DRAMA2 – An Evolutionary Leap for the DRAMA Environment for Instrumentation Software Development

Tony Farrell*, Keith Shortridge, (Australian Astronomical Observatory)
*E-mail: tony.farrell@aao.gov.au

INTRODUCTION

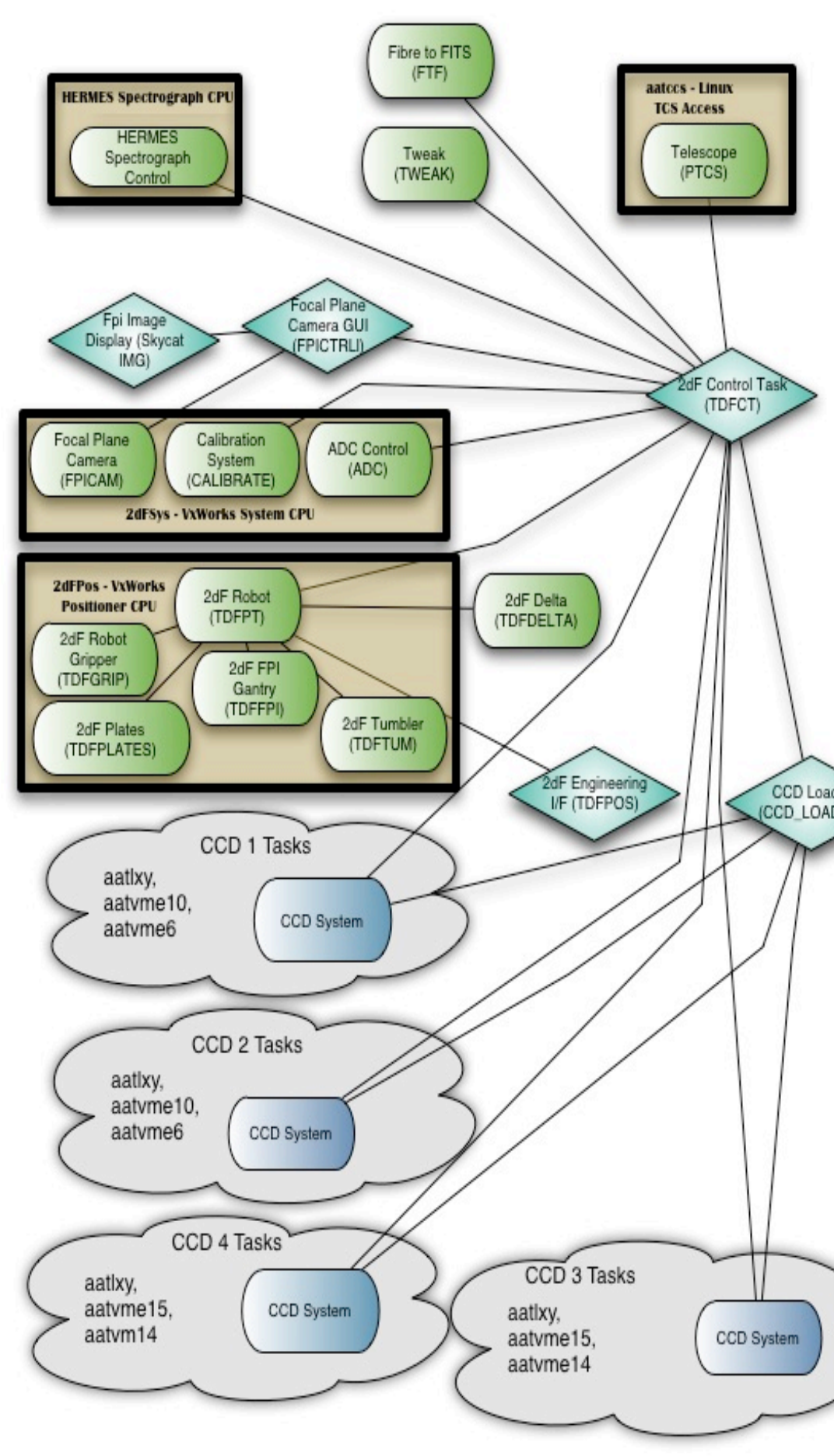
The DRAMA API [1] remains the AAO's primary tool for constructing complex instrumentation systems and has been/is being used by various other observatories. it implements a tasking model; with each named task responding to named messages of a number of different types. In a DRAMA "System", tasks can 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. It has been run on many flavours of UNIX/Linux, VMS, VxWorks and MS Windows, and provided the ability to write soft1 real-time applications and with good performance on, for example, 30Mhz 68020 CPUs. The flexibility allowed systems as complex as the AAO's 2dF system [3] to be implemented – see figure 1, making use of the most appropriate hardware for each job across a distributed system.

Most work is a DRAMA task is done in response to "Obey" messages – in effect, command messages; implementing "Actions". The design approach implements co-operative multi-tasking; multiple actions can 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). The approach has worked well and a strongly objected-oriented task design approach was implementable for tasks written in C.

Attempts were made starting about 1994 to implement C++ interfaces for DRAMA, but the results were relatively poor and various different approaches were tried. One of the early issues was the poor portability of early C++ compilers, some features such as templates and exceptions were not reliably implemented and were not portable. Another was that we were still learning the best approaches to use.

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 being more portable then threads were. In the C API, task authors must work around DRAMA when using threads; but in recent times, many libraries for component control have presumed threads are available and thread support has become widespread and is presumed to be available by most software engineers. We had been working on designs for proper thread support for DRAMA over some years, but had not yet implemented it.

C++11 [4] 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; compilers of interest (GCC and Clang in particular) have implemented the full feature set on machines of interest (Linux and Mac OS X). We have taken advantage of the upgrade of C++ to implement DRAMA2, which will simplify writing and maintaining complex DRAMA tasks.


Fibre to FITS (FTF)
HERMES Spectrograph CPU
HERMES Spectrograph Control
Tweak (TWEAK)
aatccs - Linux TCS Access
Telescope (PTCS)
Fpi Image Display (Skycat IMG)
Focal Plane Camera GUI (FPICTRL)
2dF Control Task (TDFCT)
Focal Plane Camera (FPICAM)
Calibration System (CALIBRATE)
ADC Control (ADC)
2dFSys - VxWorks System CPU
2dFPos - VxWorks Positioner CPU
2dF Robot (TDFPT)
2dF Robot Gripper (TDFGRIP)
2dF FPI Gantry (TDFFPI)
2dF Plates (TDFPLATES)
2dF Tumbler (TDFTUM)
2dF Delta (TDFDELTA)
2dF Engineering I/F (TDFPOS)
CCD Loader (CCD_LOADER)
CCD 1 Tasks
aatlxy, aatvme10, aatvme6
CCD System
CCD 2 Tasks
aatlxy, aatvme10, aatvme6
CCD System
CCD 4 Tasks
aatlxy, aatvme15, aatvm14
CCD System
CCD 3 Tasks
aatlxy, aatvme15, aatvme14
CCD System

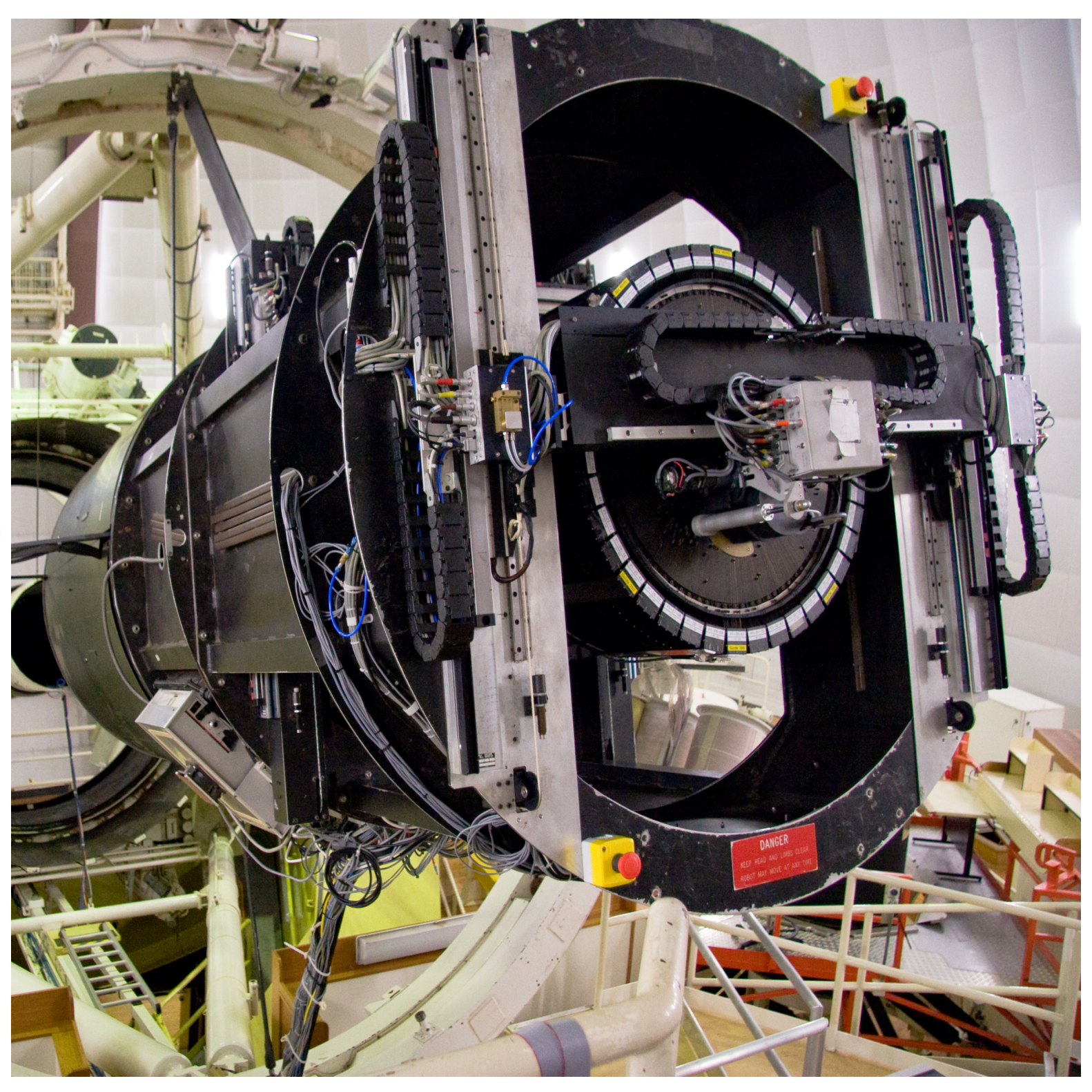


Figure 1. With over 30 DRAMA tasks spread over 10 CPUs, the 2dF/HERMES system is the largest example of an existing DRAMA system within the AAO. Its origins are in the original 2dF DRAMA system commissioned in 1994, but it has been modified many times since, including to support HERMES, showing the flexibility of DRAMA

DRAMA2 has quickly modernized the development of DRAMA tasks. It is well documented and allows reliable tasks to be written quickly. It allows sequenced code to be written for sequenced jobs, but with all the efficient non-blocking DRAMA messaging facilities available. Much of the (potentially risky) complexity of creating threaded distributed applications is hidden from programmers using DRAMA2, in the most common cases.

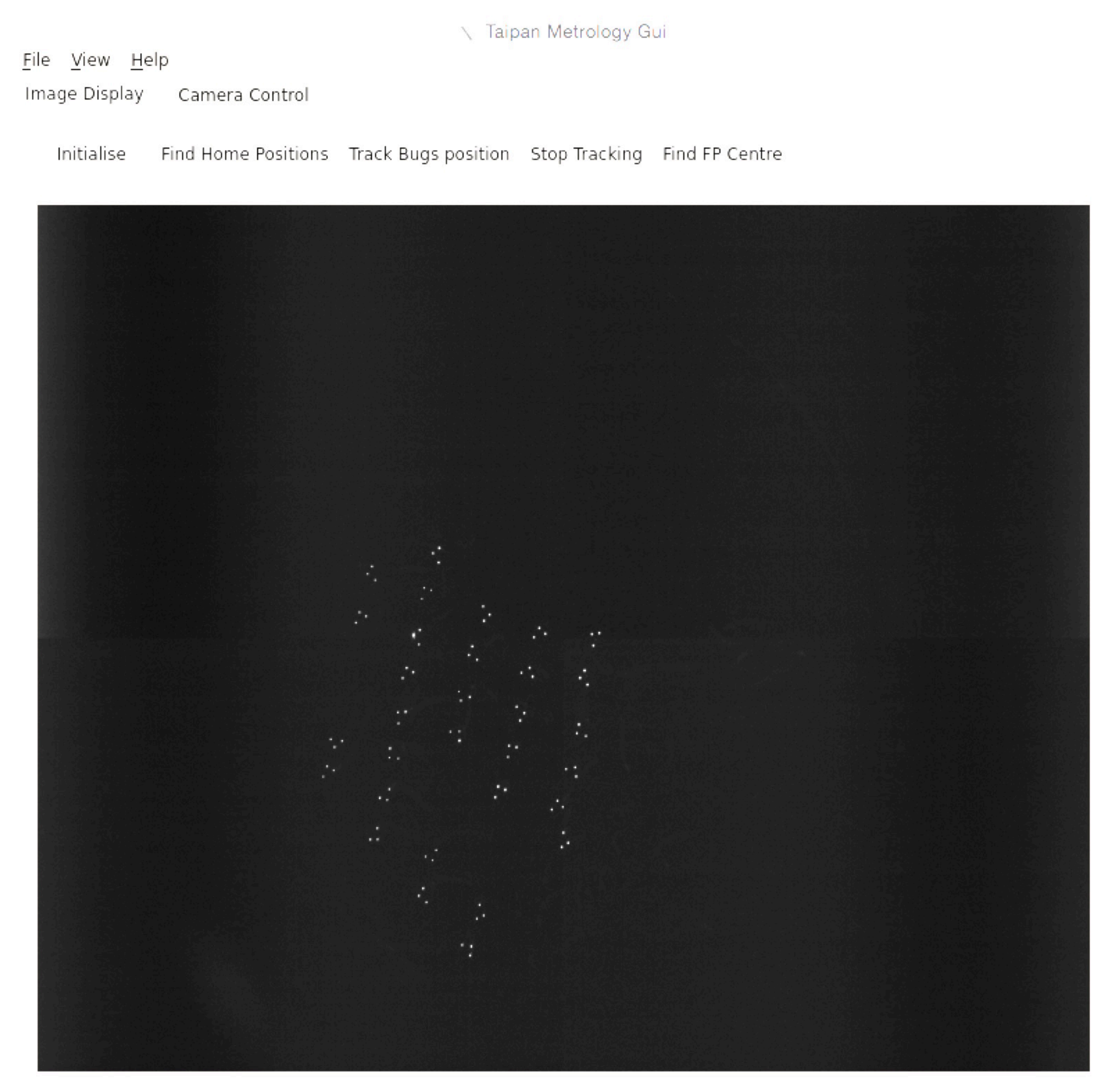

Taipan Metrology Gui
File View Help
Image Display Camera Control
Initialise Find Home Positions Track Bugs position Stop Tracking Find FP Centre

Figure 2:. The TIPAN Starbugs metrology GUI, implemented using the Qt Widget set, is the first implementation of a GUI using DRAMA2 . The buttons send DRAMA messages which move the TIPAN Starbugs. The image display area shows a collection of starbugs under test – each represented by 3 dots.

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

Basic "Hello World" example

Example 1, left, 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.

Threaded "Hello World" example

In example 1 (above), the "HELLO" action is running in the main DRAMA thread. Whilst it can "Reschedule", in the traditional DRAMA way to return control to the DRAMA message thread, the intent of DRAMA2 is to support running actions in threads.

The class "thread::TAction" is an abstract sub-class of "MessageHandler". The user of this class must provide the method "ActionThread", which is invoked within a thread when an Obey message of the specified name is received. When the thread completes, DRAMA is informed and the action is marked as completed. Importantly, all details of thread creation; joining the thread etc. is hidden by DRAMA2. Any exception thrown by the thread is reported via DRAMA2 as an action failure – the task does not abort. Example 2, right, shows a simple implementation of a thread action.

Action threads can create their own sub-threads, which can interact with DRAMA, but the implementer is then responsible for handling creation, joining the thread, dealing with exceptions in the thread, etc.

```
#include "drama.hh"
using namespace drama;

// Action definition.
class Action1 : public thread::TAction{
public:
  Action1(std::weak_ptr<Task> theTask):
    TAction(theTask) {}
private:
  void ActionThread(const sds::Id &) override {
    MessageUser("Hello World - from a thread");
  }
};
// Task Definition
class ExTask : public Task {
private:
  // actions
  Action1 Action1Obj;
public:
  ExTask(const std::string &taskName) :
    Task(taskName), Action1Obj(TaskPtr()) {
    Add("HELLO",
        MessageHandlerPtr(&Action1Obj,
                          nodel()));
    Add("EXIT", &SimpleExitAction);
  }
};
// Main program.
int main() {
  CreateRunDramaTask<ExTask>("TASK2");
  return 0;
}
```

```
#include "drama.hh"

class RunAction : public drama::thread::TAction {
  /* Class which defnies the RUN action */
public:
  RunAction(std::weak_ptr<drama::Task> theTask) :
    TAction(theTask), _theTask(theTask) {}
  ~RunAction() {}
private:
  std::weak_ptr<drama::Task>  _theTask;

  /* Implement the RUN action - as a thread. */
  void  ActionThread(const drama::sds::Id & /*obeyArg */) {
    /* Create a Path object */
    drama::Path server(_theTask,
                       "TASK1", "",
                       "./ex1");
    /* Load (if needed) and get path to TASK1 */
    server.GetPath(this);
    MessageUser("Have gotten path, will send obey");
    /* Send the HELLO message to TASK1 */
    server.Obey(this, "HELLO");
    MessageUser("Subsidiary Action completed");
  }
};

class ClientTask : public drama::Task {

private:
    RunAction RunActionObj;
public:
  /** Task Constructor, from here we add actions */
  ClientTask(const std::string &taskName) :
    drama::Task(taskName), RunActionObj(TaskPtr()) {
    /* Add actions */
    Add("RUN", drama::MessageHandlerPtr(
          &RunActionObj, drama::nodel()));
    Add("EXIT", drama::SimpleExitAction);
  }
  ~ClientTask() {
  }
};

/** main program */
int main()
{
  drama::CreateRunDramaTask<ClientTask>("TASK3");
  return 0;
}
```

Sending DRAMA Messages

A "Path" class is provided to enable sending DRAMA messages to other tasks. In traditional C DRAMA, message sending does not block and an action must explicitly reschedule to message handle replies. In DRAMA2, 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 to the left shows a task implementing the action RUN, which sends an Obey message with the name HELLO to the task TASK1 (our non-threaded example).

Optional arguments to Obey() support sending arguments. Versions of this with timeouts are available.

There a 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.

Documentation and Regression Testing

An important part of the implementation of DRAMA2 was to ensure the documentation was created with the package, rather then the tradition of being tacked on later. The "doxygen" tool was chosen as the interface documentation tool and all interfaces have been documented as the code was written.

A 130-page manual has been generated, working through all the many features of DRAMA2. The manual includes a large number of code examples, all of which is available as compilable code. Generation of the detailed manual and the required examples quickly highlighted various flaws or unnecessary complexities in the initial interfaces, allowing them to be revamped before release.

As example programs were generated to demonstrate and test features they have been added to our regression test facilities. As a result, any change to DRAMA2, or the underlying DRAMA software, is automatically subject to extensive testing.

REFERENCES

[1] T.J. Farrell, K. Shortridge, J.A. Bailey, "DRAMA: An Environment for Instrumentation Software," Bulletin of the American Astronomical Society, Volume 25, No 2, (1993).
[2] Allan P. M., "The ADAM software environment," Astronomical Data Analysis Software and Systems I, 126 (1992) [3] Bailey J. A., et al., "DRAMA: an environment for distributed instrumentation software," Proc. SPIE 2479 , 62 (1995)
[3] Taylor K., et al., "Anglo-Australian Telescope's 2dF Facility," Proc. SPIE 2871 , (1997)
[4] ISO/IEC 14882:2011 "Information technology -- Programming languages -- C++" (2011)

Basic DRAMA2 Design

Implementation as wrapper around the C API

DRAMA2 is implemented 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. Most of the "DRAMA" internal processing is done within this thread. Other threads can send DRAMA messages (and make other DRAMA API calls) but cannot actually read the messages. Much potential complexity is removed and 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

Locking is important to get right! Systems with multiple locks help to avoid lock wait delays, but significantly increase the complexity of the design required to ensure avoiding deadlocks. Since DRAMA2 is an API available to implement applications, it is much harder to avoid deadlocks when using multiple locks. As a result, 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 – a thread that has already taken the lock will not deadlock if it attempts to take it a second time.

The DRAMA2 design allows the DRAMA2 message read thread to block waiting for a message without taking the lock. That thread only takes the lock when processing a message. Since the lock is free any time the DRAMA2 message read thread is waiting for a message, there is plenty of opportunity for application threads to lock access to DRAMA and send messages themselves.

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. An Error Reporting System (ERS) enables extra contextual information to be added when errors occur.

In C++, it is natural to replace the inherited status approach and ERS by exceptions. 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. The DRAMA2 exception class stores the integer status value and information about the context and location of the 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. If the exception is the DRAMA2 exception, the original status value will be available and can be returned to the C API as the status of the call, otherwise another status value will be returned. Any extra context available in the exception will be reported using ERS.

Other Features

Self Defining Data Structures (SDS)

The `sds::Id` class provides access to DRAMA's SDS objects, used to send data between tasks. Action implementations can access any structure sent to them and can send such structures as arguments in any message they send. Some complexities of the underlying SDS system made writing a clean C++ interface hard prior to C++11. In C++11, the move assignment and move copy operators proved liberating, allowing an effective and relatively clean interface to be constructed.

Accessing Command Line Arguments.

All arguments to actions are sent in an SDS structure, but there is a standard approach to command arguments, which make allow simple generic programs to be used send obey messages to any task. Various simple methods are provided by the sds::Id class to construct such arguments. In action receiving the message, sds::Id class methods can be used, but there is also an alterative interface – via the "gitarg" namespace. These are a series of classes that create sub-classes of standard types initialised from an SDS structure. For example, a gitarg::Bool uses an SDS structure to initialise a Boolean type, accepting for example string values "YES" and "NO" to indicate the value.

GUIs

DRAMA provides a number of GUI toolkits, Java and Tcl/Tk based GUIs being commonly used at this point. These will continue to work with DRAMA2 tasks. Additionally, new toolkits will be constructed using DRAMA2, with Python likely to be the first.

The support for working with threaded systems easily ensures DRAMA2 can be used with many other modern systems. The first DRAMA2 task implemented outside the package itself was a GUI using the Qt widget set, an extensively threaded environment – this is on e of the GUIs for the AAO's TAIPAN project – see figure 2.

DRAMA Parameters

Two classes are provided to support creating and accessing DRAMA Task parameters. DRAMA Task parameters are represented within an SDS Structure, so there is much overlap with SDS support. The class "Parameter" allows parmeters to be created and accessed as if they are basic C++ types. E.g., assigned to and from.

The ParSys class is more traditional for DRAMA, allowing accessed to named parameters via Get/Put methods supporting various types.

Logging

DRAMA's logging system has been re-implemented to support threads. In addition to output the standard DRAMA log information, it outputs the ID of the thread from which is log message is generated.

Extensive use of RAII

Resource Allocation is Initialisation (RAII) is used extensively throughout DRAMA2 to ensure correct management of locks and threads. For example. Locks are always taken

.

Kicking Threaded Actions

A DRAMA Action can be "Kicked", which provides a method for other tasks to communicate with a running action. Often used for Action cancellation, Kick messages are flexible and a system design may use them to update a running action.

In DRAMA2, the "WaitForKick()" method and related methods allow a thread to wait for a kick message to be received.

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. The caller can ask the object if a kick was received and respond correctly.