DRAMA2 - DRAMA for the modern era.

Tony Farrell and Keith Shortridge

Australian Astronomical Observatory, P.O. Box 915 North Ryde, NSW 1670, Australia

Abstract. The DRAMA Environment provides an API for distributed instrument software development. It originated at the Anglo-Australian Observatory (now Australian Astronomical Observatory) in the early 1990s, in response to the need for a software environment for large distributed and heterogeneous systems, with some components requiring real-time performance. It was first used for the AAOs 2dF fibre positioner project(Lewis et al. 2002) for the Anglo-Australian Telescope. DRAMA is used for most AAO systems and is or has been used at various other observatories looking for a similar solution. Whilst DRAMA has evolved and many features were added, the overall design has not changed. It was still a largely C language based system, with some C++ wrappers. It did not provide good support for threading or exceptions. Ideas for proper thread support within DRAMA have been in development for some years, but C++11 has provided many features that allow a high quality implementation. We have taken the opportunity provided by C++11 to make significant changes to the DRAMA API, producing a modern and more reliable interface to DRAMA, known as DRAMA2.

1. Introduction

The DRAMA API (Farrell et al. 1993) remains the AAO's primary tool for constructing complex instrumentation systems and has been/is being used by various other observatories. It implements a tasking model usingh an approach based on the older Starlink ADAM Environment(Allan 1992). Each named task responds to named messages of a number of different types. In a DRAMA "System", tasks can run across different hosts in a heterogeneous environment. DRAMA was implemented from about 1992 and was designed to be highly portable at a time before ANSI C was available on all machines of interest.

Most work is a DRAMA task is done in response to "Obey" messages implementing "Actions" (commands). Co-operative multi-tasking allows multiple actions to be running at the same time but must deliberately return control to the DRAMA message reading loop between events to allow other actions to run and for the action itself to be "Kicked" - sent a message to change its behaviour in some way (typically, but not always, to cancel the action cleanly).

Due to portability issues and problems determing the best approach, early C++ interfaces to DRAMA were of poor quality.

Whilst DRAMA tasks using threads of various types have been implemented over the years, DRAMA itself has not supported using threads, with its own co-operative multi-tasking technique sufficient in most cases and more portable then threads were.

C++11(ISO 2011) was a major revamp to the C++ language: Threads are now supported using a well thought out approach, by the compilers and standard libraries; Many new features are provided by C++11 that assist library implementers to construct quality interfaces. We have taken advantage of the upgrade of C++ to implement DRAMA2, which will simplify writing and maintaining complex DRAMA tasks.

2. Basic DRAMA

A quick introduction to DRAMA is needed. A DRAMA task executes a message receive loop that dispatches control to event/message handlers when messages arrive. A "Path" is a connection to another task, which is opened as required by application specific code and then used to send messages. There are various types of messages sent between tasks, which may be running across various machines on the network.

Application specific code is provided to handle the "Obey" and "Kick" message types. The "Obey" message type causes application specific code to be run. "Kick" message communicate with a running Action of the "Name" in the message. The implementation of an action of a given "Name" is provided by application specific C language routines,invoked in response to messages by the DRAMA event loop.

Most messages can have an "Argument" attached, which provides a way of moving data across machines. These arguments are implemented using Self Defining Structures (SDS). These can be of any size and are designed to allow large amounts of data to be sent efficiently between tasks.

3. The Approach to Implementing DRAMA2

Implementation as wrapper around the C API. An older DRAMA JAVA interface proved that dramatic changes to the C level interfaces to DRAMA are not required for thread and exception support. By implementing DRAMA2 as a set of wrappers around DRAMA C APIs, compatibility with the large set of existing tasks can be easily maintained and DRAMA2 could be implemented quickly.

Only one Thread reading DRAMA messages. There is one and only one thread that actually blocks for and reads DRAMA messages from the underlying message queue. Other threads can send DRAMA messages (and make other DRAMA API calls) but cannot actually read the messages directly. No changes are required to the DRAMA C language internals when using this approach. If another thread needs to wait for a DRAMA message to occur, it must wait on a C++ condition, which is notified by the DRAMA thread when the message arrives.

Locking access to DRAMA structures. Only one lock is used and it must be taken by most methods that invoke the DRAMA C API. Use of the lock is normally internal to the DRAMA2 methods, but it can be used by application specific code to access any DRAMA C API not yet available or for application specific locking. Use of the DRAMA2 lock as the only lock in the application would avoid deadlock. The DRAMA2 lock is safe for recursive use.

Status and error reports vs. C++ Exceptions. The DRAMA C API uses an inherited status convention. Most functions have a "status" argument, which is a pointer to an integral type. Functions are expected to check the value pointed to is zero on entry. If it is not, they return immediately. If an error occurs, status is set to a non-zero value. The integer status value is passed as the result of DRAMA messages, allowing other tasks to determine if an Action has failed and to interpret the status value. An Error Reporting System (ERS) enables extra contextual information to be added when errors occur.

In DRAMA2, an exception class is provided which is a sub-class of std::exception. Any DRAMA2 method invoking a DRAMA C API must check the status returned and, if bad raise an exception.

At any point where the DRAMA C API must invoke a DRAMA2 method, there must be an interface function that has an inherited status argument. This function must catch any exception thrown by DRAMA2 and convert it to a DRAMA status. Any extra context available in the exception will be reported using ERS.

4. Task Structure

A Simple Task. The example below shows "Hello World" in DRAMA2. This program implements a task named "TASK1", which has just one Action - named "HELLO". Sending an Obey message with the name "HELLO" will result in the message "Hello World" being output and the task then exiting. The action is implemented by sub-classing the abstract class "MessageHandler" providing an implementation of "MessageReceived()". Any number of actions can be added in a similar way and they don't normally cause the task to exit, and may be invoked multiple times in sequence.

```
#include "drama.hh"
using namespace drama;
class Action1 : public MessageHandler{       // Action Definition
  Request MessageReceived() override {
    MessageUser("Hello World");
    return  RequestCode::Exit;   }
};
class ExTask : public Task {                // Task Definition
  Action1 Action1Obj;                       // Action object
  ExTask(const std::string &taskName) :
    Task(taskName) {
    Add("HELLO",
        MessageHandlerPtr(&Action1Obj,
                          nodel()));   }
};
// Main program.
int main() {                                     // Main program
  CreateRunDramaTask<ExTask>("TASK1");
  return 0;  }
```

Threaded Actions.. In the above example, the "HELLO" action is running in the main DRAMA2 thread. Whilst it can "Reschedule", in the traditional DRAMA way to return control to the message thread, the intent of DRAMA2 is to support running actions in threads. For a thread, the user must declare a sub-class of the "TAction" class and provide the method "ActionThread", which is invoked within a thread when an Obey message received. When the thread completes, DRAMA2 is informed and the action is marked as completed. All details of thread creation; joining the thread etc. is hidden by DRAMA2 and examples thrown by the thread are reported via DRAMA2 as an action failure. The example below shows a simple implementation of a thread action.

```
class Action1 : public thread::TAction{       // Action Definition
  Action1(std::weak_ptr<Task> theTask):
    TAction(theTask) {}
  void ActionThread(const sds::Id &) override {
    MessageUser("Hello World - from a thread"); }
};
```

Action threads can create their own sub-threads, which can interact with DRAMA.

Kicking Threaded Actions. The "WaitForKick" method and releated methods, allows a threaded action to wait for a kick message. Alternatively, a "KickNotifier" object may be created before say entering a CPU intensive loop. These objects create a thread that waits for a kick message. Multliple child threads of one action may be waiting.

Sending Messages. A "Path" class is provided to enable sending DRAMA messages to other tasks. Message sending is only possible from a threaded action. The thread, but not the task, is blocked to await replies. As a threaded action can have child threads, they may have any number of messages outstanding at any time. The example below shows sending a message from DRAMA2.

```
Path server(...)
...
server.Obey(this, HELLO);
```

There are various message sending methods, including the ability to monitor for changes to the values of parameters in other tasks. By default, the methods will block until the subsidiary action completes, but there are features allowing overriding of the default processing of the various possible replies to a message.

5. Other Features

Most other DRAMA features have been implemented in DRAMA2, and what remains can be implemented as requested. Doxygen has been used to generate documentation and a 130 page manual has been generated.

References

Allan, P. M. 1992, in Astronomical Data Analysis Software and Systems I, edited by D. M. Worrall, C. Biemesderfer, & J. Barnes, vol. 2 of Astronomical Society of the Pacific Conference Series, 126

Farrell, T. J., Bailey, J. A., & Shortridge, K. 1993, in Bulletin of the American Astronomical Society, vol. 25 of Bulletin of the American Astronomical Society

ISO 2011, ISO/IEC 14882:2011 "Information technology – Programming languages – C++" (International Standards Organisation)

Lewis, I. J., et al. 2002, MNRAS, 333, 279