Description and Analysis of Central Registry, a Pattern for Modular Implicit Invocation

by

Jonathan Newcomb Swirsky Whitney

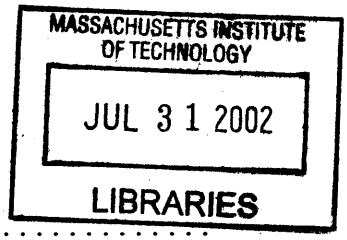
Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002
[June 2002]



Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by .....
Artur C. Smith
Chairman, Departmental Committee on Graduate Students

Description and Analysis of Central Registry, a Pattern for Modular Implicit Invocation

by

Jonathan Newcomb Swirsky Whitney

Submitted to the Department of Electrical Engineering
and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Central Registry is a generalization of several implicit-invocation design patterns. Its goal is to provide the kind of decoupling that implicit invocation provides in a more modular and flexible way than current patterns. We describe the pattern in detail and present a formal model of its key features. Using the formal model, we are able to establish, by automatic analysis, properties of any system that uses the pattern. We also describe an implementation framework for the pattern and evaluate its use in five substantial programs.
Keywords: Design Patterns, Implicit Invocation, Formal Models, Component Architectures.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I would like to thank Daniel Jackson for his advice and guidance. I would like to give special thanks to Robert Lee and Allison Waingold for helping with all aspects of this work. Ilya Shlyakhter and Manu Sridharan gave me a lot of assistance with Alloy and the Alloy Analyzer. All the members of the Software Design Group at LCS were extremely supportive and friendly for many years, and I am indebted to them for countless hours of discussion. I would finally like to thank my parents, friends and roommates for listening to me talk about this thesis for hours on end.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Implicit Invocation

Implicit invocation systems are systems in which procedure calls are made implicitly by the system. In all such systems, the major design goal is to allow for flexible assembly of components. Requests for procedure calls are made by announcing events, and procedures register interest with the system for particular events. The primary benefit of implicit invocation systems is that the components have no dependence on each other. Instead, they all rely on the underlying system to perform the communication between them. This form of system architecture allows for incremental extension and contraction of system functionality by simply adding and removing procedures for events or by adding and removing events. In [11], Garlan et al. note that this form of architecture is complementary, and not contradictory, to data abstraction techniques, one of the most popular forms of modularization [12, 11].

Design Patterns

Design patterns have become quite popular in the software engineering field since the publication in 1995 of the "Gang of Four" book [1]. The authors describe design patterns as generic solutions to problems that arise often. Design patterns have significantly contributed to software development in many ways. Primarily, they

capture years of design experience, which enables designers to avoid thinking about problems that already have good solutions. Secondly, they provide designers with a high-level vocabulary with which to communicate.

The primary goal of design patterns is to describe designs that are flexible and reusable [1]. Each pattern description is accompanied by an informal description of how the pattern achieves more flexible designs. The structure of each pattern is described somewhat formally in UML, but the behavior descriptions are quite informal. This can cause potential problems with a system designer who desires to know if a usage of a pattern is correct. It is fairly easy to convince oneself that the structure of a design matches a pattern, but as the patterns provide no formal description of behavior, it can be difficult to decide if a design behaves as the pattern prescribes. For most patterns, the behavior is extremely simple, and thus informal descriptions often suffice.

Central Registry

Central Registry is a design pattern for implicit invocation systems. It captures the structure and behavior of the underlying communication machinery of implicit invocation systems, and provides a disctinct way in which to describe these systems. Central Registry adds some novelties of its own in the form of event filtering and global state, which allows the pattern to apply in different forms to a wide range of systems.

This thesis presents a pattern that is more complex in behavior and structure than most "Gang of Four" design patterns. One of the main contributions of this thesis is a formal model of the pattern, which gives precise definitions of both the structure and behavior of the entities involved. This approach has two benefits. Primarily, it provides system designers with a better way to check if they are using the pattern correctly. Secondly, it allows for a more precise description of how the pattern achieves a reuseable and flexible design.

In summary, this thesis contributes :

- A generalized pattern for implicit invocation systems, which condenses the designs of many such systems.
- A formal model that gives precise definitions of both the structure and behavior of implicit invocation systems.
- A machine-checked argument that specific properties related to modularity hold for specific variants of the pattern.
- Some practical insights gleaned from experience using Central Reigstry, and examples of how other related patterns are inadequate.

This thesis presents Central Registry in the style of a "Gang of Four" pattern. The layout of the typical pattern description is extended to include the formal model and machine checked argument, as well as descriptions of the tradeoffs involved in using different variants of the pattern.

Chapter 2

Pattern Description

2.1 Intent

Avoid bidirectional coupling of the sender of a request and its receiver by passing all requests through a single object. Provide mechanisms to allow for dynamic behavior in response to a request.

2.2 Motivation

Consider a door locking system in a prison. Guards have access cards, which they swipe through card readers to open doors and pass from one section to another. The system determines whether or not to unlock a door and for how long. Its goal is to mediate access between compartments of a prison in a disciplined way to minimize the risk of undesirable prisoner movements.

It is natural to structure such a system based on the physical layout of the prison. The prison may be divided into sections, with different kinds of policies for the different sections. For example, high-security sections will have different rules for which doors can be open simultaneously than low-security sections. Furthermore, prison-wide rules relating to opening doors may be desirable. Lastly, requests to open a door may be generated by a warden not physically present at the door itself.

One problem is to structure the software to allow a variety of responses to door

unlocking requests, to allow these responses to change according to different states of the prison, different credentials of a guard, new administrative policies. It should be easy to change policies and to fine-tune the response of the system in different scenarios. This must be accomplished without excessive modification to existing software, such as the prisoner or guard databases.

Another problem is that requests can be generated by independent entities. It is therefore desirable that the part of the system that processes requests be independent of the parts of the system that can generate requests. Conversely, the senders of requests should remain independent of the receivers because the receivers are not always the same in different scenarios.

2.2.1 Overview of Central Registry Design

The idea of this pattern is to allow for this type of dynamic and flexible behavior while disallowing dependence between senders and receivers of the request. This is achieved by interposing a centralized event dispatching mechanism between the senders and receivers.

The various requirements of the prison system highlight the interesting features of the Central Registry pattern. *Event handlers*, which are wrapped around existing software, are registered to receive *events* in a single centralized table, which we call the *central registry.* The event handlers are registered against *event filters*, which accept or reject events based on their content, type and the state of the system.

Events are posted to a central queue. The event handlers that are registered against accepting filters will handle an event. There can be numerous event handlers registered to receive an event. The event handlers, in addition to carrying out local action, can change global state values and post new events to the queue. The central registry can maintain configurable policies for posting and queuing events and can also dynamically register and deregister handlers.

2.2.2 Example

This section describes parts of this system to highlight elements of the Central Registry design more concretely. This section will describe the behavior of event handlers that perform access-control, event handlers that unlock specific doors, and event handlers that control guard GUI consoles. These event handlers will communicate via `CardSwipedEvents` and `DoorUnlockEvents`. They will be registered with the central registry against filters that match the type of event as well as specific information contained within each individual event.

In this system, an embedded card reader is placed at every door. When a guard's card is swiped, this reader generates a `CardSwipedEvent`, containing the relevant data from the card and the door in question. The reader posts this event to the central registry. Each section of the prison is equipped with an access-control computer, which is an event handler. This handler is registered against an event filter that only matches `CardSwipedEvents` generating from doors within that section. The access-control handler then verifies the identity of the guard and grants access by generating and posting a `DoorUnlockEvent`. The role of the filter is to ensure that the access-controller for this section of the prison will not bother to verify requests to open doors in other sections.

This system may make use of event-based queuing priorities. A `DoorLockEvent`, for example, might be given a higher priority than a `DoorUnlockEvent`. Under this policy, all `DoorLockEvents` that are pending will be dispatched before any pending `DoorUnlockEvents`. In this system, an asynchronous post policy is appropriate, because card swipes happen across the prison asynchronously and should be handled as such.

Suppose the prison has a "lock-down" status for emergencies in which certain doors should not open. This can be implemented by setting a global state variable, which event filters access when determining whether a given handler should be activated for a given event. In this state, the filters for the door-unlocking handlers will not match any `CardSwipedEvents`.

Another requirement may be a GUI display of a map of the prison in which doors light up when they are unlocked. Each console running this GUI display may be a separate event handler that is *dynamically registered* when somebody logs into the console. This handler is passive. It merely listens to `DoorUnlockEvents` and `DoorLockEvents`, and updates the display.

This design achieves fairly good decoupling between the various subsystems described as event handlers. The GUI console, for example, need not depend on any of the software running at the access-control handlers or in the card-readers. Similarly, the access-control handlers and card-readers are independent of each other. All three subsystems are dependent on the specification of the events and global state variables instead.

2.3 Applicability

The Central Registry pattern is applicable to systems that are comprised of well-encapsulated subsystems whose communication can be represented as a small set of messages, and where it is desirable for the senders and receivers of requests not to have explicit knowledge of each other.

In such systems, the Central Registry pattern improves modularity – only the global state elements and the events are shared across subsystems. If the subsystem's interfaces cannot be represented by a small set of events, using Central Registry is likely to complicate rather than simplify a system. Because handlers can be dynamically registered and deregistered, Central Registry is particularly well-suited to systems whose communication targets change at run-time. Inter-module communication concerns are moved into the Central Registry, so that communication machinery need not be implemented in each module. Additionally, the pattern allows systems to achieve looser coupling and reuse of event, event filtering and event handling code.

2.4 Structure and Behavior

2.4.1 Structure

The object model in Figure 2-1 summarizes the structure of this pattern, viewed semantically. The boxes in the model represent semantic domains relevant to the pattern and the arrows connecting them represent relations between these domains. The notation on the arrowheads indicate the multiplicity of that relation where ? means "zero or one", ! means "exactly one", and * means "zero or more". While these semantic domains may be represented as classes, they do not necessarily map directly to the class structure [8]. For example, the Pairs of EventFilters and EventHandlers may be implemented as a hashtable. It is interesting to note that in Figure 2-1 the EventHandlers do not have a relation to Events. This is due to the fact that this type of diagram only captures structural properties of a design, and not behavioral properties.

The module dependence diagram shown in Figure 2-2 represents the syntactic dependences of Central Registry components on one another [8]. Edges with open arrows denote subtypes; solid edges with line heads denote dependences, and dashed edges denote weak dependences. Weak dependence results from modules depending on the existence, but not the behavior, of other modules.

As with all design patterns, there is a key dependence missing from Figure 2-2, which is the dependence between event handlers and event producers and vice-versa. This is the main achievement of the pattern.

2.4.2 Participants

Events. Events are the core of the Central Registry pattern. They encapsulate messages sent between subsystems.

EventHandlers. EventHandlers encapsulate interfaces to subsystems. They can handle certain Events in certain States. They are registered with the CentralRegistry against some set of EventFilters. EventHandlers can make changes to the State, and

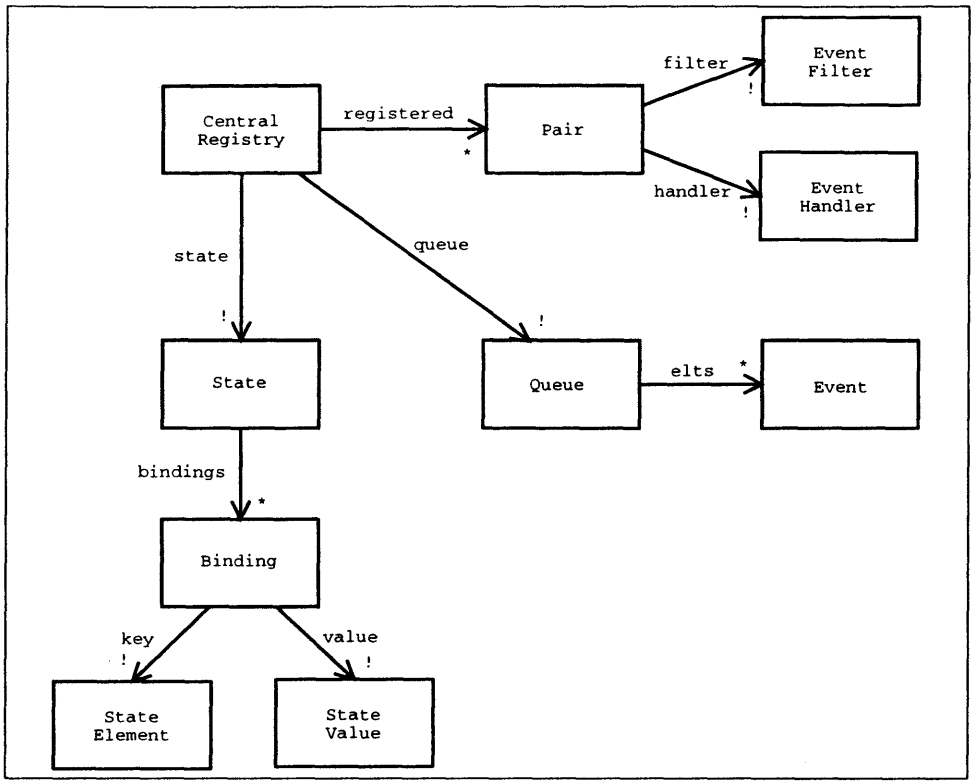


Figure 2-1: Object model for CR pattern.

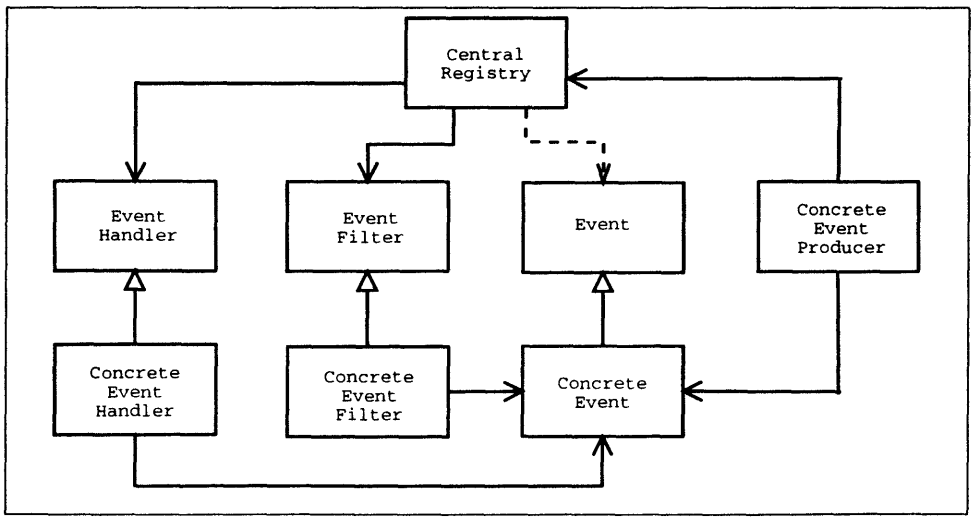


Figure 2-2: Module dependence diagram for typical CR system.

can be EventProducers.

EventFilters. EventFilters implement predicates on pairs of Events and States. They are registered with EventHandlers in the CentralRegistry.

EventProducers. EventProducers are components of the system that can post Events to the CentralRegistry. Often the EventProducers are also EventHandlers.

CentralRegistry. The CentralRegistry is a Singleton [1]. It maintains a registry of (EventFilter, EventHandler) pairs. It also maintains an Event queue, and a global State, which maps variable names to values. The CentralRegistry is in charge of dispatching Events to all interested Event-Handlers.

State. State is a mapping from state elements to values. It represents the shared global state of the system.

2.4.3 Collaborations

- EventProducers post Events to the CentralRegistry.
- The CentralRegistry retrieves an Event from the front of the queue. It matches the Event, along with the current State, against the registered EventFilters. For each matching EventFilter, the Event is dispatched to the corresponding EventHandler.
- The EventHandlers that are activated can post new Events to the CentralRegistry and can update the global State.

The interaction diagram in Figure 2-3 illustrates how the different entities collaborate with each other.

2.4.4 Features and Variants

Post Policy

Events can be posted synchronously or asynchronously to the CentralRegistry. In the synchronous case, there is no queue in the CentralRegistry. All posted events are

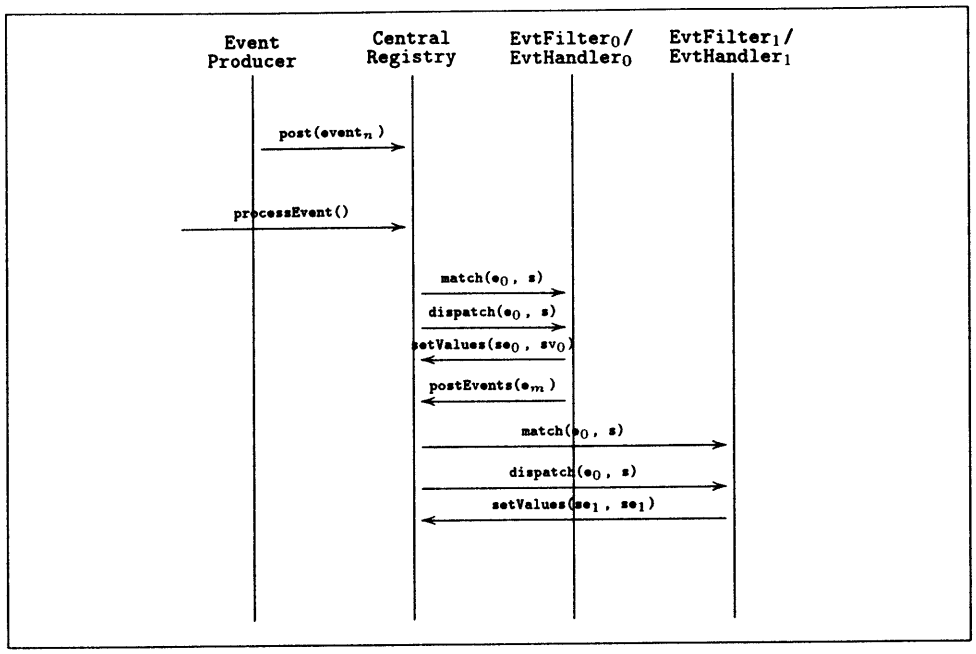


Figure 2-3: An interaction diagram of the CR processing an Event

dispatched immediately. This behavior essentially implements implicit method calls in Implicit Invocation systems [11]. This policy has a serious drawback: if an event handler responding to event e produces new events, then these events are handled before other handlers have responded to event e, meaning that e is not handled atomically.

The asynchronous policy is a more natural message-passing model. In systems using this policy, events are placed on the end of the CentralRegistry's queue and are processed some time later. Control is returned to the event producer immediately after the event is posted.

Queue Policy

When the asynchronous post policy is in use, the CentralRegistry's queue can define a queue policy that defines how events on the queue are processed. Queue policy could range from defining event priorities, to EventProducer-based fair-queueing, to dropping events as the result of congestion.

Global State

Some systems are designed with global state that is accessable to all subsystems.

Shared global state acts as a means of data communication between subsystems. Examples of global state variables range from status flags for the system to shared ADT's.

Often, global status flags can be replaced by dynamically registering and deregistering handlers. For example, two handlers which perform different actions may be swapped with one another in the registry instead of having one handler which performs both actions based on a boolean state value.

The deregistration approach might have the consequence of exposing subsystems to each other, since a handler must know about any other handlers that should be deregistered instead of just knowing about the state variable to be changed. In cases like these, the use of state is beneficial to further decoupling the system. Both event handlers depend on the global status flag, but do not refer to each other. Another benefit of this kind of design is to make the status explicit. It is often the case that much behavior must change in response to a status change. In such situations, performing dynamic registration and deregistration can get quite complicated. This can be remedied by increasing the complexity of the filtering and adding a global status flag.

Dynamic registration/deregistration

As illustrated in section 2.2, it is important to be able to modify the set of registration pairs (EventFilter, EventHandler) dynamically. We model this behavior with two special kinds of events (`DeregistrationEvent` and `RegistrationEvent`), which are handled by the CentralRegistry itself, and which contain a set of (EventHandler, EventFilter) pairs to be added or eliminated from the registry.

2.5 Consequences

This section briefly lists some of the benefits and liabilities of Central Registry. Some of these consequences will be discussed in further depth in Chapter 3.

1. *Modularity.*

Figure 2-2 shows a module dependence diagram for a typical Central Registry system. EventProducers do not depend on EventHandlers or vice versa. Instead, the EventProducers depend on the CentralRegistry (because they post events to it). The EventFilters, EventHandlers, and EventProducers all depend on the Events; the Events serve as the interface amongst them. The CentralRegistry depends on the EventFilter and EventHandler interfaces, since it makes calls to EventFilters to check if Events match and to EventHandlers to handle Events. Since the CentralRegistry just stores Events to the queue and then passes them along to EventFilters and EventHandlers, it only weakly depends on the Event interface; that is, it refers to the name of the interface without using any of its behavior.

2. *Added flexibility.*

   Central Registry provides much flexibility in modifying system behavior both statically and dynamically. New EventHandlers can be added at runtime, or the set of EventFilters associated with an EventHandler can change at runtime. Furthermore, new EventHandlers can be implemented independently of code already written. Lastly, modifications to existing EventHandlers should have little effect on other EventHandlers.

3. *Receipt is not guaranteed.*

   Systems designed using Central Registry cannot guarentee that two components that wish to communicate will always succeed. In many designs, a sender will have an explicit handle on its receiver, and is thus guaranteed that the receiver is present. However, EventHandlers can silently disapear from the CentralRegistry, which can mean that some Events will no longer have EventHandlers to handle them. Furthermore, priority-based queue policies may starve Events of low priority.

4. *Performance.*

   In Central Registry systems that make use of the asynchronous post policy,

there is a potential for long delay between the generation and handling of an Event. Even in systems that make use of the synchronous post policy, there is an extra level of method invocation that decreases performance. In most cases, the gain in flexibility and modularity will outweigh this.

Chapter 3

Properties and Analysis

3.1 A Formal Model

This section describes a formal model of the Central Registry pattern written in the Alloy [3] modeling language. The core structure of the model and some parts of the behavior are presented in Figure 3-1. The complete model is included in Appendix B; it fully captures the structure described in Figure 2-1, the behavior described in Figure 2-3, as well as some of the properties discussed in Section 3.2 of this chapter.

The signatures in the model roughly correspond to the domains in the object model in Figure 2-1. Some of the domains in Figure 2-1 are represented as relations inside other signatures. For example, the `Pair` domain is represented as the `registeredPairs` relation in the `CentralRegistryState` signature. The structural model shown here contains :

- `CentralRegistry`, which contains a set of bindings between event filters and event handlers, a queue of events, and the global state.
- Global `State` which is modeled as a mapping from `StateElements` to `StateValues`.
- A set of `EventFilters`, which match certain event types in certain system states.
- A set of `EventHandlers`, which for an (event, state) pairing, produce a new set of bindings that update the system state, and a new sequence of events which

```
/*This model is based on a FSM idiom. Each CentralRegistryState atom
  represents a point in time of the Central Registry*/
sig CentralRegistryState {
  //ef->eh is in registeredPairs if eh is registered
  //against ef in this state of the FSM.
  registeredPairs : EventFilter -> EventHandler,
  //this field represents the current global state
  globalState : State,
  //this field represents the current queue of Events
  queue : Seq[Event]
}
sig Event {}
/*if e.regChange contains ef->eh, then ef->eh
  will be added to the central registry*/
disj sig RegisterEvent extends Event {
  regChange : EventFilter -> EventHandler
}
/*if e.deregChange contains ef->eh, then ef->eh
  will be removed from the central registry*/
disj sig DeregisterEvent extends Event {
  deregChange : EventFilter -> EventHandler
}
sig Bindings {
  stateMapping : StateElement -> ? StateValue
}
/**Global state assigns a value to every variable*/
sig State extends Bindings { }
{
  //this is the constraint about "completeness"
  stateMapping.StateValue = StateElement
}
/**The trans relation describes the state changes and new events
  that result from handling an event in a given global state.*/
sig EventHandler {
  trans : Event -> State -> Bindings->Seq[Event]
}
/*If e->s is in match, then the filter accepts e in state s*/
sig EventFilter {
  match : Event -> State
}
sig StateElement {}
sig StateValue {}

/**The types of several helper functions are given here.
  The bodies of these functions can be found in Appendix B.*/

/*Returns the set of event handlers that are registered aginst
  matching event filters*/
det fun acceptingHandlers(e:Event, s:CentralRegistryState):
      set EventHandler {...}

/*returns a relation mapping eh->b->seq if ae->s->b->seq is in eh.trans*/
det fun collectChanges(ae: set EventHandler, ae:Event,
      s:State):EventHandler->Bindings->Seq[Event] {...}

/*describes how a set of Bindings is written to the global state s'*/
fun writeChanges(s, s':CentralRegistryState, sc: set Bindings) {...}

/*appends every seq in ne to the back of s.queue while removing the first event
result placed in s'.queue*/
det fun appendToQueueBackRemove(s, s':CentralRegistryState,
      ne: set Seq[Event]) {...}

/*appends every seq in ne to the back of s.queue. result placed in s'.queue*/
det fun appendToQeueuBack(s, s':CentralRegistryState, ne:
      set Seq[Event]) {...}

/*true if e1 was generated before e2*/
fun genBefore(e1, e2:Event) {...}

/*true if e1 was dispatched before e2*/
fun dispBefore(e1, e2:Event) {...}
```

```
/**This fun describes how Events are processed from the head of the
  queue in the asynchronous policy. With this policy, new Events are
  enqueued at the end of the queue. */
fun asynchronousProcess(s, s':CentralRegistryState) {
  some s.queue..SeqFirst()
  let activeEvent = s.queue..SeqFirst() {
    s'.eventJustProcessed = activeEvent
    let activeHandlers = acceptingHandlers(activeEvent, s) {
      handleSpecialEvent(activeEvent, s.registeredPairs,
        s'.registeredPairs)
      let allPossibleChanges = collectChanges(activeHandlers,
        activeEvent, s.globalState) {
        some chosenChanges:EventHandler->Bindings->Seq[Event]{
          all eh:activeHandlers {
            sole change:eh->Bindings->Seq[Event] |
              change in allPossibleChanges and
                change in chosenChanges
          }
          all eh:chosenChanges.Seq[Event].Bindings |
            eh in activeHandlers
          writeChanges(s, s',
            EventHandler.chosenChanges.Seq[Event])
          appendToQueueBackRemove(s, s',
            Bindings.(EventHandler.chosenChanges))
          Bindings.(EventHandler.chosenChanges) =
            s'.eventsJustGenerated
        }
      }
    }
  }
}
/**This fun describes the state transition of an event being
  asynchronously enqueued outside of event handling.*/
fun asynchronousEnqueue(s, s':CentralRegistryState, e:Event) {
  some seq:Seq[Event] {
    seq.seqElems = Ord[SeqIdx].first->e
    s'.eventsJustGenerated = seq
    appendToQueueBack(s, s', seq)
  }
  s'.registeredPairs = s.registeredPairs
  s'.globalState = s.globalState
  no s'.eventJustProcessed
}
/**This policy states that all event posts are asynchronous**/
fun asynchronousPolicy() {
  setup()
  limitTrans()
  all s:CentralRegistryState - Ord[CentralRegistryState].last {
    let s' = OrdNext(s) {
      ((asynchronousProcess(s, s')) ||
        (some e:Event | asynchronousEnqueue(s, s', e)))
    }
  }
}
/**This describes the property that if an event e1 is generated
  before e2, then it will be dispatched before e2.**/
fun orderedDispatch() {
  all disj e1, e2:Event {
    genBefore(e1, e2) => dispBefore(e1, e2)
  }
}
/**This states that in the asynchronous policy, with no event
  duplicates, ordering of dispatch is guarenteed**/
assert OrderOfDispatchAsync {
  asynchronousPolicy() && noDuplicateEvents() => {
    orderedDispatch()
  }
}
check OrderOfDispatchAsync for 3 but
      4 CentralRegistryState, 4 Seq[Event] //no solutions : expected
```

Figure 3-1: Core Alloy model

are added to the central registry's queue.

- Special events for registration and deregistration, which contain bindings between event filters and event handlers that are to be added to or removed from the central registry's binding, respectively.

Figure 3-1 includes the model for the behavior of the asynchronous post policy, as well as assertions to check that this policy guarantees ordered dispatch of events. We also modeled the synchronous post policy and a two-priority queue policy, which are included in Appendix B.

3.2 Properties of Central Registry

In this section, we discuss various properties of the pattern and their consequences. First, we introduce some required properties that go along with the programming model that Central Registry is describing. We then discuss the effects of these properties. All of these properties relate solely to the asynchronous post policy version of the pattern, with a simple FIFO queue.

3.2.1 Fundamental Properties

Three fundamental properties of the Central Registry pattern can be described as follows :

- Required execution.

  If the binding of (event filter f, event handler h) is registered, then any event posted to the central registry that f matches must result in the execution of h.

- Atomicity accross events.

  All handlers that are registered against filters matching an event will execute before any new events are processed.

- Ordering of events per event producer.

  If an event producer posts event e_1 and e_2 to the central registry in that order, then the handlers for e_1 will execute before the handlers for e_2.

These properties together form a model of concurrent computation that is natural to associate with asynchronous message passing systems. An event specifies a request for service. The results of an entitity performing this service can potentially influence later requests. Arbitrary interleavings of these requests would prevent reasoning about the global state after the request has been processed.

The motivating example described in Section 2.2 assumes that not all the event handlers will be on the same machine. Thus, it is natural to think of the set of event handlers that will respond to an event as executing concurrently. The next section will discuss problems that result from this model.

3.2.2 Analysis and Requirements

This section discusses the programming discipline that is necessary to achieve implementations of Central Registry systems with the properties defined in Section 3.2.1. A fundamental problem arises out of the confluence of three factors : the existence of global state, the requirement that all triggered event handlers execute, and the concurrent execution of event handlers in response to an event. Without careful thought, it may be impossible to reason about the global state after an event is processed.

In the remainder of this section, we will refer to a simple example. Pseudo-code for this example is included in Figure 3-2. This example describes an air-traffic-control system, in which there are three global boolean variables, two event handlers, and one event filter. Both event handlers in this example are registered against the same filter `Filter` in the central registry. This example describes an airport that has two power generators. One of the power generators is equiped with extra features that are used only in cases of emergency (the `SecureGenerator`). The main power generator cannot be on during an emergency. It is undesirable for both generators to be on at the same time, although the system will not crash in this state. But it is

```
global boolean StateOfEmergency;
global boolean MainGeneratorOn;
global boolean SecureGeneratorOn;

MainGenEH extends EventHandler {
  void handleEvent(Event e) {
    MainGeneratorOn := false;
    StateOfEmergency := true;
  }
}

SecureGenEH extends EventHandler {
  void handleEvent(Event e) {
    SecureGeneratorOn := false;
    StateOfEmergency := false
  }
}

Filter extends EventFilter {
  boolean acceptEvent(Event e) {
    return (MainGeneratorOn && SecureGeneratorOn);
  }
}
```

Figure 3-2: Pseudo-code for an air-traffic-control example

vital that at least one of the generators be on at all points in time. This means that (`MainGeneratorOn` || `SecureGeneratorOn`) must always be true.

The two event handlers will execute in response to the same event, because they are both registered against the same filter. The value of `StateOfEmergency` depends on the particular interleavings of assignments as the two handlers execute concurrently. This means that it is impossible to reason about the value of `StateOfEmergency` after the event is dispatched.

Rejected Solutions

This section gives a series of solutions that are natural for this sort of problem, but fail to allow local reasoning about the behavior of event handlers.

- *Change the model of event handling*

  A common solution to handling concurrency problems is to implement transactions to allow sequential execution in the abstract. In the example of Figure 3-2, if `MainGenEH` executes before `SecureGenEH`, then `MainGeneratorOn` will become false. Now, the filter `Filter` will no longer accept the event, and `SecureGenEH`

```
fun TransactionSystem() {
  all s:CentralRegistryState - Ord[CentralRegistryState].first {
    let s' = OrdNext(s) {
      TransactionTransition(s, s')
    }
  }
}
fun TransactionTransition(s, s':CentralRegistryState) {
  let activeEvent = s.queue..SeqFirst() {
    let activeHandlers = acceptingHandlers(activeEvent, s) {
      some transactionOrder: Seq[EventHandler] {
        all eh:activeHandlers | one transactionOrder.seqElems.eh
        transactionOrder..SeqElems() in activeHandlers
        ChangeReflectsOrder(s, s', transactionOrder)
      }
    }
  }
}
/*True if in the transition between s and s', the event handlers in
¡order¿ executed in that order.*/
fun ChangeReflectsOrder(s, s':CentralRegistryState, order: Seq[EventHandler]) {...}
```

Figure 3-3: Formal description of a CR system that has a transaction model

should no longer execute. The *Required Execution* property requires that all handlers whose filters will accept the event when it is dequeued must execute.

A solution to this problem is to provide each event handler with a read-only copy of the dispatch-time global state. Another way around this problem is to ensure that each event handler does not affect the state in such a way that any event filters will return a different result. This requires global reasoning about all event filters and is thus undesirable. A thrid potential solution is to compute all executing filters first, and execute the handlers in a second step. This does not work because the handlers themselves can read from the state.

- *Require commuting operations*

  Central Registry requires that all event handlers execute when invoked. Arbitrary interleavings of assignments to global variables are acceptable if every interleaving produces the same acceptable result. One way to achieve this is to require that no more than one event handler modify a state variable in response to an event. The Gizmoball system described in Section 5.3 has this requirement, which is defined in Alloy in Figure 3-4. However, this solution is undesirable because it requires global reasoning across all the event handlers, and thus eliminates decoupling between the event handlers. In the example given above, the two event handlers do not commute because they assign con-

```
fun CommutingCondition() {
  all cr:CentralRegistryState {
    //no conflicting state writings.
    !StateConflicts(cr)
  }
}
/*true of two event handlers can assign conflicting values to the
  global state*/ fun StateConflicts(cr:CentralRegistryState) {...}
```

Figure 3-4: Condition for Commuting Operations

```
/**Abstract Events define legal States before and after they are executed*/
sig Event {
  legalStateTrans : State -> State
}
/**This describes a CR system in which every handler obeys the legalStateTrans relation of Event.*/
fun LegalBindingsSystem() {
  all s:CentralRegistryState - Ord[CentralRegistryState].last {
    let s' = OrdNext(s) {
      ObeyLegalBindingsTrans(s, s')
    }
  }
}
/*Every event handler that is active in this transition must obey the
legalStateTrans relation of the active event*/
fun ObeyLegalBindingsTrans(s, s': CentralRegistryState) { ...}
```

Figure 3-5: Extension to CR model where Events specify legal transitions

flicting values to `StateOfEmergency`.

Specify legal global states

The above solutions describe ways to allow for concurrent operation while guaranteeing a deterministic behavior of the global state. Event producers do not know which handlers will handle their posted events. This suggests that requiring a deterministic result is unnecessary. One way to benefit from non-deterministic results while allowing local reasoning is to specify legal values of the global state.

This solution has two variants. In both variants, event handlers will execute in some arbitrary order, and will see a read-only copy of the dispatch-time global state.

- *Specify legal states in events*

  In this variant, each event specifies what the global state should be after the

```
SavePowerEvent extends Event {
  boolean legalStateTrans(oldState, newState) {
    if (oldState.MainGeneratorOn && oldState.SecureGeneratorOn){
      return !(newState.MainGeneratorOn && newState.SecureGeneratorOn);
    }
    else return true;
  }
}
```

Figure 3-6: Pseudo-code for an Event which specifies legal post-states

```
SomePowerInvariant{
  boolean invariantHolds() {
    return (MainGeneratorOn || SecureGeneratorOn);
  }
}
```

Figure 3-7: Pseudo-code for a global invariant

event is dispatched, as a function of what the global state was before the event was dispatched. Figure 3-5 shows an extension to the core model of Figure 3-1 in which the Event signature now contains a relation `legalStateTrans` from `State` to `State`. Figure 3-5 also describes the requirement that every event handler must assert a legal state.

Figure 3-6 shows an example of such an event. This Figure refers to global variables defined in Figure 3-2

- *Specify global invariants*

  In the other variant, there are global invariants for the state variables which must hold at all times. This variant is described in Figures 3-8, 3-9 and 3-11. Figure 3-7 gives pseudo-code for such an invariant on the air-traffic-control system.

  The invariant of Figure 3-7 will not hold if both event handlers from Figure 3-2 execute. However, both handlers preserve the invariant, as they are reading a copy of the dispatch-time global state, and they assume that the filter predicate is true. In the code for `MainGenEH`, for example, it is assumed that the value of

```
sig Invariant {
  allows : set Bindings
}
/** This describes a Central Registry system that has global invariants*/
sig CentralRegistryState {
  registeredPairs : EventFilter -> EventHandler,
  globalState : State,
  queue : Seq[Event],
  registeredInvariants : set Invariant
}
```

Figure 3-8: Extension to CR model with Invariants

```
/**This describes a CR system in which every handler obeys the registered invariants*/
fun InvariantSystem() {
  all s:CentralRegistryState-Ord[CentralRegistryState].last {
    let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = acceptingHandlers(activeEvent, s) {
        all b:activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
          MatchInvariants(s, b)
        }
      }
    }
  }
}
/*True if b is acceptable by all invariants of s*/
fun MatchInvariants(s: CentralRegistryState, b:Bindings) {...}
```

Figure 3-9: CR system with event handlers that obey invariants

```
MainGenEH extends EventHandler {
  void handleEvent(Event e) {
    MainGeneratorOn := false;
    SecureGeneratorOn := true
    StateOfEmergency := true;
  }
}

SecureGenEH extends EventHandler {
  void handleEvent(Event e) {
    SecureGeneratorOn := false;
    MainGeneratorOn := true;
    StateOfEmergency := false
  }
}
```

Figure 3-10: Pseudo-code for modified event handlers

```
/**This describes a CR system in which every handler obeys the registered invariants
and also asserts a value for every se in an invariant it potentially modifies*/
fun StrongInvariantSystem() {
  all s:CentralRegistryState-Ord[CentralRegistryState].last {
    let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = acceptingHandlers(activeEvent, s) {
        all b:activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
          MatchInvariants(s, b)
          StrongInvariantAssertion(s, b)
        }
      }
    }
  }
}
//this fun states that if a bindings intersects with an invariant's state elements,
//then it must assert a value for each element
fun StrongInvariantAssertion(s:CentralRegistryState, b:Bindings) {
  all inv:s.registeredInvariants {
    (some b':inv.allows | some b'.stateMapping.StateValue & b.stateMapping.StateValue) =>{
      all b':inv.allows {
        b'.stateMapping.StateValue in b.stateMapping.StateValue
      }
    }
  }
}
```

Figure 3-11: CR system with strong invariants

`SecureGeneratorOn` will not change. This assumption is wrong.

One way around this problem is to require that every event handler assign values to the global state without assuming anything about the filter predicate. The event handlers can assign each variable that they did not modify the old value from the dispatch-time copy, and the invariant will hold. Figure 3-10 shows modified pseudo-code for the event handlers from Figure 3-2. Any ordering of the two event handlers will result in a global state that satisfies the invariant.

In both versions of this solution, the ordering chosen for the event handlers does not matter, and local reasoning about the global property is possible using inductive techniques described in [8].

The constraints imposed by these solutions are burdensome, and can severely limit the behavior of event handlers. Furthermore, the global state must be copied for each event that is dispatched. This can be a severe performance hit if there are many global variables. This suggests that Central Registry is not applicable to systems that require a lot of global state.

Chapter 4

Implementation Framework

This chapter presents a framework implementation of the core Central Registry components in Java. This framework defines a generic `CentralRegistry` component, and interfaces for `Events`, `EventHandlers`, and `EventFilters`. The framework also defines generic interfaces for the global state to which the above modules and interfaces refer.

Systems are developed with this framework by defining concrete events, global state variables and invariants over them, and event handlers and filters. Becaues the framework provides many of the interfaces, as well as all the code for the `CentralRegistry` component, little thought or work need go into specifying these interfaces. This makes it quite simple for a system designer to make use of the pattern, as there is little extra development work involved.

4.1 Variants

The framework provides support for all the variants of the pattern discussed in Section 2.4.4. Support for queueing policies is achieved through the definition of `PolicyFilters` and `QueuePolicy` objects. `PolicyFilters` are similar to to event filters, in that they represent predicates on events and global states. These filters can also be configured based on the current state of the registry and of the queue. Filtering based on the state of the queue is beneficial in situations where a Central

Registry system needs to respond to congestion of the queue. `QueuePolicy` objects take as input an Event, the current state of the registry, the current queue, and the global state. The behavior they define is a reordering of the events on the queue. The framework provides support for dynamic loading of various policy filters and queue policies. This makes it easy for system developers to account for variations in the behavior of their system which results in more adaptable systems in general.

The two post policies defined in section 2.4.4 are also implemented. A Central Registry system can be configured to use either of these policies at startup, and they can be switched dynamically. The framework does not allow the two to be present at the same time.

Specifications for the framework can be found in Appendix A.

4.2 Global State Reasoning in Asynchronous Central Registry

As discussed in Section 3.2.2, reasoning about the correctness of a Central Registry system requires careful thought and discipline. The framework seeks to assuage this problem by providin behavior and structure to allow for use of the reasoning techniques described in Section 3.2.2. To this end, invariants are implemented as closures which are registered with the `CentralRegistry` and checked at runtime. Furthermore, the global state that the event filters and handlers read is an unmodifiable view. This allows event handlers to reason about the state at the time of event dispatch, and not concern themselves with the actions of other handlers.

One requirement for modular reasoning discussed in Section 3.2.2 is that `Event` subtypes be immutable. This is specified in the `Event` interface, but is not checked at runtime. Another requirement is that the event handlers be executed in some order, without overlaps in state modification. The current framework solves this problem by executing in a single thread. However, this is not necessarily the solution with the best performance, and does not apply to systems where the event handlers are simply

stubs for remote machines.

4.3 Benefits of the Framework

System designers can use this framework quite easily. Initial design work must provide specifications for events, global state, and invariants on the state. This type of up-front design is quite conducive to discovering early design flaws. Furthermore, it is quite simple to make use of the framework to implement stub systems, and perform integration testing before subsystems are actually written. To achieve this, one has only to write stub event handlers that have no local effects on the subsystems they are encapsulating. This type of prototyping can also lead to earlier discovery of design problems.

Another benefit of using the framework is ease of implementation. Because much of the behavior of the `CentralRegistry` component is already implemented, and the rest of the behavior is encapsulated with generic interfaces, implementation using this pattern does not require as much overhead as the size and complexity of the pattern might suggest.

Chapter 5

Known Uses

We have implemented a number of systems using the Central Registry pattern. We have also used the pattern to redesign some existing systems.

5.1 The CTAS Communications Manager

CTAS is a suite of tools developed by NASA designed to aid management of air traffic flow at large airports. The primary goal of CTAS is to increase the landing rate of aircraft through automated planning [4].

A component of this system was redesigned by a seminar of MIT graduate students and faculty [4]. This variant of the CTAS Communications Manager component is the one that we redesigned using the Central Registry pattern. In fact, the impetus behind the Central Registry pattern originated from limitations of this variant of the CM component.

Design Overview

The CTAS Communications Manager (CM) is essentially a message switching and database server. The CM is in charge of maintaining and controlling the client processes, and interfacing the client processes with the aircraft database. All client-to-client interactions are filtered through the CM. The CM must be able to handle the departure and arrival of new client processes.

The redesign of the CM is event-based. There are three types of events: messages from clients, administrative events generated by the server, and the actual addition and deletion of clients. However, the CM does not provide a common interface for these events and does not explicitly recognize messages and client arrival/departure as events. This severely limits the flexibility of the CTAS system, as the event notification paths connect the producers and consumers of the events directly.

The Central Registry Solution

We redesigned the CTAS CM using the Central Registry pattern. The subsystems for message-handling, clients, schedulers, and management of flight analyzers are each modeled as event handlers. The result is a much more flexible and decoupled system. New behavior can be added dynamically, and different implementations of components can be easily swapped. Figures 5-1 and 5-2 show module dependence diagrams of the CTAS CM not using and using Central Registry respectively. Note that for clarity, the Central Registry components have been left out in Figure 5-2. Many modules depend on the central registry module, as well as on events. However, these Figures are highlighting the missing dependences between various pre-existing modules.

This solution makes use of the asynchronous post policy described in Section 2.4.4, and does not use any special queue policy. This system does not require any global state, although some might be introduced to handle emergency cases.

Benefits:

- Achieved further decoupling between the different subsystems.
- Allowed for unification of events under a single hierarchy, which permits easier extensions to system behavior.

Liabilities:

- Delayed message passing (higher latency).
- One extra level of method invocation per event (lower throughput).

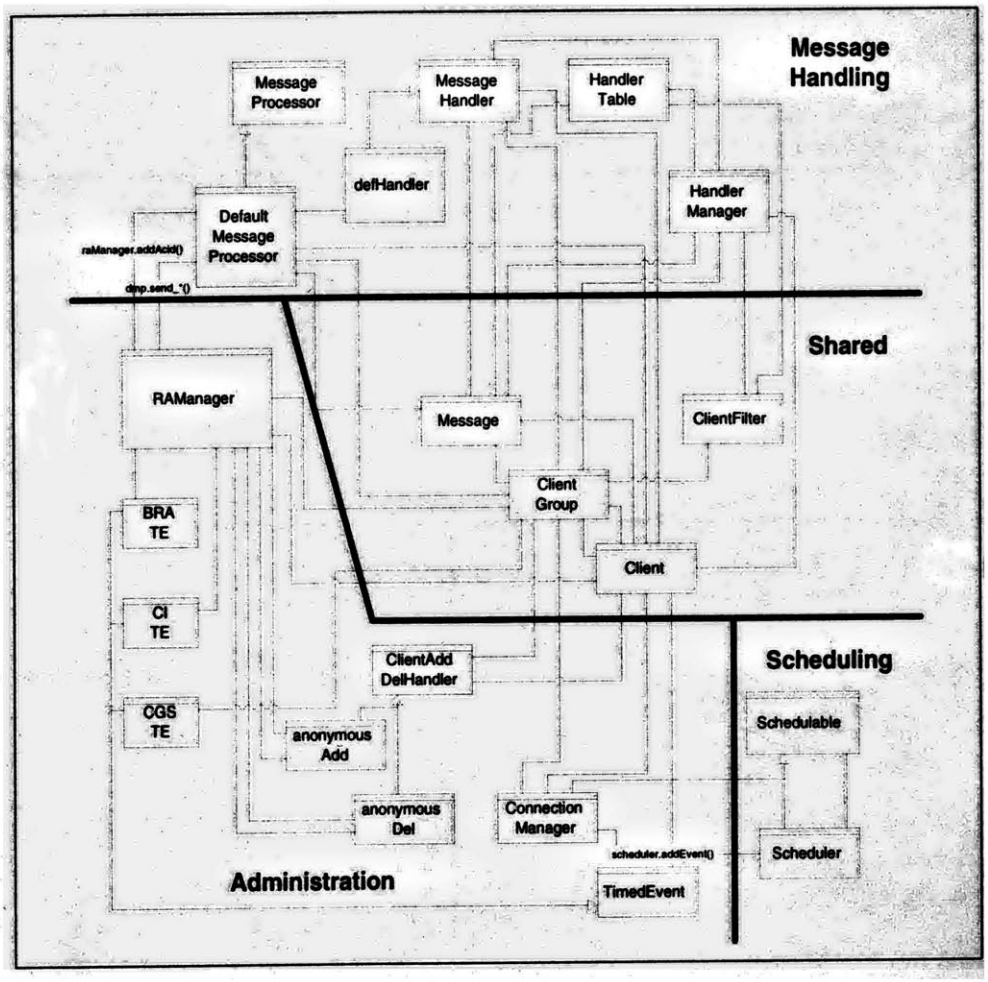


Figure 5-1: Module dependence diagram of a non-CR CTAS design.

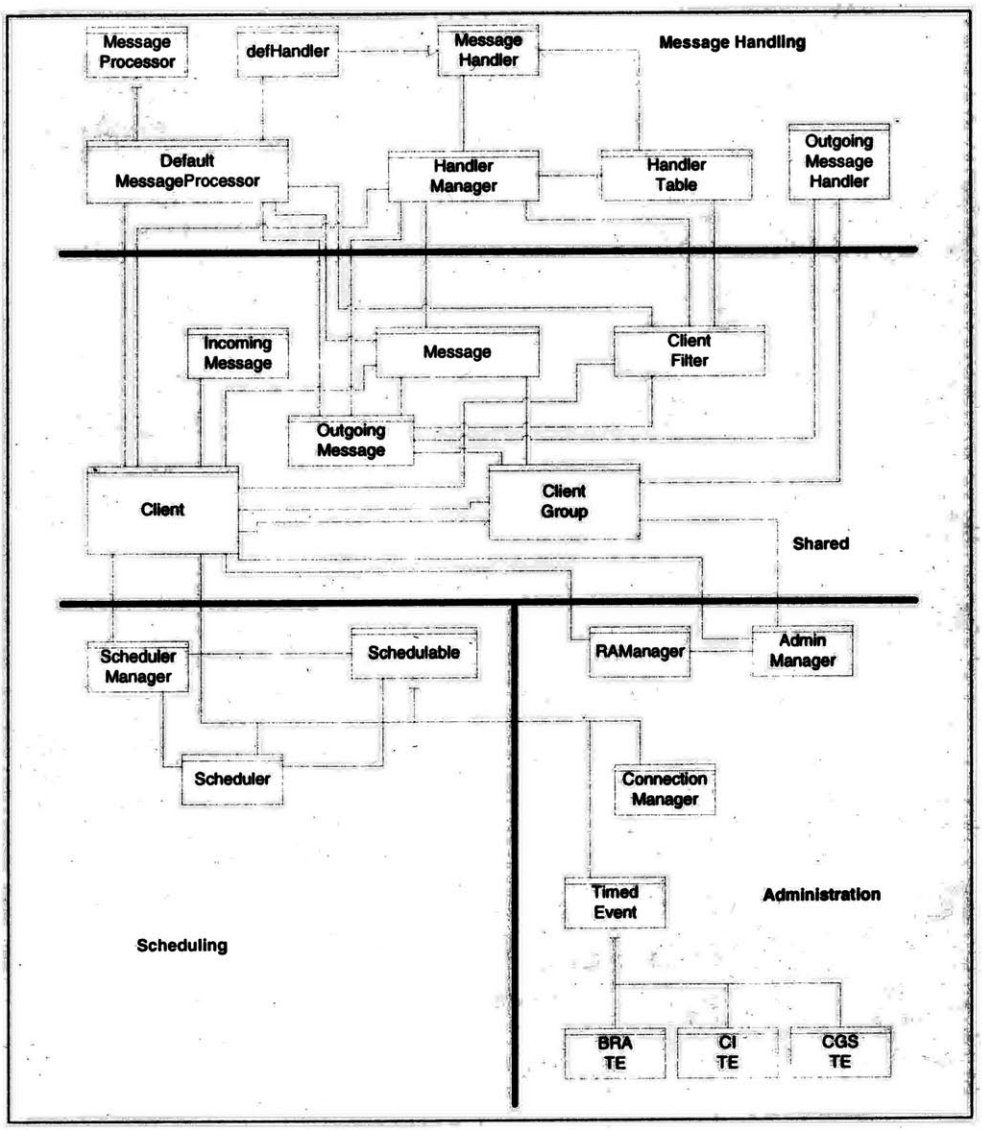


Figure 5-2: Module dependence diagram of a CR CTAS design.

5.2 Tagger

Tagger is a text-processing system written by Daniel Jackson [9]. It takes marked up text as input, and generates output for a layout program (like QuarkXpress). The design of the Tagger system largely resembles the Central Registry pattern, but differs in some key aspects. We redesigned this system to comply with Central Registry pattern, which resulted in a more flexible system.

Design Overview

Tagger uses a token registry, which accepts one event for each token generated by the input parser, and dispatches them to handlers, which are registered on the token types. Two particular problems in the design complicated the code:

The registry is not a singleton. The design is complicated by the fact that events that correspond to the numbering of sections, figures, etc. are handled by different handlers than events for normal text. The registry consists of two instances that are active at different times. The two registries maintain different sets of handlers that embody the behaviors that result from the different contexts of whether the event is for a numbering string or normal text. This interferes with decoupling: the input subsystem must decide which registry to send tokens to and the event handlers must know which registry to register with. This inflexibility highlights the resemblance of the Tagger design to the Multicast pattern described in [10].

Handlers are implicitly context-dependent. There is one specific case in the Tagger system when a handler is registered with the standard registry, and immediately removes itself after it is activated. This is to handle the start of a new paragraph: if no paragraph style token appears after a paragraph break, then a default style tag must be inserted; if a paragraph style token does appear, then it is processed as usual. Thus, a new handler must be registered when a paragraph break is reached, and then deregistered after the next token has been handled. But there is nothing in the structure of the system that makes this context change explicit, which makes it difficult to judge how modifications to the system will effect the behavior.

The Central Registry Solution

We added two state variables, `TEXT_MODE` and `PARA_START`, to address the anomalies in the original design. `TEXT_MODE` describes whether the system is in normal text mode or numbering text mode, which correspond to the behaviors of the two different registries in the original design. Event filters check the value of this variable and dispatch to the correct handler. This solution lacks the elegance of two separate registries that the original design held.

`PARA_START` keeps track of whether a paragraph break just occurred, in which case the event handler for text tokens looks up the default paragraph style and outputs that token before the text token. The state of the system is now explicit, and no longer needs to be inferred from the registered handlers.

Our implementation of Tagger uses the synchronous event posting policy, in order to process every event generated by an incoming token before processing the next token. We also implemented a variant of Tagger that did not make use of global state, but instead made use of Registration and Deregistration events. While this scheme worked, it was rather unwieldy. This system is an example where global state is not necessary, but significantly improves the ease of implementation.

Benefits:

- Centralized token dispatching, which permits greater behavioral flexibility.
- Allowed for explicit definition of shared context.

Liabilities:

- Shared data can limit modular reasoning.
- Complex filtering as a result of merging registries into a singleton.

5.3 Gizmoball

Gizmoball is a configurable pinball game, used as a final project in an introductory software engineering class at MIT [7]. The user controls a set of flippers, and must

attempt to keep balls from falling off the bottom of the playing area. Users can halt play, reconfigure the layout of the playing area, and load saved configurations from disk. The objects on the board are called *gizmos*. Gizmos may also be connected, so that if a ball hits some bumper, that collision can cause a flipper on the other side of the board to flip.

Design Overview

Some of the interesting issues in the design of this system are :

The design of the Gizmo ADT and the triggering system. A common solution presented by many students is what we call an "action-based" design. In this design, gizmos maintain a set of gizmos that they are connected to. The connections are maintained in the form of actions, which can be performed on the target gizmos. Essentially, this design is a variant of the Multicast pattern described in Section 6.4. This design hinders the ability to add new collision behavior. For instance, to increment a point counter each time that the ball collides with a bumper, a new point increment action must explicitly be registered with each bumper.

The interface between the GUI and the backend. A common solution is the use of the Observer pattern [1]. As we discuss in Section 6.2, while the Observer pattern does allow the backend to be independent of the GUI, the GUI is still dependent on the backend, since it must query the backend for update information. Thus, modifications to the backend can potentially result in modifications in the GUI.

Dealing with multi-threaded animation. One common solution is to have an animation thread that calls some method on a board ADT which causes an update to the GUI through the observer interface. However, implementors have to pay careful attention to which threads are performing which tasks and must worry about data races. Some designs simply have the animation thread take care of everything, in which case performance becomes an issue. Good Gizmoball designs usually provide some variant of the Command [1] pattern between the animation thread, and the main program thread.

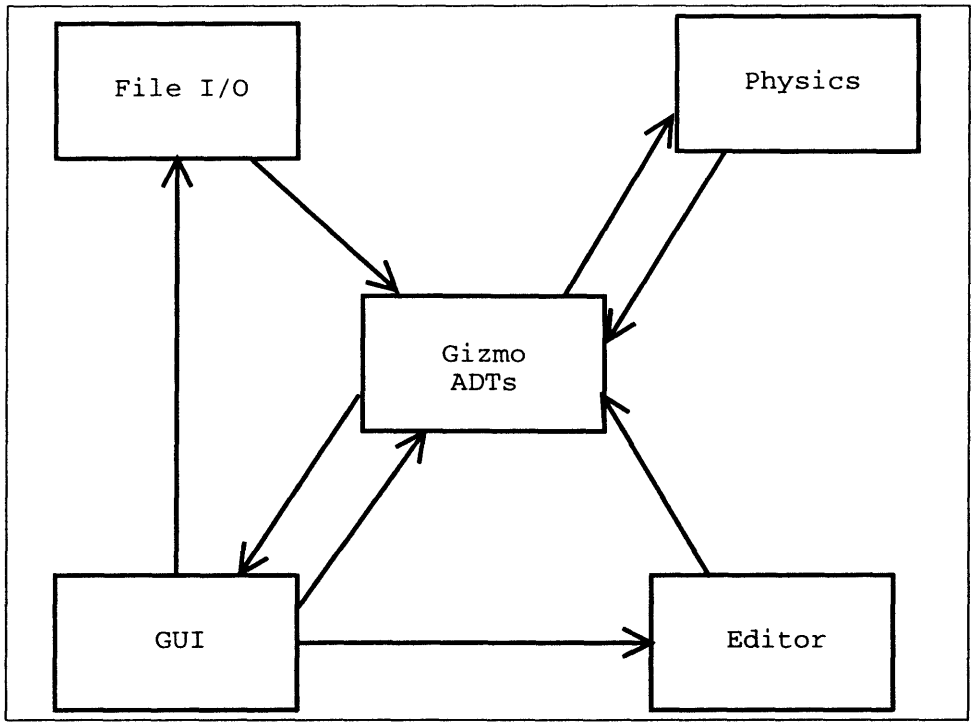


Figure 5-3: Module dependence diagram of a non-CR Gizmoball design.

The Central Registry Solution

The Central Registry version of Gizmoball was implemented with Robert Lee and Allison Waingold. Figures 5-3 and 5-4 show module dependence diagrams of Gizmoball designs not using and using Central Registry respectively. These diagrams highlight the improved decoupling and modularity in the Central Registry design.

While implementing Gizmoball, we perceived several benefits. Our design of the gizmo ADT models connections as event handlers, and triggering of connections and collisions as events. Switching between modes (editing, animating, not-animating) is modeled with global state. Communication between the backend and the GUI is event-based, severing any dependences between the two subsystems. The animation thread communicates with the rest of the system by posting events to the central registry. Our implementation uses a simple queueing policy which drops extra repaint-request events in the presence of performance slowdown.

This system exposed one potential problem with the Central Registry pattern. In Central Registry systems, much of the system state is recorded in the queue and

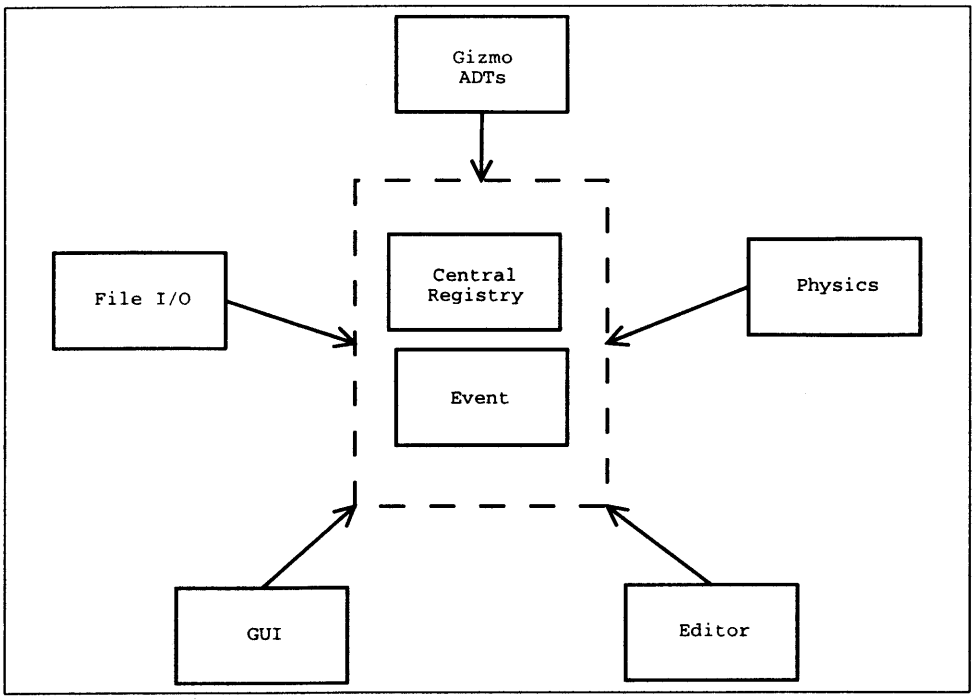


Figure 5-4: Module dependence diagram of a CR Gizmoball design.

the registry. It is therefore quite complicated to store and load this information from disk. However, this can be remedied by adding an extra event handler that logs every event and can produce a file from an event trace.

Benefits:

- Easy extension of triggering system to incorporate new actions.
- Completely decoupled the GUI and the backend.

Liabilities:

- Difficulty of saving system state to a file.
- Performance overhead, important in real-time system.

5.4 JarSheet

JarSheet is a configurable spreadsheet application developed by this author accompanied by Robert Lee and Allison Waingold.

Design Overview

Some of the interesting issues in the design of this system are :

The design of the cell grid ADT. One fundamental behavior that all spreadsheets must contain is that of permeating a new value of a cell to all cells that refer to it. A common design for this problem is to have each cell keep a list of cells that refer to it, and notify those cells when their values change. Once again, this design is a variant of the Multicast [10] pattern. Separate mechanisms must be in place to notify the GUI of a change of value, as it is unlikely that the GUI and the Cell ADT share a common behavioral interface at the code level.

Allowing for User-Defined Functions. One requirement on the JarSheet application was the ability to allow users to define their own mathematical functions based on a set of existing functions. Many designs will simply create an object that will take some generic parameter list and return a result. These objects will be stored in some global lookup table. One interesting design issue is how this table is stored, and who has access to it. The part of the system that is responsible for creating the new function objects will have to store them in the table, and the formula evaluation subsystem will have to access the same table to lookup functions.

The Central Registry Solution

The JarSheet application was designed using the Central Registry pattern. Each cell was linked to an event handler that would listen for new values in cells it was interested in. Function closures were implemented as event handlers as well. Adding a user defined function simply requires registering a new function closure handler in the central registry.

This design presents several benefits over a more common design. There is no table lookup by any code to discover which function closure to invoke. This is accomplished through writing event filters that filter on the type of operation. This means that when a cell requires a recomputation of it's value, it has merely to throw an event and wait for an update, which decouples the cell ADT from the computation algorithms. Furthermore, adding new functions is extremely simple.

JarSheet makes use of the synchronous post policy. This makes sense in the context of cell value computation. It does not seem correct to state that the computation of a new cell value is over until all sub-formula computations are complete and a new value is ready. This would not be possible using the asynchronous post policy. JarSheet does not make use of any global state.

One problem encountered while developing JarSheet came because of the use of the synchronous post policy. It is unclear what should happen if more than one event handler will respond to an event. The properties of a synchronous Central Registry system are currently ambiguous, and some of the properties that hold for asynchronous systems cannot hold in the synchronous case. One such property is the notion that all the handlers for an event will execute before the next event is processed.

Benefits:

- Decoupled the cell ADT from any computation algorithms
- Simple addition of user defined functions

Liabilities:

- Semantics of synchronous post policy are ill defined

5.5 The Visualization Tool for the Alloy Analyzer

The Alloy Analyzer [3] is accompanied by a visualization tool that allows users to inspect solutions to models or counter-examples to assertions. It is the subject of a thesis by Brian Lin [15], and was designed using the Central Registry pattern.

The pattern came in useful in decoupling customization entities from generic algorithms that respond to certain customizations. For example, one generic algorithm for a customization is the ability to project the visualization onto a certain type. Many of the data elements in the tool must respond to this change in how things are visualized. An original design used a Multicast/Observer design by having generic

interfaces for the different algorithms which would then register with customizations that could trigger these algorithms.

The Central Registry design captures the algorithm interface as an event handler interface, and makes the invocation of an algorithm into an event. Through filtering, this allows a cleaner separation between the pieces of the algorithm and the sources of the events.

Chapter 6

Comparisons

There are several patterns that can be classified as specializations of the Central Registry pattern. Of these patterns, only one has benefitted from formal modeling and analysis. Here we describe these patterns and where they fall in the feature space.

Table 6.1 summarizes where these patterns fall in terms of the features of Central Registry.

Pattern	central	filtering	state	dyn. reg	policies
Observer	No	Maybe	No	Yes	NA
EventPorts	No	Yes	No	Yes	NA
II	Yes	No	Yes	No	Yes
Multicast	No	No	No	Yes	NA
Mediator	Yes	No	No	No	NA

Table 6.1: Features of related patterns

6.1 Implicit Invocation

Implicit invocation systems defer method calls to the underlying system. A call to a procedure is made by generating a request to the system, which then determines a matching procedure and invokes it. This form of modularization is similar to Central Registry. Implicit invocation systems decouple the caller from the callee, in the same way that Central Registry decouples the event producer from the event handler.

Garlan and Khersonsky developed an infrastructure for modeling implicit invoca-

tion systems [6]. The model is similar to ours in that the internal machinery of event delivery is factored out so that a new system can be modeled by adding specifications for the event and event handling components in the system. Their modeling infrastructure handles shared variables and pluggable post and dispatch policies. However, they do not model event filtering as we do. While event filtering can be mimicked with state variables, we believe that the increased modularity of encapsulated event filters is a significant improvement. Our modeling infrastructure can accommodate systems that do not include event filters, whereas modeling event filters by introducing state variables in their system would not be as straightforward.

Garlan's model also assumes that the event bindings are fixed at startup. Again, we could use state variables to model changing event bindings, but this complicates the model and in our experience, the behavior that results from changing bindings is particularly error-prone.

6.2 Observer

In instances of the Observer pattern [1], an *observer* registers its interest in a *subject* with the subject itself. The subject keeps track of its observers, and notifies each of its observers whenever its state changes [1]. The observer then queries the subject to ascertain the nature of the state-change.

Observer is used extensively in Java's windowing toolkit (AWT). Classes can implement listener interfaces for each type of event (e.g. a mouse event, or a button pressed event), and register themselves to listen to events occurring in a particular window (or other GUI widget). In this case, the subject keeps track of listeners for each type of event, providing type-based filtering.

Observer differs from Central Registry in two ways: its distributed nature and the dependence of the observers on the subjects. Each subject keeps track of its own observers. Filtering can be implemented (as in the AWT example), but is not part of the core pattern.

6.3 EventPorts

EventPorts is an implicit invocation pattern similar to Central Registry [5]. In the EventPorts approach, the event handling components each export two interfaces in order to interact with other components: the InPorts and OutPorts. The InPorts in this model are very similar in function to the EventFilters presented in the Central-Registry model. They contain the logic to match an event. The OutPort interface is meant to decouple the event handling logic from the logic that dispatches outgoing messages. Both interfaces can be dynamically bound and unbound.

While the EventPorts approach is similar to Central Registry, there are several key differences. Without a central point of registration, it is difficult to provide the framework to reason about several key system attributes. The lack of a centralized model of global state makes it difficult to provide the groundwork for sound reasoning about state changes across event handlers. Furthermore, there are no guarantees of ordering and atomicity. Lastly, the EventPorts approach does not achieve the same re-usability of event handler/event filter pairings.

6.4 Multicast

Multicast [10] attempts to decouple event producers from the objects that handle events. Each event producer keeps a registry of event handler objects to which it passes events and dispatches the events to all registered handlers. In this sense, the pattern resembles Central Registry. However, Multicast has an explicit dependence of the event handler on the event producer. This dependence is not present in Central Registry.

Vlissides argues in [10] that the primary reason for not using a Central Registry-type approach is the lack of type-safety. In our framework, we achieve some measure of runtime type using the Visitor pattern [1] on events, event filters, and event handlers.

6.5 Mediator

Mediator [1] employs a *mediator* object to keep objects from referring explicitly to each other. The mediator defines an interface that *colleagues* use to communicate with each other. Any behavior that one colleague uses which is provided by another colleague must be included in the mediator interface. Mediator achieves some of the same goals as Central Registry – a colleague can be swapped with another that implements the same behavior, without changing other colleagues that use the behavior. However, this can only be done statically; there is no way to dynamically reconfigure the system. Furthermore, adding new behavior in the module requires expanding the mediator interface. While superficially similar to Central Registry, the mediator pattern uses encapsulation, not implicit invocation.

Chapter 7

Discussion and Conclusions

7.1 Formal Models of Patterns

The model is implemented in a framework mind-set. This means that we set up the model so that it should be simple to extend for a particular system. We believe this to be desirable to users of the pattern. Users of any design pattern would like to know if they are using it correctly. They can do this by writing assertions that would check if properties of their system are guaranteed by the generic pattern. We have not done this type of analysis for any of the systems we described in Section 5.

Our experience building a formal model of the Central Registry pattern provided us with the following benefits :

- **Clarity in understanding the pattern.** Writing precise specifications of the structure and behavior led us to understand many of the consequences of the design that we might have overlooked otherwise.

- **Encouraged discovery of properties.** Alloy is accompanied by an automatic analyzer. The availability of a tool that can check properties encouraged us to precisely define some of the properties that we might not have defined at all.

- **Usage of the pattern.** By playing with formal specifications of the pattern, we discovered interesting aspects about the various features that are not quite

properties. For example, we realized that the notion of atomicity did not really accompany the synchronous post policy in the way we had originally thought. This led to a better understanding of how the synchronous policy should be used by implementors.

- **Checking generic properties.** The Alloy Analyzer allowed us to check various properties of the pattern that are common to every Central Registry implementation.

We believe that the benefits gained from this experience could be applied to other design patterns, in particular those that exhibit complex structure and/or behavior. All current design pattern descriptions in [1] lack a precise definition of behavior, and concentrate instead on informal arguments for increased decoupling. In this thesis, a formal description of behavior, structure and properties accompanied the informal argument for increased coupling. The desired effect of this is a higher confidence in the claims made by the informal arguments. It seems that, while perhaps the patterns in [1] are perhaps so widely used and understood that they will not benefit from this, new patterns can gain credibility from this type of approach. In addition, even patterns described in [1] could benefit in precision by being described formally.

7.2 Frameworks for Design Patterns

Several languages have incorporated design patterns as language features or library elements, and have made the use of these patterns commonplace. The Iterator [1] pattern has become commonplace in the `java.util` package. The `java.awt.event` package represents a medly of several design patterns. Similarly, the Observer [1] pattern is integrated into Smalltalk-80 [1], and Java (`java.util.Observer`). Languages where procedures are first-order elements incorporate the Strategy [1] pattern as an integral part of the language. Some work has been done that investigates language extensions to incorporate design patterns [14].

The desired goal of providing a language feature or a library framework for a

design pattern is to make it more simple and natural to use the design pattern. This should improve the quality of code produced in that language as the structure and pattern-related behavior of the pattern elements is pre-defined.

For many patterns, the benefits of having library frameworks is negligeable. It is trivial, for example, to rewrite the Observer interfaces in a language such as Java. Central Registry is a somewhat more complex pattern both in structure and behavior. In addition, there is an element in the pattern, the `CentralRegistry` that is common to all instances of the pattern with few extensions. Furthermore, the extensions to this element are provided for in a modular way by the framework. This has significant benefits for designers that wish to use Central Registry.

The model of Central Registry in the Alloy language is also a framework. Alloy is not accompanied by any design pattern libraries. A benefit of having a pattern library is that it can be accompanied by a set of properties that are desirable for all instances of the pattern. Extending the framework will allow a system designer to easily profit from the power of the Alloy Analyzer to check properties of a system, which is extremely beneficial to the early detection of design flaws.

Appendix A

Framework Specifications

```
package centralreg.event;
/**
 * This class is the CentralRegistry component of Central Registry pattern framework
 * @specfield registeredPairs \\ EventFilter -> EventHandler
 * @specfield globalState \\ StateElement -> ? StateValue
 * @specfield queue \\Seq[Event]
 * @specfield registeredInvariants \\ set Invariant
 * Invariants are invariants on this.state, and are checked before and
 * after the execution of each event handler.
 */
public class CentralRegistry extends EventHandler {
/**
 * This method initializes the CentralRegistry from a properties file.
 * This file specifies the post policy, queue policy, and the global state setup.
 *
 * @param the name of the properties file
 */
public static void init(String filename);
/**
 * @requires: nothing
 * @returns: the singleton instance of CentralRegistry
 */
public static CentralRegistry getInstance();
/**
 * @requires : args != null
 * @modifies : this.registeredInvariants
 * @effects : attempts to register inv as an invariant of the system.
 *            If owner does not own all the StateElements involved in the
```

```
 *            invariant, StateElementOwnershipException, no modification to this.
 */
public void registerInvariant(EventHandler owner, StateInvariant inv)
throws StateElementOwnershipException;
/**
 * @requires : this.registeredPairs not be modified while Iterator is in use
 * @returns : a generator for the set of all EventHandlers eh s.t.
 *            this.registeredPairs[eh] is non-null
 */
public Iterator handlers();
/**
 * @requires : this.registeredPairs not be modified while Iterator is in use
 * @returns : a generator for the set of all EventFilters ef s.t.
 *            this.registeredPairs[eh] = ef
 */
public Iterator filters(EventHandler eh);
/**
 * The CentralRegistry is an EventHandler that can handle RegisterEvents.
 *
 * @requires : args != null
 * @modifies : this.registeredPairs
 * @effects : this'.registeredPairs = this.registeredPairs + e.regChange
 */
public void handleEvent(RegisterEvent e, Context currentState);
/**
 * The CentralRegistry is an EventHandler that can handle DeregisterEvents.
 *
 * @requires : args != null
 * @modifies : this.registeredPairs
 * @effects : this'.registeredPairs = this.registeredPairs - e.deregChange
 */
public void handleEvent(DeregisterEvent e, Context currentState);
/**
 * @requires : e!= null
 * @modifies : this.queue
 * @effects : e will be placed on the queue. It's location on the queue
 * will depend on the current post and queue policies.
 */
public void processEvent(Event e);
/**
 * @requires: nothing
 * @modifies: nothing
 * @returns: copy of this.globalState
 */
public Context getCurrentState();
```

```
}
```

```
package centralreg.event;
/**
 * Context defines a mutable mapping from state elements to state values.
 * @specfield state \\ StateElement -> StateValue
 * Each Context should be initialized correctly (i.e. each state element
 * should be set to some default value, and the names of the state should be
 * set correctly.
 */
public interface Context {
/**
 * @requires : this.state not be modified while iterator is in use
 * @modifies : nothing
 * @effects : returns an iterator over the global state elements
 */
public Iterator stateElements();
/**
 * @requires : nothing
 * @modifies : nothing
 * @effects : returns a new Context object that is .equals() to this
 */
public Context copy();
/**
 * @requires : arg != null
 * @modifies : nothing
 * @effects : if stateElement is a valid StateElement
 *            returns this.state[stateElement]
 *            otherwise, NoSuchElementException
 */
public Object pollElement(String stateElement)
throws NoSuchElementException;
/**
 * @requires : arg != null
 * @modifies : this.state
 * @effects : if stateElement is a valid StateElement,
 *            and if value is a legal value,
 *            this'.state[stateElement] = value
 *            if stateElement is not valid : NoSuchElementException
 *            if value is not valid : InvalidValueForElementException
 */
public void assertElement(String stateElement, Object value)
throws NoSuchElementException, InvalidValueForElementException;
}
```

```
package centralreg.event;
```

```
/**
 * An Event is a polymorphic data type used to encapsulate an event in the system.
 * An Event is a triple : (source, body, state), where source is the EventProducer
 * that generated this event, body is a generic object, and state is a virtual
 * context which represents the status of the Context at the time of
 * creation of the Event.
 * @specfield body \\ generic Object
 * @specfield source \\ EventProducer that generated this.
 * The relationship between Event and EventHandler is an instance of the Visitor
 * pattern. EventHandlers are Visitors, and Event's can be visited.
 * They accept EventHandlers, and call eh.handleEvent(this, c) inside
 * the accept(EventHandler eh, Context c) method.
 * The same relationship exists between Event and EventFilter objects.
 */
public interface Event {
/**
 * @requires : nothing
 * @returns : this.body
 */
public Object getBody();
/**
 * @requires : nothing
 * @returns : this.source
 */
public EventProducer getSource();
/**
 * @requires : nothing
 * @effects : implements the abstract accept operation as an element
 *            that can be visited by a Visitor. Calls the abstract
 *            handleEvent(this, c) method on EventHandler.
 */
public void accept(EventHandler eh, Context c);
/**
 * @requires : nothing
 * @effects : implements the abstract accept operation as an element
 *            that can be visited by a Visitor. Calls the abstract
 *            acceptsEvent(this, c) method on EventFilter.
public boolean accept(EventFilter ef, Context c);
}

package centralreg.event;
/**
 * This class defines a generic EventFilter. It has an acceptsEvent
 * method which returns true if this filter matches the event in the
 * given state.
```

```
 * This class is a Visitor in the Visitor pattern. The visit method is
 * the acceptsEvent method
 */
public abstract class EventFilter {
/**
 * @requires : all args != null
 * @returns : true if the EventHandler that this is registered for in
 *            the CentralRegistry can handle <e> in the currentState
 *            note : the default behavior is to return false.
 */
public boolean acceptsEvent(Event e, Context currentState);

/**
 * Most EventHandlers will not want to accept RegisterEvents.
 * This defaults to false;
 */
public boolean acceptsEvent(RegisterEvent e, Context currentState);
/**
 * Most EventHandlers will not want to accept DeregisterEvents.
 * This defaults to false;
 */
public boolean acceptsEvent(DeregisterEvent e, Context currentState);
}

package centralreg.event;
/**
 * An EventHandler defines the behavior for handling an Event.
 * This class is a Visitor in the Visitor pattern. The visit method
 * from the Visitor pattern is the handleEvent method in this class.
 */
public abstract class EventHandler
{
/**
 * The top level handleEvent method. This defaults
 * by throwing a runtime exception.
 */
public void handleEvent(Event e, Context currentState);
/**
 * Most EventHandlers will not respond to RegisterEvents.
 * Therefore, this defaults by throwing a runtime exception.
 */
public void handleEvent(RegisterEvent e, Context currentState);
/**
 * Most EventHandlers will not respond to DeregisterEvents.
 * Therefore, this defaults by throwing a runtime exception.
```

```
 */
public void handleEvent(DeregisterEvent e, Context currentState);
}

package centralreg.event;
/**
 * This class represents an abstract invariant on global state.
 * The check method performs the verification of the invariant
 * and throws a runtime exception if the invariant does not hold
 */
public interface StateInvariant {
    public boolean check(Context state);
}
```

Appendix B

Complete Alloy Model

```
/**
Alloy Model of the Central Registry design pattern.
Rob Lee, Allison Waingold, Jonathan Whitney. 3-15-02.

This model describes the core features of the Central Registry design pattern.
Central Registry is a generalization of various implicit invocation patterns.
This model describes the basic structure of a system using Central Registry,
as well as the various behaviors these systems can have.

Central Registry is a pattern for component-based message passing systems.
The core elements are :

Events.
  Events encapsulte a message passed between subsystems

EventHandlers.
  EventHandlers encapsulte interfaces to subsystems. They can handle certain Events in certain States.
  They are registered with the CentralRegistry against some set of EventFilters. EventHandlers
  can make changes to the STate, and can be EventProducers

EventFilters.
  EventFilters are predicates on Events and States. They are registered with EventHandlers
  in the CentralRegistry.

EventProducers.
  EventProducers are parts of the system that can post Events to the CentralRegistry.

CentralRegistry.
  The CentralRegistry maintains a registry of EventHandlers paired with EventFilters. It also
  maintains a queue of Events, as well as a global State, which maps variable names to values.
  The CentralRegistry is in charge of dispatching Events to all interested EventHandlers.


In the formal model presented below, Events, EventHandlers and EventFitlers are modeled with signatures.
The CentralRegistry is modeled by a finite state machine, called CentralRegistryState. One atom of this signature describes
the state of the CentralRegistry at one point in time. The model describes  the behavior of the pattern by defining
the legal transitions of this finite state machine. We have modeled special events for registration and deregistration.
Properties of the structure are represented as invariant constraints. Some of the generic behavior is also described
as invariant constraints on the state transitions of the finite state machine. Behavior that belogs to various different
policies is modeled in functions, which are constraints that can be applied at will.

We have also described some of the properties that we would like to investigate. For the most part, we
investigate the results of applying some policy to the finite state machine. These properties are described
in assertions. The Alloy Analyzer will attempt to find counter-examples to these assertions in some scope.
```

```
*/

/**We called our model helen after our inspiration**/
module prophelen


/*****************************************************************************************************
STRUCTURE AND CONSTRAINTS

This part of the model describes the basic structure of the pattern. Sigs represent types, and relations are
declared inside them. Sigs can be appended with constraints that are applied to every atom of the sig.
*****************************************************************************************************/



/**These are some standard libraries of Alloy constraints and signatures. Seq describes polymorphic sequences. Ord
describes polymorphic total orderings. SeqUtil describes some  utilities for sequences, such as constraints for
subsequence.*/
open std/seq
open std/ord
open SeqUtil


/*This model is based on a FSM idiom. Each CentralRegistryState
  represents a point in time of the Central Registry*/
sig CentralRegistryState {

   //this field is non null if this state  was reached by an event being processed.
   //in which case it is the event that was just processed
   eventJustProcessed: option Event,

   //if in the state transition that resulted in this state included any new events
   //these events are in some sequences in this set.
   eventsJustGenerated : set Seq[Event],

   //eh->ef is in registeredPairs if eh is registered against ef in this state
   registeredPairs : EventFilter -> EventHandler,
   //this field represents the current global state
   globalState : State,
   //this field represents the current queue of Events
   queue : Seq[Event],
   //this field represents the set of Invariants that will be checked.
   registeredInvariants : set Invariant

}


/**Abstract events specify legal global states.*/
sig Event {
   legalStateTrans : State -> State
}

/*regChange and deregChange are the ef->eh pairs that will be
  added or removed from the central registry*/
disj sig RegisterEvent extends Event {
   regChange : EventFilter -> EventHandler
}
disj sig DeregisterEvent extends Event {
   deregChange : EventFilter -> EventHandler
}

sig Bindings {
   stateMapping : StateElement -> ? StateValue
}

/*Global state assigns a value to every variable*/
```

```
sig State extends Bindings {
}
{
   //this is the constraint about "completeness"
   stateMapping.StateValue = StateElement
}

/*The trans relation describes the state changes and new events
  that result from handling an event in a given global state.*/
sig EventHandler {
   trans : Event -> State -> Bindings->Seq[Event]
}

/*If e->s is in match, then the filter accepts e in state s*/
sig EventFilter {
   match : Event -> State
}

sig StateElement {}
sig StateValue {}

/**This sig describes an invariant set on the global State. All global states must match the invariant*/
sig Invariant {
   allows : set Bindings
}

/**
Enqueue, Process and Reorder are sigs that are simply used to get a better visualization of the behavior in
the Analyzer. They are not part of the core model*/
sig Enqueue {}
sig Process {}
sig Reorder {}



/*****************************************************************************************
STATE TRANSITIONS

these functions describe various legal state transitions that CentralRegistryStates can take.
*****************************************************************************************/

/**This fun describes how Events are processed from the head of the queue in the synchronous policy.
In the synchronous policy, there is no queue in the CentralRegistry. However, we have to model the recursive nature
of the synchronous policy with a queue, as Alloy doesn't support recursion. Therefore, we model the synchronous policy
by enqueuing new events at the head of the queue.*/
fun synchronousProcess(s, s':CentralRegistryState) {
   some s.queue..SeqFirst()
   precondition()
   let activeEvent = s.queue..SeqFirst() {
      s'.eventJustProcessed = activeEvent
      let activeHandlers = acceptingHandlers(activeEvent, s) {
 handleSpecialEvent(activeEvent, s.registeredPairs, s'.registeredPairs)

 let allPossibleChanges = collectChanges(activeHandlers, activeEvent, s.globalState) {
//OR MAYBE THIS COULD READ : some chosenChanges: allPossibleChanges
    some chosenChanges:EventHandler->Bindings->Seq[Event] {
       all eh:activeHandlers {
  sole change:eh->Bindings->Seq[Event] |
     change in allPossibleChanges and change in chosenChanges
       }
       all eh:chosenChanges.Seq[Event].Bindings | eh in activeHandlers
       //this causes the chosen state changes to be written in arbitrary order.
       writeChanges(s, s', EventHandler.chosenChanges.Seq[Event])
       appendToQueueFrontRemove(s, s', Bindings.(EventHandler.chosenChanges))
       Bindings.(EventHandler.chosenChanges) = s'.eventsJustGenerated
   }

 }
```

```
      }
   }
}
/**This fun describes the state transition of an event being synchronously enqueued outside of event handling.*/
fun synchronousEnqueue(s, s':CentralRegistryState, e:Event) {
   some seq:Seq[Event] {
      s'.eventsJustGenerated = seq
      seq.seqElems = Ord[SeqIdx].first->e
      appendToQueueFront(s, s', seq)
   }
   s'.registeredPairs = s.registeredPairs
   s'.globalState = s.globalState
   no s'.eventJustProcessed
}

/**This fun describes how Events are processed from the head of the queue in the asynchronous policy.
The this policy, new Events are enqueued at the end of the queue. */
fun asynchronousProcess(s, s':CentralRegistryState) {
   some s.queue..SeqFirst()
   let activeEvent = s.queue..SeqFirst() {
      s'.eventJustProcessed = activeEvent
      let activeHandlers = acceptingHandlers(activeEvent, s) {
 handleSpecialEvent(activeEvent, s.registeredPairs, s'.registeredPairs)
 let allPossibleChanges = collectChanges(activeHandlers, activeEvent, s.globalState) {
    some chosenChanges:EventHandler->Bindings->Seq[Event] {
       all eh:activeHandlers {
  sole change:eh->Bindings->Seq[Event] |
     change in allPossibleChanges and change in chosenChanges
       }
       all eh:chosenChanges.Seq[Event].Bindings | eh in activeHandlers
       writeChanges(s, s', EventHandler.chosenChanges.Seq[Event])
       appendToQueueBackRemove(s, s', Bindings.(EventHandler.chosenChanges))
       Bindings.(EventHandler.chosenChanges) = s'.eventsJustGenerated
    }
 }
      }
   }
}
/**This fun describes the state transition of an event being asynchronously enqueued outside of event handling.*/
fun asynchronousEnqueue(s, s':CentralRegistryState, e:Event) {
   some seq:Seq[Event] {
      seq.seqElems = Ord[SeqIdx].first->e
      s'.eventsJustGenerated = seq
      appendToQueueBack(s, s', seq)
   }
   s'.registeredPairs = s.registeredPairs
   s'.globalState = s.globalState
   no s'.eventJustProcessed

}



/**This describes the state transition of reordering the queue with two priorities. Deregister and Register events
have higher priority than any other event.*/
fun twoPriorityQueueReordering(s, s':CentralRegistryState) {

//   some s.queue.seqElems
   reorderedQueues(s, s')

   no e:Event - (RegisterEvent+DeregisterEvent) {
      some e':RegisterEvent+DeregisterEvent | e' in s'.queue..SeqNexts(e)
   }
   no s'.eventJustProcessed
   no s'.eventsJustGenerated

}
```

```
/*****************************************************************************
POLICIES


Policies are constraints on which state transitions are legal.
*****************************************************************************/


/**This policy states that all state transitions must be synchronous**/
fun synchronousPolicy() {
   setup()
   all s:CentralRegistryState - Ord[CentralRegistryState].last {
      let s' = OrdNext(s) {
 ((synchronousProcess(s, s')) || (some e:Event | synchronousEnqueue(s, s', e)))
      }
   }
}



/**This policy states that all state transitions must be asynchronous**/
fun asynchronousPolicy() {
   setup()
   limitTrans()
   all s:CentralRegistryState - Ord[CentralRegistryState].last {
      let s' = OrdNext(s) {
        ((asynchronousProcess(s, s')) || (some e:Event | asynchronousEnqueue(s, s', e)))
      }
   }
}

/**This policy states that enqueue and process state transitions are asynchronous, and interleaved
with state transitions that reorder the queue.**/
fun twoPriorityQueuePolicy() {
   setup()
   limitTrans()
   forceInterestingBehavior()

   some s:CentralRegistryState-Ord[CentralRegistryState].last | twoPriorityQueueReordering(s, OrdNext(s))

   all s:CentralRegistryState - Ord[CentralRegistryState].last {
      let s' = OrdNext(s) {
   ((asynchronousProcess(s, s')) || (some e:Event | asynchronousEnqueue(s, s', e)) || twoPriorityQueueReordering(s, s'))


 (((asynchronousProcess(s, s')) || (some e:Event | asynchronousEnqueue(s, s', e)))&& some OrdNext(s')) =>
      twoPriorityQueueReordering(s', OrdNext(s'))
 (twoPriorityQueueReordering(s, s') && some OrdNext(s')) =>
    ((asynchronousProcess(s', OrdNext(s'))) || (some e:Event | asynchronousEnqueue(s', OrdNext(s'), e)))

      }
   }
}

/*****************************************************************************
HELPER FUNCTIONS
*****************************************************************************/

//utility function to limit the size of the trans relation, so we can actually see things
det fun limitTrans() {
   all eh:EventHandler {
      #eh.trans < 3
      some eh.trans
   }
}
```

```
//utility function that forces some Reordering based on priorities in the two priority scheme
fun forceInterestingBehavior() {
   some e:Event - (DeregisterEvent + RegisterEvent) {
      Ord[CentralRegistryState].first.queue..SeqFirst() = e
      #Ord[CentralRegistryState].first.queue..SeqElems() >1
      some e:(DeregisterEvent+RegisterEvent) {
         e in Ord[CentralRegistryState].first.queue..SeqElems()
      }
   }
   noDuplicateEvents()
}


//utility function that gives the set of EventFilters that can
//accept a specific Event
det fun acceptingFilters(e:Event, dispatchState:State):set EventFilter
{
   result = {ef:EventFilter| (dispatchState in e.(ef.match)) }
}
//utility function that gives the set of EventHandlers that
//are registered against filters that accept the parameter event
det fun acceptingHandlers(e:Event, dispatch:CentralRegistryState): set EventHandler {
   result = {eh:EventHandler | some ef:EventFilter { ef in acceptingFilters(e, dispatch.globalState)
and eh in ef.(dispatch.registeredPairs) } }
}

//this collects the changes resulting from dispatching the active event to all
//the event handlers. It makes an non-deterministic choice if there are many possibilities in the trans relation
det fun collectChanges(activeHandlers:set EventHandler, activeEvent:Event, dispatchState:State):
EventHandler->Bindings->Seq[Event] {

   result = {eh:activeHandlers, b:Bindings, seq:Seq[Event] |
      eh->activeEvent->dispatchState->b->seq in EventHandler$trans
   }
}

/** this updates the registeredPairs for Register and Deregister events*/
det fun handleSpecialEvent(activeEvent:scalar Event, registeredPairs: EventFilter -> EventHandler,
outputRegisteredPairs : EventFilter -> EventHandler){
   outputRegisteredPairs = registeredPairs + activeEvent.regChange - activeEvent.deregChange
}


/**This fun describes the precondition for the synchronous post policy. In otherwords, it is a constraint that
requires that there be never be more than one EventHandler that handles an Event.*/
fun precondition() {
   //this says that there is at most one EventHandler that can ever handle an event

   all crs:CentralRegistryState {
      let e = crs.queue..SeqFirst() {
 sole eh:crs.registeredPairs[EventFilter] | eh->e->crs.globalState->Bindings->Seq[Event] in EventHandler$trans
      }
   }


}


/**This fun describes how the set of Bindings <stateChanges> is written to the global state of s'.*/
fun writeChanges(s, s': CentralRegistryState, stateChanges:set Bindings) {
/*
   let modifiedElements = stateChanges.stateMapping.StateValue {
      all se:modifiedElements {
 one b:stateChanges {
    se->b.stateMapping[se] in s'.globalState.stateMapping
 }
      }
      all se:StateElement-modifiedElements {
```

```
 s'.globalState.stateMapping[se] = s.globalState.stateMapping[se]
      }
   }
  */

   (s'.globalState).stateMapping = (s.globalState).stateMapping -
   (((stateChanges.stateMapping).StateValue)->(StateValue - (StateElement.(stateChanges.stateMapping)))) +
   stateChanges.stateMapping

}

/**This is a helper fun to describe appending a set of sequences to the back of a queue while removing the front element*
det fun appendToQueueBackRemove(s, s':CentralRegistryState, newEvents:set Seq[Event]) {
   s'.queue..SeqStartsWith(SeqRest(s.queue))
   all seqEvents: newEvents {
      SeqUtilities[Event]..subSequence(seqEvents, s'.queue)
   }
   #s'.queue.seqElems+1 = (sum seq:newEvents+s.queue| #seq.seqElems)
}

/**This is a helper fun to describe appending a set of sequences to the back of a queue*/
det fun appendToQueueBack(s, s':CentralRegistryState, newEvents:set Seq[Event]) {
   s'.queue..SeqStartsWith(s.queue)
   all seqEvents:newEvents {
      SeqUtilities[Event]..subSequence(seqEvents, s'.queue)
   }
   #s'.queue.seqElems = (sum seq:newEvents+s.queue | #seq.seqElems)
}

/**this is a helper fun to descript appending a set of sequences to the head of a queue while removing what was the head*
det fun appendToQueueFrontRemove(s, s':CentralRegistryState, newEvents: set Seq[Event]) {
   SeqUtilities[Event]..SeqEndsWith(s'.queue, SeqRest(s.queue))
   all seqEvents:newEvents {
      SeqUtilities[Event]..subSequence(seqEvents, s'.queue)
   }
   #s'.queue.seqElems+1 = (sum seq:newEvents+s.queue | #seq.seqElems)
}

/**This is a helper fun to describe appending a set of sequences to the head of a queue*/
det fun appendToQueueFront(s, s':CentralRegistryState, newEvents: set Seq[Event]) {
   SeqUtilities[Event]..SeqEndsWith(s'.queue, s.queue)
   all seqEvents:newEvents {
      SeqUtilities[Event]..subSequence(seqEvents, s'.queue)
   }
   #s'.queue.seqElems = (sum seq:newEvents+s.queue | #seq.seqElems)
}


/**This is a helper fun to describe the condition that the queues are in some different order*/
fun reorderedQueues(s, s':CentralRegistryState) {

   //the queue's don't change size
   #s'.queue.seqElems = #s.queue.seqElems

   all ind:((s.queue).seqElems).Event {
      let e = (s.queue).seqElems[ind] {
 #((s'.queue).seqElems).e = #((s.queue).seqElems).e
      }
   }
}


fun setup() {
   no Ord[CentralRegistryState].first.eventJustProcessed
   no Ord[CentralRegistryState].first.eventsJustGenerated
}
```

```
/**This encapsulates the condition that e1 is generated before e2 in the presence of asynchronous state transitions**/
fun genBefore(e1, e2:Event) {

   let crgen1 = {cr:CentralRegistryState | e1 in cr.eventsJustGenerated..SeqElems()} {
      let crgen2 = {cr:CentralRegistryState | e2 in cr.eventsJustGenerated..SeqElems()} {
        ((some crgen1) and (some crgen2) and ((OrdLT(crgen1, crgen2)) or ((crgen1 = crgen2) and
   (some seq:crgen1.eventsJustGenerated | (e1+e2 in seq..SeqElems()) and seq..SeqPrev(e1, e2)))))
      }
   }
}

/**This encapsulates the condition that e1 is dispatched before e2 in the presence of asynchronous state transitions**/
fun dispBefore(e1, e2:Event) {
   let crdisp1 = CentralRegistryState$eventJustProcessed.e1 {
      let crdisp2 = CentralRegistryState$eventJustProcessed.e2 {
 ((some crdisp1) and (some crdisp2)) => OrdLT(crdisp1, crdisp2)
      }
   }
}

/**This describes the condition that Events are never reused**/
fun noDuplicateEvents() {
   all crs:CentralRegistryState {
      not crs.queue..SeqHasDups()
   }
   all ev:Event {
      sole crs:CentralRegistryState {
 ev in crs.eventsJustGenerated..SeqElems()
      }

      all crs:CentralRegistryState-Ord[CentralRegistryState].first {
 no (Ord[CentralRegistryState].first.queue..SeqElems() &
     crs.eventsJustGenerated..SeqElems())
      }
   }
}

/**This fun describes the condition that a sequence of events is unsorted**/
fun Unsorted(seq:Seq[Event]) {

   some e:Event - (RegisterEvent+DeregisterEvent), e':(RegisterEvent+DeregisterEvent) |
      e in seq..SeqPrevs(e')
}


/**This describes the condition that two events have the same priority**/
fun samePriority(e1, e2:Event) {
   (e1+e2 in RegisterEvent+DeregisterEvent) || (e1+e2 in Event - (RegisterEvent + DeregisterEvent))
}

fun TransactionTransition(s, s':CentralRegistryState) {
   let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = acceptingHandlers(activeEvent, s) {
 some transactionOrder: Seq[EventHandler] {
    all eh:activeHandlers | one transactionOrder.seqElems.eh

//     all eh:activeHandlers | one ind:SeqIdx | transactionOrder[ind] = eh
    transactionOrder..SeqElems() in activeHandlers

    ChangeReflectsOrder(s, s', transactionOrder)
 }
      }
   }
}

fun ChangeReflectsOrder(s, s':CentralRegistryState, order: Seq[EventHandler]) {
   let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = order..SeqElems() {
```

```
 some emptyBinding: Bindings {
    no emptyBinding.stateMapping
    some bindingsOrder: Seq[Bindings], evOrder:Seq[Seq[Event]] {
       #bindingsOrder.seqElems = #order.seqElems
       #evOrder.seqElems = #order.seqElems
       //this universal quantifier states that the two sequences represent the chosen transitions per event handler
       //and that the indeces match up
       all ind: order.seqElems.EventHandler {
  let b = bindingsOrder.seqElems[ind] {
     let evseq = evOrder.seqElems[ind] {
      let eh = order.seqElems[ind] {
   activeEvent->s.globalState->b->evseq in eh.trans
}
     }
  }
       }

       //have to say now that :
       //1. The bindings changes were applied in order.
       //2. The seqs were appended in order.

       //1.
       //changedPair is the set of se->sv pairs that are in s' but not s.
       all changedPair:s'.globalState.stateMapping-s.globalState.stateMapping {
  let se = changedPair.StateValue {
     let sv = changedPair[se] {
//there must be some bindings that established
some b:bindingsOrder..SeqElems() {
   se->sv in b.stateMapping
   all b':bindingsOrder..SeqNexts(b) {
      no sv':StateValue | se->sv' in b.stateMapping
   }
}
     }
  }
       }
       //2.
       //we will use a flatten function on seqs, and state that this flattened seq is
       //what is appended to the queue...
       let flattened = FlattenSeqs(evOrder) {
  appendToQueueBackRemove(s, s', flattened)
       }

    }
 }
     }
   }
}


det fun FlattenSeqs(s:Seq[Seq[Event]]):Seq[Event] {

   //these conditions help a lot :)
   !s..SeqHasDups()
   all seq:s.seqElems[SeqIdx] {
      !seq..SeqHasDups()
   }


   //special case for the beginning :
   let i0 = Ord[SeqIdx].first {
      result.seqElems[i0] = (s.seqElems[i0]).seqElems[i0]
   }
   //this says that every sequence in the sequence is a subsequence in the result
   all i:s.seqElems.Seq[Event] {
      SeqUtilities[Event]..subSequence(s.seqElems[i], result)
   }
   //as subsequence preserves ordering, all we need to say now is
```

```
   //each adjacent sequence connect by first and last elements
   all i':s.seqElems.Seq[Event]-Ord[SeqIdx].first {
      let i = OrdPrev(i') {
 let lasti = s.seqElems[i]..SeqLast() {
    let firsti' = s.seqElems[i']..SeqFirst() {
       let resultindiceslast = result.seqElems.lasti {
  let resultindicesfirst = result.seqElems.firsti' {
     some resi:resultindiceslast, resi':resultindicesfirst | resi = OrdPrev(resi')
  }
       }
    }
 }
      }
   }
}

fun OverlappingStateValues(s: CentralRegistryState) {
   let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = acceptingHandlers(activeEvent, s) {
 let potentialBindings = activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
    no se:potentialBindings.stateMapping.StateValue {
       some disj eh1, eh2:activeHandlers {
  (eh1.trans[activeEvent][s.globalState].Seq[Event]).stateMapping[se] !=
  (eh2.trans[activeEvent][s.globalState].Seq[Event]).stateMapping[se]
       }
    }
 }
      }
   }
}

fun ObeyLegalBindingsTrans(s, s': CentralRegistryState) {
   let activeEvent = s.queue..SeqFirst() {
      let activeHandlers = acceptingHandlers(activeEvent, s) {
 let potentialBindings = activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
    all b:potentialBindings {
       //exState will be the extension to a State of b
       some exState:State {
  //ex state has all the se's of b
  all se:b.stateMapping.StateValue {
     se->b.stateMapping[se] in exState.stateMapping
  }
  //and it is extended to match the rest of what the global state was.
  all se:StateElement-b.stateMapping.StateValue {
     se->s.globalState.stateMapping[se] in exState.stateMapping
  }

  //now all we have to say is that s.globalState->exState is a legal trans
  s.globalState->exState in activeEvent.legalStateTrans
       }
    }
 }
      }
   }
}

fun AllowState(s:CentralRegistryState, exState:State) {
   all inv:s.registeredInvariants {
      some b':inv.allows {
 b'.stateMapping in exState.stateMapping
      }
   }
}



//utility fun that constrains a Bindings to match all Invariant from a given CRState
fun MatchInvariants(s: CentralRegistryState, b:Bindings) {
```

```
   some exState:State {
      //ex state has all the se's of b
      all se:b.stateMapping.StateValue {
 se->b.stateMapping[se] in exState.stateMapping
      }
      //and it is extended to match the rest of what the global state was.
      all se:StateElement-b.stateMapping.StateValue {
 se->s.globalState.stateMapping[se] in exState.stateMapping
      }
      //now exState is what will be the result of only asserting b.
      //we have to check that exState matches the invariants that are registered.
      AllowState(s, exState)
   }
}

//this fun states that if a bindings intersects with an invariant's state elements,
//then it must assert a value for each element
fun StrongInvariantAssertion(s:CentralRegistryState, b:Bindings) {
   all inv:s.registeredInvariants {
      (some b':inv.allows | some b'.stateMapping.StateValue & b.stateMapping.StateValue) =>
      {
 all b':inv.allows {
    b'.stateMapping.StateValue in b.stateMapping.StateValue
 }

      }
   }
}

/**This describes a CR system which has the property of allowing conflicting registrations for State*/
fun StateConflicts(cr:CentralRegistryState) {
   //there is a state conflict if there exists some event and some state for which there are two bindings produced that
   //agree on a se, but not an sv
   some disj eh1, eh2:EventHandler, disj ef1, ef2:EventFilter {
      ef1->eh1 in cr.registeredPairs
      ef2->eh2 in cr.registeredPairs
      some e:Event, s:State {
 e->s in ef1.match
 e->s in ef2.match
 let b1 = eh1.trans[e][s].Seq[Event] {
    let b2 = eh2.trans[e][s].Seq[Event] {
       some se:StateElement {
  some b1.stateMapping[se]
  some b2.stateMapping[se]
  b1.stateMapping[se] != b2.stateMapping[se]
       }
    }
 }
      }
   }
}



/*****************************************************************************
PROPERTIES

Properties are things about the pattern that we would like to check
*****************************************************************************/

/**This describes a CR system in which there is no global state*/
fun NoGlobalState() {
   no StateElement
   no StateValue
}

fun AsyncNoGlobalState() {
   asynchronousPolicy()
```

```
   NoGlobalState()
   TransactionSystem()
}

run AsyncNoGlobalState for 2 but 1 CentralRegistryState

fun NoConflictsSystem() {
   all s:CentralRegistryState {
      !StateConflicts(s)
   }
}

fun AsyncNoConflictsSystem() {
   NoConflictsSystem()
   TransactionSystem()
   asynchronousPolicy()
}
run AsyncNoConflictsSystem for 2 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]

/**This describes a CR system that has the property of guarenteeing a transaction behavior of event handlers*/
fun TransactionSystem() {
   all s:CentralRegistryState - Ord[CentralRegistryState].first {
      let s' = OrdNext(s) {
 TransactionTransition(s, s')
      }
   }
}

fun AsyncTransactionSystem() {
   TransactionSystem()
   asynchronousPolicy()
}

run AsyncTransactionSystem for 2 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]


/**This describes a CR system that has only commuting operations*/
fun CommutingOperationsSystem() {
   all s:CentralRegistryState {
      let s' = OrdNext(s) {
        OverlappingStateValues(s)
      }
   }
}

fun AsyncCommutingOperationsSystem() {
   CommutingOperationsSystem()
   TransactionSystem()
   asynchronousPolicy()
}

run AsyncCommutingOperationsSystem for 3 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]


/**This describes a CR system in which every handler obeys the legalStateTrans relation of Event.*/
fun LegalBindingsSystem() {
   all s:CentralRegistryState - Ord[CentralRegistryState].last {
      let s' = OrdNext(s) {
        ObeyLegalBindingsTrans(s, s')
      }
   }
}

fun AsyncLegalBindingsSystem() {
   LegalBindingsSystem()
   TransactionSystem()
   asynchronousPolicy()
}
```

```
run AsyncLegalBindingsSystem for 3 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]

/**This describes a CR system in which every handler obeys the legalStateTrans relation but asserts an entire State*/
fun LegalStatesSystem() {
   LegalBindingsSystem()
   all eh:EventHandler {
      eh.trans[Event][State].Seq[Event] in State
   }
}


fun AsyncLegalStatesSystem() {
   LegalStatesSystem()
   TransactionSystem()
   asynchronousPolicy()
}

run AsyncLegalStatesSystem for 3 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]


/**This describes a CR system in which every handler obeys the registered invariants*/
fun InvariantSystem() {
   all s:CentralRegistryState-Ord[CentralRegistryState].last {
      let activeEvent = s.queue..SeqFirst() {
 let activeHandlers = acceptingHandlers(activeEvent, s) {
    all b:activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
         MatchInvariants(s, b)
    }
 }
      }
   }
}

fun AsyncInvariantSystem() {
   InvariantSystem()
   TransactionSystem()
   asynchronousPolicy()
}

//this claims that if all the invariants hold in the pre-state, t hey will hold in the post state.
fun InvariantsHold() {
   all s:CentralRegistryState-Ord[CentralRegistryState].last {
      let s' = OrdNext(s) {
 AllowState(s, s.globalState) => AllowState(s', s'.globalState)
      }
   }
}

assert InvariantsFollow {
   InvariantSystem() => InvariantsHold()
}


assert StrongInvariantsFollow {
   AsyncStrongInvariantSystem() => InvariantsHold()
}


check InvariantsFollow for 2 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]
check StrongInvariantsFollow for 2 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]

run AsyncInvariantSystem for 3 but 2 CentralRegistryState, 3 Seq[Event], 2 Seq[Seq[Event]]

/**This describes a CR system in which every handler obeys the registered invariants
and also asserts a value for every se in an invariant it potentially modifies*/
fun StrongInvariantSystem() {
   all s:CentralRegistryState-Ord[CentralRegistryState].last {
```

```
      let activeEvent = s.queue..SeqFirst() {
 let activeHandlers = acceptingHandlers(activeEvent, s) {
    all b:activeHandlers.trans[activeEvent][s.globalState].Seq[Event] {
       MatchInvariants(s, b)
       StrongInvariantAssertion(s, b)
    }
 }
      }
   }
}

fun AsyncStrongInvariantSystem() {
   StrongInvariantSystem()
   TransactionSystem()
   asynchronousPolicy()
}

run AsyncStrongInvariantSystem for 3 but 2 CentralRegistryState, 3 Seq[Event], 3 Seq[Seq[Event]]



/**This describes the property that if an event e1 is generated before e2, then it will be dispatched before e2.**/
fun orderedDispatch() {
   all disj e1, e2:Event {
      genBefore(e1, e2) => dispBefore(e1, e2)
   }
}

/**This describes the property that if two events have the same priority, and if one is generated before the other,
the the order of generation will be the order of dispatch**/
fun orderedWithinTwoPriority() {
   all disj e1, e2:Event {
      samePriority(e1, e2) && genBefore(e1, e2) =>
 dispBefore(e1, e2)
   }
}

/**This describes the property that after sorting, the queue always has the higher priority events before the lower
priority events**/
fun HighPriorityBeforeLow() {
   no crs:CentralRegistryState {
      some crs.eventJustProcessed
      crs.eventJustProcessed in (Event - (RegisterEvent+DeregisterEvent))
      Unsorted(OrdPrev(crs).queue)
   }
}


/*****************************************************************************
ASSERTIONS

In each assertion, we check to see if a certain policy implies a certain property.
The Alloy Analyzer will search for counter examples to these assertions
*****************************************************************************/

/**This states that in the asynchronous policy, with no event duplicates, ordering of dispatch is guarenteed**/
assert OrderOfDispatchAsync {
   asynchronousPolicy() && noDuplicateEvents() => {
      orderedDispatch()
   }
}

/**This asserts that in the synchronous policy, with no event duplicates, ordering of dispatch is guarenteed**/
assert OrderOfDispatchSync {
   synchronousPolicy() && noDuplicateEvents() => {
      orderedDispatch()
   }
}
```

```
/**this asserts that in the presence of the reordering queue policy, order of dispatch is guarenteed.
We expect this to be false**/
assert OrderOfDispatchWithReordering {

   twoPriorityQueuePolicy() => {
      orderedDispatch()
   }
}

/**This asserts that, within a priority, order of dispatch is guarenteed**/
assert OrderOfDispatchWithinPriority {
   twoPriorityQueuePolicy() => {
      orderedWithinTwoPriority()
   }
}

/**This asserts that the queue policy maintains queues that are correctly sorted.**/
assert PrioritiesHold {
   twoPriorityQueuePolicy() => {
      HighPriorityBeforeLow()
   }
}


/*******************************************************************************
EXECUTING ALLOY

We have written several commands that the Alloy Analyzer can execute.
*******************************************************************************/

check PrioritiesHold for 3 but 4 CentralRegistryState, 4 Seq[Event]
check OrderOfDispatchWithinPriority for 3 but 4 CentralRegistryState, 4 Seq[Event]
check OrderOfDispatchWithReordering for 3 but 5 CentralRegistryState, 5 Seq[Event]
check OrderOfDispatchAsync for 3 but 4 CentralRegistryState, 4 Seq[Event]
check OrderOfDispatchSync for 3 but 4 CentralRegistryState, 4 Seq[Event]
run synchronousPolicy for 3 but 4 CentralRegistryState, 4 Seq[Event]
run asynchronousPolicy for 3 but 4 CentralRegistryState, 4 Seq[Event]
run twoPriorityQueuePolicy for 3 but 4 CentralRegistryState, 4 Seq[Event]
```

Bibliography

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[2] J. Dingel, D. Garlan, S. Jha, D. Notkin. Reasoning about implicit invocation. In *International Symposium on the Foundations of Software Engineering*, Nov. 1998.

[3] D. Jackson, I. Shlyakhter, M. Sridharan. A Micromodularity Mechanism. In *Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01)*, September 2001.

[4] D. Jackson, J. Chapin. Redesigning Air-Traffic Control: A Case Study in Software Design. In *IEEE Software*, May/June 2000.

[5] A. Lauder. EventPorts. In *ECOOP Workshop for PhD Students in OO Systems*, 1999.

[6] D. Garlan, S. Khersonsky. Model Checking Implicit-Invocation Systems. In *10th International Workshop on Software Specification and Design*, November 2000.

[7] Course material for MIT Laboratory in Software Engineering (6.170). Available electronically at http://web.mit.edu/6.170/archive/Old-Fall01/psets/gb/gizmoball.html.

[8] D. Jackson *Lecture Notes on Software Design.* Available electronically at http://theory.lcs.mit.edu/~dnj/pubs/fall00-lectures.pdf. Fall 2000.

[9] D. Jackson, A. Fekete. Lightweight Analysis of Object Interactions. In *Fourth International Symposium on Theoretical Aspects of Computer Software*, Japan, October 2001.

[10] Vlissides, J. "Pattern Hatching: Multicast" C++ Report, February 1997.

[11] D. Garlan, G. Kaiser, D. Notkin. Using Tool Abstraction to Compose Systems. In *IEEE Computer* June 1992.

[12] B. Liskov, J. Guttag. *Program Development in Java; Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley, 2001.

[13] D. Parnas. Designing software for ease of extension and contraction. In *IEEE Transactions on Software Engineering*, 1979.

[14] G. Sullivan. *Design Patterns in a Dynamic Language.* Guest lecture in MIT Advanced Topics in Software Design (6.898). Available electronically at http://www.ai.mit.edu/~gregs/proglangsandsofteng.pdf. April 2002.

[15] L. Lin. *Visualization Framework for Software Design Analysis* MIT Thesis, May 2002.