘read()’ behavior (method) goes into effect. We illustrate the states of a stated ‘File’ object, in the table below.

State	Description
openfile	The file is open but not at the last record.
closefile	The file is closed.
Eof	The file is open but at the last record.

Table 3.14 ‘File’ Stated Object States

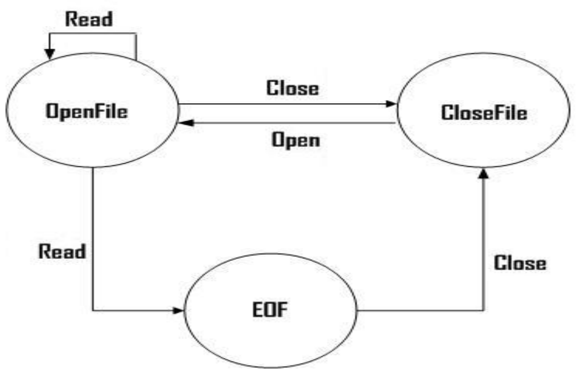


Figure 3.6 ‘File’ Stated Object Model

Chapter 4

4 Background and Related Work

Finite state automata (FSA) have been a significant tool for modeling diverse kinds of systems. State charts [15] are an extended version of FSA which can model more sophisticated aspects of systems such as nested states. State charts have also been effectively used in modeling the design of software systems [15]. Due to their significance, software engineers have attempted to use state charts directly in the implementation of software in addition to simply modeling the design of software systems. One of the attempts to leverage state charts in the implementation of software is the state design pattern [16]. Conventionally the design and implementation of control-related software applications has been widely modeled by state charts [15]. The significance of state charts in software modeling can be seen by the fact that the design of business-related applications is increasingly modeled by workflow [46]. “Workflow” is a concept similar to state charts that is primarily used to model the design of business-related applications. Researchers have attempted to make use of workflow directly in the implementation of business-related software [47] so that state can be directly encoded by the programming language.

The main drawback preventing software engineers from exploiting these tools for direct use in programming is the lack of explicit support for typestate in programming languages.

4.1 Typestate

The term “typestate” was initially used from the perspective of imperative programming [23] for software reliability and software validation using primitive data types. Later, the concept of typestate was incorporated into the object concept of OOP [17 - 22, 24, 28].

Typestate related behavior in standard OOP is analyzed by analysis tools [4, 5, 14, 26, 60] that attempt to capture the typestates of an object separately from an object oriented program. This separately captured typestate is called typestate property. The object oriented program is then analyzed to check whether the program violates that typestate property. In contrast, SCOOP allows ‘typestate’ as an explicitly defined integral part of an object that can be accessed directly and analyzed as proposed by Sheng [17], Aldrich [18, 24] and Fähndrich [20].

Typestate-Oriented programming, abbreviated as TSOP, [18, 24] is a previously proposed programming paradigm similar to SCOOP. TSOP requires modeling software in terms of the state of an object rather than in terms of the object itself. The strength of TSOP is that it represents the state of an object as a first class language concept and as a unit entity of programming which helps to implement the state transition of the object by the TSOP language. But this strength exists at the cost of not representing the object as a first class entity as in conventional OOP. Our proposed SCOOP language, described in Chapter 5, adds the state feature to objects similar to the approach proposed by Sheng [17], Aldrich [18, 28], Fähndrich [20] and Sterkin [21], but in addition, we propose both the object as well as its typestates as first class language entities.

Our approach to static and dynamic typestate checking does not require any additional annotation like “access permission” or “state guarantee”, as proposed by Aldrich [22, 24, 28], nor does it require annotations like “key”, as proposed by Fahndrich [27]. We refer to our approach as “user friendly” because it does not require the programmer to learn and use any additional annotations except typestate annotations. Our approach does not constrain typestate transitioning in the presence of aliasing, unlike the approach of Fähndrich [20], where the typestate of “maybe aliased” objects cannot transition. Fähndrich [20] proposes transitioning of the typestate of an object only if the object is not aliased. In our setting, all objects are “maybe aliased”. Our typestate aware stated type system tracks the current typestate transitions of the object and reflects it to all of its aliases.

We introduce a transition function `trans()` as used in the iterator example, in Section 6.4.1. This function has not yet been proposed by any of the earlier approaches that are similar to SCOOP. Our proposed `trans()` function, illustrated in Section 6.4.1, introduces an additional capability for the programmer to perform a valid transition on an object from outside of the object at any point in time during the lifetime of the object. SCOOP, with the help of the `trans()` function, enables an object to act like a statechart as proposed by Sterkin [21]. Sheng [17], Aldrich [18, 28] and Fähndrich [20] proposed state transitioning of an object only by method invocation. Our proposed additional `trans()` method can be used for direct transition of an object to a valid typestate, as illustrated in the `step()` function of the iterator example in 6.4.1. Sterkin [21] proposed an approach similar to our proposed `trans()` function but in his approach the overall language design lacks object oriented features.

4.2 Typestate Extension and Subclassing

SCOOP also supports typestate extension along with subclassing (behavioral subtyping) of stated objects. Fähndrich [20] allowed subclassing of stated objects but typestate extension is not supported. Aldrich [24] allowed typestate extension but subclassing is not defined when programming with typestate. Aldrich's [28] work allowed typestate extension with behavioral subtyping but neither an implementation architecture for typestate nor aliasing or static typestate checking are discussed. We illustrate typestate extension in Section 6.6 and 6.7.

4.3 Typestate Tracking

Typestate tracking, as mentioned in Section 5.16, refers to the process by which a stated type system is aware of the current typestate of the stated object at any point in time.

Fähndrich [27] present a technique that associates the concept of a "key" with an object such that the "key", rather than the object itself, actually switches between typestates and assumes typestate transitions. By definition, the compiler always uses the object's "key" to reference the actual object in order to track typestate transitions through aliases. However, it is not defined how the type checker will detect the "key" of an object when it detects an alias of that object. Furthermore, the type checker needs to maintain this extra "key" for each object, which is a burden. The programmer also needs to learn to use a new notion and syntax to declare the objects with "keys" so that the type checker can track the transition of typestates through the "keys". Fähndrich [27] acknowledges that their proposed technique has limitations for aliasing. The notion of typestate invariant is not discussed.

Jonathan Aldrich [28] presents typestate as a separate but built-in feature of the object that is similar to our approach. But, the state of an object is not assumed to be a first class language concept. In our approach, both the object and its state are first class language concepts. Furthermore, Jonathan Aldrich [28] does not define typestate tracking or typestate checking.

4.4 Aliasing

At an abstract level, our approach to alias tracking is similar to the "integrated verifier technique" of Fink [26]; but, this approach treats typestate as a separate entity whereas our approach treats typestate as a built-in feature of a stated object. Fink's [26] proposed technique does not define the implementation of how to keep the current typestate of a stated object. Neither does it define how to update the current typestate, nor how to keep the aliases. We detail our technique as compared to the work of Fink [26] by mentioning the implementation aspects. For the implementation of typestate, we introduce 'state class' as a concrete representation of the typestate of a stated object and also introduce an alias table. Further we introduce a single location in the symbol table to keep the current typestate of an object so that all aliases may be statically aware of their current typestate.

4.5 Typestate Invariant

Jonathan Aldrich [28] illustrates that Java I/O and JDBC can be modeled with typestate invariants. We illustrate the use of typestate invariants in Chapter 6. Aldrich [28] defines typestate invariance, but the architecture of how typestate invariant is maintained by the language is not mentioned. We present an implementation technique for typestate invariants in Section 6.5. Moreover, Aldrich [28] does not define typestate checking.

Manuel Fähndrich [29] illustrates that an array reader and a lexer can be modeled with typestate invariants. He incorporates the notion of typestate into objects in a similar way to our concept in that there can be explicitly defined external typestates of objects. However, Fähndrich limits an object to only one possible typestate in such a way that the object will either be in its one possible typestate, i.e. the object is valid, or it will not be in that typestate, i.e. the object is invalid. Furthermore, Fähndrich defines the notion of internal representation of typestate by the data members of an object and describes that the external typestate is bound with the internal typestate data, i.e. typestate invariants. Fähndrich's [29] presented internal typestate data is termed "object invariant" rather than typestate invariant because there is only one typestate that controls an object. In our case, we allow multiple typestates for an object and each typestate may have distinct internal typestate invariants.

Manuel Fähndrich [20] notes that a typestate holds a predicate over an attribute (data member) of an object and its values. However, there is no construct given by Fähndrich [20] which the programmer can use to define the predicate that binds the typestate with the attribute and its values.

In order to facilitate aliasing, Manuel Fähndrich [30] employs "adoption and focus" operations to a linear type system. With these operations, the type checker can assume must-alias properties for a limited program scope. In contrast, our approach does not limit aliasing over the program scope.

4.6 Dynamic Behavior Adaptation

Dynamic behavior adaptation is one of the well-stated requirements of autonomic computing [32, 34, 35, 36]. The MAPE-K loop, shown in Figure 7.7, of autonomic computing provides an architecture for dynamic behavior adaptation. Different approaches have been proposed for capturing the sensor and effector requirement of the MAPE-K loop. We propose that state controlled object oriented programming can be effectively exploited in order to fulfill the sensor and effector requirement of the MAPE-K loop. Considering Example 3.10, the variable of the print manager that holds the number of connected clients will serve as a sensor, and the corresponding polymorphic, adaptable, 'print' behaviors will serve as effectors of the MAPE-K loop.

Many programming language based approaches to 'dynamic compositional adaptation' have been proposed [41, 42] to achieve autonomic and autonomous computing. Sadjadi [48] studied an earlier taxonomy of dynamic compositional adaptation approaches. To the best of our knowledge, no earlier state oriented programming models [17, 18, 20, 21, 24] support typestate-based dynamic compositional adaptation, nor have any prior compositional adaptation approaches proposed the dynamic adaptation of state related behavior of software objects.

Chapter 5

5 A Static Typestate Checking Technique under Aliasing

5.1 Introduction

In this chapter we introduce the aliasing behavior of our proposed SCOOP programming language. We also introduce several components of SCOOP that collectively contribute to a proposed typestate checking technique.

An elegant typestate checking technique in the presence of aliasing is presented in [18, 22, 24], but this technique comes at the cost of requiring the programmer to learn the new concepts of "access permission" and "state guarantee" and use their corresponding notations. In this chapter, we illustrate that the same results can be achieved with SCOOP without using any extra annotations. Our proposed typestate checking does not constrain aliasing in any way. Our proposed typestate checking technique statically computes the set of current possible typestates of a stated object and also makes any aliases of that object aware of that set of typestates.

5.2 Presentation

In order to address the issues of typestate, we need to have a type system with support for typestates [26] rather than the conventional type system of imperative programming languages. We refer to such a type system with typestate support as a "stated type system". In the literature, a stated type system is also referred to as a "typestate system". We propose a "stated type system" along with its implementation technique, "proxy and stated class" architecture. For the implementation of typestate by the compiler, we introduce a new kind of nested class, in Section 5.8, which we call “state class” that is hidden from the programmer and implemented internally by the compiler.

Our newly introduced constructs can be used by the programmer to declare the typestates of an object. However, these typestate declaration constructs are optional. The “typestate structure” is declared in the stated class. It has a lexical scope in its stated class that encloses the attributes and methods of a particular typestate. Transforming each programmer-defined typestate structure, as in Figure 5.1, to a state class, as in Figure 5.2, allows the typestates to be used as a unit entity and first class language concept.

In the following sections, we introduce the programming constructs which declare the external typestates of a stated object from the programmer’s point of view and present the language implementer’s view. In subsequent sections, we introduce our “proxy and state class” architecture.

5.3 The SCOOP Language

In Appendix A we have given a context free grammar for parsing SCOOP programs. This is an earlier Java-like grammar with some added and some modified productions to parse SCOOP language constructs. In the appendix, we have marked some productions with the letter ‘S’ to indicate that these productions are particular to our proposed SCOOP language. As usual in a parser grammar, there are two kinds of symbols. We define our convention for the grammar symbols in the table below.

Kind of Symbols	Description	Examples
Terminals	Keywords and Operators in small alphabets	int, public, statebinding, +, ++,
	Identifiers in small alphabets	id, o_id, st_id
Non Terminals	enclosed in angle brackets	<var_dec>, <class>, <stm>

Table 5.1 SCOOP Grammar Symbols

We discuss in some detail below a few of the productions for parsing SCOOP language constructs. The numbering of the productions we use is according to the numbering in Appendix A.

Production no. 4 in Appendix A is as follows

<class> → <st_dec_list> class cls_id { <cls_body> }

This production allows parsing the stated class declaration, e.g. the `File` class in Figure 5.1. The <st_dec_list> non-terminal would parse the declaration of state name of a class just before parsing the keyword `class`.

Production no. 5 in Appendix A is as follows

<st_dec_list> → [state (<st_list>)]

This production allows parsing the state name of a stated class using the <st_list> non-terminal. It is derived from Production no. 4.

Production no. 6 in Appendix A is as follows

<st_dec_list> → Є

This production allows stopping the expansion of parsing the list of state names of a class.

Production no. 7 in Appendix A is as follows

<st_list> → <st_list> , <st_id>

This production allows parsing the repetition of state names while parsing the declaration of the list of state names using Production no. 5.

Production no. 8 in Appendix A is as follows

<st_list> → <st_id>

This production allows parsing the single state name when there is only one state declared for a class.

Production no. 9 in Appendix A is as follows

<st_id> → st_id

This production allows parsing the actual state name given by the programmer. In this production we represent the actual state name by the terminal   st_id.

Production no. 10 in Appendix A is as follows

<cls_body> → <var_dec_list> < st_binding> <cls_m_list> <c_s_m_list>

This production allows parsing the internal body of a class and is derived from Production no. 4. As is obvious, the class body should first of all allow the variable declaration, followed by a state binding, followed by a list of the methods of the class, followed by the list of the state structures in the class.

Production no. 18 in Appendix A is as follows

<st_binding> → statebinding { <b_list> }

This production allows parsing the state binding and is derived from Production no. 10. The <b_list> non-terminal allows parsing the list of binding state name, with a binding rule.

Production no. 19 in Appendix A is as follows

<b_list> → <b_list> , <bind>

This production allows parsing the binding of state names with the binding rule. This is derived from Production no. 18.

Production no. 77 in Appendix A is as follows

<c_s_m_list>→<c_s_m_list> <c_s_method>

This production allows parsing of the list of state structures, i.e. a possible repetition of many state structures. It is derived from Production no. 10. An example of a state structure is given in Figure 7.1.

Production no. 78 in Appendix A is as follows

<c_s_method> → <s_list> { <s_v_d_lst><s_m_list> }

This production allows parsing a single state structure. It is derived from Production no. 78. The <s_list> non-terminal would parse the state name of the state structure. An example of a state structure is given in Figure 7.1

Production no. 79 in Appendix A is as follows

<s_m_list> →<s_m_list> <s_method>

This production allows parsing the list of methods in the state structure, i.e. a list of methods particular to a state. It is derived from Production no. 79.

Production no. 81 in Appendix A is as follows

<s_method>→<r_type><m_id>(<arg_list>)<s_end_list>{<s_m_body>}

This production allows parsing a method in the state structure, i.e. a method particular to a state. The <s_end_list> non-terminal would parse the output state of this method. It is derived from Production no. 80.

Production no. 90 in Appendix A is as follows

<st_stm>→this.trans(st_id);

This production allows parsing the ‘trans()’ method as a language keyword. Note that this parsing is possible only from the body of a method i.e. it is particular to a state.

Production no. 115, in appendix A is as follows

<s_list> → [st_id]:[st_id]

This production allows parsing the state name of a state structure. The colon separates the state names of the parent class and the subclass. It is derived from Production no. 79.

5.4 Stated Type System

Our flow sensitive “stated type system” validates typestate declarations, typestate transitioning, typestate equivalence, typestate casting, and static and dynamic typestate checking for stated objects. It performs this static analysis with the help of several features. These features include state class (Section 5.8), typestate coercion (Section 5.13), finding aliases with the help of an alias table (Section 5.14) and typestate checking (Section 5.17).

Any operation on a stated object or occurrence of any statement that refers to a stated object or its alias is validated by the stated type system. For validation, the stated type system evaluates the current typestate of a stated object. More specifically, upon any occurrence of a method invocation on a stated object that causes a typestate transition, the stated type system statically updates the typestate (output state) of the stated object. This static update of the typestate is achieved by means of typestate coercion as mentioned in Section 5.13. If a definite output typestate cannot be decided, then the set of possible output typestates is statically associated with the stated object reference.

5.5 Default Typestates

The SCOOP language supports two default typestates for any stated object. These typestates are always part of a stated object typestate set by default. Any field invocation will be invalid while the object is in either of these default typestates.

The first default typestate is *null*. For each stated object, as soon as an object is deleted from memory or its reference is set to null, the stated object reference is assumed to be in the *null* typestate. An object reference that has been declared and not instantiated yet also assumes the *null* typestate.

The other default typestate is *undefined*. Each stated object is assumed to be in one of its declared typestates at any point in time. If, due to some problem (e.g. a function that changes the typestate of the object does not return properly) the current typestate of a stated object cannot be determined, then the stated type system transitions the object to its default *undefined* typestate. Occasionally, an object could be instantiated but its internal typestate representation data may not yet be assigned values compatible with any of its declared external typestates. In this case, the object also assumes the *undefined* typestate.

The *undefined* typestate of a stated object corresponds to what is typically referred to as the *error* state of finite state automata. When a finite state automaton receives an invalid input for which there is no transition defined from the current state, it transitions to the *error* state. Our SCOOP language contains a built-in "typestate interface" from which each stated object is extended (inherited) by default. The "typestate interface" provides the *null* and *undefined* default typestates for each stated object.

5.6 A Programmer's View

We present example code for the classic `File` stated object in Figure 5.1 according to our proposed SCOOP syntax.

```
[state(openfile,closefile=start,eof)]  //declaring the typestates of File
class File{
  string name;                         //this field is accessible  in every state
  public sharedmethod(){}              //this method is accessible in every state
  [openfile]{
    public string file desc;
    public read()[openfile | eof]
    {..
    }
    public display()[openfile]
    {…
      print("original openfile");….
    }
    public  close()[closefile]
    {….
      this.trans (closefile);…..
    }
  }
  [eof]:[openfile]{                    //eof typestate structure
    public override close()[closefile]
    {….
                                       //eof is the extension of openfile typestate
      base.close();..                  // therefore read() method is not available
    }                                  // because at eof further reading is not possible
  }
  [closefile]{                         //closefile typestate structure
    public open()[openfile]
    {..
      this.trans(openfile);...
    }
  }
}
class client{
  File f=new File ();                  //an object of closefile typestate  is created
  f.open();                            //f transitions to openfile typestate
  f.name="file1";
  f.close();                           //f transitions to closefile typestate
  f.name="file2";
  prinf(f.name);                       //this statements prints  "file2"
  f.close();                           //error detected statically i.e.
                                       //"close() not found in the current typestate
                                       //closefile of f"
}
```

Figure 5.1: A SCOOP Program

5.7 A Language Implementer's View

The programmer-defined code in Figure 5.1 is transformed to the intermediate code in Figure 5.2 by the SCOOP compiler according to our proposed “proxy and state class” architecture.

```
class File{                                    //transformed proxy stated class File
  enum state{openfile,closefile,eof}           // translated by compiler
  string name;
  state curr st;
  public File()
   {
      openfile  of=new openfile();
      closefile cf=new closefile();
      closefile ef=new eof ();
  }
  public sharedmethod(){}
  public void trans(state st)
   {
      if(st==openfile)
       this= of;                               //typestate of this is statically coerced  to openfile
      else if (st==closefile)
       this= cf;                               //typestate of this is statically coerced to closefile
   }

class openfile{                                // a special kind of nested class i.e. “state class”
    public string file desc;
    public close()[closefile]
    {….
       this.trans (closefile);
    }
    public read() [openfile | eof]
    {...
    }
    public display()[openfile]
    {…
       print(“original openfile state display”);….
     }
}
class eof:openfile{……                          //a special kind of nested class i.e. “state class”
   public override close()[closefile]
    {….
       base.close();..
     }
}
class closefile{                               //a special kind of nested class i.e “state class”
   public open()[openfile]
    {….
       this.trans(openfile);…..
    }
 }
}
```

```
class client{
   File f=new File();            //an object of initial typestate ( e.g. closefile) is created
                                 //by overloaded 'new' operator for stateful class. The type
                                 //system upon occurrence of '=' would coerce the typestate of
                                 //'f' to closefile statically.
    f.open();
    f.name="file1";
    f.close();
    f.name="file2";
    prinf(f.name);               //it prints file2
    f.close();                   //error detected statically i.e. "close() not found in the
                                 //current typestate closefile of 'f'"
}
```

Figure 5.2 SCOOP language-generated `File` Stated Object.

As in the code given in Figure 5.2, the "stated class", i.e. `File` class, serves as the proxy class for the encapsulated 'state classes', i.e. `openfile`, `closefile` and `eof`. The instances of 'state classes', e.g. `of` and `cf`, are called typestate objects. The `File` stated class in Figure 5.2 is transformed according to our "proxy and state class" architecture and shown in Figure 5.3.

5.8 State Classes

We introduce a new kind of class and name it "state class" as it represents a state of a stated object. A "state class" is an implementation of a typestate. A state class is an inner class coupled with a few additional characteristics so that it can fulfill the typestate requirements, according to our proposed architecture. However, the programmer writing a stated class, as shown in Figure 5.1, declares the typestate and typestate dependent fields (data members and methods) inside the stated class and is unaware of the existence of the "state class". The "state class" generated by the language and hidden from the programmer, as shown in Figure 5.2, contains only the fields specific to its typestate, e.g. `file_desc` in the *openfile* typestate class. Since a state class represents a typestate of an object and the typestate of an object is an inherent characteristic, it is quite natural to define the "state class" as a nested class within the stated class.

5.9 Characteristics of State Classes

We introduce the characteristics of state classes, which are designed such that that they can not only be used as an appropriate implementation of typestate but can also fit into our proposed “proxy and state class” architecture to achieve our desired objectives. Each state class keeps a reference to its proxy class.

Proxy-State Compatibility: All objects of state classes, e.g. `of` or `cf` in Figure 5.2, are compatible with their encapsulating proxy stated object, e.g. `f` in Figure 5.2, such that an instance of any state class can be assigned to the reference of stated class, e.g. `this=of`, in Figure 5.2. This characteristic is technically similar to the inheritance relation, in which subclass objects can be assigned to a parent class reference. However, state classes are not subclasses of their encapsulating stated class. This characteristic is required to implement typestate transition for the stated object.

Single-Proxy Data: Each object of a state class, encapsulated in its stated object, will have access and share a single copy of the fields of its proxy stated object. This characteristic is similar to the inheritance relation, in which subclass objects share the same fields as their parent class. However, state classes are not subclasses of their encapsulating stated class. Contrary to the inheritance relation, an instance of a state class does not keep a separate instance of its stated class. Each object of a state class shares a single instance of its proxy stated object and the same data members of the single instance of the stated object are shared among each typestate object. This characteristic is required so that in the case of typestate transition of the stated object, all shared data of the encapsulating “stated object” can be accessed seamlessly. Let us consider the snippet below:

```
f.open();
f.name=”file1”;
f.close();
f.name=”file2”;
f.open();
prinf(f.name);
```

The last statement of this snippet prints “file2” because the attribute ‘name’ of the ‘File’ class is shared among every state of `f` and is not particular to one state class. Each state class keeps a reference to its proxy class.

5.10 The Proxy and State Class Architecture

The proxy and state class architecture is the scheme by which the typestates are organized in stated objects, as illustrated in Figure 5.3. A state controlled object oriented compiler would transform the programmer defined stated class, e.g. `File` in Figure 5.1, to the “proxy and state class” code as in Figure 5.2.

An alias table and symbol table are used as supporting components within the proxy and state class architecture for stated objects to achieve the desired typestate checking. The alias table is defined in Section 5.14. The use of the symbol table is mentioned in Section 5.14 and 5.16. Figure 5.3 sketches the stated object of the `File` class of Figure 5.2 that gives an idea of how a stated object is organized in memory by the SCOOP language. In Figure 5.4, we demonstrate an expanded sketch of the same object of the `File` class with the supporting components of our proxy and state class architecture, i.e. the typestate entry in the symbol table, the alias table for the object of the `File` class while in *openfile* typestate and the invariant table with an entry of typestate invariant for the *openfile* typestate. The ‘g’ and ‘h’ are the aliases of ‘f’. The invariant table, in Section 6.5, is also a relevant component but it is not used directly for typestate checking. The stated type system, in Section 5.4, performs typestate checking with the help of these components as demonstrated in Figure 5.4.

For each typestate of the programmer-defined stated class, a state class is generated as an inner class inside the transformed stated class. Objects of inner state classes, i.e. typestate objects, are created inside the transformed stated class. The typestate objects are referred to by the instance references, e.g. `of` and `cf`, of the transformed stated class. An additional ‘trans()’ function is generated to implement typestate transitions. The transformed stated class serves as a proxy for the state classes and typestate objects.

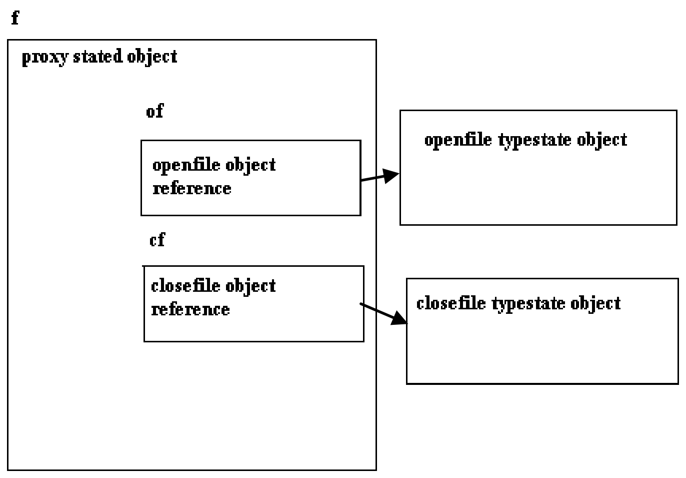

f
proxy stated object
of
openfile object reference
cf
closefile object reference
openfile typestate object
closefile typestate object

Figure 5.3 Proxy and State Architecture of `File` Stated Object

Figure 5.1 illustrates a `File` stated class from the perspective of a programmer. The `File` stated object has two typestates: *closefile* and *openfile*. Any typestate can be extended. For example, *eof* is an extension of the *openfile* state, as in the following snippet of Figure 5.1.

```
[eof]:[openfile]{       }
```

Extended typestates, e.g. `eof`, are called state subclasses and are represented by a subclass of the parent state class in the transformed code, as in Figure 5.2. Inheritance of an object, e.g. File, is the same as in conventional OOP. A specialized File, e.g. ImageFile or TextFile, can seamlessly extend the File class as in conventional OOP. Therefore, the extension of a typestate to its specialized typestate and the extension of an object to its specialized child object are not in conflict. Typestate extension is illustrated in Section 6.6 and 6.7.

In order to achieve typestate checking in a language with aliasing, our proposed architecture resolves the following challenges.

- A typestate transition of a stated object either via the stated object referent or via any of its aliases should be detected statically to prevent any illegal invocation of a field of the stated object.

- All aliases of a stated object should be statically aware of the current possible set of typestates of the stated object preventing any illegal method invocation or field reference.

The strength of our architecture is that it resolves these two above-mentioned challenges without requiring a programmer to learn new annotations to address these issues. The components and features which work together to achieve the desired objectives of our architecture are presented in the following sections.

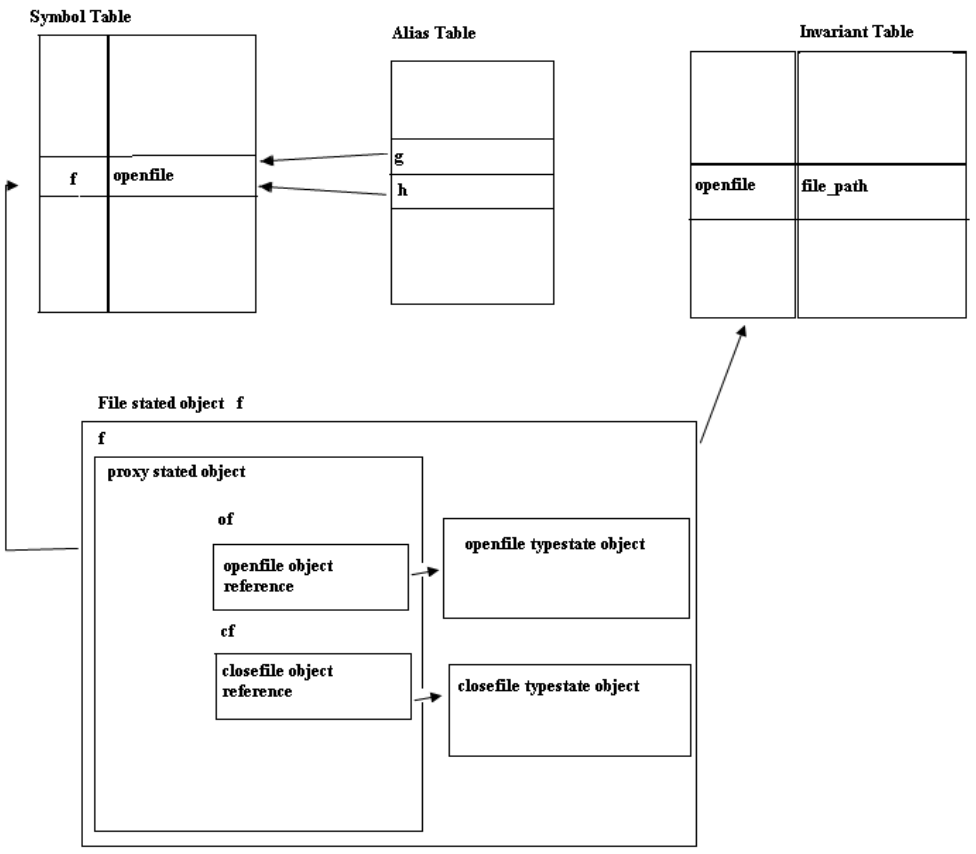


Figure 5.4 A Sketch of Proxy and State Class Architecture for a `File` Stated Object in the *openfile* Typestate with Alias Table, Symbol Table and Invariant Table

5.11 Creating Stated Objects

For creating an instance of a stated object, SCOOP provides an overloaded `new` operator to instantiate a stated object. The overloaded `new` operator has the following characteristics, illustrated with reference to the code in Figure 5.2.

- The overloaded `new` operator instantiates a stated object, e.g. `f`, such that the inner typestate objects, e.g. `of` or `cf`, of its nested state classes will be compatible with `f` (just as subclass objects are compatible and can be assigned to super class references).
- When a stated object `f` of a type, e.g. `File`, is created then it can be coerced to any of its state classes, e.g. `openfile` or `closefile`. Even if it is coerced and points to any of its typestate objects, the fields of the original instance of `File` are accessible by `f`.

Stated Objects in Memory

As we have mentioned, a transformed stated object has two major components i.e. the proxy class and the state class. Each state class will keep a reference to its proxy class. If a stated object is in a particular typestate then it means that the stated object reference is assigned the instance of that particular 'state class'. A state transition of a stated object from its current typestate to another typestate requires that the current 'state class' instance in the stated object reference be replaced by another "state class" instance. In order to replace the current "state class" instance with another "state class" instance, we require that each "state class" should hold the reference of its proxy class instance. This is because only the proxy class instance knows the references of all its state class instances.

5.12 Functional Interface of a Stated Object

Associating the output typestate (post condition) with the functional interface of the object is at the core of SCOOP's objective to achieve static typestate checking. As in Figure 5.1, each function declaration has associated square brackets [] to define its output typestate as below:

```
public void close()[closefile]{...this.trans (closefile);..}
```

The associated output typestate of a method enforces that the object must transition to its associated output typestate when the function returns. Furthermore, the stated type system statically coerces the current typestate of the receiver of `close()` to the output typestate, i.e. `closefile`, when the invocation statement of `close()` occurs. However, such coercion is only possible if there is only one possible output typestate of a function. The reasoning for associating output typestate with the stated object reference is given in Section 5.17.1.

It should be noted that the functions of a class can be enclosed in any of the typestates of the object. The typestate encapsulating a function represents that the function belongs to that particular typestate only. This implies that the encapsulating typestate is the precondition for that function, i.e. the function is only available if the stated object is in that particular typestate. Different functions which have the same name, return type and arguments may coexist in different typestates leading to typestate based polymorphism. A data member or a method of an object can also be explicitly defined as available in more than one typestate, e.g. the `step()` method of the iterator example in Section 6.4.1. If a field is not encapsulated in any typestate or there is no precondition typestate associated with a field, then that field is available to the stated object instance irrespective of any typestate.

5.13 Typestate Coercion

The "Proxy-State Compatibility" characteristic, mentioned in Section 5.9, allows the assignment of "typestate objects" into the reference of the proxy stated object. This is how our architecture implements state transitions of stated objects. As in Figure 5.2, at any point in time, one of the typestate objects, e.g. `of` or `cf`, must be assigned to the reference of the stated object, e.g. `f`. If `f` is assigned `of`, i.e. `f=of`, then this implies that `f` is transitioned to the *openfile* typestate. The stated type system, before assigning `of` to `f`, statically coerces (or casts) `f` from its current typestate to the *openfile* typestate. Similarly, if `f` is assigned `cf`, i.e. `f=cf`, this implies that `f` is transitioned to the *closefile* typestate. Therefore, the stated type system, before assigning `cf` to `f`, statically coerces `f` from its current typestate, *openfile* to the other typestate, *closefile*.

From an implementation viewpoint, typestates, e.g. *openfile* or *closefile*, are actually represented by “state classes”. The current typestate of an object is kept at a single location, i.e. at a symbol table entry for the original reference of the object. Therefore, the static typestate coercion of `f` actually inserts the name of either of the “state classes” in the symbol table entry corresponding to `f`. When an object `f` of a class, e.g. `File`, is created, it can be coerced to any of its state classes, e.g. *openfile* or *closefile*. Even if it is coerced, the common fields of the original instance of `File` are accessible through the object instances of the state classes.

5.14 The Alias Table

There may be many aliases of an object’s original referent. Similar to a symbol table, the stated type system statically maintains an alias table as a repository for all of the aliases of the original referent of each object. There is a single alias table for each stated object. The current typestate of an object is kept at a single location only, i.e. at a symbol table entry of the original reference of the object. All aliases and the original referent of the stated object recognize their current typestate through that single location. All aliases of an object in the alias table point to the symbol table entry of their original object reference to read their current typestate. We need to maintain an alias table as well as a single location in the symbol table to hold the current typestate of the object, so that whenever a stated object changes its typestate, the typestate transition is updated to that one single location and reflected to all of the aliases. Updating a typestate transition to all the aliases of a stated object statically prevents any illegal field invocation of the stated object by any of its aliases.

An operation on a stated object may cause the transition of the object in either of the many possible output typestates. In that case, one possible current typestate cannot be determined statically and cannot be assigned to that symbol table entry statically. Therefore, the symbol table entry for holding the current typestate is instead assigned a set of possible current typestates. The actual current typestate of the object taken from the set of possible current typestates is determined at run time, i.e. when the object actually transitions to a typestate at run time. We have introduced the implementation of typestate transitions with the help of typestate coercion and proxy-state compatibility.

5.15 Tracking Aliases

The SCOOP language would perform an "automated alias analysis" during the parsing of the whole program. Therefore, this alias analysis performs whole program analysis. All aliases of each object are detected while parsing the program as described below.

Keeping track of the aliases of a stated object requires checking of:

- the assignment operator (=)
- parameter passing to method calls

While parsing the source code, upon any occurrence of the assignment operator (=) for a stated object, the stated type system, in addition to the usual type checking of the LHS and RHS of the assignment operator, also performs the following operations:

1. Store Aliases: If the RHS of the assignment operator is a stated object reference and the LHS is the intended alias of the RHS, then the LHS is added to the alias table.

2. Typestate Casting: The pointer of newly inserted LHS alias would be set to the typestate entry of the original object reference in the symbol table.

5.16 Static Typestate Tracking

The term “static typestate tracking” represents the mechanism for the previously discussed stated type system. This statically keeps the typestate of objects and their aliases updated in the symbol table with the help of typestate coercion. The coercion operation is completely defined in Section 5.13.

Below we discuss a few code segments with reference to the program in Figure 5.1.
Upon any occurrence of the `trans()` function within each static lexical scope, e.g. an if-else lexical scope or a for-loop lexical scope, the stated type system performs the following operations statically:

- The typestate of the receiver of the `trans()` function is coerced (casted) into the typestate to which transition is intended.

- The typestate of all aliases of the receiver of the `trans()` function is coerced (casted) into the typestate to which the transition is intended.

Methods can transition the typestate of the receiver object upon their return. It is important that such methods specify their output typestate, so that the typestate of the receiver can be appropriately coerced. Let us consider the following code snippet:

```
public void close()[closefile]{..this.trans (closefile);..}
```

When the invocation statement of `close()` occurs, e.g. `f.close()`, the stated type system statically coerces the current typestate of the receiver of `close()` to the *closefile* typestate. This coercion is implemented by updating the typestate entry in the symbol table. The significance of associating the output typestate with each method signature is illustrated in Section 5.17.1.

5.17 Typestate Checking

If the current typestate of an object is known, then the accessible fields of the stated object can be determined. Therefore, checking the current typestate of an object is crucial to SCOOP. The current typestate of an object can be determined at compile time i.e. statically. However, the prerequisite of static typestate checking is that every method must describe only one possible output typestate (post condition) of its receiver. As long as this prerequisite is met, our technique completely resolves static typestate checking in the presence of aliasing without requiring any additional annotations.

Many practical situations require methods for allowing more than one possible output typestate (post condition). For example, the `step()` method of our iterator object `HIter`, in Section 6.4, has two possible output typestates, *traversal* and *end*. In this case, the exact current typestate of the object cannot be determined statically when such a method would have returned. If the exact current typestate of the object cannot be determined statically then we have to rely on dynamic typestate checking. However, dynamic typestate analysis is able to exploit the information, i.e. a set of possible typestates, already deduced by static typestate analysis.

5.17.1 Static Typestate Checking

Static typestate checking is challenging, especially in the presence of aliasing. However, with the help of our proposed architecture, given in Section 5.10, our stated type system accomplishes typestate checking statically without requiring any extra annotation as in [18, 24]. With the help of static typestate tracking, mentioned above, each stated object and aliases statically knows its current typestate. Since a stated object knows its current and updated typestate, the stated type system can statically compute all available fields (data members and methods) of the current typestate of the object. Now, let us consider the following snippet from the client-side code of Figure 5.1.

```
    f.close();
    f.name="file2";
    prinf(f.name);
    f.close();
```

The second call to `close()` method is invalid because the first occurrence of `f.close()` has statically coerced `f` to the *closefile* typestate. A file already in *closefile* typestate cannot be closed again. Therefore, static typestate checking would statically prompt the error as below:

"`close()` not found in the current typestate *closefile* of `f`".

Such an error can be statically detected if the following two conditions are met:

- The current typestate of the object is statically known at any point in time.
- The available fields of the object, depending upon its current typestate, can be computed statically.

In our setting the first condition is met by "static typestate tracking". When the first call of `f.close()` occurs, the typestate of `f` is statically coerced to *closefile* by the "static typestate tracking" mechanism of the stated type system. The stated type system can perform such a coercion only because it knows from the signature of `close()` that the output state of `close()` is *closefile*. It is, therefore, necessary for each method to specify its output state for the appropriate typestate coercion of the receiver.

The second condition is met easily. Upon the second occurrence of `f.close()`, the current typestate of `f`, i.e. *closefile*, is statically known. Therefore, the available fields of the current *closefile* typestate can be computed from its state class, i.e. `closefile`, and it can be determined that the method `close()` is not available in the `closefile` state class. This second condition is met so conveniently due to our natural representation of typestate as a distinct state class. Let us consider another code snippet:.

```
void m(File f, File g){  f.close();print(g.file_desc); }
```

The field `file_desc` of a File object points to a low level resource only when the file is in *openfile* typestate. Due to possible aliasing, `f` and `g` may refer to the same object. In this case the method signature and body are still well typed due to our technique for handling aliases. Upon the occurrence of `f.close()`, the stated type system would coerce the typestate of `f` to *closefile*. This coerced typestate will also be reflected in all of its aliases, including `g`. Since the typestate of `g` is statically known to be *closefile* and in the *closefile* typestate the statement `g.file_desc` is invalid, the field `file_desc` is not available in the *closefile* typestate. Therefore, the stated type system would statically prompt the error as below:

"The field `file_desc` does not belong to the current typestate *closefile* of `g`".

In order to illustrate the generality of our proposed typestate checking technique, we take another example of a simple iterator client from [22] as below:

```
Collection c=new….                                  //legal
Iterator it=c.iterator();                           //legal
if(it.hasNext()  &&  c.size() == 3) {               //legal
   c.remove(it.next());                             // legal
if(it.hasNext())                                    //ILLEGAL
}
```

Figure 5.5 A Simple Iterator Client

This sample code is modified from the sample code in [22] for simplicity, but illustrates the same intentionally seeded illegal invocation of a function. We show that our proposed technique can smoothly detect this illegal invocation statically without requiring the programmer to write any complex annotations as in [22]. In this example code, the object `it` of the `iterator` class iterates over the `Collection c`. This code presents an interesting scenario in which the `iterator` iterates to its next data member, and stays (points) at that data member, but that data member is deleted by `Collection c` at which the iterator is located. Since the current data member has been deleted, calling the `it.hasNext()` function on the deleted (or null) data member is an invalid function invocation. In [22], an elegant but complex technique is presented to statically catch this invalid invocation. However, our technique is very simple and does not require the programmer to write any extra annotation. With the global analysis of our stated type system, as soon as the `it.next()` data member is passed on as an argument to the `c.remove()` function, the stated type system detects that the parameter `n` in the signature of `c.remove(Node n)` is an alias of `it.next()`. As soon as `c.remove(Node n)` deletes its received parameter `n` from `c`, the stated type system detects that the typestate of this alias `n` is transitioned to the *null* typestate. Therefore, the typestate of the original argument `it.next()` is also transitioned to the *null* typestate because all aliases of an object share the same typestate. In the subsequent statement, `it.hasNext()` is invoked and the stated type system knows statically that the data member to which `it.hasNext()` points has transitioned to the *null* typestate. A data member in its *null* typestate cannot access any of its methods or attributes, therefore, it is statically detected.

5.17.2 Dynamic Typestate Checking

The typestate analysis problem requires detecting the current typestate of a stated object at any location of the program either statically or dynamically. In other words, it requires finding what typestate the object has reached at any location of the program. The typestate analysis problem is undecidable statically. There are many practical scenarios in which the typestate of a stated object cannot be detected at compile time. This is because we may never know until run time what the output typestate will be when an invoked method returns. Therefore, violations of invoking a typestate dependent field cannot be determined unless the object has actually transitioned to a particular typestate at run time. Let us consider the code snippet below:

```
File f=new File();
f.open();
f.read();
f.read();
```

Intuitively the first call to `read()` can either transition `f` to the *eof* typestate or keep it in the same *openfile* typestate because there are two possible output typestates for `read()`. Therefore, only at run time, i.e. via dynamic typestate checking, can it be determined whether the second call of `read()` is a valid call or not. This is because the `read()` method is not available if `f` has transitioned to *eof* typestate as a result of first call to `read()`. Since, the `read()` method has two possible output typestates, our static typestate tracking mechanism cannot coerce `f` into any one possible typestate so the static typestate check is not very helpful. In such cases, the compiler can generate runtime assertions to find the actual typestate of `f` at run time and coerce `f` accordingly. The "proxy and state class" architecture would still work seamlessly at run time and with the help of dynamic typestate coercion, the available fields of the current typestate of `f` can be detected.

In this example, the current typestate of `f` cannot be statically determined when the first call of `f.read()` returns because the method `f.read()` has more than one output typestate. The second call to `f.read()` may be invalid if the first call to `f.read()` has transitioned `f` to the *eof* typestate. In such cases, if the second call to `f.read()` is invalid then an exception would be raised at run time. If the programmer has not handled the exception in the code then the program will crash at run time. However a SCOOP compiler can statically detect that the second call of `f.read()` may be invalid because of the first call that transitions `f` to the *eof* typestate. Therefore, upon such a detection, the SCOOP compiler can statically check whether the programmer has enclosed the second call of `f.read()` inside the proper exception handling scope. If the compiler discovers that the second call to `f.read()` is not enclosed in exception handling code, then the SCOOP compiler would generate a warning statically for the programmer to enclose the second call of `f.read()` within appropriate exception handling scope. Such a warning generated by the SCOOP compiler will make a significant impact on avoiding unexpected program crashes due to inconsistent typestates. This is our contribution as no previous study has attempted to avoid such typestate inconsistencies by generating these warnings statically.

5.18 Conclusion

To the best of our knowledge, the presented static and dynamic typestate checking techniques are the only techniques presented so far that include both of the following:

- Defines the implementation architecture, i.e. given in Section 5.10, for typestate checking.
- Works in the presence of aliasing and does not require a programmer to learn and use any additional annotations.

We claim that our proposed technique allows effective software testing through user friendly typestate checking.

Chapter 6

6 The Typestate Invariant

6.1 Introduction

In state controlled object oriented programming, the explicit external typestates of an object are declared in the object definition. The binding rule associated with each typestate is verified by the stated type system. In this chapter we introduce how the stated type system would implement the validation of the typestate binding rule. We represent the set of explicitly declared typestates of a stated object as T_O.

“T_O” is the set of typestates associated to an object O.

The SCOOP language recognizes the attributes and the values of the attributes that represent the internal typestate of that object. These attributes along with their defined values are referred to as the typestate invariant. The external typestate and its corresponding typestate invariants are bound with each other in the object definition. This binding is defined via an invariant binding rule (state predicate).

For instance, the *openfile* typestate of a `File` object can be bound with the `file_path` attribute of `f by` having a binding rule such that if `f` is in the *openfile* typestate then the `file_path` attribute should hold a valid memory address of an opened file, i.e. it cannot be null. Therefore, the typestate invariant rule for the *openfile* typestate can be represented as below.

```
file_path !=null
```

The typestate and its invariant are bound as illustrated in the snippet below from Figure 5.2.

```
[statebinding{ openfile : ( file_path !=null ) }]
```

The stated type system validates at any point in time whether each of the external typestates of the object complies with the binding rule that binds the typestate invariant with its external typestate. A modification in the value of a bound typestate invariant would also be verified by the stated type system to check whether the modified value complies with its binding rule. In case there is a violation of a binding rule of the current typestate, either the object would be forced to transition to another typestate that complies with the binding rule, or the object would be transitioned to an *undefined* typestate as mentioned in Section 5.5. It should be noted that the *undefined* typestate is a default typestate available for each stated object.

We argue that the typestate and the typestate invariant are associated with the stated object instance instead of the stated class. Therefore, the typestate and typestate invariant are not captured with respect to the subclass hierarchy.

6.2 Motivation

An explicitly defined external typestate of an object that is bound with its internal typestate data (typestate invariant) allows simpler and easier programming for the user of the stated object because the stated object itself ensures that the external typestate and internal typestate representing data are synchronized.

The advantage of a typestate invariant is that the programmer does not need to know the complex internal invariant binding rule to realize the current typestate of the object. The user of a stated object can simply access the external typestate, i.e. a single piece of information, to find the current typestate rather than bothering with the complex rules concerning the typestate invariant to realize the current typestate of the object.

Binding external typestate with the internal typestate invariants provides better information hiding. An internal typestate invariant does not need to be exposed outside the object, and the user of the object can realize the internal typestate invariants through the external typestates.

For instance, the *fullcharged* external typestate of a printer stated object may abstract over typestate invariants such as ‘cartridge ink level’ and ‘cartridge last replaced’. The values of each of these typestate invariants may collectively or individually cause the printer typestate to transition from the *fullcharged* to the *halfcharged* typestate in the case that either ‘cartridge ink level’ goes below 50% or ‘cartridge last replaced’ exceeds 20 days. Such a transition of state may also depend on some correlation between the internal typestate invariants (although this is beyond our present scope). The user of the printer object does not need to know the internal typestate representation, e.g. whether the ‘cartridge ink level’ goes below 50% or ‘cartridge last replaced’ exceeds 20 days, in order to realize the current typestate of the printer stated object. Therefore, to determine the current typestate of an object, the user of the stated object can write a simpler piece of code, as shown below:

```
       if printer.curr_state == printer.fullcharged then
```

rather than writing the following piece of code:

```
if(printer.cartridge_ink_level>50AND
                     printer.cartridge_last replaced>20) then
```

SCOOP-supported typestate invariants imply that the compiler provides the memento design pattern by default, and the programmer does not need to bother with writing code to extract, save, and restore the state specific data in order to preserve the state of an object as illustrated in Section 6.8

6.3 Implementation of a Typestate Invariant

In the literature, the notion of a typestate invariant has already been studied. In this chapter, we study how to implement typestate invariants.

In order to implement typestate tracking and invariant binding, we propose the use of an invariant table. The invariant table serves as the architecture for maintaining and tracking typestate invariants. In Section 6.5, we explain how typestate invariants are maintained with the help of an invariant table. The proposed use of the invariant table also supports typestate extension and subclassing simultaneously.

6.4 Binding Typestate with a Typestate Invariant

The binding of an object's external typestate with its typestate invariants is defined by our proposed state binding construct as illustrated in Section 6.1

We refer to the state predicate concept of [20] as the binding rule and introduce our `[statebinding]` keyword to declare a binding rule. The binding rule can optionally be used by the programmer to bind the values of the object's typestate invariant with its typestate.

Our `[statebinding]` keyword enforces that the current external typestate should be compliant with its binding rule at every point in time.

For any method that changes the typestate of an object, the stated type system verifies that the typestate invariant binding rule holds for the post condition typestate when that method returns. This verification performed by the stated type system also applies to the overridden method in a subclass. Therefore, the type system also ensures that overridden methods in the subclass leave the object according to the post condition typestate of the method.

6.4.1 Iterator Example

An iterator is a programming construct that performs custom iteration over a data structure. During an iteration, the iterator object switches between different states that can be suitably modeled by a stated object. We take an object based iterator example from [6] and present it using our proposed state controlled syntax in Figure 6.1. This example is also referred to in Case Study 3.12.

In this example, we illustrate the typestate binding keyword as below.

```
    [statebinding]{ initial :( j== -1 AND i < ht→buckc)]
```

The above statement binds the instance variables, i.e. the internal representation of the typestate, with the typestate name, i.e. the externally accessible typestate, of the iterator object `HIter`, in Figure 6.1. Here, the binding keyword binds the *initial* external typestate with the range of values of `j`, `i` and `ht→buckc`.

The instance variable `i` represents the index of the element currently pointed by the iterator. The value of `ht→buckc` represents the index of the last element of the iterator. The defined range of values is the invariant of the *initial* external typestate of the iterator object `HIter`. Once the binding between a typestate and its typestate invariant is defined, then the typestate transition also modifies the values of the bound typestate invariant by default as mentioned in Section 6.4.3. Alternatively, modification of the values of the bound typestate invariant may also force the transition of the corresponding typestate of the object as mentioned in Section 6.4.2.

```
[state{ initial , traversal , end }]
template <typename Key, typename Val>
class HIter : public Iter<Val> {
  HTable<Key,Val> *ht;
  HBlock<Key,Val> *blk;
  int i, j;
  [statebinding{initial  :( j== -1 AND i < ht->buckc),
                traversal:( j==0 AND i < ht->buckc),
                end      :( i== ht->buckc)}]
  public:
  HIter(HTable<Key,Val> *ht0) [initial]{
    this.trans(initial);
    ht = ht0;
    i = 0;
    j = -1;                               // ++j will gives entv[0] i.e. j is 0
    while (i < ht->buckc) {               //this loop find first non-empty
       blk = ht->buckv[i];                // block
      blk = ht->buckv[i];
      if (blk && blk->entc > 0) break;
      i++;
    }
    this.step();
  }
  [initial,traversal]
  void step()[ traversal | end]
  {
    this.trans(traversal);
    if (++j < blk->entc) return;
    j = 0;                                // Try start of a block.
    blk = blk->next;                      // Try next block in chain.
    if (blk && blk->entc > 0) return;
    i++;                                  // Try next chain.
    while (i < ht->buckc) {
      blk = ht->buckv[i];
      if (blk && blk->entc > 0) break;
      i++;
    }
    if(i==ht->buckc)                      //these two statements in grey are
       this.trans('end');                 //shown for brevity but  are not
                                          // part of this code.
  }
  Val value() { return blk->entv[j].val; }
  bool empty() { return i == ht->buckc; }
};
```

Figure 6.1 Iterator Stated Object

The binding rule that binds a typestate with its internal typestate invariant data members, is true as long as the iterator is in that typestate. For instance, for the *traversal* typestate, the stated type system makes sure that as long as the binding rule for the *traversal* typestate is true, then the `step()` function is accessible. For the *traversal* typestate, the typestate invariants are `i, j` and `t→buckc`. The *traversal* typestate will hold as long as the following rule is true:

```
[statebinding{traversal:( j==0 AND i < ht→buckc)}]
```

Similarly for the *end* typestate of the iterator, the typestate invariants (`i` and `ht→buckc`) are bound as below:

```
[statebinding{ end :( i== ht→buckc)}]
```

And for the *initial* typestate, the typestate invariants (`j, i and ht→buckc`) are bound as below:

```
[statebinding{ initial :( j== -1 AND i < ht→buckc)}]
```

The following statement

```
[statebinding{ initial :( j== -1AND i < ht→buckc),
traversal :( j==0 AND i < ht→buckc),  end :( i== ht->buckc)}]
```

from Figure 6.1 collectively declares the binding of the internal typestate invariants with the corresponding external typestates. Furthermore, it defines the range of values (binding rules) of those internal typestate invariants for which the corresponding external typestate holds.

Once the binding is defined, it is the responsibility of the stated type system to monitor the internal typestate invariants and their current values with the help of the invariant table according to the typestate binding rule. The stated type system then keeps track of any non-compliance of the binding rules and makes the typestate transition accordingly.

The two statements in grey, in Figure 6.1, are shown for brevity but are not the part of the code. This conditional transition is deduced and internally implemented by the SCOOP language due to the defined typestate invariant rule and post condition of the `step()` method.

6.4.2 Typestate Invariant Based Default Transition of Typestate

To the best of our knowledge, previous studies have only considered the effect of typestate transitions on typestate invariants. The converse, i.e. the effect of changes in typestate invariants on typestate transitions, has not yet been studied.

While binding the typestate invariant with an external typestate, we can also define the range of data domains for which the typestate invariant should hold under that external typestate. This range of values of a typestate invariant serves as the internal representation for the corresponding external typestate. As soon as the values of a bound typestate invariant go out of the specified range, i.e. the binding rule is violated for that specific external typestate, then the bounded external typestate is transitioned to the other external typestate according to the current values of its invariant.

For instance, at the return of the `step()` function, the stated type system validates the binding rule of the *end* typestate, if `i == ht→buckc`. The grey lines of code in the `step()` function show that the typestate transition conditional to the bound typestate invariant is internally implemented by the SCOOP language. This default transition of typestate is achieved with the help of the invariant table that is discussed in Section 6.5.

6.4.3 Typestate Transition Based Modification to Typestate Invariant

The user of a stated object can explicitly transition the stated object from its current typestate to another typestate. For instance, while the iterator object is in the *traversal* typestate, the client-side code can invoke the following method:

```
    iterator.trans(end)
```

The above code not only transitions the iterator object to the *end* typestate but also enforces the *end* typestate invariants. Therefore, the stated type system sets up the *end* typestate invariant, i.e. `i == ht→buckc`, by modifying the value of `i`. This default modification to the typestate invariant is recorded in the invariant table mentioned in Section 6.5.

6.5 The Invariant Table

The SCOOP compiler creates a specific hash table data structure called the "invariant table" for each stated object. The invariant table is created along with the symbol table. One of its main uses is to hold the bindings between typestates and their invariants. Each typestate name of a stated object is an entry in the invariant table in the same way that each variable or object name is an entry in the symbol table. Each row in the invariant table holds the typestate name, its typestate invariant, its binding rule, and a flag to mark whether the binding rule is true or false. If the flag for a specific typestate is true then that specific typestate is the current typestate of the object. The stated type system, at any point in time, evaluates the binding rule and sets the flag accordingly for any given typestate name entry in the invariant table.

The invariant table always holds two default typestate names for each stated object, i.e. the *null* and *undefined* typestates. As previously mentioned, whenever a stated object points to a null reference then it is in the *null* typestate. Whenever there is no typestate binding rule that evaluates to be true for a stated object then the object is transitioned to the *undefined* typestate.

The stated type system always points to a typestate name in the invariant table corresponding to the current typestate of a stated object. In other words, for each stated object, the stated type system always points to that typestate entry in the invariant table which has its flag set to true. If the binding rule of the current typestate is evaluated as false, then a recovery operation is performed by the SCOOP compiler that either transitions the typestate of the object to the typestate for which the binding rule is true in the invariant table, or transitions the object to the default *undefined* typestate. The default *undefined* typestate is mentioned in Section 5.5.

Table 6.1 illustrates an invariant table for the iterator stated object, given in Figure 6.1, while it is in the *traversal* typestate.

Typestate	**Invariant**	**Invariant rule**	**Valid**
null	This	this == null	False
error	This	this $\neq$ null AND this.curr_tstate $\notin T_{iterator}$	False
initial	`j,i, ht→buckc`	`j== -1 AND i < ht→buckc`	False
traversal	`j, i, ht→buckc`	`j==0 AND i < ht→buckc`	True
end	`j, i, ht→buckc`	`i== ht→buckc`	False

Table 6.1

6.6 Typestate Extension

In this section we illustrate typestate extension with a few variations in our code fragments. The sample code in Figure 6.2 is a SCOOP program. It includes a `File` class with an *openfile* typestate, *closefile* typestate and *eof* as an extension of the *openfile* typestate.

```
[state(openfile,closefile,eof)]
class File{
  string name;

  [openfile]{
    public  read()[openfile | eof]{..}
  }

  [eof]:[openfile]{
       …………
  }

  [closefile]{
      ………
  }
}
```

Figure 6.2 SCOOP `File` Stated Class

The sample code in Figure 6.3 is the SCOOP generated translation of the code given in Figure 6.2.

```
class File{
  enum state {openfile, closefile,eof}
  string name;
  state curr_st;

  class openfile{                              // openfile 'state class'
    public read() [openfile | eof]{...}
  }

  class eof:openfile{……                        // eof 'state class'
          …..
   }
  class closefile{                             // closefile 'state class'
          ………….
  }
}
```

Figure 6.3 SCOOP-generated `File` Stated Class

The sample code in Figure 6.4 is a SCOOP program. It includes a `File` class and an `Imagefile` as its subclass. The `File` class has an *openfile* typestate, a *closefile* typestate and an *eof* typestate as an extension of the *openfile* typestate. The `Imagefile` overrides the *openfile*, *closefile* and *eof* typestates of the parent class `File`.

```
[state(openfile,closefile,eof)]
class File{
  string name;

  [openfile]{
    public read()[openfile | eof]{...
    }
  }

  [eof]:[openfile]{
       ...........
  }

  [closefile]{
      .........
  }
}

class Imagefile:File{
     ...
  [openfile]{
        ....
  }
  [closefile]{
    ...
  }
  [eof]:[openfile]{
       ...........
  }

  }
}
```

Figure 6.4 SCOOP `File` and `Imagefile` Stated Class

The sample code in Figure 6.5 is the SCOOP generated translation of the code given in Figure 6.4.

```
class File{
  enum state {openfile, closefile, eof}
  string name;
  state curr_st;

  class openfile{                                  //openfile ‘state class’

    public read() [openfile | eof]{...
    }
  }

  class eof:openfile{……                            // eof ‘state class’
          …..
   }
  class closefile{                                 //closefile ‘state class’
          …………
  }
}
class Imagefile:File{
   enum state {openfile, closefile, eof}

  class openfile{                                  //openfile ‘state class’
    public read() [openfile | eof]{...
    }
  }

  class eof:openfile{……                            // eof ‘state class’
          …..
   }
  class closefile{                                 // closefile ‘state class’
          …………
  }
}
```

Figure 6.5 SCOOP-generated `File` and `Imagefile` Stated Class

The sample code in Figure 6.6 is a SCOOP program. It includes a `File` class and an `Imagefile` as its subclass. The `File` class has the *openfile* and *closefile* typestates. The programmer in the `Imagefile` subclass overrides the *openfile* typestate and then extends the *openfile* typestate by the *eof* typestate.

```
[state(openfile,closefile,eof)]
class File{
  string name;

  [openfile]{
    public read()[openfile]{...
    }
  }

  [closefile]{
      .........
  }
}
class Imagefile:File{
     ...
  [openfile]{
        ....
  }
  [closefile]{
    ...
  }
  [eof]:[openfile]{
       ............
  }

  }
}
```

Figure 6.6 SCOOP `File` and `Imagefile` Stated Class With Typestate Extension

The sample code in Figure 6.7 is the SCOOP generated translation of the code given in Figure 6.6.

```
class File{
  enum state {openfile, closefile, eof}
  string name;
  state curr_st;

  class openfile{                                         // openfile ‘state class’

    public read() [openfile | eof]{...
    }
  }

  class closefile{                                        // closefile ‘state class’
          …………
  }
}

class Imagefile:File{
   enum state {openfile, closefile, eof}
  class openfile{                                         // openfile ‘state class’

    public read() [openfile | eof]{...
    }
  }

  class eof:openfile{……                                   // eof ‘state class’
          …..
   }
  class closefile{                                        // closefile ‘state class’
          …………
  }
}
```

Figure 6.7 SCOOP-generated `File` and `Imagefile` Stated Class With Typestate Extension

6.7 Typestate Invariants With Typestate Extension and Subclassing

As mentioned, SCOOP supports “typestate” as a third kind of field for an object. In Figure 6.8, we illustrate a `File` stated object definition and `ImageFile` as its subclass definition.

$$T_{File} = \{ \texttt{openfile, closefile, eof} \}$$

$$\text{and, } T_{ImageFile} = \{ \texttt{openfile, closefile, eof} \}$$

In our setting, each typestate of an object is always public for the object instance. The typestates of the superclass are inherited by the subclass so that the subclass knows the typestates of its superclass. However, in order to make the typestates of the superclass part of the subclass interface, the typestates of the superclass need to be explicitly declared (overridden) in the subclass so that the overridden typestates are accessible to the object instance of the subclass along with any new typestates of the subclass. A subclass can choose to override as many typestates of its superclass as it needs to. Figure 7.1 illustrates a typestate structure. The complete typestate structure from the superclass can also be optionally overridden in the subclass. Moreover, the subclass allows for overriding of the typestate invariant for the overridden typestate with a new binding as illustrated in Figure 6.8. For instance, for a `File` superclass, there is a field `file_path` and the typestate invariant rule for *openfile* typestate of `File` can be coded as below:

```
file_path!=null
```

The `Imagefile` subclass has an attribute `image`. The instance of the `Imagefile` subclass reads an image from the location `file_path` and loads it in its `image` attribute. Therefore, the `Imagefile` subclass can override the binding rule for its *openfile* typestate that formulates the overall binding rule as shown below:

```
file_path != null AND image!=null
```

```
[state(openfile,closefile=start,eof)]
class File{                              //beginning of File class
 string name;                            //this field is accessible in every states
 string file path;
  [statebinding{ openfile :( file path !=null ) }]
  public sharedmethod(){}                //this method is accessible in every typestates

  [openfile]{
    public string file desc;
     public  read()[openfile | eof]{...}
     public display()[openfile]{
       print("original openfile state display");..}
       public  close()[closefile]{..
          this.trans (closefile);...
       }
   }
   [closefile]{
       public  open(string file path arg)[openfile]{...
          this.trans(openfile);...
       }
   }
   [eof]:[openfile]{ ...
   }
}                                        //end of File class


[state(openfile,closefile=start,eof)]
class Imagefile:File{
 [statebinding{ openfile:( base.file path !=null AND
                                    this.image != null )}]
  [openfile]{
    string image;
    public override read()[openfile]{
    }
    public override display()[openfile]{..
      print("image of openfile state");..
    }
    public  override close()[closefile]{..
      this.trans (closefile); ...
    }
  }
 [closefile]{
    public open(string file path arg)[openfile]{...
      this.trans(openfile);...
    }
  }
}



class cls main{
   static void main(){
     Imagefile imfl = new Imagefile();
     Imfl.open("d:\imfl.bmp");
   }
}
```

Figure 6.8 Typestate Extension With Subclassing

SCOOP allows the typestate extension of the *openfile* typestate to the *eof* typestate and the inheritance of `File` to `ImageFile` along with the overriding of the binding rule in the `ImageFile` subclass, which can be composed in a single program, as illustrated in Figure 6.2. Table 6.2 shows a partial illustration of the invariant table for the `File` stated object while it is in the *openfile* typestate.

Typestate	**Invariant**	**Invariant rule**	**Valid**
openfile	file_path	file_path != null	True

Table 6.2. Partial Invariant Table of `File` Stated Object

Table 6.3 shows a partial illustration of the invariant table for the `ImageFile` stated object while it is in the *openfile* typestate.

Typestate	**Invariant**	**Invariant rule**	**Valid**
openfile	base.file_path, this.image	base.file_path !=null AND this.image!= null	True

Table 6.3. Partial Invariant Table of `ImageFile` Stated Object

Each stated object instance has only one invariant table regardless of the fact that the stated object instance is an instance of a subclass or superclass. In [20], Manuel Fähndrich proposes that each subclass instance should keep a separate frame for each of its superclasses. This can become cumbersome.

6.8 State Preservation

Since SCOOP inherently captures the state of a stated object, it can preserve the state data by default. Case Study 3.11 illustrates the capability of the SCOOP language to provide the memento design pattern by default to preserve the state of an object.

6.8.1 Memento Design Pattern

In conventional OOP, the memento design pattern [44] is a behavioral design pattern that extracts the state (data members) of an object outside that object, holds the state of the object and then restores the held state back to the same object. The ‘originator’ is the object whose state is extracted and saved outside of it. The object that temporarily stores the state of the originator object is called the ‘memento’ object, and the ‘caretaker’ is another object that holds the ‘memento’ object. In addition to these objects (memento and caretaker), the programmer needs to write two additional functions in the originator object definition. One of those functions returns the original state so that the original state of the object can be saved outside of the object. Another function simply receives the originally saved state so that it can be restored. The memento design pattern is implemented using these additional objects and functions. Note that in the context of conventional OOP, the term “state” means the data members of the object.

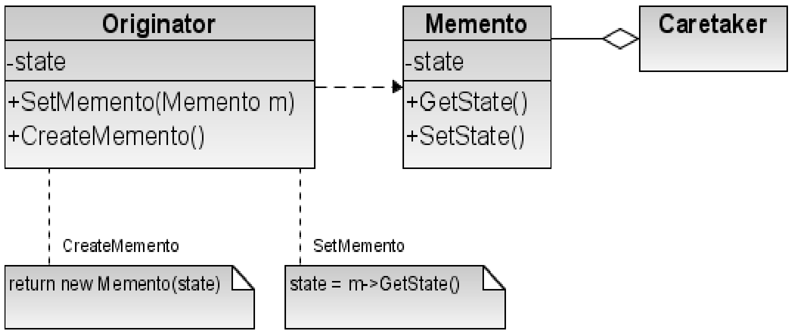


Figure 6.9 Memento Design Pattern

6.8.2 Memento by SCOOP

As an implication of the use of the invariant table, SCOOP provides the memento design pattern by default. In order to take advantage of memento functionality, the programmer neither needs to write any additional functions to extract and restore state nor code any additional memento or caretaker classes (in contrast to conventional OOP). Therefore, in comparison to conventional object oriented programming, SCOOP reduces the burden on the programmer of writing additional code. SCOOP can provide such built-in functionality due to the following characteristics:

- Each typestate is a first class language concept.
- Each external typestate and its associated typestate invariant data have already been defined in the object definition.
- Typestate invariant data members are already recorded separately in the invariant table.

A user of a stated object within client-side code can therefore extract the internal state because the SCOOP compiler already knows the data members particular to a typestate of the stated object. Furthermore, in SCOOP, we can define an object of a "typestate" that serves as an equivalent to the memento object of conventional OOP. In Figure 7.1, we present the *openfile* typestate structure. Internally, SCOOP implements each typestate structure as a "state class" that allows the declaration and creation of an object instance of a typestate structure.

We illustrate the memento design pattern with the SCOOP language with reference to the `ImageFile` example given in Figure 6.8. The client-side SCOOP code snippet below creates an object `im_fl` that serves as an equivalent to the memento object of the memento design pattern in conventional OOP.

```
Imagefile:[openfile] im_fl_saved = new Imagefile:[openfile];
```

The above client-side code snippet creates an object of the *openfile* typestate which is valid in our proposed SCOOP language because, internal to the language, each typestate is represented by a distinct "state class".

The code snippet below extracts the *openfile* typestate of the object.

```
    im_fl_saved = imfl.openfile;
```

The above statement copies the *openfile* typestate data members to the `im_fl_saved` object. The code snippet below closes the already opened file, opens another file, and loads its image.

```
imfl.close();
imfl.open("d:\imfl1.bmp");
```

Now the `imfl` contains the image of the newly opened `ImageFile`.

The code snippet below restores the original state of the image from the `im_fl_saved` object.

```
imfl.openfile = im_fl_saved;
```

6.9 Conclusion

In this chapter, we investigate how typestate invariants can be used to specify the properties of an object. Typestate invariants allow the synchronization of the internal and external typestates of an object. A modification in the value of the bound internal typestate invariants is verified by the stated type system when the object changes its typestate and vice versa. The implementation mechanism for typestate invariants presented in this chapter is compatible with subtyping and typestate extension simultaneously. Our technique also provides a default memento pattern from the language.

Chapter 7

7 Typestate Based Dynamic Compositional Adaptation

The direct support of dynamic behavior by a programming language is called “dynamic compositional adaptation” [41]. Context-oriented programming [37] and aspect-oriented programming [59] have been exploited for dynamic behavior adaptation in autonomic computing [33]. We propose that dynamic compositional adaptation can be realized by the dynamic replacement of partial behaviors of software objects, where the behavior specific to a given state or typestate of an object is replaced at run time. Meaning, a specific typestate structure of an object is replaced with a new typestate structure at run time. Such dynamic composition is referred to as typestate-based dynamic compositional adaptation. In this chapter, we investigate typestate-based dynamic compositional adaptation using SCOOP. We also present algorithms, in Section 7.4, to replace the typestate specific part of a software object.

7.1 Introduction

Dynamic Software Update (DSU) [51] is a desired feature in some contexts of software engineering. In the context of autonomic computing [33, 34], DSU is referred to as dynamic behavior adaptation [35, 40] and it is one of the well-stated requirements of autonomic computing. DSU requires software to adapt new functional or nonfunctional features at run time, i.e. to update the code without restarting or recompiling the software. The dynamic adaptation of the software assists in the self-configuration, self-healing, self-management, self-optimization and self-protection requirements of autonomic computing.

We propose dynamic compositional adaptation by dynamically replacing typestates of objects. A stated object allows some of its methods (behaviors) to be accessible regardless of its current typestate, but some methods (behaviors) are particular to a specific typestate and are accessible only if the object is currently in that typestate. The set of methods (behavior) particular to the typestate of an object represents a partial behavior of that object. Therefore, the complete behavior of a stated object is distributed among many partial behaviors. SCOOP allows dynamic replacement of partial behaviors such that only typestate specific partial behavior of an object is replaced, rather than replacing the entire object. Typestate-based compositional adaptation of objects is desirable in many software applications, as mentioned in [41, 43]. We believe that typestate-based dynamic compositional adaptation benefits mostly the self-healing and self-configuration aspects of autonomic computing.

The implementation of typestate replacement depends mainly on the internal architecture of the objects. In our setting, the organization of typestates in stated objects is based on our architecture, which is explained in Section 5.10. We argue that our proposed ‘proxy and state class’ architecture allows typestate swapping at run time as easily as it would be performed by the algorithms presented in Section 7.4.

7.2 Typestate as Context

Context-aware adaptation is also one of the desired and well-studied functionalities of software [50]. We argue that a specific typestate of an object, say ‘O’, can be viewed as the context for other objects that interact with the object ‘O’. A transition of a typestate of object ‘O’ reflects a change in context for the other objects. Therefore, our proposed typestate-based dynamic compositional adaptation serves the purpose of context-aware adaptation.

There are many programming scenarios that require software to be able to adapt to a varying context. Such a scenario can typically occur in dynamic wireless network conditions, TCP network congestion, fault tolerant components, air traffic control and life-support systems where the cost and safety of application restart can be prohibitive. Typestate-based dynamic compositional adaptation overcomes the cost of application restarts whenever typestate-associated behavior adaptation or context-associated behavior adaptation is desired.

Built-in support for typestate-based dynamic adaptation of software objects according to their changing context allows for easier means to achieve dynamic adaptation. Let us consider a simple cryptography scenario for a computer network. A ‘message’ is sent over a network through a bus in its *encrypted* state. As long as the ‘message’ is passing across the network, it is supposed to be in the *encrypted* state. However, as soon as the ‘message’ is received at the terminal, its environment or context has changed from bus to terminal and requires the ‘message’ to transition to its *decrypted* state. The functional or behavioral aspect of the decryption of a ‘message’ may need to adapt depending on the terminal where it is received. Our proposed stated object not only has the built-in capability to transition to its *decrypted* typestate, but it can also adapt to a new decryption functionality according to the kind of terminal where it is received.

7.3 Typestate as a First Class Language Concept

In Figure 7.1, we illustrate the typestate structure from Figure 6.8. As already mentioned, we propose that in addition to the object, the object’s typestate is also a first class language concept in SCOOP. Therefore, typestate structure can be passed to a function as a parameter, it can be received as a function argument and it can also be returned from a function. This implies that a stated object can dynamically replace its existing typestate (i.e. typestate structure) with an entirely new typestate structure. However, the new typestate structure must be compatible with the previous “typestate object” that adapts to it. The new typestate structure is assumed to be compatible with the previous one if it retains the same interface as that of the previous one. This compatibility is required so that all interdependent objects keep working as before.

For instance, suppose we are creating an image application. An `imagefile` object, as in Figure 6.8, may need to read an image from a device, e.g. a scanner or a camera. The programmer writes the `read()` function of the *openfile* typestate of the `ImageFile` object, which can read the image from these devices that were known at the time of compiling the `ImageFile` object. After the `ImageFile` has been compiled and executed, a new device is connected to the system. The `read()` mechanism to read an image from the newly connected device is different from the `read()` mechanism with which the `ImageFile` was initially compiled. Therefore, an `ImageFile` object may need to dynamically adapt a new `read()` method so that it can read an image from a newly connected device that was unknown at compile time. In particular, the `ImageFile` object needs to dynamically adapt the new `read()` method without recompiling. We propose that a separately compiled new `read()` method replaces the earlier `read()` method of the `ImageFile` object. Since `read()` is a behavior particular to the *openfile* typestate of the `ImageFile` object, it is very likely that the related data, or any other function of the *openfile* typestate also need to adapt with the newly replaced `read()` method. Therefore, the behavior that is associated with only the *openfile* typestate of the `ImageFile` needs to be dynamically replaced or adapted. The programmer-defined "typestate structure" of the *openfile* typestate of the `ImageFile` taken from Figure 6.8, which is intended to be replaced, is shown in Figure 7.1.

```
[openfile]{
 string image;
 public override read()[openfile]  {}
 public override display()[openfile]{
        print("image of openfile state");
                                   }
 public override close()[closefile]{..this.trans (closefile);..}

}
```

Figure 7.1 *Openfile* Typestate Structure

In SCOOP, each typestate structure defined by the programmer is translated to a specific kind of state class that is hidden from the programmer. The instances of each state class are called typestate objects.

7.4 Architecture

We propose that at run time a stated object allows the replacement of the partial behavior associated with any of its typestates other than its current typestate. Replacing the typestate associated behavior will depend mainly on the architecture with which the typestate is organized inside the stated object.

As we propose, a stated object internally implements each of its typestate structures as a distinct and lightweight “typestate object”. This lightweight “typestate object”, shown in Figure 5.2, is hidden from the programmer.

We need to maintain typestate associated behavior in typestate objects, so that the stated object that encapsulates all typestate objects does not become cumbersome in the memory. There are as many internal typestate objects as the number of typestates of a stated object. Therefore, the problem of dynamically modifying typestate-associated behavior reduces to replace the existing “typestate object” reference with the new “typestate object” reference. All existing and new typestate objects are extended (inherited) by a built-in ‘typestate interface’ provided by the compiler, so that they are type compatible. We investigate the minimum attributes required for the typestate associated reference object so that the stated object can be as lightweight as possible. The overall swapping of the original typestate object with a new typestate object is implemented by the four basic algorithms below.

- Swap the new “state class” with the original “state class”, as discussed in Section 7.4.1.
- If the original “typestate object” is an instance of the “state subclass”, adjust the new “typestate object” to the “state subclass” hierarchy, as discussed in Section 7.4.2.
- Swap the new “typestate object” with the original “typestate object”, as discussed in Section 7.4.3.
- Adjust the new typestate object entry in the symbol table, as discussed in Section 7.4.4.

7.4.1 Dynamic State Class Swapping

In order to replace an existing “typestate object” with a new “typestate object”, we first need to swap the state class corresponding to the existing typestate object with the state class of the new typestate object.

We introduce a `swap_stateclass (Type typeNew, String replaceWith)` algorithm, in Figure 7.3, that performs all the checks and operations to convert a new standalone state class, namely `typeNew`, into the existing nested state class of the proxy stated class. In order to implement this algorithm, a reflection technique to dynamically load a standalone class is used. The argument `typeNew` is the new standalone state class. The second argument `replaceWith` is the typestate name of the original state class that needs to be swapped with `typeNew`. This way of dynamically loading a class, instantiating its instance, and invoking its functions is possible in modern OO languages like C++, Java and C# using the reflection technique [45]. For instance, in C#, a standalone class, say `new_cls`, from a standalone assembly file, say `asm`, can be loaded, as `tp`, in a program using the code in Figure 7.2:

```
String new_cls;       //name of the new class to load
Assembly asm;
String type;          //fully qualified name of the new class to be loaded
Type tp;              // this will hold the new loaded class
String asm_path;      //path of assembly file that contains the standalone class
asm=Assembly.LoadFile(@asm_path);
type= asm.toString + “.” + new_cls;
tp= asm.GetType(type);
```

Figure 7.2 Dynamic Loading by Reflection

The above code fragment can be used to load a standalone class, as in the algorithm shown in Figure 7.3. Once a separately compiled standalone class has been loaded into memory then its instance can be created in the usual way.

```
swap_stateclass (Type typeNew, String replaceWith)
Type cls;
original_cls= classOf(replaceWith);//class of 'replaceWith' typestate is
                                   //returned
if (typeNew.IsClass) AND interfaceOf(typeNew)==
interfaceOf(original_cls) then
   o  load the 'typeNew' class using reflection technique.
   o  replace the 'original_cls' class with the 'typeNew' class.
   if the 'original_cls' class has state subclass(s) then
      o  make the 'typeNew' class the parent of all these state
         subclass(s).
   end if
end if
end swap_stateclass
```

Figure 7.3 Swap State Class Algorithm

7.4.2 Checking for State Subclass

It is possible that the new state class, i.e. `typeNew`, has to be replaced with the existing state class, i.e. `exst_cls`, which is a state subclass, i.e. an extension of a typestate. For instance as in Figure 6.8, the `exst_cls` class, i.e. `eof`, is an extension of the *openfile* typestate. Therefore, in the code of Figure 5.2, it is a subclass of the `openfile` state class. The new state class `typeNew` may need to replace the `eof` state subclass. In this case, the algorithm, shown in Figure 7.4, is invoked to dynamically transform a standalone state class `typeNew` into a state subclass.

```
check_state_subclass(Type typeNew ,Type exst_cls)
  if the existing state class ‘exst_cls’ is the state
  subclass then
    o ‘typeNew’ class should already have overridden
      methods of all virtual methods of the parent class
      corresponding to the its parent typestate.
    o vtable of ‘typeNew’ class will already have entries
      for any of its virtual, non virtual and overridden
      methods.
    o create a new vtable by copying the vtable from
      ‘exst_cls’ state subclass.
    o replace virtual method entries of newly created
      vtable by the corresponding entries of overridden
      methods from vtable of standalone class.
    o Append all other virtual and non-virtual method
      entries of ‘typeNew’ vtable into the newly created
      vtabel.
    o replace the original vtable of ‘typeNew’ class by
      newly created vtable.
  end if
end check_state_subclass
```

Figure 7.4 Check State Subclass Algorithm

7.4.3 Dynamic Typestate Object Swapping

We put forward the algorithm, `swap_tstate_object`, in Figure 7.5, to dynamically swap a new typestate object instance of a new state class with an existing typestate object instance. The existing typestate objects are instances of nested state classes encapsulated within the stated class. Each stated class, as given in Section 5.7, contains the referent of each existing typestate object. The new typestate object remains a standalone typestate object until it replaces the existing typestate object. The algorithm `swap_tstate_object` validates all preliminaries while replacing a new typestate object with an existing typestate object. The new typestate object is an instance of its corresponding state class.

```
swap_tstate_object ()
    o create a typestate object of newly replaced state
      class.
    o assign the state data of the original typestate
      object to the new typestate object using built-in
      memento pattern.
    o Assign the newly created typestate object to its
      referent declared in the encapsulating stated class.
end swap_tstate_object
```

Figure 7.5 Swap Typestate Object Algorithm

This algorithm exploits SCOOP's built-in memento design pattern, given in Section 6.8.2, so that the new typestate object retains the same state as that of the original typestate object. The last step of this algorithm invokes the algorithm `adjust_sym_table`, described in Section 7.4.4, to adjust the corresponding entry in the symbol table for the new "typestate object".

7.4.4 Dynamic Name-to-Object Binding

Name-to-Object binding is the mechanism by which a name (object reference) is assigned the address of an object instance allocated in the heap. The set of object instance types, which can be assigned to a name or object reference, is known at compile time. This set of object instance types includes either the instance of the declared type of the object reference or its subclasses. This name-to-object binding mechanism is dynamic in case an instance of subclass is assigned to a name (referent) of the base class. This is because such a binding can only be resolved at runtime as the compiler cannot be sure of what subtype object instance this name of the base type would be pointing to at runtime. However, as we have mentioned in Section 5.9, an instance of a "state class" is compatible with a "stated object" and therefore may be assigned to the referent of the stated object. We propose that a fully and entirely standalone state class be loaded at run time and then transformed into a nested class of the stated class.

In a language with static scoping, the compiler can use an ‘insert’ operation to place a name-to-object or reference-to-instance binding into the symbol table. This binding makes up the referencing environment [32]. We propose a “dynamic name-to-object binding” mechanism which allows assigning at run time an object instance of an entirely new class to the name (object reference) of the declared type. However, the mechanism requires that the new state class type be compatible with the declared type of the object reference. “Dynamic name-to-object binding” works similarly to “dynamic method binding”. Dynamic method binding is achieved by replacing a new method’s address in the virtual method table (vtable) of an object. Similarly, ‘dynamic name-to-object binding’ is achieved by replacing, at run time, the address of a new typestate object instance in the symbol table for static scoping and in the central reference table for dynamic scoping. This new typestate object instance is created from an entirely new state class and replaces the address of the previous typestate object instance in the symbol table in order to establish the new binding of name (object reference) with newly created typestate object instance. Therefore, such a new binding can only be resolved at runtime as the compiler cannot be sure of the object instance this name (object reference) would be pointing to at runtime. The SCOOP compiler invokes the algorithm `adjust_sym_table` of Figure 7.6 to adjust the address of the new typestate object instance in the symbol table. This algorithm is invoked from the last step of the algorithm given in Figure 7.4.

```
adjust_sym_table(prevObj, newObj)
    int add;
    add=lookup(prevObj)        //the address of symbol table entry for prevObj is
                               //looked up in symbol table and returned
    if typeof(prevObj)== typeof(newObj) then
        insertAt(add,newObj)   //newObj name is replaced with
                               //prevObj name in the symbol table
    else
            return error       // prevObj and newObj are not compatible
    end if
end adjust_sym_table
```

Figure 7.6 Adjust Symbol Table Algorithm

7.5 MAPE-K Loop

Dynamic behavior adaptation [35, 36, 37, 38, 39, 40] is one of the well-stated requirements of autonomic computing [33, 34]. The so-called MAPE-K loop, Figure 7.7, of autonomic computing defines an abstract architecture to achieve dynamic behavior adaptation [33, 34]. Different approaches have been proposed for capturing the sensor and effector requirements of the MAPE-K loop, such as layers in context oriented programming [37] and refraction/transmutation in Adaptive Java in [42, 52]. We exploit typestate and typestate-based polymorphism as a natural fit for the sensor and effector requirements, respectively. According to our proposed approach, dynamic behavior adaptation is achieved by swapping the typestates at run time. Typestate based polymorphism may be more flexible if a typestate is dynamically adapted.

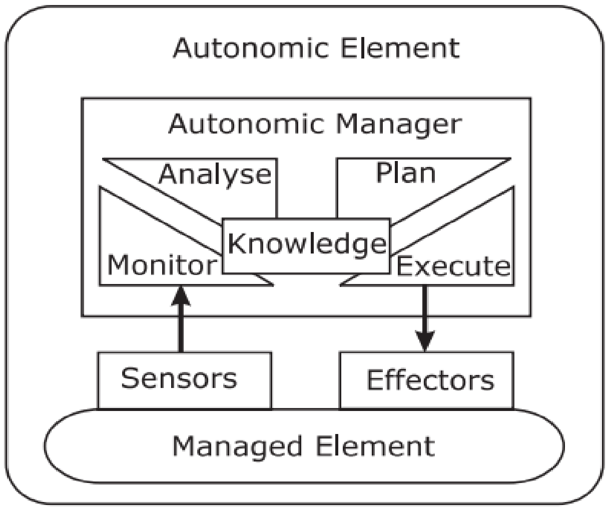


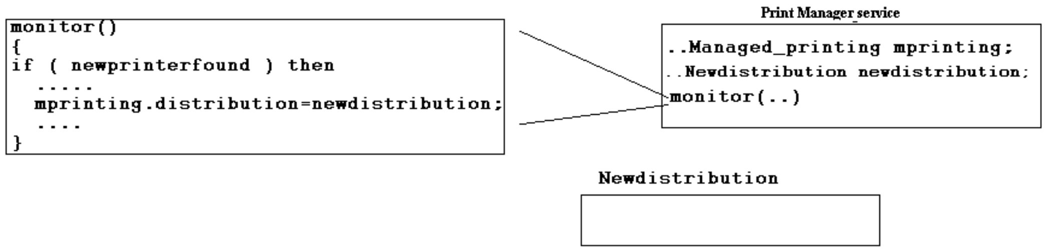
Figure 7.7 IBM's MAPE-K Loop

7.5.1 A Case Study

We return to the online banking application case study first presented in Section 3.10. In this example, the number of connected clients will determine exactly what typestate the 'managed printing service' will assume. If the number of clients increases beyond a set threshold, the 'managed printing service' transitions to the *distribution* typestate, otherwise it transitions or stays in the *streaming* typestate. The number of connected clients is used as an internal typestate invariant bound with the external typestate through a binding rule.

We show the 'print manager service' in Figure 7.8 that corresponds to the autonomic manager of the MAPE-K loop of Figure 7.7. In Figure 7.8, we also illustrate a standalone *newdistribution* typestate. The 'print manager service' loads the *newdistribution* typestate object using the code given in Figure 7.2. The 'monitor' function of the 'print manager service' periodically analyzes the relevant information and plans to trigger a typestate adaptation to the 'managed printing service'. The events received by the 'print manager' serve as the 'sensors' of the MAPE-K loop. An example of such an event is when a new support printer is deployed in the system. This 'monitor' function is invoked according to a policy defined by the 'print manager'. This policy includes the monitoring, analysis, planning and execution phases of the MAPE-K loop (that is beyond our scope at present). For the sake of simplicity, we assume that the ‘monitor’ function is invoked by an event.

If the administrator decides to deploy an additional support printer to the application, the ‘print’ behavior of the *distribution* typestate must realize the existence of a new support printer at run time. In such a circumstance, the ‘managed printing service’ can dynamically adapt to a standalone *newdistribution* typestate object encapsulating the new ‘print’ behavior that distributes print requests among all printers.



```
monitor()
{
if ( newprinterfound ) then
  .....
  mprinting.distribution=newdistribution;
  ....
}
```

```
..Managed_printing mprinting;
..Newdistribution newdistribution;
monitor(..)
```

Figure 7.8 Printing Manager Service

The ‘print manager service’ may keep a reference to the ‘managed printing service’, i.e. `mprinting`. As shown in Figure 7.8, the ‘print manager service’ triggers the following statement:

```
mprinting.distribution=newdistribution;
```

The above statement replaces the existing *distribution* typestate of `mprinting` with the *newdistribution* typestate. This statement internally uses the four algorithms, given in Section 7.5, to ensure seamless typestate-based dynamic adaptation. Furthermore, this statement serves as an ‘effector’ of the MAPE-K loop.

We show in Figure 7.9 that the previous *distribution* typestate object of the ‘managed printing service’ is replaced with a *newdistribution* typestate object.

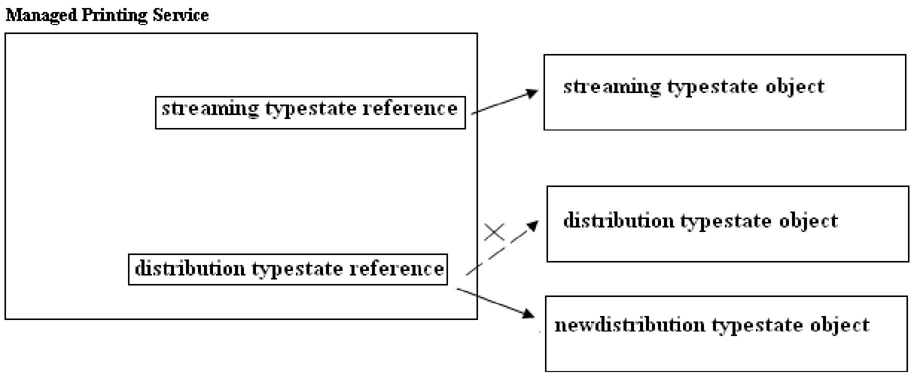

Managed Printing Service
streaming typestate reference
distribution typestate reference
streaming typestate object
distribution typestate object
newdistribution typestate object

Figure 7.9 Managed Printing Service

7.6 Conclusion

We have shown how the typestates of an object can be exploited for dynamic behavior adaptation. Furthermore, typestate is also an intuitive requirement for the dynamic adaptation of software components. The case study examined integrates the use of typestate for dynamic behavior adaptation.

Chapter 8

8 Model Checking

Model checking is usually applied at the software design level rather than at the software source code level, which is at least partially due to the fact that source code does not have any representation suitable for model checking. There are very few approaches [58] to model checking the source code of conventional object oriented programs. We observe that a program may contain some statements that violate the intended design of its model. For example, a program may have statements that violate the object protocol. Such statements are bugs that may be caught by model checking tools. However, the inability to model check the source code results in the inability to catch these kinds of bugs. In order to catch these kinds of bugs in the source code through model checking, we require the source code to have an abstraction with a suitable representation for model checking. We argue that a state controlled object oriented language allows specifying the object behavior by abstracting the state of the object. Transitions among the states or typestates can also be specified.

8.1 Model Checking SCOOP

Imperative programming languages, including OOP, are more useful in terms of describing software systems with operations, flow of operations, and performing operations on data. Formal verification tools are purpose-built to specify simple state machine representations of a system [55]. SCOOP combines both the simplicity of specifying a state machine representation of a software object and the strength of OOP.

State controlled object oriented source code exhibits a representation of objects that can be directly captured by formal verification tools, such as NuSMV [57], for model checking. Conventional object oriented language lacks such a representation of objects. Case study 3.14

of the `File` stated object is described in the SCOOP source code in Figure 5.1 and can be read conveniently as a finite state model as shown in Figure 3.7.

Mbeddr [56] is a tool that allows specifying the state machine representation of software objects. However, mbeddr [56] source code lacks the strength of OOP. In mbeddr, a programmer writes the source code of an object in the form of a state machine using mbeddr [56]. Mbeddr [56] can directly translate the programmer defined source code to another source code that is understandable as a model by NuSMV [57].

In this chapter, we compare SCOOP code with that of mbeddr. The description of objects, i.e. in the form of a state machine, written in mbeddr is quite analogous to the SCOOP source code. Therefore, we assume that SCOOP source code can also be directly translated to the source code that is understandable by NuSMV as a model.

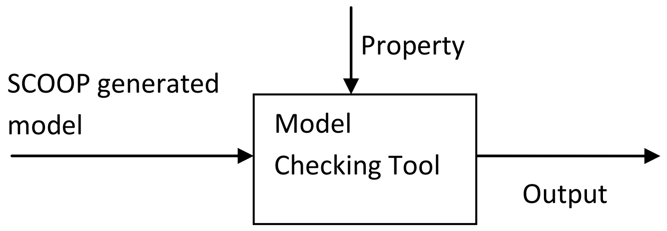


Figure 8.1 Model Checking SCOOP

Figure 8.1 demonstrates a model checking tool, e.g. NuSMV, that can input a SCOOP generated model and a property to check whether the SCOOP software object complies with the property.

8.2 Symbolic Model Checking

As discussed in [55], there are many properties that can be checked on the state machine description of a software object. The checking of a property over a state machine model verifies whether the property holds true for that model. The model checker checks the property in every possible system run. If a required property holds true for a given model then it assures that the model correctly holds that property. Otherwise, the model does not correctly represent the required property and therefore should be rectified. All these properties may be divided into two categories i.e. automatic checking and manual checking. In the following sections, we illustrate properties of each of these categories that can also be checked for the SCOOP-generated model.

8.3 Automatic Checking

Some properties can be checked by the model checking tool itself as default properties. We therefore refer to such property checking as "automatic checking". Below we discuss one such property which checks whether all states of the model are reachable.

The Reachability Property

The "reachability property" checks whether all states of a model are reachable. We illustrate the computation tree logic formula, CTL [54], to represent the reachability property from [55] for the *openfile* state of the 'File' stated object model as below.

```
SPEC  AG  _current_state!= openfile
```

The above property written in CTL formulae can be defined in the following words.

"In all possible system runs, it is true in every state that
none of these each states is the *openfile* state".

If the NuSMV finds the above property to be False for the `File` stated object model, it implies that the *openfile* state is reachable, hence it is a success scenario.
If the NuSMV finds the above property to be True for the `File` stated object model, it implies that the *openfile* state is not reachable and hence it is not a success scenario.

8.4 Manual Checking

There may be many custom properties that a programmer intends to check for his or her model using temporal logic formulae [54]. One of these properties is discussed below.

The 'P is false before R Property'

This property, from [55], checks whether a condition P can be true before R. We illustrate this property in connection with the model of the 'File' stated object discussed in Case Study 3.14. The same model is specified in Figure 5.1 and drawn in Figure 3.7.

P is false Before R.
P: file is read
R: file is in *openfile* state

Therefore, this property is evaluated as below.

"file is read" is false Before "file is in *openfile* state"

The above property says that the stated File object should not be read unless it is in the *openfile* state. As long as this property is true for the File stated object model, the model is correct for that property. Otherwise, the model attempts to read the File that is not in the *openfile* state. Clearly these kinds of bugs introduced in the software are a violation of object protocol.

8.5 Conclusion

Software testing and analysis of programs performed by a programming language includes static analysis to detect any violation of object protocols. In this chapter, we demonstrated that the capability of SCOOP to specify object protocols can be exploited to detect violations of these protocols statically through model checking.

Chapter 9

9 Conclusion and Future Work

9.1 Conclusion

This thesis introduces applications of a particular formal method in software engineering. We have investigated multiple aspects of integrating finite state machines into an object oriented language, creating a new style we call “state controlled object oriented programming”, which we abbreviate as SCOOP. Situations arise in many software settings where objects are based either implicitly or explicitly on finite state machines. When the machine's state is encoded in ordinary programming variables, it obscures the natural abstractions. Through SCOOP, it is possible for programs to be easier to understand by people and by software tools.

The notion of typestate checking is used to enforce object protocol, taking into account object state. There are basically two types of approaches that are used to capture the typestate of an object. One approach is to perform an analysis of the programs that are coded in a conventional imperative programming language. This analysis deduces the typestate information of the objects and uses this information to perform typestate checking. Another approach, the one we have advocated, is to write programs in a programming language with explicit support of typestate so that programs inherently capture the typestate information. In this approach, the programming language can directly perform typestate checking.

Static typestate checking cannot absolutely find the current typestate of an object because in some cases the current typestate of the object cannot be determined until run time. We have shown how the typestate information can be used for dynamic typestate checking as well as for dynamic form of polymorphism.

Our proposed language does not require a programmer to learn and use any extra annotations except typestate annotations and does not impose any constraints on aliasing.

One feature of SCOOP is a typestate checking technique that enforces an object protocol consistent with the internal state of objects. An important aspect of typestate is that the external and internal representations of typestate can be synchronized. We introduce an implementation technique to perform this operation effectively. An implication of this implementation technique is that SCOOP provides the “memento” design pattern by default.

We further propose to exploit typestate for dynamic compositional adaptation, allowing an object to effectively and dynamically adapt its typestate-related partial behavior. By doing so, we propose that typestate can also serve autonomic computing. Typestate adaptation without state preservation may not be very helpful in practical scenarios. We have shown that our typestate adaptation technique can exploit the default “memento” feature for easier state preservation. Since we are not only exploring novel techniques but also drawing a synergy between them, a realm of related implementation research exists. The improvement of algorithms from the implementation viewpoint should be investigated.

We illustrate that objects with explicit support of state are more intuitive to use as both the managed component and manager component of autonomic computing. We also illustrate that the explicit support of state in the object can benefit GUI frameworks as well as data structures.

Object Protocol is the central concept of typestate checking and software validation as discussed in this thesis.

In order to obtain practical benefit of our proposed techniques, we have to allocate resources to implement the SCOOP language. Developing a new programming language is a time and resource consuming process. However, the training of the software developers to adapt SCOOP is not a challenge due to its intuitive and OOP like constructs.

9.2 Future Work

The proposed SCOOP language can be implemented with a compiler. Before or during the implementation of such a compiler, the following aspects should be further investigated for the implementation of details:

- Algorithms that can generate graphical state machine diagrams of the stated objects based on the source code. Similarly, algorithms that generate the diagrams that show the interaction between the related stated objects.
- According to the proposed ‘proxy and state class’ technique, the representation of the “state class” and the lightweight “typestate object” for the optimized implementation of the typestate of stated object can be explored. For instance, as in [54], the ‘state’ of the program is proposed to be implemented symbolically rather than explicitly with the help of binary decision diagrams to effectively combat the state explosion problem.
- The proposed architecture in Section 5.15 identifies certain statements of the SCOOP code that create aliases of the stated objects. Optimized analyses similar to “pointer analysis” algorithms can be investigated that efficiently find these statements that create aliases.
- In Figure 6.1, we show that the SCOOP language can deduce the conditional transition based on the post-condition of the method and typestate invariant binding rule. We can investigate further to find an algorithm that analyze the programmer defined source code and deduce the check points in the code where such conditional transitions should be implemented by the SCOOP language.

10 Reference

[1] Timothy Budd. An Introduction to Object-Oriented Programming. (3rd Edition).

[2] Martin Abadi and Luca Cardlli. A Theory of Objects. Springer, Chapter 6, pp 57–76. Chapter 11, pp 141–152. 1996.

[3] Samek Miro. Portable Inheritance and Polymorphism in C. Embedded Systems Programming, Volume 10. pp 54–67, March 1997.

[4] Nels E. Beckman, Duri Kim, Jonathan Aldrich. An Empirical Study of Object Protocols in the Wild. Proceedings of the 25th European Conference on Object-Oriented Programming, Lancaster, UK. pp 2–26, July 2011.

[5] Y. Yang, A. Gringauze, D. Wu, H. Rohde, J. Yang. Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis. Tech. Rep. MSR-TR-2008-108, Microsoft Research, Aug. 2008.

[6] Stephen M. Watt. A Technique for Generic Iteration and its optimization. In Proceedings of the ACM SIGPLAN Workshop on Generic Programming. pp 76–86. 2006.

[7] Girish Keshav Palshikar. An Introduction to Model Checking. http://www.embedded.com/design/embedded/4024929/An-introduction-to-model-checking
[Valid on 19 March, 2014]

[8] William Stalling. Operating System-Internals and Design Principles. 7th Edition. 2011.

[9] Linda Null, Julia Lobur. The Essentials of Computer Organization and Architecture. Third Edition. pp 146–157. 2012

[10] Miro Samek, Paul Montgomery. State-Oriented Programming. Embedded Systems Programming. pp 22–43, Aug, 2000.

[11] Basic Object Oriented Programming and State Machines. pp 1-15.
http://courses.csail.mit.edu/6.01/spring08/handouts/week3/week3-notes.pdf
[Valid on 19 March, 2014]

[12] Walter Cazzola, Ahmed Ghoneim, Gunter Saake. Software Evolution Through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, Objects, Agents and Features: Structuring Mechanisms for Contemporary Software, Lecture Notes in Computer Science 2975, pp 69–84. Springer, July 2004.
http://link.springer.com/content/pdf/10.1007%2F978-3-540-25930-5_5.pdf

[13] Jamil Ahmed, Sheng Yu. Adding Autonomy into Object. In Proceedings of the International Conference on the Foundation of Computer Science. pp 136–141. 2011. http://world-comp.org/p2011/FCS4822.pdf
[Valid on 19 March, 2014]

[14] Patrick Lam,Viktor Kuncak , Martin Rinard. Generalized Typestate Checking for Data Structure Consistency. In proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation. pp 430–447. 2005.

[15] David Harel. Statecharts: A Visual Formalism For Complex Systems. In Science of Computer Programming. pp 231–274, 1987.

[16] Erich Gamma et al. Design Patterns–Elements of Reusable Object-Oriented Software. Addison-Wesley. pp 316–325. 1995.

[17] Haitong Wu, Sheng Yu. Adding State into Object. In Proceedings of the International Conference on Programming Languages and Compilers, pp 101–107. 2005.

[18] Jonathan Aldrich, Joshua Sunshine. Typestate-Oriented Programming. In Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, pp 1015–1022. ACM, 2009.

[19] H. Xu, Sheng Yu. Type Theory and Language Constructs for Objects with States. Electronic Notes in Theoretical Computer Science,135(3). pp 141–151, 2006.

[20] Robert DeLine and Manuel Fähndrich. Typestates for Objects. In European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 3086, pp. 465–490. Springer-Verlag (2004).

[21] Asher Sterkin. State-Oriented Programming. In Proceedings of Multiparadigm Programming with Object-Oriented Languages. pp 1–32. 2008.

[22] Kevin Bierhoff and Jonathan Aldrich. Modular Typestate Checking of Aliased Objects. In Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). pp 301–320, October 2007.

[23] Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. IEEE Transactions on Software Engineering, VOL. SE-12, NO.1. pp 157–171, Jan 1986.

[24] R. Wolff, R. Garcia, Eric Tanter, and J. Aldrich. Gradual Typestate. In Proceeding of the 25th European Conference on Object-Oriented Programming (ECOOP). pp 459–483. July 2011.

[25] Sheng Yu. Class-is-Type is Inadequate for Object Reuse. ACM SIGPLAN Notices, v.36 n.6. pp 50–59, June 2001.

[26] Stephen Fink , Eran Yahav , Nurit Dor , G. Ramalingam , Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. In Proceedings of the 2006 International Symposium on Software Testing and Analysis, Portland, USA. pp 133–144. July 2006.

[27] Robert DeLine, Manuel Fahndrich. Enforcing High-Level Protocols in Low-Level Software. In Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation (PLDI). pp 59–69. May 2001.

[28] Kevin Bierhoff, Jonathan Aldrich. Lightweight Object Specification with Typestates. In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering. pp 217–226. Sep 2005.

[29] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, Wolfram Schulte. Verification of Object-Oriented Programs with Invariants. In Journal of Object Technology, vol. 3, no. 6, Special issue: ECOOP 2003 Workshop on FTfJP, pp. 27–56. June 2004.

[30] Manuel Fahndrich, Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), Germany. pp. 13–24, May 2002.

[31] Viktor Kuncak, Patrick Lam, Martin Rinard. Role Analysis. In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL). pp 17–32. Jan 2002.

[32] Michael L. Scott. Programming Language Pragmatics. Third Edition.

[33] IBM. An Architectural Blueprint for Autonomic Computing. Technical Report, 2003. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf [Valid on 19 March, 2014]

[34] Kephart, J. O. Research Challenges of Autonomic Computing. In Proceedings of the 27th international conference on Software engineering (ICSE). pp 15–22. May 2005.

[35] Klein C, Schmid R, Leuxner C, Sitou W, Spanfelner B. A Survey of Context Adaptation in Autonomic Computing. In Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems (ICAS). pp 106–111. March 2008.

[36] Jorge Fox , Siobhán Clarke. Exploring Approaches to Dynamic Adaptation. Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction. pp 19–24, Lisbon, Portugal. June 2009.

[37] Guido Salvaneschi, Carlo Ghezzi, Matteo Pradella. Context-Oriented Programming: A Programming Paradigm for Autonomic Systems. In Journal the Computing Research Repository (CoRR). May 2011.

[38] Marc Schanne, Tom Gelhausen, Walter F. Tichy. Adding Autonomic Functionality to Object-Oriented Applications. In Proceedings of the 14th International Workshop on Database and Expert Systems Applications. pp 725, September 2003.

[39] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM). pp 649. October 2002.

[40] E. P. Kasten, P. K. Mckinley, S. M. Sadjadi, R. E. K. Stirewalt. Separating Introspection and Intercession to Support Metamorphic Distributed Systems. In Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), pp 465–472. July 2002.

[41] Kasten, E.P, McKinley, P.K. Perimorph: Run-Time Composition and State Management for Adaptive Systems. In Proceedings of the 24th International Conference on Distributed Computing Systems Workshops, pp 332–337, March 2004.

[42] Philip K. McKinley, Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing Adaptive Software. In Journal Computer,volume 37, pp 56–64, July 2004.

[43] Christopher M. Hayden, Edward K. Smith, Michael Hicks, Jeffrey S. Foster. State Transfer for Clear and Efficient Runtime Upgrades. In Proceedings of the 27th International Conference on Data Engineering Workshops. p.179–184. April 2011.

[44] Erich Gamma , Richard Helm , Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. pp 283.

[45] http://msdn.microsoft.com/en-us/library/vstudio/f7ykdhsy(v=vs.100).aspx;
[Valid on 19 March, 2014]
http://msdn.microsoft.com/en-us/library/ms227224(v=vs.80).aspx.
[Valid on 19 March, 2014]

[46] D.Hollingsworth. The Workflow Reference Model. Workflow Management Coalition. Document number TC00-1003-Issue 1.1 edn. January 1995.

[47] Zef Hemel, Ruben Verhaaf, Eelco Visser. WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Application. In Proceeding of the Model Driven Engineering Languages and Systems (MODELS). pp 113–127. October 2008.
http://link.springer.com/chapter/10.1007%2F978-3-540-87875-9_8#page-1
[Valid on 19 March, 2014]

[48] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A Taxonomy of Compositional Adaptation. Technical Report MSU-CSE-04-17, 2004.

[49] N. Gui, V. De Florio, H. Sun and C. Blondia. ACCADA: A Framework for Continuous Context-Aware Deployment and Adaptation. In Proceeding of 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, Lyon, France, LNCS 5873, pp 325–340, Springer Verlag, November 2009.

[50] Amir Hammami, Thierry Villemur, Tom Guerout. Towards Context-aware Deployment and Reconfiguration. 22nd Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. pp 86–91. June 2013.

[51] Miedes, E., Munoz-Escoi, F.D. A Survey About Dynamic Software Updating. Tech. Rep. ITI-SIDI-2012/004, Instituto Universitario Mixto Tecnologico de Inform-atica, Universitat ´Politecnica de Valencia. 2012.
http://web.iti.upv.es/~fmunyoz/research/pdf/TR-ITI-SIDI-2012003.pdf
[Valid on 19 March, 2014]

[52] E. P. Kasten and P. K. McKinley. Adaptive Java: Refractive and Transmutative Support for Adaptive Software. Tech. Rep MSU-CSE-01-30 Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA, December 2001.

[53] M. Voelter and D. Ratiu, Language Engineering as an Enabler for Incrementally Defined Formal Analyses. In Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA). pp 9–15 June 2012.

[54] J. R. Burch , E. M. Clarke , K. L. McMillan , D. L. Dill , L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In Journal Information and Computation, v.98 n.2, pp 142–170. June 1992.

[55] Christoph Rosenberger. Model Checking for State Machine with mbeddr and NuSMV. HSR, Hochschule für Technik Rapperswil, 2013.
http://mbeddr.com/files/modelcheckingforstate-machineswithmbeddrandnusmv.pdf
[Valid on 19 March, 2014]

[56] mbeddr c userguide. http://mbeddr.com. [Valid on 19 March, 2014]

[57] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore and Marco Roveri. NuSMV 2.5 Tutorial.
http://nusmv.fbk.eu/NuSMV/tutorial/index.html.
[Valid on 19 March, 2014]

[58] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. In Journal Automated Software Engineering, Volume 10 Issue 2. pp 203–232. April 2003.

[59] Philip Greenwood, Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an autonomic System. In Proceeding of Dynamic Aspects Workshop (DAW04), pp 76–88. March 2004.

[60] Eric Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, South Africa. pp 5–14. May 2010.

APPENDIX A

LALR Context Free Grammar for SCOOP

Prd #	Productions	
1	<program> → <cls_list> $$	
2	<cls_list> → <cls_list> <class>	
3	<cls_list> → Є	
4	<class> → <st_dec_list> class cls_id { <cls_body> }	S
5	<st_dec_list> → [state (<st_list>)]	S
6	<st_dec_list> → Є	S
7	<st_list> → <st_list> , <st_id>	S
8	<st_list> → <st_id>	S
9	<st_id> → st_id	S
10	<cls_body> → <var_dec_list> < st_binding> <cls_m_list> <c_s_m_list>	S
11	<var_dec_list>→<var_dec_list> <var_dec> ;	
12	<var_dec_list> → Є	
13	<var_dec>→<vd_start><vd_rest>	
14	<vd_start> → <pub-prv><dt> id	
15	<dt> → int	
16	<dt> → string	
17	<dt> → cl_id	

18	<st_binding> → statebinding { <b_list> }	S
19	<b_list> → <b_list> , <bind>	S
20	<b_list> → <bind>	S
21	<bind> → st_id : (<rel_exp>)	S
22	< st_binding> → Є	S
23	<cls_m_list> → <cls_m_list> <cls_method>	
24	<cls_m_list> → Є	
25	<cls_method> → <pub-prv> <r_type> <m_id> (<arg_list>) { <m_body> }	
26	<pub-prv> → public	
27	<pub-prv>→ private	
28	<r_type> → <dt>	
29	<m_id> → m_id	
30	<arg_list> → Є	
31	<arg_list> → <args>	
32	<args>→ <args>, <arg>	
33	<args>→<arg>	
34	<arg> →<dt> id	
35	<m_body> → <var_dec_list> <stm_list>	
36	<stm_list> → <stm_list> <stm>	
37	<stm_list> → Є	
38	<stm> → <lhs> = <exp>;	
39	<stm> → read (id) ;	
40	<stm> → print <exp>;	

41 <stm> → o_id.m_id(<param_list>);

42 <vd_new> → = new ID ()

43 <stm> → if (<rel_exp>) {<stm_list> } <else>

44 <else>→ else {<stm_list>}

45 <else> → Є

46 <stm> →while(<rel_exp>){<stm_list> }

47 <stm>→for(<init><mid>;<last>) { <stm_list>}

48 <rel_exp>→<rel_exp> <rel_op> <rel_term>

49 <rel_term> →<rel_bool_fac>

50 <rel_exp> →<rel_term>

51 <rel_term> → <rel_fac><comp_op><rel_fac>

52 <rel_bool_fac> → bool_id

53 <rel_fac> → id

54 <rel_fac> → num

55 < rel_fac > → o_id.p_id

56 <rel_op> → AND

57 <rel_op> → OR

58 <comp_op> → ==

59 <comp_op> → !=

60 <comp_op> → >

61 <comp_op> → <

62 <comp_op> → <=

63	<comp_op> → >=	
64	<exp> → <term>	
65	<exp> → <exp> <add_op> <term>	
66	<term> → <factor>	
67	<term> → <term> <mult_op> <factor>	
68	<factor> → (<exp>)	
69	<factor> → id	
70	<factor> → o_id.p_id	
71	<factor> → num	
72	<add_op> → +	
73	<add_op> → -	
74	<mult_op> →*	
75	<mult_op> →/	
76	<c_s_m_list>→ Є	
77	<c_s_m_list>→<c_s_m_list> <c_s_method>	S
78	<c_s_method> → <s_list> { <s_v_d_lst><s_m_list> }	S
79	<s_m_list> →<s_m_list> <s_method>	S
80	<s_m_list> → <s_method>	S
81	<s_method>→<r_type><m_id>(<arg_list>)<s_end_list>{<s_m_body>}	S
82	<s_end_list> → [<s_e_list>]	S
83	<s_list> → <s_list> , <s_id>	S
84	<s_list> → <s_id>	S

85	<s_e_list> →<s_e_list> | st_id	S
86	<s_e_list> → st_id	S
87	<s_m_body> →<var_dec_list> <s_stm_list>	
88	<s_stm_list> →<s_stm_list> <st_stm>	
89	<s_stm_list> → Є	
90	<st_stm>→this.trans(st_id);	S
91	<st_stm>→<lhs> = <exp>;	
92	<st_stm>→ read (id) ;	
93	<st_stm>→ print <exp>;	
94	<st_stm> →o_id.m_id(<param_list>);	
95	<st_stm> →o_id.m_id(<param_list>);	
96	<st_stm>→ while(<rel_exp>){<s_stm_list> }	
97	<st_stm>→ for(<init><mid>;<last>) { <s_stm_list> }	
98	< st_stm> → if (<rel_exp>) {<s_stm_list>}<s_else>	
99	<s_else> → else {<s_stm_list>}	
100	<s_else> → Є	
101	<init>→ id = num;	
102	<mid>→<rel_exp>	
103	<last>→id <in_dec>	
104	<in_dec>→ ++	
105	<in_dec>→ --	
106	<lhs>→id	

107	<lhs>→ o_id.p_id	
108	<param_list>→<params>	
109	<params>→<params>,<factor>	
110	<params>→<factor>	
111	<param_list>→ Є	
112	<new>→ Є	
113	<cl_id>→ cl_id	
114	<s_id> → [st_id]	S
115	<s_list> → [st_id]:[st_id]	S
116	<vd_rest> → <vd_simple>	S
117	<vd_rest> →<vd_new>	S
118	<vd_simple> → <vd_simple> , id	S
119	<vd_simple> → Є	S
120	<vd_new> → Є	S
121	<s_v_d_lst> → <s_v_d_lst><s_v_d> ;	S
122	<s_v_d_lst> → Є	S
123	<s_v_d>→<s_vd_start><vd_rest>	S
124	< s_vd_start> → <dt> id	S

APPENDIX B

SCOOP vs OOP

Example 1

This is an example code for the printing object, `managed_printing`, of a banking application that prints the account statement of the account holders. The printing is carried out in either *streaming* or *distribution* state. We illustrate the code in SCOOP as well as in two possible variations in OOP. We argue that this example shows the increased readability and easier encoding of SCOOP as compare to OOP while encoding these kinds of objects.

SCOOP Managed Printing Service

```
[state(streaming=start, distribution)]
class managed_printing     //managed_printing_service
{
     [streaming]
     {
          void print_acc_statement()[streaming]{
               //send print to the main printer only
               System.out.println("print in streaming");
          }
     }
     [distribution]
     {
           void print_acc_statement()[distribution]{
               //send print to the all connected printers
               System.out.println("print in distribution");
          }
      }
}
class client{
   public static void main(String args[]){
     managed_printing  prnt_client=new managed_printing();
     prnt_client.print_acc_statement();
     prnt_client.trans(distribution);
     prnt_client.print_acc_statement();
    }
}
```

OOP Managed Printing Service **//using conditional constructs**

```
public class managed_printing {   //managed_printing_service

     private String state="";
     public void setState(String state){
          this.state=state;
     }
     public void print_acc_statement (){
          if(state.equalsIgnoreCase("streaming")){
               System.out.println("print in streaming");
          }
          else if(state.equalsIgnoreCase("distribution")){
               System.out.println("print in distribution");
          }
     }
}
class client{

     public static void main(String args[]){

        managed_printing  prnt_client=new managed_printing();
        prnt_client.setState("streaming");
        prnt_client.print_acc_statement();
        prnt_client.setState("distribution");
        prnt_client.print_acc_statement();
      }
}
```

OOP Managed Printing Service **//using state design pattern**

```
public interface State {
     public void print_acc_statement ();
}

public class streaming_state implements State {
     public void print_acc_statement() {           //override
          System.out.println("streaming");
     }
}

public class distribution_state implements State {
     public void print_acc_statement(){            //override
          System.out.println("distribution");
     }
}

public class PrintContext implements State {

     private State print_State;

     public void setState(State state) {
          this.print_State = state;
     }
     public State getState() {
          return this.print_State;
     }
     public void print_acc_statement(){
          this.print_State.print_acc_statement();
     }
}

public class client {
     public static void main(String[] args) {
        PrintContext prnt_client = new PrintContext();
        State print_streaming = new streaming_state();
        State print_distribution = new distribution_state ();
        prnt_client.setState(print_streaming);
        prnt_client.print_acc_statement();
        prnt_client.setState(print_distribution);
        prnt_client.print_acc_statement();
     }
}
```

Curriculum Vitae

Name	Jamil Ahmed	
Post-Secondary Education	Masters of Computer Science The University of Karachi, Pakistan.	06/2000–6/2002
	Bachelor of Science The University of Karachi, Pakistan.	06/1996–05/1999
Related Work Experience	Research Assistant The University of Western Ontario Canada.	01/2009–30/2014
	Teaching Assistant The University of Western Ontario Canada.	01/2012–04/2012, 01/2013–04/2013, 01/2014–04/2014
	Lecturer The University of Karachi, Pakistan.	06/2003–06/2009
	Software Engineer Softech Microsystem, Karachi, Pakistan.	06/2002–05/2003
AWARDS	PhD WGRS Scholarship The University of Western Ontario, Canada.	01/2012–04/2012, 01/2013–08/2013, 01/2014–04/2014
	PhD Scholarship The Higher Education Commission, Pakistan.	01/2009–30/2012