SOFTENG 461 - Formal Methods

Assignment 3
Alloy Research Project

Modelling Resource Allocation and Deadlock Avoidance

Jagannath VSB
Krishneel Prasad
Prasanthan Thaveenthiran
Sunil Lath

May 31, 2004

Abstract

Deadlock avoidance is a method of preventing deadlocks by prohibiting any allocations which result in cyclic patterns in a special purpose graph called Resource Allocation Graph. The Alloy model for deadlock avoidance consists of signatures representing Processes, Resources and System States. The signature for System States contains mappings for claims, requests and allocations which form the Resource Allocation Graph. The request, allocate and de-allocate state transition functions are used to control transitions from one state to another state. The transition functions also maintain the Resource Allocation Graph and implement the deadlock avoidance algorithm. Automatic execution of the model is ensured by ordering the system states in such a way that each state is the result of applying a transition function on the previous state.

Contents

1 Introduction 2
2 Assumptions 2
3 Deadlock Avoidance — Key Concepts 2
  3.1 Resource Allocation 2
  3.2 Deadlock 2
  3.3 Implementing Deadlock Avoidance 4
4 Final Model 4
  4.1 Signatures 4
  4.2 Functions 6
  4.3 Cyclic Detection 9
  4.4 Model Execution 11
5 Earlier Partial Models 12
6 Conclusions 13
7 Appendices 15
  7.1 Appendix 1: Final Model 15
  7.2 Appendix 2: Model 1 19
  7.3 Appendix 3: Model 2 20
  7.4 Appendix 4: Model 3 21
  7.5 Appendix 5: Meeting Minutes 24

1 Introduction

Deadlock avoidance is one of methodologies used by Operating Systems to prevent deadlocks among running processes which need certain resources to complete their tasks. It depends on controlling the resource allocation process by ensuring that no allocations which might lead to deadlocks are made. This is achieved by maintaining a special graph called a resource allocation graph and checking for certain patterns in that graph.

This report discusses an Alloy model developed to describe and observe the deadlock avoidance methodology. Key concepts related to deadlock avoidance are described in Section 3. The final model evolved out of many partial models which are briefly described Section 5. The final model is described in detail in Section 4.

2 Assumptions

- There is only one instance of each type of resource.
- Execution of a process is instantaneous and the resources held by a process are only released once all required resources are allocated to the process.
- The process resource requirements are known *a priori*. So this means that the resources required by a process never change and that resource requirements are not conditional.

3 Deadlock Avoidance — Key Concepts

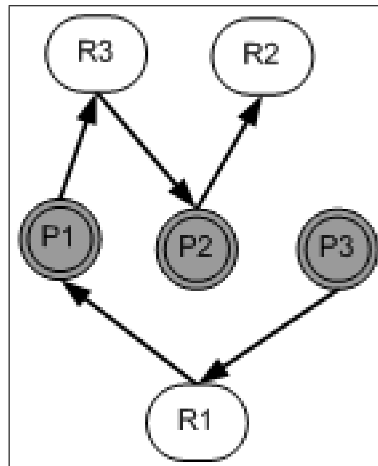
3.1 Resource Allocation

The basis of deadlock is the need for resources by processes. A process has a set of resources it needs in order to complete. Any process will make a request for a resource it needs when it requires it. The system will allocate the resources requested. The manner in which the resources are allocated will not be arbitrary and is the basis behind Deadlock Avoidance. Below is a diagram that depicts resource allocation and resource needs. The directed edge from R1 to P2 is an allocation of Resource 3 to Process 2. The edge from P1 to R3 is a request for Resource 3 which is need by Process 1.

3.2 Deadlock

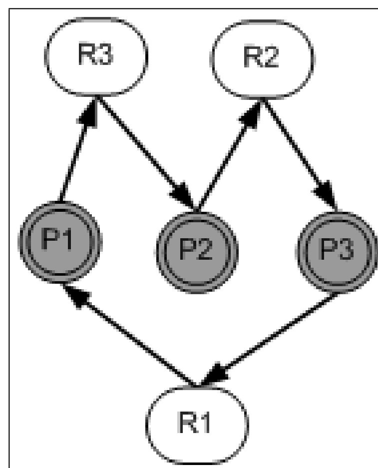
Often situations arise where a process enters a waiting state because it requires an unavailable resource. This resource will be held by some other process. Deadlock can arise in such a situation if the process holding the resource is also

Figure 1: Showing resource allocation and request by processes



waiting. This can be best illustrated as a circular dependency. The figure below describes deadlock in a resource allocation graph. As seen in the diagram every process is requesting a resource but all those resources are currently allocated to another process.

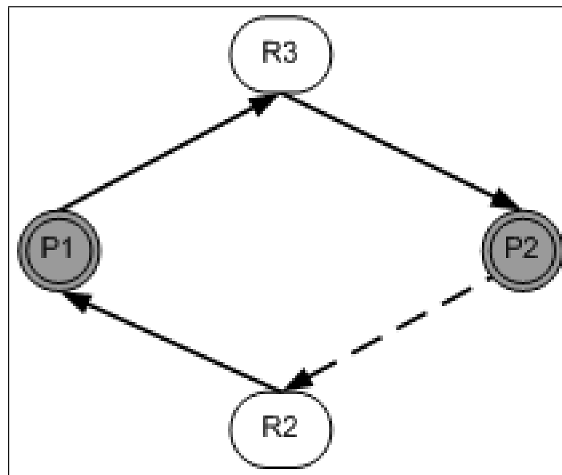
Figure 2: Showing a deadlock situation in a resource allocation graph



3.3 Implementing Deadlock Avoidance

To implement avoidance a third kind of edge needs to be incorporated into the resource allocation graph. This is called a claim edge. What this edge signifies is that a process will, at some time, require the resource pointed to. This also implies that all the resources that may be requested by a process are known in advance. With this edge we can control requests made by a process. Before an allocation is made the claim edges can be checked to see if a request would result in a circular dependency or deadlock. The dotted line in the diagram represents a claim edge. When making a request we check the possible effect of adding this request. If deadlock results then the request is denied.

Figure 3: Showing a deadlock situation in a resource allocation graph



4 Final Model

4.1 Signatures

Resource

A system state consists of a finite number of resources to be distributed among competing processes. The `Resource` signature is used to represent an instance of a resource in the system without having fields of its own.

```
sig Resource {
}
```

Process

The system state consists of resources as well as processes that use those resources. A process may use a resource in only the following sequence, request, allocate and then de-allocate. The `Process` signature is used to model a process. The property of requesting and allocating resources is time dependent and is therefore modeled as separate functions.

```
sig Process {
}
```

System State

The `SystemState` signature models the set of resource claims, requests and allocations by processes.

```
sig SystemState {
    claims: Process -> Resource,
    requests: Process -> Resource,
    allocations: Resource -> Process
}{
    all r: Resource | sole r.allocations
    no claims & requests
    no requests & ~allocations
    this != FinalState => runFunction(this, OrdNext(this))
}
```

Both `claims` and `requests` are a ternary relation mapping processes to resources. Allocations on the other hand map resources to processes. All three have a similar structure and it is convenient to place them in `SystemState`, allowing us to place constraints on them easily. By joining all system resources `r: Resource` with allocations `r.allocations`, we get a set of processes. By using `sole r.allocations`, we ensure that at most one process is returned which means that every resource is allocated to at most one process.

The use of the `no` operator is to ensure there are no common mappings between `claims` and `requests` i.e. a claim and request cannot be the same. Similarly, `no requests & ~allocations` ensures that a request and an allocation cannot be the same.

Initial State

The `InitialState` extends `SystemState` because it represents the system state as it would be at the start of the system. It inherits all the properties of `SystemState` and places additional constraints. The model could thus refer

to the `InitialState` directly instead of calling `SystemState` with constraints wrapped around it.

```
static disj sig InitialState extends SystemState {
}
{
    all p:Process, r:Resource | some p.claims && some claims.r
    no requests
    no allocations
    Ord[SystemState].first = InitialState
}
```

Final State

The `FinalState` signature extends `SystemState` and is similar to `InitialState` except that it adds the extra constraint `no claims` instead of having `some claims`. The final state is also the last state in the ordering of the states.

```
static disj sig FinalState extends SystemState {
}
{
    no claims
    no requests
    no allocations
    Ord[SystemState].last = FinalState
}
```

4.2 Functions

Run Function

```
fun runFunction(state, state': SystemState) {
    (some state.requests && allocate(state, state')) ||
    (some state.allocations && deAllocate(state, state')) ||
    (some state.claims && request(state, state'))
}
```

The `runFunction` represents the types of functions that can be invoked which cause a change in the system state. It runs only one of the three functions, resource request, allocation or de-allocation. In order for a de-allocation, there must already be an allocation present in the state. An allocation requires there be a request present in the state and a request requires a claim to be present in the state. These constraints ensure that a more applicable solution is found without wasting the Alloy search space.

Allocate Function

```
fun allocate(state, state': SystemState){
    all process: state.requests.Resource |
    allocateOneResource(process, state, state') ||
    doNotAllocateAnyResource(process, state, state')
    Resource.(state'.allocations - state.allocations)
    in state.requests.Resource
    state'.requests in state.requests
    state.allocations in state'.allocations
    state.claims = state'.claims
}
```

The purpose of the above mentioned function is to depict the transition from one SystemState to the next under a resource allocation. If possible it will allocate a resource to each process that is waiting on resources. The following line of code manages the conditional allocation of resources:

```
allocateOneResource(process, state, state') ||
doNotAllocateAnyResource (process, state, state')
```

The `allocateOneResource` function ensures:

- That there is only resource allocated to the specified process.
- That the resource allocated to the process is a resource that was requested by the process.
- The request that was converted to an allocation should no longer remain a request. This will prohibit inherent cycles that we wish to avoid.
- Finally a check is made that the allocation results in no cycles that could result in a deadlock.

If the above mentioned constraints can not be satisfied it is expected that the `doNotAllocateAnyResource` function will be called. This function ensures:

- That no resource is allocated to the specified process.
- It also ensures that all the previous requests are maintained in the next SystemState and that all allocations are carried over as well.

Once the allocation decision is made the function performs a few checks on whether the allocations were completed successfully. One major constraint applied is that only resources that had requests were allocated a resource. This is achieved by the following line of code:

```
Resource.(state'.allocations - state.allocations)
in state.requests.Resource
```

The function also adds constraints to ensure no new requests are made and that claims are maintained over the allocation

De-Allocate

```
fun deAllocate(state,state':SystemState){
    all process: state.allocations[Resource] |
    deAllocateIfComplete(process, state, state')
    state'.allocations in state.allocations
    state.claims = state'.claims
    state.requests = state'.requests
}
```

This function is used to free any resources allocated to a process that is completed. A process is considered completed when it has been allocated all the resources it has claimed. The function checks all processes that have been allocated resources and calls the `deAllocateIfComplete` function. The central part of this function is ensuring that a process has been allocated all its resources before de-allocating it. This is achieved by this line in the `deAllocateIfComplete` function: `no state.requests[process] && no state.claims[process]`. It implies that there are no claims or requests for the specified process as the process has been allocated its requests.

Apart from de-allocating the resources the function also ensures that no new allocations are made at this stage and that all previous claims and requests for resources are maintained.

Request Function

```
fun request(state, state': SystemState) {
    all process: state.claims.Resource |
    makeSomeRequests(process, state, state')
    (state'.requests - state.requests).Resource
    in state.claims.Resource
    state'.claims in state.claims
    state.requests in state'.requests
    state'.allocations = state.allocations
}
```

The request function intakes a system state and tries to convert some of the claims (resources required initially by the process) of all processes into requests and updates the state as the output (`state'`).

The segment of code `all process: state.claims.Resource` returns all the process in the current state those have some claims, and each of these processes are passed on to the function `makeSomeRequests`.

The line `state'.requests - state.requests`. Resource in `state.claims .Resource` ensures that the claims in the new state are updated according to the requests made by the processes.

The lines `state.requests in state'.requests` and `state'.allocations = state.allocations` just makes sure existing requests and allocations are carried over to the new state.

Make Some Requests

```
fun makeSomeRequests(process: Process, state, state': SystemState) {
        {some process.(state'.requests - state.requests)
        process.(state'.requests - state.requests)
        in state.claims[process]
        state'.claims[process] = state.claims[process] - process.
        (state'.requests - state.requests)}
}
```

The function `makeSomeRequests` intakes a particular process to convert some of its claims into requests. The segment of code `some process.(state'.requests - state.requests)` ensures that the modified process's requests are changed from previous state.

The line `process.(state'.requests - state.requests) in state .claims[process]` obtains the set of resources that are involved in new requests by performing set difference operation `state'.requests - state.requests` and join operation on it with the current process. The resources obtained are restricted to be from the claims of the process in previous state. This ensures that only resources involved in claims are converted into requests.

The last line `state'.claims[process] = state.claims[process] - process.(state'.requests - state.requests)` updates the claims of the process by removing the claims which are converted into requests.

The segment of code `state'.claims[process]` obtains resources left in the claims that aren't converted to requests and equates it to difference of all the resources in previous state claims of the process `state.claims[process]` with claim resources those are converted to requests `process.(state'.requests - state.requests)`.

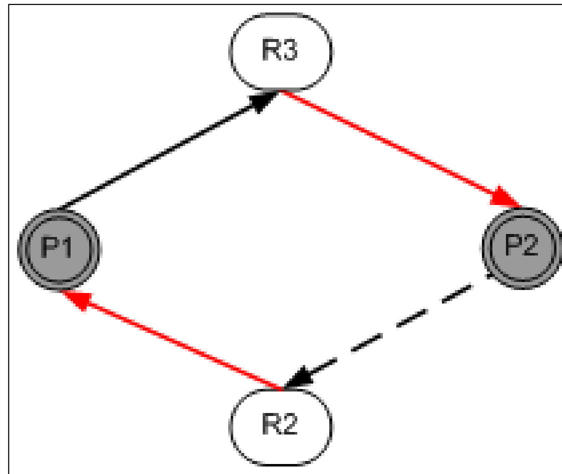
4.3 Cyclic Detection

```
fun noCycleCreated(state: SystemState) {
    no (iden[Process] &
```

```
    ^((state.requests + state.claims).state::allocations))
}
```

This section of code is written to detect the common cyclic request/allocation situations which are certainly guaranteed to produce a deadlock. The code takes advantage of built in functions *transitive closure* and *iden[Set]* to detect the deadlock. The transitive function work on homogenous binary relation in the form of union of multiple joins such as $r + r.r + r.r.r + ...$ limit of the series. While *iden[Set]* produces reflexive relation with all of Set's elements map to themselves. As explained earlier, deadlock arises from processes competing for the same resources to achieve completion. For example if two processes are competing for two same resources, there is a chance that order of allocation of resources could produce simple deadlock situation.

Figure 4: Graph showing a binary deadlock situation



Clearly this is a deadlock situation where Resource 3 is being held by Process 2 and Resource 2 is being held by Process 1, while Process 1 needs Resource 3 and Process 2 needs Resource 2.

Union of `state.requests + state.claims` returns the binary relation with right type containing elements from Process set and left type containing elements from Resource set. The `state::allocation` simply returns another binary relation with Resource mapping to Process (inverse of previous). By joining these two binary relations, a homogenous relation can be obtained with its type being Process. This is where *iden[Set]* becomes highly useful. Transitive closure of the homogenous relation obtained by join operation will have at least one reflexive mapping (where a process maps to itself) when there is deadlock situation. Hence intersection of *iden[Set]* and resultant of transitive closure

operation should be empty for non deadlock situations. At the first glance it might look as a bold statement but in close inspection this is a vigorous condition.

In the above example union of `state.requests + state.claims` will contain `P1->R3` and `P2->R2`, while `state::allocation` will return relation containing mappings `R3-> P2` and `R2-> P1`. The result of the joined operation on these relations is such as below

$$\begin{array}{|c|c|}\hline P1 & R3 \\ \hline P2 & R2 \\ \hline \end{array} \bullet \begin{array}{|c|c|}\hline R3 & P2 \\ \hline R2 & P1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline P1 & P2 \\ \hline P2 & P1 \\ \hline \end{array}$$

Transitive closure of resultant relation (`P1 -> P2` and `P1 -> P2`) will produce another homogenous relation with mappings `P1 -> P2, P1 -> P2, P1 -> P1` and `P2 -> P2`. There are two reflexive mappings `P1 -> P1` and `P2 -> P2` which indicates common interest of resource for processes.

$$\hat{} \begin{array}{|c|c|}\hline P1 & P2 \\ \hline P2 & P1 \\ \hline \end{array} = \begin{array}{|c|c|}\hline P1 & P2 \\ \hline P2 & P1 \\ \hline P1 & P1 \\ \hline P2 & P2 \\ \hline \end{array}$$

4.4 Model Execution

The execution of the model is somewhat automated. All the execution requires is the number of System States, Resources and Processes. Therefore running a simple function like the one described below will result in a sequence of System States that provide completed resource allocation without deadlock.

```
fun System() {}

run System for 2 Process, 5 Resource, 10 SystemState"
```

The way in which this automated execution is achieved is described below:

- Firstly there is a need to have an ordering on the SystemState.
- The critical aspect to the automated execution is that the sequence of System States is based on the runFunction described in Section 4.2. The code below found in the signature for SystemState shows how the next system state is determined.

```
this != FinalState => runFunction(this, OrdNext(this))
```

The term this refers to the current System State. Another outcome of this automated execution is that each system state signifies an allocation, request or

de-allocation. Alternatives to running the above mentioned system execution function would be to specify details such as the how many process can simultaneously request or claim a resource. Examples of such functions are provided in the appendix showing the entire final model.

5 Earlier Partial Models

In Model 1, the system state was represented by a set of resources mapped to a set of processes by the relation `allocations`. An allocation was constrained such that each resource could be allocated to at most one process. Model 1 has a function `allocate` that checks that a resource has not yet been allocated to a process and allocates it to the given process.

```
fun allocate(state, state': SystemState, process: Process,resource:
    set Resource){
    no (resource & process.(state.allocations))
    some resource
    all r:Resource | r in resource => r
    in process.(state'.allocations)
}
```

The `allocate` function takes in a process and a set of resources. After the resource is allocated to the process, the system state changes and is represented by `state'` which basically contains an extra process to resource mapping in its `allocations` relation.

Model 2 introduced the relation `needs` which are a set of resources that a system process uses to perform jobs. So a process now needs a specific set of resources which is more realistic e.g. in a given system, a process would only need the tape drive and printer resources to complete a certain printing job and wouldn't care about other resources in that system. The `allocate` function no longer has to take in a process and a set of resources since a process has a needs relation which defines the set of resource that it requires. Model 2 also explicitly defines a set of waiting processes in the system state under the relation `waitingProcesses`. A waiting process is a process that hasn't satisfied all its resource needs. This is indicated by `all p: Process |some (p.needs - p.allocations) <=> p in waitingProcesses`. An `InitialState` was also introduced which ensured there was no allocations made at the start of the system. Deadlock can easily arise in such a system since resources are randomly assigned to waiting processes and no constraints are placed to avoid this yet.

Model 3 implements a deadlock avoidance scheme. The model replaces the `needs` relation with a `claims` relation and introduces an extra function requests. The `claims` relation says that a process will, at some time, require the resource pointed to. Thus if a process claims some resources, a request can be made for that claim. The new requests should only be for resources that are claimed,

`process.(state'.requests - state.requests) in state.claims[process]`. The system will only allocate after checking that no circular dependencies are found in the system and the process has previously made a request.

6 Conclusions

Modelling deadlock avoidance using Alloy proved to be a challenging and educational experience. It has given us a very good understanding of the powerful modelling and analysing capabilities of Alloy. The part of the model that we take most pride in is the automatic running of the model achieved by ordering the system states using the Ordering module provided with Alloy. The state transition functions (request, allocate and de-allocate) were the most challenging parts of the model and required numerous refinements to ensure all situations were dealt with correctly. The model reiterates that deadlock avoidance ensures that the system never encounters a deadlock albeit being a very cautious approach which may unnecessarily prevent some allocations.

Bibliography

[1] G. G. Abraham Silberschatz, Peter Galvin, *Applied Operating System Concepts*. 605 3rd Avenue, New York, NY, 1051—0012: John Wiley & Sons, 2003.

[2] Operating System Notes. 2003; Available at URL: `http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/deadlockAvoidance.htm` Accessed May 20, 2004.

7 Appendices

7.1 Appendix 1: Final Model

```
module DeadlockAvoidance
open std/ord

// A system process which uses resources to perform jobs.
sig Process {
}

// A resource which is used by processes to complete a job.
sig Resource {
}

// A snapshot of the state of resource allocations in a system at a particular
time.
sig SystemState {

        // A set of mappings from processes to resources indicating that a
        // process needs a resouce to complete.
        claims: Process -> Resource,
        // A set of mappings from processes to resources indicating that the
        // process has requested allocation of a resource.
        requests: Process -> Resource,
        // A set of mappings from resources to processes indicating that a
        // resource has been allocated to a process.
        allocations: Resource -> Process
}{
        // Each resource can be allocated to at most one process
        // There can be no claims for which requests exist.
        // There can be no requests for which allocations exist.
        all r: Resource | sole r.allocations
        no claims & requests
        no requests & ~allocations

        // If this state is not the final state, the next state should be
        // the result of running a state transition function on this state.
        this != FinalState => runFunction(this, OrdNext(this))
}

// The initial system state where no allocations and requests are made.
static disj sig InitialState extends SystemState {
} {
        // Each process should have some claims similarly each
```

```
        // Resource should be claimed by atleast one process.
        all p:Process, r:Resource | some p.claims && some claims.r
        no requests
        no allocations

        // The initial state has to be the first state in the ordering of System States
        Ord[SystemState].first = InitialState
}

static disj sig FinalState extends SystemState {
} {
        no claims
        no requests
        no allocations

        // The initial state has to be the first state in the ordering of System States
        Ord[SystemState].last = FinalState
}

fact {
        // All the processes, resources and systemStates should be included in the
model.
        Process = univ[Process]
        Resource = univ[Resource]
        SystemState = univ[SystemState]
}

// A function which runs one of three possible state transition functions
// on state to produce state'(which comes after "state" in the ordering).
fun runFunction(state, state': SystemState) {
        // Run the allocate transition function if there are some requests
        // Run the deAllocate transition function if there are some allocations
        // Run the request transition function if there are some claims
        (some state.requests && allocate(state, state')) ||
        (some state.allocations && deAllocate(state, state')) ||
        (some state.claims && request(state, state'))
}

// A function which allocates(if possible) a single resource to each waiting
process.
fun allocate(state, state': SystemState){

        // For all processes which have requests make one allocation if possible.
        all process: state.requests.Resource |
        allocateOneResource(process, state, state') ||
        doNotAllocateAnyResource(process, state, state')
```

```
        //Ensuring that the processes that have got new allocations are processes
        // which had requests.
        // No new requests should be made.
        // Previous allocations should be carried over.
        // All claims have to be carried over.
        Resource.(state'.allocations - state.allocations) in state.requests.Resource
        state'.requests in state.requests
        state.allocations in state'.allocations
        state.claims = state'.claims
}

// Allocate one resource to the process if deadlock can be avoided.
fun allocateOneResource(process: Process, state, state': SystemState) {
        // One new allocation should be made for this process.
        {one process[state'.allocations - state.allocations]

        // The allocation should be for a resource that was requested.
        process[state'.allocations - state.allocations] in state.requests[process]

        // The request for the allocated process should be removed.
        state'.requests[process] = state.requests[process] - process[state'.allocations
- state.allocations]

        // No cycle should be created by the allocation (avoiding deadlock!!!).
        noCycleCreated(state')}
}

// A function that ensures that there is no cycle in the mappings of the
SystemState.
fun noCycleCreated(state: SystemState) {
        no (iden[Process] & ^((state.requests + state.claims).state::allocations))
}

// A function that ensures that there is a cycle in the mappings of the
SystemState.
fun CycleCreated(state: SystemState){
        some (iden[Process] & ^((state.requests + state.claims).state::allocations))
}

// Make no allocations and ensure that the allocation information is carried
over.
fun doNotAllocateAnyResource(process: Process, state, state': SystemState) {
        // There should be no new allocations for this process.
        {no process[state'.allocations - state.allocations]
```

```
        // All requests and allocations should be carried over.
        state.requests = state'.requests
        state.allocations = state'.allocations}
}

// A function which de-allocates resources which are allocated to completed
processes.
// A completed process has no claims and requests left.
fun deAllocate(state,state':SystemState){

        // For all processes de-allocate resources if completed.
        all process: state.allocations[Resource] | deAllocateIfComplete(process, state,
state')

        // No new allocations should be made.
        // All claims and requests should be carried over.
        state'.allocations in state.allocations
        state.claims = state'.claims
        state.requests = state'.requests
}

// If a process is complete then de-allocate all resources otherwise
// carry over all the present allocations.
fun deAllocateIfComplete(process: Process, state, state': SystemState) {
        no state.requests[process] && no state.claims[process] =>
        (no state'.allocations.process),
        (state.allocations.process = state'.allocations.process)
}



// A function which makes a single request for each process.
fun request(state, state': SystemState) {

        //If a process claims some resources a request is made for some of those
resources.
        all process: state.claims.Resource | makeSomeRequests(process, state, state')

        // All new requests should be for processes which had claims.
        // No new claims are made.
        //Already present requests are copied over to the next state.
        (state'.requests - state.requests).Resource in state.claims.Resource
        state'.claims in state.claims
        state.requests in state'.requests

        //Allocations are not changes between the two states.
```

```
        state'.allocations = state.allocations
}

// Some new requests should be made for each process with claims.
// The new requests should be for resources that were claimed.
// All claims that were not requested should be copied over to the next state.
fun makeSomeRequests(process: Process, state, state': SystemState) {
        {some process.(state'.requests - state.requests)
        process.(state'.requests - state.requests) in state.claims[process]
        state'.claims[process] = state.claims[process] - process.(state'.requests -
state.requests)}
}

assert InterleavingClaims {
        all r: Resource | sole InitialState.claims.r
}

assert DeadlocksAvoided {
         all state: SystemState | some state.allocations => noCycleCreated(state)
}

fun System() {
        two r:Resource | #(InitialState.claims.r) > 1
}

run System for 2 Process, 5 Resource, 10 SystemState
check InterleavingClaims for 2 Process, 6 Resource, 8 SystemState
check DeadlocksAvoided for 2 Process, 6 Resource, 8 SystemState
```

7.2 Appendix 2: Model 1

```
module DeadlockAvoidance

// A system process which uses resources to perform jobs.
sig Process {

}

// A resource which is used by processes to complete a job.
sig Resource {

}

// A snapshot of the state of resource allocations in a system
// at a particular time.
```

```
sig SystemState {
    processes: set Process,
    resources: set Resource,
    allocations: processes -> resources
}{
    processes = univ[Process]
    resources = univ[Resource]
    // Each resource can be allocated to at most one process
    all r: Resource | sole r[allocations]
}

// allocates a set of resources to a process
fun allocate(state, state': SystemState, process: Process,
                resource: set Resource){
    no (resource & process.(state.allocations))
    some resource
    all r:Resource | r in resource => r in
                    process.(state'.allocations)
}

fun test (){
    some SystemState.allocations
}
```

7.3 Appendix 3: Model 2

```
module DeadlockAvoidance

// A system process which uses resources to perform jobs.
sig Process {
    needs: set Resource
} {
    some needs
    Process = univ[Process]
}

// A resource which is used by processes to complete a job.
sig Resource {

}

// A snapshot of the state of resource allocations in a system
// at a particular time.
sig SystemState {
    waitingProcesses: set Process,
```

```
    allocations: Process -> Resource
}{
    // Each resource can be allocated to at most one process
    all r: Resource | sole r[allocations]
    all p: Process | some (p.needs - p.allocations)
                    <=> p in waitingProcesses
}

// Initial system state where all the processes are waiting
// and no allocations are made.
static disj sig InitialState extends SystemState {
} {
    waitingProcesses = univ[Process]
        no allocations
}

// allocates a set of resources to a process
fun allocate(state, state': SystemState){

    all wp: state.waitingProcesses |
            some (wp.needs - Process.(state.allocations))
            => one ((wp.(state'.allocations - state.allocations))
            & (wp.needs - Process.(state.allocations)))
    state.allocations & state'.allocations = state.allocations
}

fun test (){
    some SystemState.allocations
}
```

7.4 Appendix 4: Model 3

```
module DeadlockAvoidance

// A system process which uses resources to perform jobs.
sig Process {
    // Set of resources needed by this process to complete.
    needs: set Resource
} {
    // Each process should need atleast one resource to complete.
    some needs
}

// A resource which is used by processes to complete a job.
```

```
sig Resource {
}

// A set of facts constraining the Process and Resource definitions.
fact{
    // Each resource should be needed by aleast one process.
    all r: Resource | some Process$needs.r
}

// A snapshot of the state of resource allocations in a system at
// a particular time.
sig SystemState {
    // Processes which are not yet complete (waiting to be
    // allocated some needed resources).
    waitingProcesses: set Process,
    // A set of mappings from a process to a resource allocated to that process.
    allocations: Process -> Resource
}{
    // Each resource can be allocated to at most one process
    all r: Resource | sole r[allocations]
    // If a process has been allocated all its needed resources,
    // it should not be in 'waitingProcesses'
    // If a process has not been allocated all its needed resources,
    // it should be in 'waitingProcesses'
    all p: Process | some (p.needs - p.allocations) <=> p in waitingProcesses
}

// Initial system state where all the processes are waiting and no
// allocations are made.
static disj sig InitialState extends SystemState {
} {
    waitingProcesses = univ[Process]
        no allocations
}

fact {
    // All the processes, resources and systemStates should be
    // included in the model.
    Process = univ[Process]
    Resource = univ[Resource]
    SystemState = univ[SystemState]
}

// A function which allocates(if possible) a single resource to
// each waiting process.
fun allocate(state, state': SystemState){
```

```
    // state and state’ cannot be the same SystemState
    state != state’
    /* If a process needs a resource which has not been allocated (to any process),
       that resource
       should be allocated to that process in the next system state.
       At most one resource is allocated to a process by this function.
    */
    all wp: state.waitingProcesses | some (wp.needs - Process.(state.allocations))
            => one (wp.(state’.allocations - state.allocations)) &&
            (wp.(state’.allocations - state.allocations)) in
            (wp.needs - Process.(state.allocations))
    // All allocations from the present SystemState should be carried
    // over to the next SystemState.
    state.allocations & state’.allocations = state.allocations
}

assert NoDeadlock {
   all s, s’: SystemState |
   {
    allocate(s, s’)
   } => s.allocations
}

run allocate for 4 Process , 10 Resource ,4 SystemState
```

7.5 Appendix 5: Meeting Minutes

Date	Duration	Location	Attendees	Apologists	Agenda	Summary
14th May 2004	17:00-18:00	SE Labs	All	n/a	Choose a proposed topic	Deadlock Avoidance was chosen as the proposed topic
15th May 2004	13:00-15:00	SE Labs	All	n/a	Form the structure of initial model 1	A set of resources can be allocated to a process.
22nd May 2004	13:00-18:00	SE Labs	All	n/a	Refine model 1 to incorporate initial state.	Initial system state where all processes are waiting and no allocations are made.
28th May 2004	14:00-15-00	SE Labs	All	n/a	Redesign model to incorporate deadlock avoidance constraints. Read text.	A claim edge needed in implementing deadlock avoidance.
29th May 2004	10:00-20:00	SE Labs	All	n/a	Complete the model and make alloy code more readable.	*Added claim edge. *Added fun to determine if a cycle exists. *Added fun request. *Added fun allocate. *Refactored alloy code to make it more readable.
30th May 2004	10:00-23:59	SE Labs	All	n/a	Complete the model complete report in LaTeX.	* Added ordering of system states. * Split up functions into smaller functions.