Using Synchronization Contracts in ECharts

1 Introduction

ECharts is a programming language for creating finite state-machines. A state machine is translated into a Java class, whose instance is executed by ECharts runtime on a dedicated thread.

ECharts runtime executes state actions or transition actions as it cycles through a state machine. These actions must embed synchronization details to avoid data-race and deadlock among concurrent machine instances.

Manually implementing synchronization details is complex and error-prone. In high-concurrent SIP containers, for example, a service execution must examine a series of session objects, shared beans and database connections, which are possibly also shared by other concurrent service executions, to know how to process a message and what other shared resources it also needs to use in processing the message.

This document describes an extension to ECharts that decouples the synchronization concern. This extension consists of a notational add-on to ECharts language and an operational add-on to ECharts runtime.

2 Overview

The basic idea of the extension is to leverage synchronization contracts to separate the synchronization concern, which is automatically handled by a synchronization middleware composed of negotiators. Synchronization is simplified because contracts are declarative and because negotiators are created by experts. Negotiators are reusable and pluggable, which facilitate performance-tuning.

Using the extension is straightforward. Instead of embedding low-level synchronization details in actions, a programmer uses high-level contracts to specify the shared resources that a machine instance needs to safely access during a transition; the runtime then uses negotiators to enforce contracts by automatically scheduling concurrent machine instances. For example, the runtime never fires two transitions concurrently if their contracts overlap.

Contracts are specified using @need annotations. These annotations can be embedded at various places of a state-machine to affect different transitions. Each annotation declares a set of resources. The contract of a transition subsumes those resources that are declared in any of the annotations that affect the transition.
A negotiator class implements a synchronization algorithm, which acquires a set of resources without incurring deadlock. Example algorithms include resource numbering, gatekeeper, global locking, and so on. These algorithms differ from each other in allowed concurrency, fairness, and introduced overhead. The runtime can be configured to use any negotiator class.

3 @need annotation

This section describes the usage of @need annotations in more detail. Refer to ECharts documentation for the details specific to ECharts language.

3.1 Types

Depending on its placement, a @need annotation belongs to one of three types: guard-action need, action need, and common need.

A guard-action need is placed between a transition’s port event (if any) and the same transition’s guard. An action need is placed between a transition’s guard (if any) and the same transition’s action. A common need is placed within a state-machine and on the same level with its transitions.

![Placement of guard needs and action needs](image)

Figure 1: Placement of guard needs and action needs

**Figure 1** illustrates the placement of guard-action needs and action needs. The placement of common needs is much more flexible, so long as the annotation is placed on the same level with state and transition definitions. **Figure 2** gives some example cases.

![Some examples of placing common needs](image)

Figure 2: Some examples of placing common needs
3.2 Scopes

A guard-action need or an action need applies to an individual transition, whereas a common need applies to all the transitions of a state-machine, including those in the nested state-machines.

The scope of a guard-action need includes the guard and the action of its applied transition. The scope of an action need includes the action of its applied transition. The scope of a common need includes the guards and the actions of all its applied transitions.

3.3 Accessibility

<table>
<thead>
<tr>
<th>Resources specified in ...</th>
<th>t's guard-action need</th>
<th>t's action need</th>
<th>the common need of ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>t's port event</td>
<td>Unsafe</td>
<td>Unsafe</td>
<td>Unsafe</td>
</tr>
<tr>
<td>t's guard</td>
<td>Read</td>
<td>Unsafe</td>
<td>Read</td>
</tr>
<tr>
<td>t's action block</td>
<td>Write</td>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>t's entry action</td>
<td>Write</td>
<td>Write</td>
<td>Write</td>
</tr>
<tr>
<td>t's exit action</td>
<td>Write</td>
<td>Write</td>
<td>Write</td>
</tr>
</tbody>
</table>

Table 1: Resource accessibility for transition t of state-machine m

In evaluating a guard or executing an action, a machine instance can safely access only those resources specified in the @need annotations whose scopes subsume the guard or the action. Table 1 illustrates when a transition can safely use shared resources.

3.4 Resource declaration

A @need annotation declares one or more shared resources. A resource declaration consists of three parts: a type, a name, and an expression to locate the actual resource. The name is optional, whereas the other parts are mandatory.

A resource can be declared in one of two ways. The first way is use a member variable to directly refer to the actual resource. For example, the following action need declares two resources of type Fork, which can be located using member variables left and right that are initialized in the machine constructor. Notice that resource names are omitted.

```java
public machine Philosopher {
    <* Fork left, right: *>
    public Philosopher(Fork left, Fork right) {
        this.left = left;
        this.right = right;
    }
    initial state Thinking;
```
The second way to declare a resource is to call a member method or a global function that returns the reference to the actual resource. The returned reference is assigned to a resource name that can be used in a transition’s guard or action. For example, the above code can be also written as follows.

```java
public machine Philosopher {
    <∗ Fork left, right; ∗>
    public Philosopher(Fork left, Fork right) {
        this.left = left;
        this.right = right;
    }
    <∗ public Fork getLeftFork() { return left; } ∗>
    <∗ public Fork getRightFork() { return right; } ∗>

    initial state Thinking;
    transition Thinking −/
    @need {
        Fork l = getLeftFork();
        Fork r = getRightFork();
    }
    eatWith(l, r) −> Thinking;
}
```

A resource name can be null, meaning that it does not have to point to a concrete resource.

## 4 Negotiators

A negotiator implements a synchronization algorithm, which acquires a set of shared resources without incurring deadlock. An optimal negotiator should also promote concurrency, reduce overhead, and ensure fairness. A negotiator class can be reused by application programmers.

A negotiator class uses `template` pattern. It has a firm internal logic that determines when to acquire resources. It relies on its provided interface implementation to determine how to acquire resources. The knowledge about what resources to acquire is specified in contracts.
Normally, the interface of a negotiator class is implemented by experts. This section describes negotiators in more detail.

4.1 Life cycle

A negotiator is one-to-one correspondent to a thread. A negotiator is created before a thread runs a machine instance and is destroyed after the machine instance is quiescent.

4.2 Operations

A negotiator performs four operations on a shared resource—claim, cancel, commit, and release. A thread can read a resource only after it calls its negotiator’s claim operation; it can write a resource only after it calls its negotiator’s commit operation. Figure 3 illustrates the idea.

4.3 Implementing synchronization algorithms

Various synchronization algorithms can be implemented by specializing the negotiator operations. A key task of these operations is coordinate calls from multiple negotiators. For example, in order to permit multiple readers or a single writer, an algorithm would allow multiple negotiators to be claiming a resource at the same time and allow at most one negotiator to be owning a resource at any time. Calls to cancel or release would activate one or more waiting negotiators.

As an example, the following code implements a simple locking algorithm.

```java
public class NegotiatorSimpleLocking extends SyncNegotiator {
    public void claim(Object X) {
        lock(X);
    }
    public void cancel(Object X) {
        unlock(X);
    }
    public void commit(Object X) { }
    public void release(Object X) {
        unlock(X);
    }
}
Although simple, the above algorithm is deadlock prone. A global locking algorithm or a gatekeeper algorithm can be implemented to avoid such deadlock. The former can be implemented if $\text{claim}(X)$ acquires a global lock; the latter can be implemented if $\text{claim}(X)$ acquires a global lock and if $\text{commit}(X)$ acquires $X$ and releases the global lock. More complex algorithms, such as resource numbering and Szumo, can also be implemented via specializing these four operations.

### 4.4 Internal template logic

In firing a transition $T$ of machine $M$, a thread uses its negotiator to claim resources $S_1$ before evaluating $T$’s guard. Therefore, the guard can safely read the resources in $S_1$. If the guard’s value is false, the negotiator cancels the resources in $S_1$. Otherwise, the negotiator claims additional resources $S_2$ and then commits $S_1$ and $S_2$. Therefore, the actions executed subsequently can safely read and write the resources in the both sets. These actions include the transition’s action block, the exist action block of the source state, and the entry action block of the target state. When the execution of these actions completes, the negotiator releases all the resources and repeats if more transitions need to fire.

In the above template, $S_1$ is created from the following sources.

- $T$’s guard-action need
- $M$’s common need
- $P$’s common need, if $P$ is a parent machine of $M$

$S_2$ is created from $T$’s action need. It is up to the negotiator to decide in which order resources within $S_1$ and $S_2$ should be claimed. However, such order must respect data dependency. In the following annotation, for example, resource $u$ should be claimed after resource $um$.

```java
@need {
    UserManager um;
    User u = um.getByID(uid);
}
```

### 5 Example

This section illustrates the ideas described above using a conference service. This service is implemented as a SIP application using ECharts for SIP Servlets (E4SS). In this service, each machine instance models the behavior of one participant. The following code fragment is used by one participant to terminate the entire conference.
Terminating a conference is performed in a super-state (line 3) and involves tearing down each participant in the conference one after another (line 4–9). Iterator `uiitr` is initialized before entering the super-state. The transition action (line 7) is executed for each participant that `uiitr` returns. This action requires two resources: the current machine instance (line 2) and the machine instance of the participant to tear down (line 6). The former is specified as a common need of the top-level machine; whereas the latter is specified as the action need of a transition in the sub-machine. The current machine instance leaves the conference, when it tears down all the participants (line 10).