# Interaction, Concurrency, and OOP in the Curriculum: a Sophomore Course*

Christopher Colby    Radha Jagadeesan    Konstantin Läufer
Chandra Sekharan

Department of Math and CS, Loyola University Chicago
6525 N. Sheridan Road, Chicago, IL 60626
{colby,radha,laufer,chandra}@cs.luc.edu
http://www.cs.luc.edu/~{colby,radha,laufer,chandra}

**Abstract**

*We argue that a computer-science curriculum should introduce the principles of concurrent programming in an integrated, coherent, and application-independent fashion early in the major. We have incorporated current research into our curriculum. We describe a sophomore-level course on the fundamentals of concurrent and interactive programming that is the fruit of this work. We offered this course to about 60 students in Fall 1997 and Spring 1998.*

*Information regarding the software frameworks for the programming assignments can be obtained by contacting the authors.*

## 1 Introduction

### What is wrong?

The world is full of concurrent systems. Consider

- real-time systems
- signal-processing systems
- telecommunication systems
- graphical user interfaces
- operating systems
- multiprocessors and pipelining
- distributed algorithms

to name but a few ubiquitous examples. What all of these systems have in common is the notion of multiple subsystems that communicate with each other or with their environment.

How do these systems come into being? Computer scientists always learn the fundamentals of *sequential* computing early in their education, but rarely ever learn the fundamentals of concurrent computing. To cover concurrent systems, the typical undergraduate curriculum instead offers a series of *application-specific* courses, usually at the junior or senior level. For instance:

1. Operating Systems. For example, every part of Nachos, a popular educational operating system, uses threads to manage concurrency.

2. (Telecommunication) Networks. Concurrency is used to describe and analyze protocols at various levels of a hierarchy, such as the physical level, the data link level, and so forth.

3. Software Engineering. For instance, interactive graphical user interfaces are

*R. Jagadeesan and C. Colby were supported in part by NSF CAREER grants.

an important class of applications studied and implemented in such courses.

4. Computer Architecture. The combination of circuits is arguably the simplest and most natural example of concurrent computation. Furthermore, concurrency arises as a natural abstraction in understanding topics such as pipelining and multiprocessors.

A fixed set of application-specific courses is bound to be both an incomplete coverage of present technology and an inadequate preparation for future technology. In contrast, the typical courses in sequential computation—programming, data structures, algorithms, etc.—teach application-independent problem-solving skills and are hence far more adaptable.

The problem, as we see it, is that concurrency and interaction are not taught at a conceptual level, but at an application level. It is not taught as a means, but as an end.

## A better approach

We argue that a computer-science curriculum should introduce the principles of concurrent programming in an integrated, coherent, and application-independent fashion early in the major. We believe that this is now possible, having successfully adapted our current research to the undergraduate computer-science curriculum at Loyola University Chicago in a course called *Introduction to Concurrency*. Furthermore, we believe that students should absorb this material at an early stage of their education.

A coherent introduction to interactive and concurrent programming at an early stage offers the following advantages.

- *Programming methodology.* Interactive and concurrent computing introduces students to a powerful programming idiom and structuring methodology—the *conceptual* decomposition into modules that run concurrently. Pedagogically, we believe the right time to introduce students to these ideas is *before* the idioms and methodology of sequential programming become ingrained as the instinctive and primary way of thinking about problem solving, algorithms, and programming.
- *Specification, Testing, and Verification.* Early in their education, undergraduate students should learn about the formal specification of programs, the generation of test cases from specifications, and the formal verification of programs as an integral part of the programming process, whether concurrent or sequential. From a pedagogical point of view, the subtleties of concurrent programs such as time-dependent behaviors and nondeterminism form a pedagogical setting to emphasize these ideas.

## But how?

Most complex concurrent applications combine two ideas:

- *Reactivity.* Reactive systems are event-driven systems that interact continuously with their environment at a rate controlled by the environment. Execution in such systems proceeds as bursts of activity. In each phase, the environment stimulates the system with an input, obtains a response within a bounded amount of time, and may then be dormant for an arbitrary period of time before initiating the next burst.
- *Asynchrony.* In asynchronous systems, processes are loosely coupled and communication can take an arbitrary amount of time.

To understand these systems in a coherent and application-independent manner, we look to the body of research in concurrent programming. There are two mature and relevant strands of research into the specification and design of concurrent systems:

1. process algebras and synchronous programming languages

2. temporal logic, testing, and verification

We discuss each of these in turn to suggest how they can help achieve our pedagogical goal.

*1. Process algebras and synchronous programming languages.* The key feature of process algebras (e.g., CCS [6], CSP [4]) and synchronous programming languages (e.g., see [3]) is a notion of abstract *behavior*, which in a concurrent system is essentially the interaction of the system with its environment. Communication takes place via (labeled) events that are abstractions of names of communication channels.

Programs can be combined freely with combinators, and one need only be concerned about the desired effects on the resulting behavior. That is, combinators operate on behaviors, and the results of the combinators are behaviors. For instance:

- The parallel-composition combinator allows two (interactive) programs to run concurrently and communicate with each other. The result is a single (interactive) program that is indistinguishable from a simple one (in much the same way that an object built by object composition has the same status as a simple object). Thus, one may use parallel composition freely for the modular decomposition of designs.

- Preemption combinators allow programs to be controlled by events. For example, the watchdog combinator `DO` $P$ `WATCHING` $e$ yields a process that behaves like $P$ until event $e$ happens, upon which execution of $P$ is terminated (in the spirit of "Ctrl–C"). Analogous to exception mechanisms in traditional programming languages, the preemption combinators aid in program modularity; for example, the watchdog above avoids the pollution of $P$ with information about the event $e$. Exceptions have first class status; *any* event can be an exception and can be used in the place of $e$ in the watchdog. This allows exceptions to play an integral role in in the programming of systems. Nesting preemption operators establishes priorities on events; for example, the event $e_2$ has higher priority than the event $e_1$ in the program fragment `DO (DO` $P$ `WATCHING` $e_1$`) WATCHING` $e_2$. We note that the programming language does not constrain these priorities; rather, the priorities are determined by the program itself.

*2. Temporal logic, testing and verification.* Temporal logic augments traditional logic with modal operators that operate on the time domain (see [5] for a survey). Extensive research has yielded an ample collection of examples and specification methodologies establishing temporal logic as an expressive language for the description of properties of reactive systems. An alternative approach for specifying temporal properties of reactive systems is to use various forms of automata. Because there exists a correspondence between temporal logic formulas and automata, the two approaches are related. Temporal logic serves as a basis for automated testing and verification.

- From temporal specifications, the testing tools automatically generate finite-state machines that accept the language of input-output traces that violate the properties. These finite-state machines can be used to generate test inputs that can be fed to the actual system to determine whether or not its output violates one of the safety properties. The tools that implement these methodologies automatically alert the user to violation of properties and provide an execution trace that witnesses the violation.

- Automated verification is based on model-checking. First, one uses a formal notation to describe a high-level model of the system under design is described. Next, the verifier automatically checks if the given model satisfies correctness properties by exploring the entire state space, i.e. by exploring all possible interactions of the concurrent components. Current pragmatic experience indicates that existing tools are successful in handling real-world problems in the hardware domain.

### Our research contribution: Triveni

Triveni is a process-algebraic framework for concurrent object-oriented programming with threads and events [2, 1]. Thus, Triveni enhances the practice of threads programming with ideas from the theory of concurrency. We have implemented Triveni as a collection of tools for the Java programming language [1]. This implementation of Triveni has the following features.

- Any Java thread that uses an Observer-based interface for events can serve as a primitive Triveni process. In other words, programmers can fit existing Java code into Triveni unchanged.
- Triveni includes a specification-based testing environment that automates testing of safety properties expressed in (propositional) linear time temporal logic.

This implementation of Triveni has been used to implement and analyze a case study involving the re-implementation of a piece of telecommunication software; see [2].

### Our course

This paper describes the design and implementation of the sophomore level course *Introduction to Concurrency*. This course introduces students to process algebras and logic-based specification and testing as tools in the engineering of concurrent systems. The current implementation of Triveni in Java serves as the programming environment for the projects in the course. We taught this course at Loyola University Chicago in Fall 1997 and Spring 1998 with enrollments of about 30 students each for a total enrollment of about 60 students.

## 2 The Course

Our course uses a running theme, the game of BATTLE from [2]. Figure 1 describes the most general form of the game. For the purposes of this paper, we use BATTLE to motivate the contents of this course.

**Interaction and concurrency.** Concurrency arises in BATTLE in two ways. First, it is natural to think of the game as a concurrent composition of the players and the game controller. In fact, because this game does not enforce turns among players, the resulting true asynchrony of the players essentially forces concurrency on the program.

A second way in which concurrency arises in BATTLE is as an *abstraction* mechanism in the programming of the user interface. Concurrency offers decomposition mechanisms that support the modular design of the user-interface behavior. For instance, concurrency helps to abstract over nonessential dependencies among independent user events. In this case, the concurrency is not forced on the program, but is merely an artifice to manage complexity. However, one could argue that this is indeed the most important use of concurrency—to realize reactive behavior.

BATTLE provides simple examples of some issues that arise in concurrent programming. For example, *priorities* ensure that a submarine that has dived cannot be hit, or that player $i$ cannot receive shots from player $j$ after $j$ notifies $i$ that $j$ has aborted the game.

BATTLE also provides natural ways to explore the interaction of OOP and concurrency. For example, to ensure extensibility

BATTLE is an $n$-player variation of the 2-player board game Battleship. New players cannot join the game once it has begun. A player loses by manually aborting the game or when all his/her ships are destroyed.

**Oceans.** Each player has a collection of ships on an individual ocean grid. The $n$ ocean grids are disjoint. Each player's screen displays all $n$ oceans, but a player can see only his/her own ships. A player's ships are confined to the player's ocean.

**Ships.** Each ship occupies a rectangular sub-grid of the player's ocean and sinks after each point in its grid area has been hit. There are two kinds of ships:

1. Battleships that can move on the surface of the player's ocean.
2. Submarines that can dive, but remain at a stationary position with respect to the player ocean's surface.

**Moves.** *A player can move as fast as the user-interface/reflexes allow.* Player $i$'s moves:

1. Fire a round of ammunition on a square of another player $j$'s ocean by clicking on it. The ammunition may hit a previously unhit point on one of player $j$'s ships, in which case an **X** is displayed at that point in player $j$'s ocean on all players' screens. No information is reported in case of a miss. The **X** marks are static; when a wounded battleship moves, or a wounded submarine dives, it does not affect previously displayed **X** marks on players' screens. When a ship is sunk, its position is revealed to all players.
2. Impart a velocity to a battleship that lasts until it receives another velocity command.
3. Make a submarine dive for a game-specific interval of time.
4. Raise a shield over his/her entire ocean for a game-specific interval of time, during which player $i$'s ships are invulnerable. When a player raises an ocean-wide shield, his/her ocean becomes dim on the screens of all players. Each player has a limited supply of shields.

Figure 1: Rules of BATTLE

it is natural to build an abstract interface to model the ship widgets of the player. Inheritance arises naturally as a way to extend (concurrent) behaviors; e.g., a submarine can be implemented by adding diving capability to a generic ship.

**Program correctness.** The presence of concurrency and interaction in the BATTLE program leads to well known problems.

First, there are consistency problems associated with the concurrent access to data structures. For example, the state of each player can be impacted by several simultaneous processes such as shots from other players, ship motion, and GUI commands. In the literature, consistency of data structures is expressed by *safety properties*; the programmer desires to ensure that "something bad does not happen".

A dual issue arises in such programs concerning progress. For example, each BATTLE player must be given a chance to proceed, and the actions of each player must eventually be transmitted to the game controller and to all other players. In the literature, these are called *liveness properties*.

The aforementioned issues take on particular potency because of the potential lack of repeatability in concurrent programs. This phenomenon, termed *nondeterminism* in the literature, can arise from abstraction over the details of scheduling. For example, in BATTLE, a shot to a player can be either a hit or a miss depending on the time of delivery of the shot to a moving battleship.

### 2.1 Integration into curriculum

In the freshman year, computer science majors take CS 170: *Structured Programming* (in C++) and CS 271: *Structured Programming and Data Structures* (in C++). In the sophomore year, they take the required courses CS 272: *Data Abstraction and Object-Oriented Programming*, CS 211: *Discrete Structures*, and CS 275: *Computer Architecture*.

The prerequisites for *Introduction to Concurrency* are CS 272 and CS 211. CS 272 ensures knowledge in basic programming, data structures, algorithms, object-oriented paradigms, and basic Java. CS 211 ensures familiarity with basic logic, elementary mathematical proof techniques including induction, and rudimentary knowledge of finite-state automata.

### 2.2 Course modules

Below are the modules that compose the course. These modules are intended as a representative collection of the topics that the students will learn in the course.

1. Specification.
   - (a) Temporal logic. Safety, liveness, and bounded-response properties.
   - (b) Automata as specifications.
   - (c) Generating test cases from specifications.
2. Design of concurrent and interactive programs.
   - (a) Process algebras.
   - (b) Parallel composition as a structuring mechanism.
   - (c) Preemption mechanisms: process abortion, process suspension, interrupts.
3. Triveni. Case studies in Triveni.
4. Implementation of concurrent programming languages.
   - (a) Automata as abstractions of interaction.
   - (b) Thread programming for managing concurrency.

The instructor can package these course modules in several different ways for a one-semester course. In the first rendering of the course, we focused on the design and implementation of concurrent programs in Triveni and the design and implementation of Triveni itself (topics (2), (3), and (4)). In the second rendering, we deemphasized (4) and spent some time on the case studies and testing issues (topics (1c), (2), (3)).

### 2.3 Assignments

Our assignments were organized around the game of BATTLE.

#### Assignment 1

*Aim:* Introduction to events and event handlers.

*Sketch:* Implement a 2-player version of BATTLE with following restrictions on Player moves. The players takes turns. At each turn, a player can do one of two things: fire a shot or place a ship. If a shot hits an Opponent ship then the same player gets to shoot again. This restriction on the flow of the game removes the genuine asynchrony present in the full version of BATTLE and allows the students to implement the program without threads and with a knowledge of only AWT events.

### Assignment 2

*Aim:* Introduction to (Mealy) automata as abstractions of controllers. Implementation of combinators, including parallel composition and preemption, on finite-state automata.

*Sketch:* Implement a framework for finite-state machines with combinators. The students are provided with skeleton code for most of the classes as well as a driver for testing. The students then reimplement the controller of Assignment 1 using this framework. Apropos, this assignment introduces some design patterns. For instance, FSM transitions are equipped with actions that are realized via the Command pattern, and the Observer pattern connects the FSM with the actual AWT events.

### Assignment 3

*Aim:* A medium-scale group project with two primary goals: illustrate the use of concurrency as a decomposition tool at the time of design and implementation, and illustrate the use of logic-based specifications as assertions to aid in debugging programs.

*Sketch:* Implement a full-scale version of BATTLE. The user interface itself is written as a Triveni program to handle all possible sequences of interactions with the user. The controller is also written as a Triveni program. The students are given glue code to connect the user interface to the controller, i.e., the transmission of events from the user interface to the game controller is automatic and transparent from the student's point of view.

## 3 Conclusions

Because our course is modular in structure, in the long run we propose to make the course an integral component of our undergraduate curriculum by (a) continuing to offer the course as an elective with periodic revisions and upgrades and (b) migrating the course materials into existing courses such as CS 272 and CS 375: *Software Engineering*. We also envision a series of courses centered around the themes of the course that develop the course modules in greater depth and detail than is possible in a sophomore one-semester course.

We are also investigating the possibility of introducing this material at the introductory level in the curriculum.

## References


[1] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Design and implementation of Triveni: A process-algebraic API for threads + events. In *Proceedings of the 1998 IEEE ICCL*. IEEE Computer Press.

[2] C. Colby, L. J. Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol. Objects and processes in Triveni: A telecommunication case study in java. In *Proceedings of the 1998 Usenix COOTS*.

[3] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic publishers, 1993.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[5] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991. 427 pp.

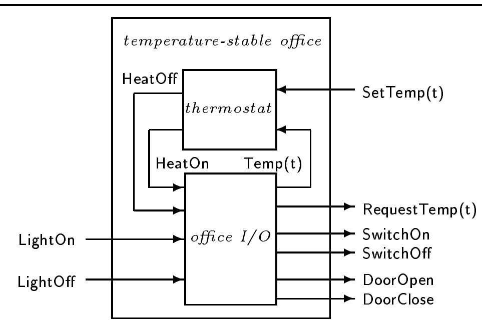
[6] R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice Hall, 1989.

## A Triveni: an example

To introduce Triveni and illustrate various features of Triveni, we describe an environmental control system for an office building from [1]. This example occupied about 7 hours of class lecture time.

We begin with the notion of an *office I/O*, which is a system that accepts as input the events that control the environment of an office (heating and lighting) and emits as output the various events necessary to communicate with the rest of the environment-control system. Some of these emitted events may originate from an action by a human occupant (switch on/off, door open/close, and temperature request). The remaining output event is a physical temperature reading, which may be automatically generated from time to time.

A *thermostat* partially automates the temperature control of an office. The pseudocode realization in Triveni of these processes is shown below, along with a diagram giving the interface of each process in terms of the events that it emits and accepts. Note that some events carry temperature data.




```
thermostat:

  temp actual_temp = INITIAL_ACTUAL_TEMP;
  temp target_temp = INITIAL_TARGET_TEMP;

  LOOP
       Temp(t) -> { actual_temp = t; }
          EMIT (t < target_temp) ? HeatOn : HeatOff
    ||
       SetTemp(t) -> { target_temp = t; }
          EMIT (actual_temp < t) ? HeatOn : HeatOff
```

The LOOP combinator in the thermostat implements an "event loop". The body of the loop is a parallel composition (using the || combinator) of two processes. The first process responds if the current event is of the form Temp(t) (i.e., a physical temperature reading); on any other event, it terminates silently. It is similar for the second process and events of form SetTemp(t). In both parallel components, two things happen on receipt of the specified event: an assignment takes place and an event is emitted to control a heater. The assignment is an *action*, written between braces, and may in general be any code in the host programming language (typically something that terminates quickly). The EMIT combinator emits an event. Events are delivered eventually (and simultaneously) to all interested listeners and the emitting process terminates.

A thermostat is attached to an office I/O simply by composing them in parallel, yielding a *temperature-stable office*. The parallel composition automatically ensures that the HeatOn, HeatOff, and Temp(t) events are transmitted between the two subprocesses. In this case, these three events are hidden with the LOCAL combinator so that they are not accessible externally as either inputs or outputs, as shown in the diagram above..

```
temperature_stable_office:

  office_IO io;
  thermostat therm;

  LOCAL  HeatOn HeatOff Temp
  IN     io || therm
```

The occupant of an office has manual control over the heat and lights via the *occupant control* process.

```
occupant_control:

  LOOP
       RequestTemp(t) -> EMIT SetTemp(t)
    || SwitchOn -> EMIT LightOn
    || SwitchOff -> EMIT LightOff
```

Upon SwitchOn, the above process will eventually emit LightOn. Triveni makes no guarantee as to the timing of event emission, so it is possible that SwitchOff could arrive *before* LightOn is emitted and thus would not actually turn off the light. Later, we will show a programming style to bulletproof against such cases. But in this case, SwitchOn and SwitchOff originate from human actions, and because we can reasonably assume that the light comes on faster than a human can flip the switch, we would not expect the bad case ever to occur. Triveni supports a notion of "`assert`" statements appropriate for concurrent programs, namely temporal-logic formulas, to express such safety properties. For instance,

$$LightOnPending =_{\text{def}} \neg \mathsf{LightOn}\ \mathcal{S}\ \mathsf{SwitchOn}$$

expresses the property of a single point during an execution run that "LightOn did not occur since the most recent SwitchOn." Then, the formula *SwOffSafety* defined as: $\Box(\mathsf{SwitchOff} \rightarrow \neg LightOnPending)$ expresses the property of an entire execution (read $\Box$ as "always") that "whenever SwitchOff occurs, there is no pending LightOn". Adding *SwOffSafety* (and the symmetric property for SwitchOn) to the office program generates a run-time error whenever the property is violated.

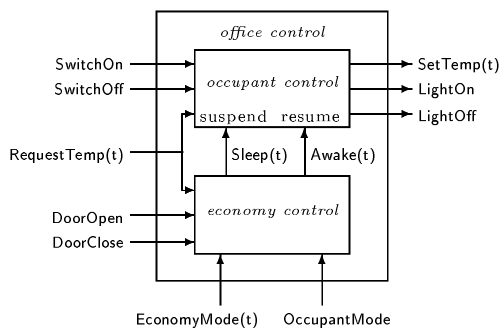
An office can be in two modes, *occupant mode* and *economy mode*. Occupant mode is the normal mode of operation, as implemented by the occupant-control process above. In economy mode, the temperature is reduced to and held at a specified value, despite any requests otherwise, and the lights are turned off and the switch disabled. The EconomyMode(t) event puts an office into economy mode, lowering the temperature to t, and the OccupantMode event returns the office to occupant mode, restoring the requested temperature to the most recent observed request. In addition, if an office is in economy mode, it should temporarily revert to occupant mode when the door is open, in case someone arrives in the middle of the night to work; in that case, the office returns to economy mode when the door is closed.

The *economy control* process implements this control, emitting Sleep(t) whenever the office should enter economy mode, lowering the temperature to t, and emitting Awake(t) whenever the office should return to occupant mode, restoring the temperature to t. The process runs three subprocesses in parallel. The first one monitors continuously the vlast requested temperature. The second and third parallel components determine when the office should change modes. Note that the code establishes mutual exclusion between *occupant* and *economy mode*.

On receipt of event EconomyMode, a process enters a loop that monitors the status of the office door. The invariant upon entry to the loop is that the office has just been placed in economy mode and needs to be put to sleep. While Sleep is being emitted, the `AWAIT` combinator waits until DoorOpen occurs. In the case that DoorOpen arrives while the emission of Sleep is still pending, the emission is aborted via the `DO/WATCHING` combinator to ensure consistency. When the door becomes open, a symmetric process emits Awake and waits for DoorClose. On receipt of OccupantMode, the door-monitoring loop is preempted and the office returns to occupant mode.

```
economy_control:

  temp last_temp = INITIAL_TARGET_TEMP;
  temp economy;

    LOOP RequestTemp(t) -> { last_temp = t; }
 || LOOP EconomyMode(t) -> { economy = t; }
           DO
             LOOP
                   DO EMIT Sleep WATCHING DoorOpen
                || AWAIT DoorOpen ->
                        DO EMIT Awake
                          WATCHING DoorClose
                     || AWAIT DoorClose -> DONE
           WATCHING OccupantMode
 || LOOP OccupantMode ->
           DO
             EMIT Awake(last_temp)
           WATCHING EconomyMode
```

Now we build an *office control* process from an occupant-control process and an economy-control process. The occupant-control process is disabled during economy mode using the `SUSPEND/RESUME` combinator, which suspends a process on receipt of a specified event (Sleep in this case) and resumes it on another event (Awake in this case). Thus, whenever the economy control sends a Sleep event, the occupant will lose control of the light and heat until the economy control sends an Awake event. Two mutually exclusive processes (not shown in the picture) run in parallel with the occupant control and the economy control to adjust the light and heat appropriately whenever the office toggles modes.



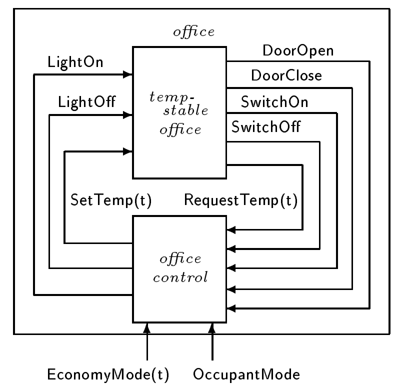

```
office_control:

  occupant_control oc;
  economy_control ec;

  LOCAL Sleep Awake
  IN
       ec
    || oc SUSPEND Sleep RESUME Awake
    || LOOP
         Sleep(t) -> DO    EMIT SetTemp(t)
                        || EMIT LightOff
                     WATCHING Awake
    || LOOP
         Awake(t) -> DO EMIT SetTemp(t)
                     WATCHING Sleep
```

Let us sprinkle in some temporal safety properties to be checked during execution.

$$Awake =_{\text{def}} (\neg\mathsf{Sleep}\ \mathcal{S}\ \mathsf{Awake}) \vee \boxminus(\neg\mathsf{Sleep})$$
$$SwOn =_{\text{def}} \neg\mathsf{SwitchOff}\ \mathcal{S}\ \mathsf{SwitchOn}$$

where $\boxminus(\neg\mathsf{Sleep})$ means that Sleep never occurred (i.e., an office is initially awake), we can specify that whenever LightOn occurs, both the office must be awake (no Sleep since the last Awake) and the switch must be on: $\Box(\mathsf{LightOn} \rightarrow Awake \wedge SwOn)$

To complete the implementation of a single office, we compose a temperature-stable office with an office control. The resulting *office* process emits no events and accepts only events EconomyMode(t) and OccupantMode. The `LOCAL` combinator hides all other events. Finally, multiple offices are combined into an entire floor of an office building.




```
office:

  temperature_stable_office tso;
  office_control oc;

  LOCAL SwitchOn SwitchOff LightOn LightOff
        RequestTemp SetTemp DoorOpen DoorClose
  IN    tso || oc

building_floor:

  building_floor bf;
  office o;

  bf || o
```

The implementation above allows offices to be added one by one. The entire floor is commanded to be placed in economy mode and to be restored to occupant mode as a whole. However, while in economy mode, individual offices may temporarily revert to occupant mode due to door activity, as described above.