



PathMATE Transformation Maps Mutex Controls

Version 0.2
June 7, 2006

PathMATE Technical Notes

Pathfinder Solutions LLC
33 Commercial Drive, Suite 2
Foxboro, MA 02035 USA
www.PathfinderMDA.com
888-662-7284

Table Of Contents

1. Introduction.....	1
2. Feature Description	1
Specifying Mutex Controls - IntertaskMutex.....	1
<i>AUTO</i>	1
<i>SINGLE</i>	1
<i>User-Specified</i>	1
Applying the Controls	1
<i>Domain IntertaskProtect</i>	1
<i>Domain Service IntertaskProtect</i>	1
Class IntertaskProtectClassExtent	2
<i>Class IntertaskProtectClassInstance</i>	2
<i>Association Control</i>	2
<i>State Pipelining</i>	4
ReentrantMutex	4
Acquiring and Releasing Controls	4
<i>Single Domain Controls - Domain Service</i>	4
<i>With Class and Association Controls</i>	5
Appendix A – Generated Mutex Code.....	6
Appendix B – Why Multiple Mutexes Must Secured in a Group instead of Inline	10
Deadly Embrace With Inline Mutexes	10
Action Called from Task A.....	10
Action Called from Task B.....	11
The Deadlock.....	11
No Deadlock With Grouped Mutexes	12

1. Introduction

This Technical Note describes intertask mutual exclusion ("mutex") control mechanisms for the protection of analyzed domain contents generated by PathMATE Transformation Maps.

2. Feature Description

Specifying Mutex Controls - IntertaskMutex

The domain IntertaskMutex property specifies what mutex control(s) may be applied as required by IntertaskProtect settings. All mutexes are generated, managed and used as singletons - one per process.

AUTO

The default value is AUTO, indicating no single domain mutex is to be used, and if any Classes or Associations specify mutex control, that a specifically generated control for that Class or Association will be used. If domain IntertaskProtect == ALL, then this marking's AUTO value is overridden and forced to SINGLE.

SINGLE

The value "SINGLE" specifies a single generated mutex control for the domain, used for all accesses to the domain.

User-Specified

The user can also provide the name of a mutex control coming from a realized source, which must resolve to a SW_ReentrantMutex_handle_t, available from the compiler global name scope. Any realized file include files must be added to sys_incl.h, and the control itself must be initialized and deconstructed manually in realized code.

Applying the Controls

Code to secure and release mutex controls protects structures that are marked with the set of IntertaskProtect properties.

Domain IntertaskProtect

By default, mutex controls are applied only as required by Domain Service, Class and Association marking. Specifying "ALL" applies a single domain mutex control entering and returning from all domain services, acting the same as if all the domain's services specified their IntertaskProtect = "ON".

Domain Service IntertaskProtect

By default, no mutex controls are applied around a domain service. Specifying "ON" applies a single domain mutex control entering and returning from this domain service.

Specifying "SINGLE" will apply a domain service specific mutex when entering and returning from this domain service.

Class IntertaskProtectClassExtent

Setting this to ALWAYS indicates a mutex control to protect the class extent structures of this class is always applied.

The applied mutex control regulates the following accesses:

- CREATE
- DELETE
- FIND CLASS
- FOREACH CLASS

If the domain IntertaskMutex marking is AUTO, then a class extent mutex control `<class name>_classMutexControl` is generated for each Class (one per class per process – singleton).

Class IntertaskProtectClassInstance

Setting this to ALWAYS indicates a mutex control to protect this class' instance structures is always applied.

The applied mutex control regulates accesses to class instance data members:

- Attribute read or write accesses
- FOREACH/WHERE or FIND/WHERE using the class's attributes
- DELETE of the class instance
- State updating via state machine event transitions (if a class is allocated to SYS_TASK_ANY).
- Association accesses from this participant structures.

If the domain IntertaskMutex marking is AUTO, then an instance mutex control `<class name>_instanceMutexControl` is generated for each Class (one per class per process – singleton).

Association Control

Setting this to ALWAYS indicates a mutex control to protect the association's structures is always applied.

The applied mutex control regulates the following accesses:

- LINK or UNLINK (either explicitly through LINK or UNLINK action language or implicitly upon deletion of one of the participating instances)
- FOREACH `<nav>` or FIND `<nav>`

If the domain IntertaskMutex marking is AUTO, then an association mutex control `A<association number>mutexControl` generated for each Association (one per association per process – singleton). This is the default level of control, and is also provided when class IntertaskProtect == "ASSOCIATION".

<i>model element</i>	<i>marking</i>	<i>description</i>
Domain	IntertaskMutex	Allows the user to specify what mutex control is to be used to protect this domain.

		No single domain mutex is to be used, and if any Classes or Associations specify mutex control, that a specifically generated control for that Class or Association will be used. If domain IntertaskProtect == ALL, or if any of the domain's services have IntertaskProtect == ON, then this marking's AUTO value is overridden and forced to SINGLE.
	AUTO (default)	
	SINGLE	A single mutex control is generated for the domain (per process), and whenever a mutex is needed (based on various IntertaskProtect markings) that this mutex is used.
	<user specified>	Similar to SINGLE, but this specifies the name of the mutex control to be used. It is assumed this is a SW_ReentrantMutex_handle_t, available from the compiler global name scope. Any realized file include files must be added to sys_incl.h, and the control itself must be initialized and deconstructed manually in realized code.
Domain	IntertaskProtect	Allows the user to manually specify the if domain is to be protected by a single mutex, applied in every domain service.
	AUTO (default)	Allows domain mutex control to be applied as specified by appropriate markings on domain operations, classes and associations.
	ALL	Requires the domain mutex to be secured at the entry of each domain service, and released on return. If IntertaskMutex == AUTO (or unset) this forces the generation of a single domain mutex control as if IntertaskMutex were SINGLE. This prevents the automatic generation of mutex controls at the class, class instance, association and link, and overrides any IntertaskProtect setting they may have.
Domain Service	IntertaskProtect	Allows the user to manually specify if this domain service is to be protected by domain mutex controls.
	OFF (default)	No special protection is applied entering or returning from this service.
	ON	Upon entering this domain operation, the domain mutex control is secured, and released upon return. If the domain's IntertaskMutex == AUTO, then it is overridden to SINGLE.
	SINGLE	Upon entering this domain operation, the domain service mutex control is secured, and released upon return. Setting SINGLE has no effect on the domain's IntertaskMutex setting.
Class	IntertaskProtectClassExtent	Specify the use of Class Extent mutex controls to protect the class extent structures of this class.
	AUTO (default)	Similar to ALWAYS but only applied in the case two a domain is deployed to two or more tasks, and contention for this class is detected. (Not currently supported, resulting in NEVER)
	ALWAYS	Always apply a mutex control to protect the class extent structures of this class. If the domain IntertaskMutex marking is AUTO, then generate a class extent mutex control for this class.
	NEVER	Do not apply a mutex control to protect the class extent structures of this class.
Class	IntertaskProtectInstance	Specify the use of Class Instance mutex controls to protect each instance's data.
	AUTO (default)	Similar to ALWAYS but only applied in the case two a domain is deployed to two or more tasks, and contention for this class' instances is detected. (Not currently supported; resulting in NEVER)
	ALWAYS	Always apply a mutex control to protect the instance data of this class. If the domain IntertaskMutex marking is AUTO, then generate an instance mutex control for this class.
	NEVER	Do not apply a mutex control to protect the class extent structures of this class.
Association	IntertaskProtect	Specify the use of mutex controls to protect the association's structures.
	AUTO (default)	Similar to ALWAYS but only applied in the case two a domain is deployed to two or more tasks, and contention for this association is detected. (Not currently supported, resulting in NEVER)

	ALWAYS	Always apply a mutex control to protect storage structures of this association. If the domain IntertaskMutex marking is AUTO, then generate an association mutex control.
	NEVER	Do not apply a mutex control to protect the storage structures of this association.
State	IntertaskProtect	Specify mutex controls on state entry.
	PIPELINED	Secure the state's mutex when transitioning into the state. Release the mutex after the entry actions are complete.

Table 1: Mutex Control Markings Table

Please note – with the Class properties specified above – the specification of IntertaskProtectClassExtent and IntertaskProtectInstance for a supertype class will override any mutex control settings for its subtypes.

State Pipelining

Setting the IntertaskProtect property of a state to "PIPELINED" allows only one instance of a class to be entering state at a time. A mutex is generated for each pipelined state. The state pipeline mutex is acquired before transitioning into the state and released after the transition to the state is complete.

ReentrantMutex

The ReentrantMutex, used in a manner similar to the CriticalSection mechanism, will be provided to allow for multiple calls to `enter()` from the same task all to return immediately as long as a different task does not have it. This is implemented in a manner that uses a simple mapping to an appropriate RTOS mechanism from each supported target platform for best efficiency.

Acquiring and Releasing Controls

Single Domain Controls - Domain Service

When a domain IntertaskMutex == SINGLE or <user specified>, a single domain control is used to control access. When applied to a domain service, the mutex is secured upon entry to the service, and released upon return.

When a single control is applied in the context of Class and Association accesses, these accesses are controlled by securing and releasing the control as close to the protected access as possible to reduce the potential duration of contention. This table outlines how each access is protected:

element	access	location
Class Extent	CREATE	Within accessor function
	DELETE	Within accessor function
	FIND CLASS	Within accessor function
	FOREACH CLASS	Within the invoking action
Class Instance	Attribute	Within the invoking action

	FOREACH/WHERE	Within the invoking action
	FIND/WHERE	Within accessor function
	DELETE	Within accessor function
	current state update	Within accessor function
	Association (from this participant)	Within accessor function
Association	LINK or UNLINK	Within accessor function
	FIND <nav>	Within accessor function
	FOREACH <nav>	Within the invoking action

Table 2: Single Control Locations***With Class and Association Controls***

In the case where class and association controls are generated, the proliferation of mutex control can dramatically increase the potential for a “deadly embrace” deadlock when one task (task A) secures some of the mutexes it needs, and finds that another task (task B) has other of the mutexes it needs, and blocks waiting for these to be released. If task B is itself waiting for some of the mutexes locked by task A, then this creates a “deadly embrace”. (For a more detailed illumination of this case see Appendix B – Why Multiple Mutexes Must .)

In order to avoid deadlocks, it is important that all the mutex controls needed for an action are secured before the action itself is started. The strategy applied in this feature implementation is to acquire before the start of the action all the mutex controls needed for an action and for all domain-local services called synchronously within the action. For example, if a domain service D:S() requires mutex control M1, and D:S() synchronously and locally calls the class service D.C:S() which requires mutex control M2, then both M1 and M2 are acquired at the beginning of D:S().

Even with this strategy it is still possible to create a multi-task topology with deadlocks, and the designer must take the utmost care to design avoid these. Some techniques to help avoid these are when a domain runs in multiple tasks and requires mutexes:

- Ensure all domain that it synchronously (locally) calls do not require mutexes
- Allocate server domains to their own tasks
- Review the domain-level scenarios with tasks in mind and understand which mutexes are acquired and released, and when.

Appendix A – Generated Mutex Code

This appendix highlights the code generated from manually specified mutex controls (per section 4.5). The test model BlockingTest has the following domains:

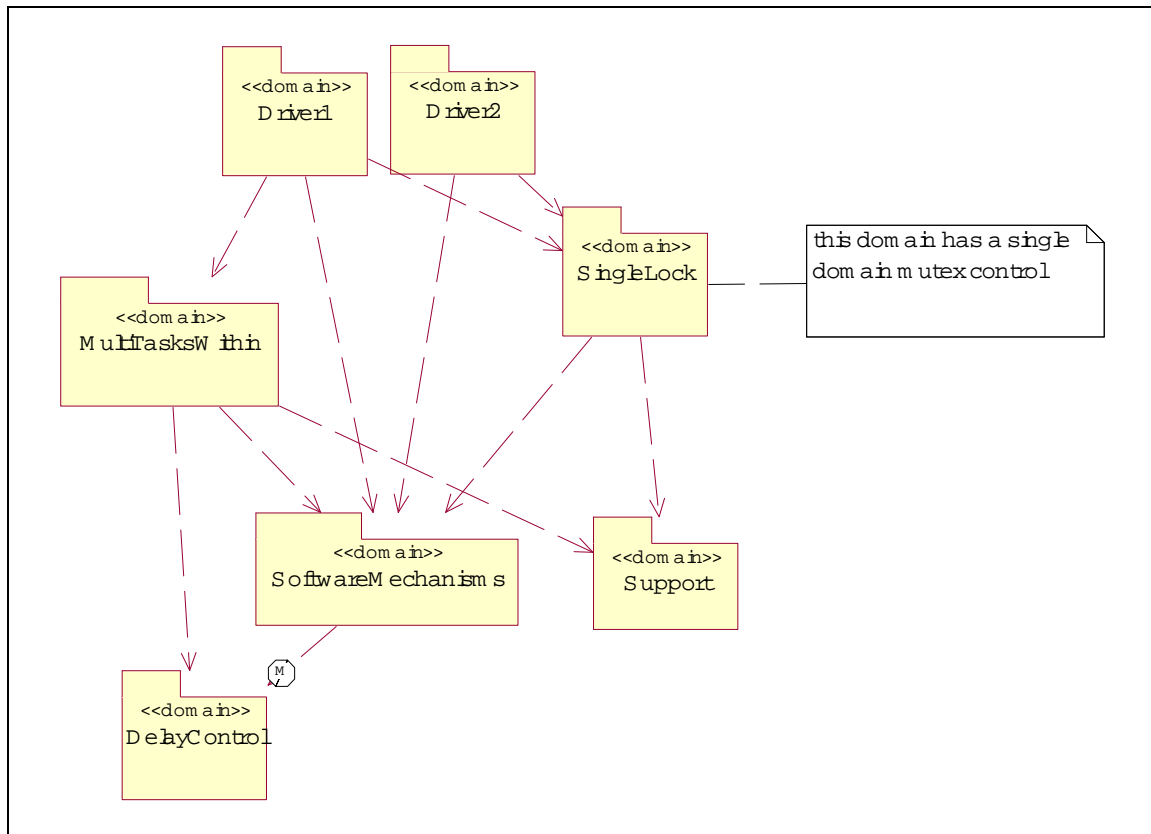


Figure 1 – BlockingTest Mutex Test System

The BlockingTest model has the following domain MultiTasksWithin:

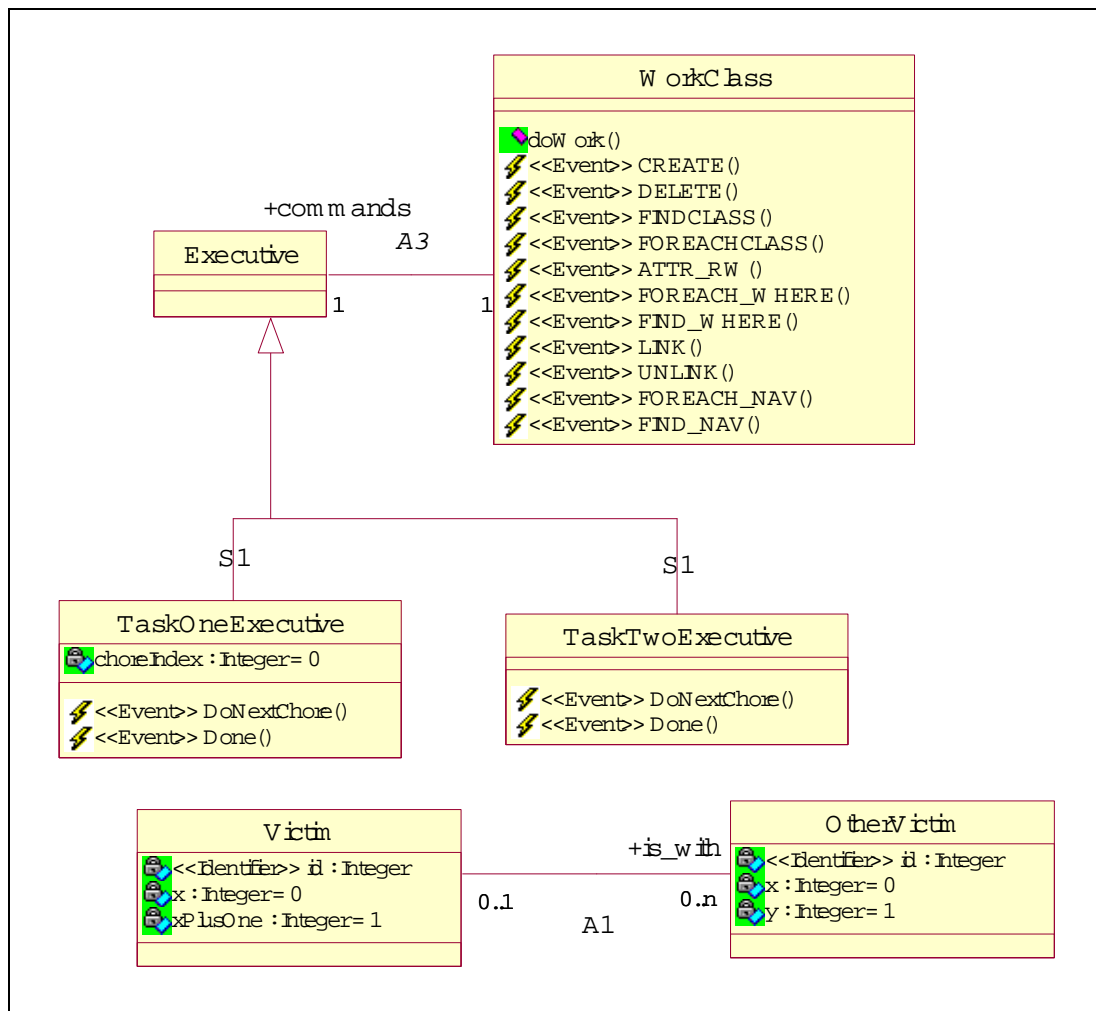


Figure 2 – BlockingTest.MultiTasksWithin(MT) Domain

To build contention, the **TaskOneExecutive** class is allocated to `SYS_TASK_ID_MAIN`, and the **TaskTwoExecutive** class is allocated to `SYS_TASK_ID_TASK2`, via `properties.txt`:

```

# The following markings are used to set up the Visual C++ project files
Domain,BlockingTest.*,SpotlightEnabled,F

Domain,BlockingTest.SW,RealizedPath,c:\pathmate\design\c\mechanisms
Domain,BlockingTest.SUPP,RealizedPath,realized
System,BlockingTest,Defines,PATH_NO_INPUT_DRIVER;PATH_DELAY_MUTEX_BLOCKING

# MULTITASK SETTINGS -----
Domain,BlockingTest.D1,TaskID,SYS_TASK_ID_MAIN
Domain,BlockingTest.D2,TaskID,SYS_TASK_ID_TASK2
Domain,BlockingTest.SL,TaskID,SYS_TASK_ANY
Domain,BlockingTest.SW,TaskID,SYS_TASK_ANY
Domain,BlockingTest.SUPP,TaskID,SYS_TASK_ANY
Domain,BlockingTest.DelayControl,TaskID,SYS_TASK_ANY
  
```

```

Domain,BlockingTest.MT,TaskID,SYS_TASK_ANY
Object,BlockingTest.MT.TaskOneExecutive,TaskID,SYS_TASK_ID_MAIN
Object,BlockingTest.MT.TaskTwoExecutive,TaskID,SYS_TASK_ID_TASK2

# MUTEX SETTINGS -----
Domain,BlockingTest.SL,IntertaskProtect,ALL
Object,BlockingTest.MT.Victim,IntertaskProtectClassExtent,ALWAYS
Object,BlockingTest.MT.Victim,IntertaskProtectInstance,ALWAYS
Object,BlockingTest.MT.OtherVictim,IntertaskProtectClassExtent,ALWAYS
Object,BlockingTest.MT.OtherVictim,IntertaskProtectInstance,ALWAYS
BinaryRel,BlockingTest.MT.A1,IntertaskProtect,ALWAYS

# MULTIPROCESS SETTINGS -----
#System,BlockingTest,GeneratedPath,..\\gc
#Domain,BlockingTest.SUPP,RealizedPath,..\\realized_c

```

Table 3 – BlockingTest Properties

As outlined in section 0 Acquiring and Releasing Controls above, the mutex controls for each action are acquired at the start of each action, so when an RTOS task prepares to enter an action, it secures all needed mutex locks at once, and will not be deadlocked by securing a partial set and then waiting on the remainder. Showing class-extent and instance locking, the entry action for MT.WorkClass.Attr_ReadWriting is:

```

String full_msg = msg;

DelayControl:ResetDelay();

FOREACH oov = CLASS OtherVictim
{
    // First check to see if the attr values are valid
    IF (oov.x != oov.y-1)
    {
        SUPP:WriteString("ERROR - invalid OtherVictim attribute values.");
    }
    oov.x = oov.x + 113;
    oov.y = oov.x + 1;
}
full_msg = full_msg + " ATTR_RW";

MT:CheckDelay(before_time, full_msg, expect_delay);
DelayControl:ClearDelay();

```

Table 4 - MT.WorkClass.Attr_ReadWriting Entry Action PAL

Just before the generated body of this action, the class extent and instance mutex controls for the OtherVictim class are secured by:

```

SW_ReentrantMutex_enter(&MT_OtherVictim_classMutexControl);
SW_ReentrantMutex_enter(&MT_OtherVictim_instanceMutexControl);

```

Table 5 – Generated Fragment: Class Extent and Instance Mutexes

They are released at the end of this action.

If an action has a number of mutexes accessed at various points throughout the action, they are all secured at the beginning of the action to avoid deadlock. The entry action for MT.WorkClass.Linking

```

Ref<Victim> v;
Ref<OtherVictim> ov;
String full_msg = msg;

DelayControl:ResetDelay();

v = FIND CLASS Victim;
ov = CREATE OtherVictim(id = 9999);
LINK v A1 ov;
full_msg = full_msg + " LINK";

MT:CheckDelay(before_time, full_msg, expect_delay);
DelayControl:ClearDelay();

```

Table 6 - MT.WorkClass.Linking Entry Action PAL

Just before the generated body of this action, the class extent and association mutex controls for Victim, OtherVictim and A1 are secured by:

```

SW_ReentrantMutex_enter(&MT_Victim_classMutexControl);
SW_ReentrantMutex_enter(&MT_OtherVictim_classMutexControl);
SW_ReentrantMutex_enter(&MT_A1_mutexControl);

```

Table 7 – Generated Fragment: Class Extent and Instance Mutexes

They are released at the end of this action.

Appendix B – Why Multiple Mutexes Must Secured in a Group instead of Inline

Section 0 Acquiring and Releasing Controls above introduces that a proliferation of mutex controls can result in the case where two tasks are blocking on each other to release mutexes they each need. Unless mutexes are secured carefully, this is a startlingly easy situation to encounter.

Deadly Embrace With Inline Mutexes

The following example presumes mutexes are secured when they are needed, *inline* with the PAL statements or statement blocks they are associated with.

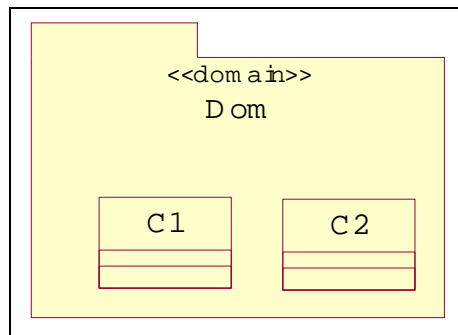


Figure 3 – Domain Dom with Classes C1 and C2

A domain Dom has classes C1 and C2 that are both accessed by tasks A and B, so the mutexes Dom_C1_classMutexControl and Dom_C2_classMutexControl are generated based on the properties:

Object, Sys.Dom.C1, IntertaskProtectInstance, ALWAYS
Object, Sys.Dom.C2, IntertaskProtectInstance, ALWAYS

Table 8 – Properties for Classes C1 and C2

Action Called from Task A

The action ActA which is called from task A does:

<pre>FOREACH this_one = CLASS C1; { Ref<C2> other = FIND FIRST C2; // continue on and do other things }</pre>

Table 9 – Action ActA

If the mutexes were generated inline with the statements that needed them (and not before), they would appear in this relative position to the PAL:

```

SW_ReentrantMutex_enter(&Dom_C1_classMutexControl);
FOREACH this_one = CLASS C1;
{
    SW_ReentrantMutex_enter(&Dom_C2_classMutexControl);
    Ref<C2> other = FIND FIRST C2;
    SW_ReentrantMutex_leave(&Dom_C2_classMutexControl);
    // continue on and do other things
}
SW_ReentrantMutex_leave(&Dom_C1_classMutexControl);

```

Table 10 – Action ActA with generated Inline Mutexes Highlighted

Action Called from Task B

The action ActB which is called from task B does:

```

FOREACH mine = CLASS C2;
{
    Ref<C1> yours = FIND FIRST C1;
    // continue on and do other things
}

```

Table 11 – Action ActB

Mutexes generated inline are:

```

SW_ReentrantMutex_enter(&Dom_C2_classMutexControl);
FOREACH mine = CLASS C2;
{
    SW_ReentrantMutex_enter(&Dom_C1_classMutexControl);
    Ref<C1> yours = FIND FIRST C1;
    SW_ReentrantMutex_leave(&Dom_C1_classMutexControl);
    // continue on and do other things
}
SW_ReentrantMutex_leave(&Dom_C2_classMutexControl);

```

Table 12 – Action ActB with generated Inline Mutexes Highlighted

The Deadlock

In the implementation-level sequence chart below, the timing shows that both tasks A and B have secured their first mutex control before they get to the second – which is held by the other task, causing a deadly embrace:

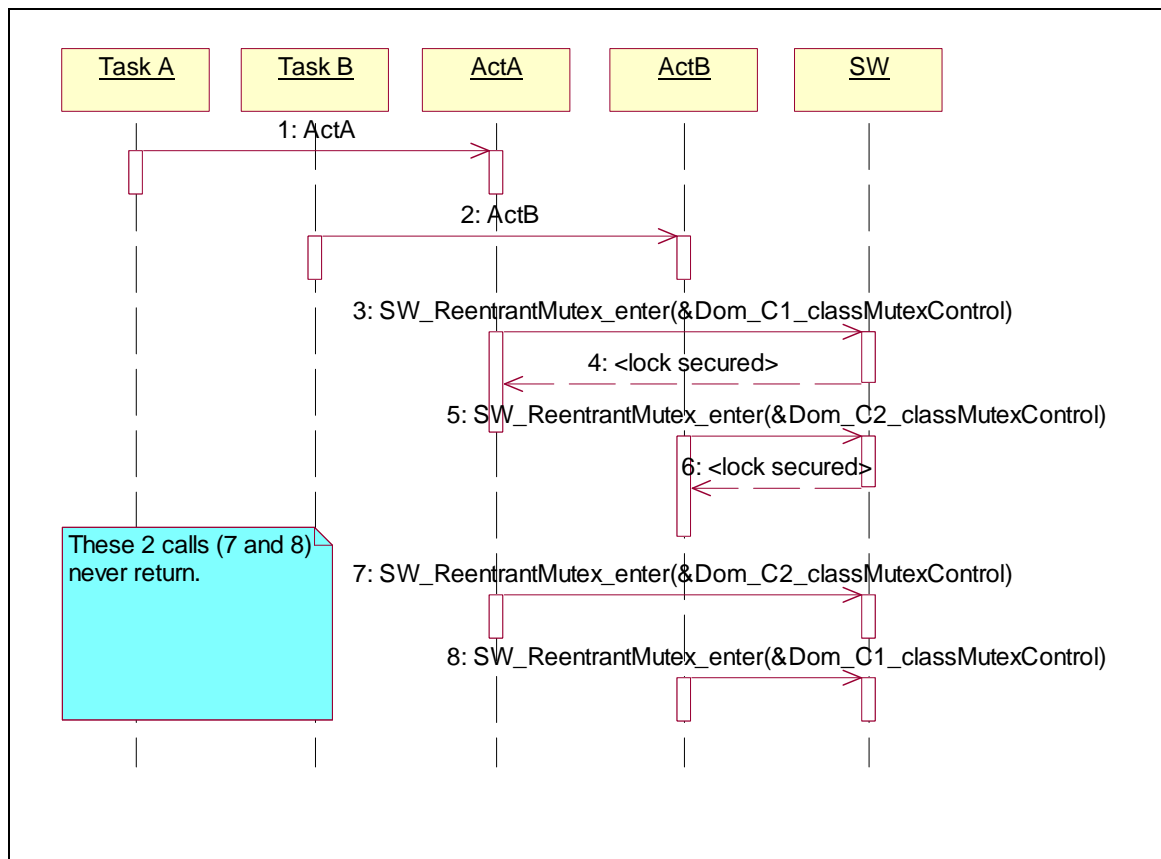


Figure 4 – Sequence of Inline Locking Making a Deadly Embrace

No Deadlock With Grouped Mutexes

The alternative to generating code for entering and leaving mutexes inline is to secure all mutexes at the beginning of an action, and generate all mutex exits at the end of an action. Another critical element to this strategy is to always secure mutexes in the same order. So actions ActA and ActB above would instead have mutexes accessors generated in the following locations:

```

SW_ReentrantMutex_enter(&Dom_C1_classMutexControl);
SW_ReentrantMutex_enter(&Dom_C2_classMutexControl);
FOREACH this_one = CLASS C1;
{
    Ref<C2> other = FIND FIRST C2;
    // continue on and do other things
}
SW_ReentrantMutex_leave(&Dom_C1_classMutexControl);
SW_ReentrantMutex_leave(&Dom_C2_classMutexControl);
  
```

Table 13 – Action ActA with generated Grouped Mutexes Highlighted

```

SW_ReentrantMutex_enter(&Dom_C1_classMutexControl);
SW_ReentrantMutex_enter(&Dom_C2_classMutexControl);
FOREACH mine = CLASS C2;
  
```

```
{  
    Ref<C1> yours = FIND FIRST C1;  
    // continue on and do other things  
}  
SW_ReentrantMutex_leave(&Dom_C1_classMutexControl);  
SW_ReentrantMutex_leave(&Dom_C2_classMutexControl);
```

Table 14 – Action ActB with generated Grouped Mutexes Highlighted

Now the same initial timing as shown above has a much different result:

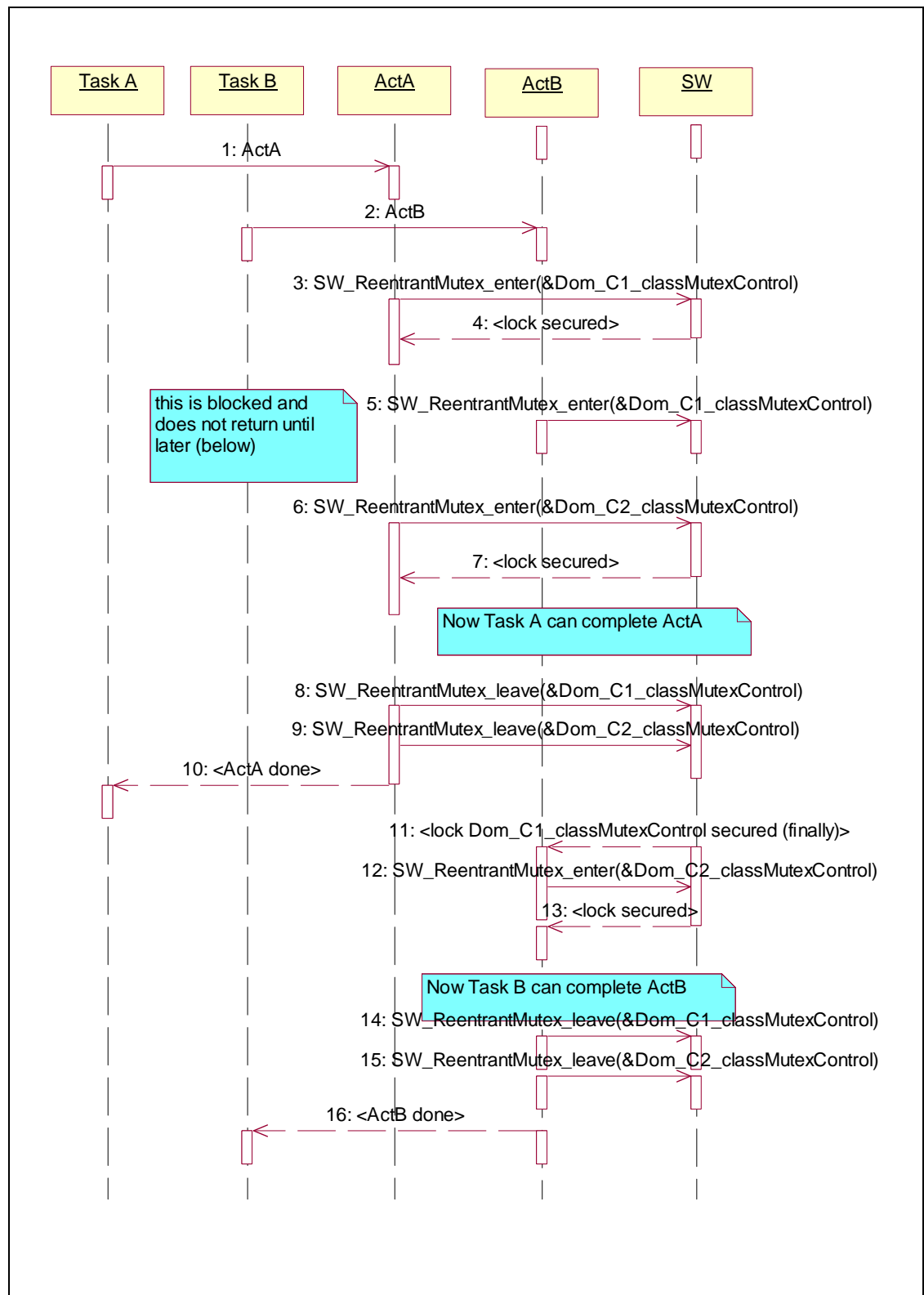


Figure 5 – Sequence of Grouped Locking Avoiding a Deadly Embrace