Rhapsody Advanced Training

Frameworks

WILLERT.

Index

IBM® Rational® Rhapsody®

IBM Rational Rhapsody is a UML Tool. It is a UML Editor and a UML Code generator. Code can be generated in several languages, C, C++, Java and ADA.

For C and C++ it is not possible to generate code from all UML constructs. For instance in UML it is possible to set a Class to Active. This means that the objects of this class will run in their own thread. Since neither C nor C++ has built-in statements for starting a thread (or sending messages or setting timers) it is not possible to directly generate code for that.

The solution is to implement the required functionality in a library, most of the functionality is RTOS related so the Framework could include a COTS RTOS

Why a Framework?

Well first of all: You don't need a framework if you don't want one!
You can generate code from UML diagrams and then fill the functions with your own code, and call state-charts and or activity diagram code from your own routines. But it is far from convenient to do so. Rhapsody will do a lot of work for you in the framework. Also you can be sure that you use the UML in a correct way if you generate "production code".

The framework will implement the functionality that the language of the generated code does not offer directly. The 'C' language for instance does not have statements for starting a timer, sending and receiving messages and starting a thread.

An RTOS does have that functionality, a Rhapsody framework will use that to implement this functionality.

What types of Framework are there?

One of the parameters that determines what framework you must/can use is the used language. You can use 'C' or 'C++' which both will use different frameworks. Java and C# use almost no framework, they use the Java VM or .net to implement the functionality that the UML needs.

Another difference is the size/speed of the framework. You can imagine that generating code on a multi gigabyte RAM Core-i7 is something different then generating on a 32k ROM 8-bit embedded CPU.

There are many Frameworks available with different features.

RXF, Realtime eXecution Framework

This is the Willert Software Tools Framework, especially created for realtime embedded targets. Has a small footprint and uses very little RAM. It is very configurable so that it can be precisely tailored for the smallest of targets. It is highly integrated with the target environment so that adapting to a new environment can take longer. Is available in C and C++. This framework uses a normal RTOS to implement most functions. For very limited target environments there is a built-in single-threaded RTOS (OO-RTX) that decreases the size and speed dramatically. Supports Model level debugging with the UML target debugger.

RXF-Cert

The certifiable version of the RXF. All needed documentation for a certification against IEC61508 SIL 3 are delivered. Some functionality is removed for safety purposes.

OXF, Object eXecution Framework

This is the IBM standard. Is not very large although larger then the RXF. is very easily adaptable to a new environment due to the OSAL that includes all functions that require adaptation. Is available for all Rhapsody languages C, C++, C#, Java, ADA. Many functions that could be implemented by an RTOS are implemented in the framework itself to save adaptation. This makes the OXF larger than e.g. the RXF. The OXF is not very deterministic since interrupts are disabled quite often and for a non-deterministic duration. Supports Animation based on Code instrumentation and TCP/IP for communication.

MXF, Automotive Framework

Also delivered from IBM, connects to an AUTOSAR, RTE (OSEK) also called MicroC Framework. Must be adapted to the users environment. Available in C.

The Extended C Framework is currently available only when the Automotive C profile is selected during installation. This framework (which will eventually replace the IDF and SF) is designed primarily for automotive applications where the following is required:

No dynamic memory allocation

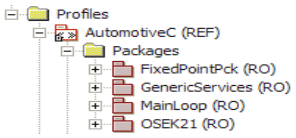
Compile-time initialization

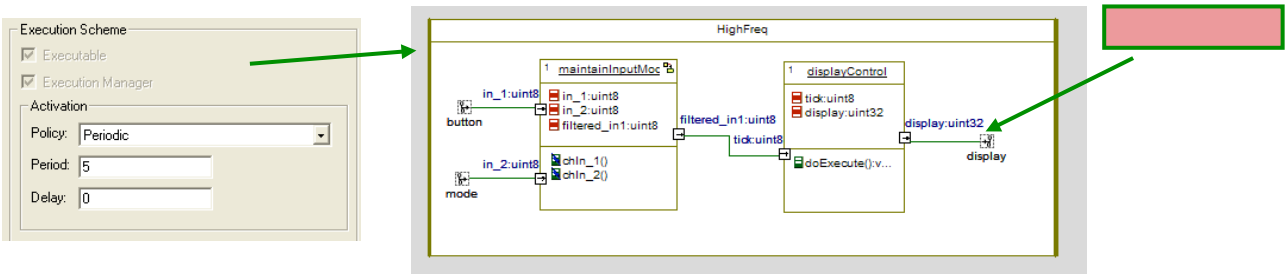
Compact and Configurable

MISRA-98 Compliance

OSEK integration or No-OS with a simple scheduler

Time Triggered Systems





SMXF, Safety Framework

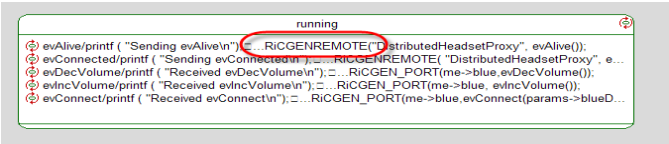
Also from IBM, includes Requirements and tests that are required for a certification. This is actually a Rhapsody model, the framework code is also generated from Rhapsody. Available in C and C++

DOX, Distributed Framework

An IBM Framework dedicated for RTOSes with protected memory. Supports functionality that can use shared memory to send events between objects in different memory spaces.

The framework allows events to be sent across address spaces.

To do so, the following properties will need setting:



C_CG::Configuration::MultipleAddressSpaces

C_CG::Class::PublishedName

C_CG::Class::PublishInstance

For each event, it will be necessary to provide a serialize and unserialize function:

C_CG::Event::SerializationFunction

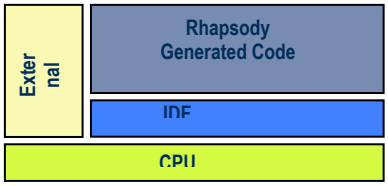
C_CG::Event::UnserializationFunction

A macro RiCGENREMOTE allows events to be sent to a remote address space ex:

IDF, Interrupt Drive Framework

The IDF was developed by an I-Logix Application Engineer, Marc Richardson. It lacks the use of active classes, therefor it is very small. it was (before Rhapsody 7.6) delivered as a Rhapsody model and the code was generated from that. It is a very good Framework for understanding how framework actually work.

By using the Interrupt Driven Framework (IDF), it is possible to run Rhapsody in C without the need for an Operating System



The IDF can replace the OXF and RTOS

It can be used without malloc \ free

It generates smaller code. For example a simple stopwatch example takes 20K on an ARM7 (framework + libraries + generated code)

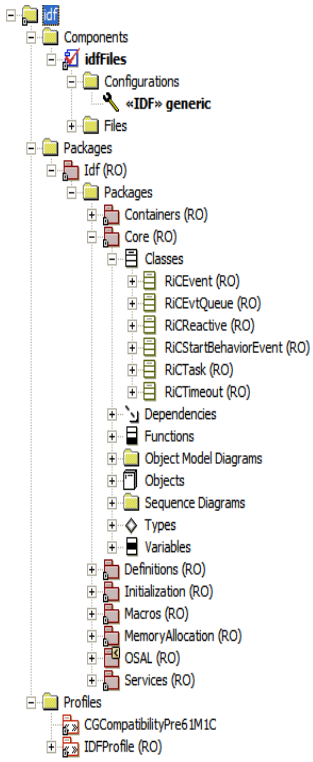
It requires just a periodic interrupt to be setup (so that timeouts can be used in State-charts)

Out of the box, there are two IDF models. Idf.rpy is the generic framework and then there will be specific models for each "Adapter" targeting a specific particular compiler and cpu

The model idf.rpy is the "generic" framework that contains the principal framework classes such as RiCReactive

Everything in this model is independent of the actual compiler and cpu used.

Everything that depends on the cpu and compiler is included in a separate "adapter" model



SF, Synchronous Framework

A very tiny Framework that can only use triggered operations, no time-outs and events.

The standard Rhapsody frameworks (OXF and IDF) both allow asynchronous and synchronous communication.

If only synchronous communication is desired so that the behavior is deterministic, then triggered operations can be used instead of events.

In this case, the majority of the framework is redundant.

The Synchronous Framework is just the bare minimum framework necessary to allow the use of triggered operations.

The Synchronous Framework does not require an OS and as such, active classes are not supported.

Timeouts and events are not supported.

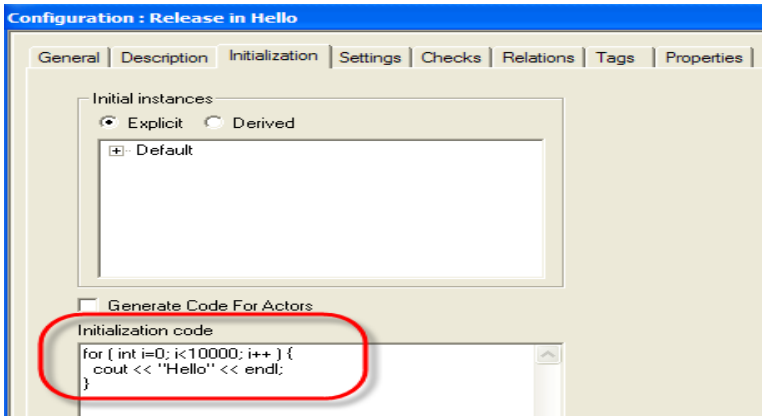
Has a smaller footprint than the IDF

NF, No Framework

Not a real framework, of course but a possibility.

If the framework is not required, then you can simply set the stereotype «NoFramework» to the configuration.

Of course without the framework, there will be no code generation for active classes, ports, state charts, relations with unbounded multiplicity, …



What must be adapted?

First the framework itself. Depending on the type of framework there are functions that have dependencies with the hardware and the development environment. They must be adapted.

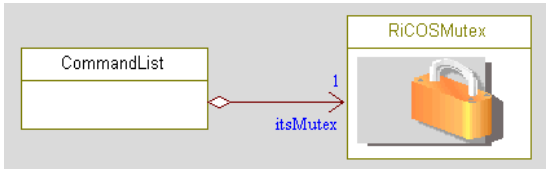
Then there is the build environment. There are frameworks that come as a Rhapsody model, the must be adapted in Rhapsody. Most frameworks use the "build in Rhapsody" way to build applications. Only RXF and RXF-Cert use a deployer to integrate the generated sources in an IDE project and build from there.

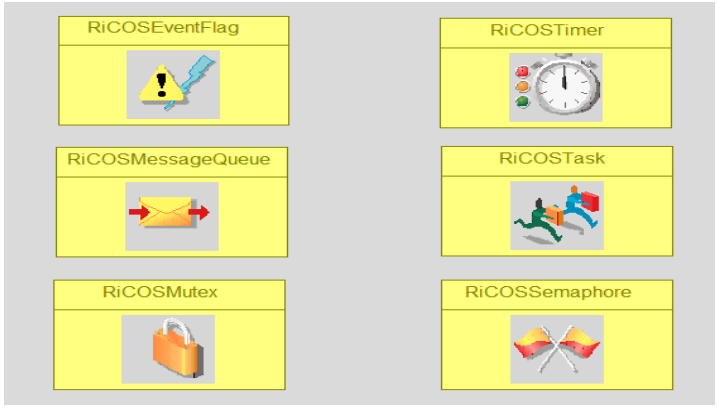
```
/*********************************/
/**      Mutex interface          **/
/*********************************/
/* init and cleanup */
RiCBoolean RiCOSMutex_init(RiCOSMutex * const me);
void RiCOSMutex_cleanup(RiCOSMutex * const me);

/* create and destroy */
RiCOSMutex * RiCOSMutex_create();
void RiCOSMutex_destroy(RiCOSMutex * const me);


/* lock the mutex */
/* in environments other than psos this is a macro, which implements the same interface */
#ifdef pSOS
RiCOSResult RiCOSMutex_lock(RiCOSMutex * const me);
#endif

/* free the mutex */
/* in environments other than psos this is a macro, which implements the same interface */
#ifdef pSOS
RiCOSResult RiCOSMutex_free(RiCOSMutex * const me);
#endif
```


```
RiCOSResult RiCOSMutex_lock(RiCOSMutex * const me) {
if (me == NULL) { return WAIT_FAILED; }

  return WaitForSingleObject(me->hMutex, INFINITE);
}


RiCOSResult RiCOSMutex_free(RiCOSMutex * const me) {
if (me == NULL) { return 0; }

  return (RiCOSResult) ReleaseMutex(me->hMutex);
  }

RiCBoolean RiCOSMutex_init(RiCOSMutex * const me) {
    if (me == NULL) return 0;

    me->hMutex = CreateMutex( NULL, FALSE, NULL );
    return (me->hMutex != NULL);
}

void RiCOSMutex_cleanup(RiCOSMutex * const me) {
    if (me != NULL) (void) CloseHandle(me->hMutex);
}

RiCOSMutex * RiCOSMutex_create() {
    RiCOSMutex * me = malloc(sizeof(RiCOSMutex));
    (void) RiCOSMutex_init(me);
    return me;
}

void RiCOSMutex_destroy(RiCOSMutex * const me) {
    if (me != NULL) {
        RiCOSMutex_cleanup(me);
        free(me);
    }
}
```

```
RiCOSResult RiCOSMutex_lock(RiCOSMutex * const me) {
  if (me == NULL) { return 0; }
  if (semTake(me->hMutex, WAIT_FOREVER)==OK)
    return 1;
  else
    return 0;
}

RiCOSResult RiCOSMutex_free(RiCOSMutex * const me) {
  if (me == NULL) { return 0; }
  if (semGive(me->hMutex)==OK)
    return 1;
  else
    return 0;
}

RiCBoolean RiCOSMutex_init(RiCOSMutex * const me) {
    if (me == NULL) return 0;

    me->hMutex = semMCreate(SEM_Q_FIFO);
    return (me->hMutex != NULL);
}

void RiCOSMutex_cleanup(RiCOSMutex * const me) {
    if (me != NULL && me->hMutex !=NULL) {
        semDelete(me->hMutex);
        me->hMutex = NULL;
    }
}

RiCOSMutex * RiCOSMutex_create() {
    RiCOSMutex * me = malloc(sizeof(RiCOSMutex));
    RiCOSMutex_init(me);
    return me;
}

void RiCOSMutex_destroy(RiCOSMutex * const me) {
    if (me != NULL) {
        RiCOSMutex_cleanup(me);
        free(me);
    }
}
```

These are the principle classes that need adaptation.





The interface stays the same, the implementation is different. Creating an adapter for a new environment is not that difficult although the documentation is not very thorough. Looking at the delivered example adapters will help you a lot. The advantage of using the OSAL is that you can use RTOS functions in your code without caring about the underlying RTOS.

Create using

```
me->itsMutex = RiCOSMutex_create();
```

Locking:

```
RiCOSMutex_lock(me->itsMutex);
```

Freeing:

```
RiCOSMutex_free(me->itsMutex);
```

How does it work

These are the principal classes in the Event Driven Framework:

RiCTask	WST_TSK (Task)
RiCReactive	WST_FSM (Finite State Machine)
RiCEvent	WST_EVT (Event)
	WST_MSQ (Message Queue)
RiCMonitor	WST_MTX (Mutex)
	WST_TMR (Timer)
	WST_TMM (Timer Manager)
RiCMemory	WST_MEM (Memory Manager)

RiCTask (WST_TSK)

This class contains the base function for the framework. Every active object has an instance of this function (hence the RiCTask * me argument)

It waits for an event and process the event by calling the generated function from the receiving object (rootstate_dispatchevent())

```
RiCReactive* RiCTask_execute(RiCTask * const me) {

    RiCTakeEventStatus result;
    RiCEvent * ev = NULL;
    RiCReactive * dest = NULL;

    while (true) {
        /* Wait for an event */
        RiCOSMessageQueue_pend(&me->eventQueue);

        /* Get the next event */
        ev = RiCOSMessageQueue_get(&me->eventQueue);

        /* Get reactive object waiting for this event */
        dest = RiCEvent_getDestination(ev);

        /* Dispatch the event */
        if (RiCEvent_getId(ev) != RiCCancelledEvent_id) {
            result = RiCReactive_takeEvent(dest, ev);
        }

        /* If necessary, delete the event */
        if (RiCEvent_isDeleteAfterConsume(ev)) {
            if (ev->eventDestroyOp != NULL) {
                ev->eventDestroyOp((void*)ev);
            } else {
                RiCEvent_destroy(ev);
            }
        }
    }
}
```

RiCReactive (WST_FSM)

A reactive class is one that reacts to events / timeouts.

Each class that has a State-chart, contains an instance of RiCReactive. This is a class that knows how to handle timeouts RiCTimeout and events RiCEvent.

RiCReactive has two main operations

The operation gen() allows events to be sent to the object, it calls an operation on its thread to queue the event onto the thread's event queue.

When the event / timeout is taken off the event queue, the takeEvent() operation is called to handle the event / timeout.

RiCEvent (WST_EVT)

```
typedef struct RiCEvent {
    struct RiCReactive * destination;
    RiCEventDestroyOp eventDestroyOp;
    RiCBoolean deleteAfterConsume;
    RiCBoolean frameworkEvent;
    short lId;
} RiCEvent ;
```

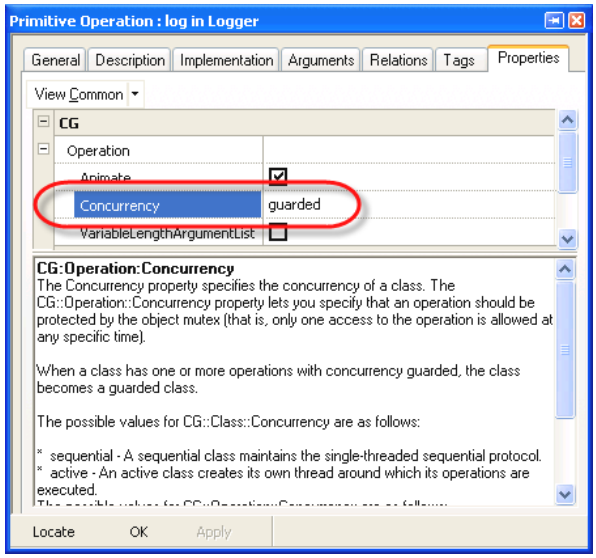
```
RiCBoolean RiCEvent_init(RiCEvent * const me, short lId, const struct RiCReactive * dest);

void RiCEvent_cleanup(RiCEvent * const me);

RiCEvent * RiCEvent_create( short lId, const struct RiCReactive * const dest);

void RiCEvent_destroy(RiCEvent * const me);
```

RiCMonitor (WST_MTX)

Rhapsody provides a class called RiCMonitor

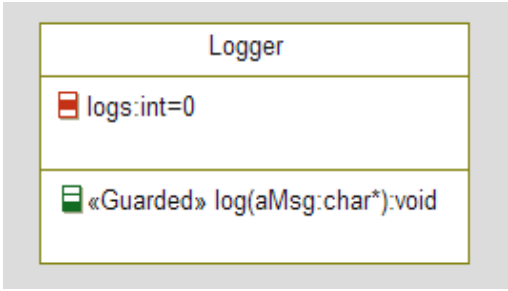
This can be used to automatically protect operations which need guarding with a mutex.

This is done by simply setting the property concurrency to guarded for the operation




```
void Logger_Init(Logger* const me) {
    RiCMonitor_init(&(me->ric_monitor));
        me->logs = 0;
}

void Logger_log(Logger* const me, char* aMsg) {
        RIC_OPERATION_LOCK(&(me->ric_monitor));
        Logger_log_guarded(me, aMsg);
        RIC_OPERATION_FREE(&(me->ric_monitor));
}

/*## operation log(char*) */
static void Logger_log_guarded(Logger* const me, char* aMsg) {
        /*#[ operation log(char*) */
        save(aMsg);
        me->logs++;
        /*#]*/
}
```

```
struct Logger {
    int logs;       /*## attribute logs */
    RiCMonitor ric_monitor;
};
```



These are the basic Framework Functions.

In other chapters we will explain the other functions like Container Classes.