AstroGrid-D

Deliverable D6.4

Prototype Implementation of grid-enabled Monitoring Methods – Documentation and Test Report[1]

Deliverable	D6.4
Authors	Thomas Radke (AEI)
Editors	Thomas Radke (AEI)
Date	5 March 2007
Document Version	1.0.0
Current Version	1.0.0
Previous Versions	

A: Status of this Document

Officially approved document for project deliverable D6.4.

B: Reference to project plan

This deliverable document refers to the task TA VI-III *"Implementierung gridfähiger Zugriffsmethoden für Monitoring"* and milestone M18 of work package WP-6 in the project plan.

[1]This work is part of the D-Grid initiative and is funded by the German Federal Ministry of Education and Research (BMBF).

C: Abstract

This document describes the prototype implementation of grid-enabled monitoring methods in selected AstroGrid-D applications, available as of month 18 into the project (February 2007).

D: Changes History

Version	Date	Name	Brief summary
0.0.1	2 February 2007	Thomas Radke	Working Draft Creation
0.0.2	14 February 2007	Thomas Radke	Description of Cactus Monitoring/Steering Thorns
0.0.3	15 February 2007	Thomas Radke	Description of Cactus Metadata Management in the Portal
0.0.4	20 February 2007	Thomas Radke	Introduction and Summary Finalised Version of Internal Working Draft
0.1.0	23 February 2007	Thomas Radke	Announcement as Official Working Draft
1.0.0	5 March 2007	Thomas Radke	Published as official AstroGrid Deliverable Document

E:

Contents

1 Introduction

Following the design of the basic structure of grid-enabled monitoring & steering methods for AstroGrid applications, the next task in the project workplan for working group WG-VI *"Grid Job Monitoring & Steering"* was to implement a first prototype for selected AstroGrid use cases. Although the initial version of such methods so far provides only limited functionality, they can already be used to monitor simple Grid jobs running on a small subset of resources available in the AstroGrid testbed.

This deliverable document describes the grid-enabled monitoring and steering methods implemented for the AstroGrid use case *Cactus* [1], a simulation framework used in gravitational wave analysis at AEI to numerically solve Einstein's equations of general relativity. Their basic design was defined in working group WG-VI's architecture design document [2] and has also been presented at various AstroGrid project meetings [3, 4, 5].

The methods described in this document have been specifically developed for and are closely integrated into the Cactus framework to monitor and steer Cactus simulations, however their design should be generic and flexible enough to be incorporated in other AstroGrid use cases as well. Two important preconditions for this are the availability of sufficient user documentation for the software modules and the interfaces implemented, together with free public access to the code for other software developers. The issue of code documentation is addressed in sections 2 and 3 in this deliverable document. Section 4 describes how the implemented prototypes are disseminated within AstroGrid, and how other software developers can access them. Finally, section 5 gives a short summary on practical experiences gathered in the process of developing and testing the software prototypes of working group WG-VI and related services.

2 Implementation of integrated grid-enabled Monitoring Methods as Cactus Thorns

For the Cactus use case (described in full detail in the AstroGrid use case survey[1]), the work within AstroGrid's working group VI focused both on the improvement and grid-enabling of existing Cactus monitoring/steering methods as well as on the concrete design and prototype implementation of new code modules providing new features for Cactus users. All modules follow the Cactus philosophy of encapsulating the implemented functionality in the form of Cactus *thorns* and can therefore be integrated seemlessly into the Cactus Computational Toolkit.

A considerable amount of work was also spent in implementing a specific Cactus application use case scenario for monitoring the results of *Cactus Integration Tests* in a *Cactus User Portal*, based on other AstroGrid technology developed in work package WP-II (an AstroGrid metadata information service) and work package WP-VII (a web portal based on the standard GridSphere portal framework).

2.1 Cactus Thorn FORMALINE

FORMALINE was originally written by Erik Schnetter as *a Cactus thorn to send meta information about a Cactus simulation run to a server, so that it is kept there forever* (excerpt from the original documentation). Within AstroGrid, thorn FORMALINE was specifically adopted to make use of the services developed in work package WP-II *"Provision and Management of Metadata"* and work package VII *"User Interfaces and APIs"*.

FORMALINE is now able to collect Cactus metadata, generate an RDF representation for it, and send it to one or more AstroGrid information services. While this information is then immediately available to users and can be accessed in order to monitor the status of their running simulations, the idea beyond this approach is to store metadata about *all* Cactus simulations of *all* users, no matter whether their simulations are running on a local machine or within a Grid context; the metadata will be stored and archived in the external information service and can be accessed and further processed at any later time, independent of the actual simulation and the environment it ran in.

Access to the metadata in various ways, eg. as an overall summary status list of all simulations or as user-defined queries for specific metadata information, is possible through a Cactus-specific user portal as developed together with work package VII (see section 3.2 on page 11 for details).

2.1.1 Implemented Functionality

FORMALINE's existing implementation was extended by several C functions to (1) query and collect specific metadata about the running simulation (using the Cactus thorn programming API[8]), to (2) translate this metadata into a dynamically generated temporary RDF/XML document, and to (3) establish a TCP/IP connection to an external information service and upload the RDF/XML document (using the AstroGrid information service programming interface).

Metadata information collected by thorn FORMALINE includes:

- the exact start date/time of the simulation
- the number of processors used by this simulation, and the host where the run was started on (processor 0 for a parallel run)
- the user name of the job's owner
- the name, location, and code release of the Cactus executable
- the name and location of the parameter file
- the current working directory and the location where Cactus output data will be written to
- a full listing of all parameters (names and typed values) set in the parameter file for this simulation

This information is regarded as static metadata and therefore sent once at simulation startup to an external AstroGrid information service.

At periodic intervals during the simulation's runtime, thorn FORMALINE can also send dynamic metadata such as the current iteration number, the current physical simulation time or the termination condition and time in case the run is about to finish.

In addition to gathering and uploading the above-mentioned predefined simulation metadata, thorn FORMALINE also supports the PUBLISH API (as described in section 2.2). FORMALINE can register callback functions to process metadata defined and published by other code modules activated in the same Cactus simulation.

2.1.2 Parameters of Thorn FORMALINE

The functionality of thorn FORMALINE can be controlled via parameter settings in a simulation's parameter file. Some of these parameters have additional logic built-in so that they can also be steered at runtime.

`boolean Formaline::send_as_rdf`
whether to send Cactus metadata from this simulation to an external information service in RDF format

`string Formaline::rdf_hostname[5]`
array parameter to specify the hostname for one or more (up to 5 different) external information services

`integer Formaline::rdf_port[5]`
array parameter to specify the port number to connect to for one or more (up to 5 different) external information services identified via their hostname(s)

`boolean Formaline::use_relay_host`
whether to use relaying to establish a TCP network connection between the simulation and an external information service (necessary when running on an internal compute node with no direct access to the outside)

`string Formaline::relay_host`
the name of the relay host if relaying is used

`integer Formaline::timeout`
timeout (in seconds) for sending meta information to an external information server

`integer Formaline::update_interval`
the update interval (in seconds) for publishing dynamic simulation metadata

`integer Formaline::publish_level`
the importance level for metadata to be published via the PUBLISH API

While the connection to an external information service is by default established directly, it can be relayed through a proxy host. This is usually necessary for the case when the simulation is running on a cluster or supercomputer where the compute nodes are *hidden* in an internal/VPN network and therefore cannot talk to outside services directly (as described in [7, 2]). Relaying is implemented in thorn FORMALINE as a function which – if activated by the user via a parameter file setting (see above) – starts a remote shell on the cluster's headnode and relays the TCP/IP communication through a proxy process there.

2.2 Cactus Thorn PUBLISH

Thorn PUBLISH was developed in AstroGrid's work package WP-VI, and in close collaboration with work package WP-II, as a new thorn for the Cactus computational framework. It provides generic functionality to announce and publish user-defined information about running Cactus simulations. User functions are defined for publishing arbitrary metadata in a structured format. Callback functions can be registered to publish the announced metadata in such a way that it is easily retrievable at a later time through external information services.

2.2.1 Implemented Functionality

Thorn PUBLISH uses the general concept of metadata – *information about data* – in order to define a flexible way for describing arbitrary user-defined runtime information about a simulation. The most basic entity of metadata is described as a *key/value* pair; a scalar *value* of defined datatype holding the actual information contents, and an associated *key* as a character string uniquely identifying that value. Based on this basic scalar value entity, it is also possible to construct a structured metadata entity by supplying a Cactus table of key/value pairs as its value. Optionally, the Cactus Publish infrastructure allows each metadata entry to be tagged with additional information, eg. a name identifying the source of the published metadata, the current iteration number and physical time or a date/time stamp to place the metadata publication in a runtime context.

It should be noted here that metadata entities are – in contrast to actual data (such as output files) generated during the simulation – assumed to be small in their overall size, making it possible to transparently process and publish them without (much) user-visible impact on the runtime performance of the ongoing simulation.

Thorn PUBLISH provides two APIs, each one consisting of a set of aliased functions:

1. a user API to publish user-defined metadata

   Application thorns can use this set of aliased functions to publish user-defined metadata describing specific runtime information about the ongoing simulation.

   Metadata can be published as an entity of a single scalar value with a generic CCTK[2] datatype, or as compound entity of multiple such scalar values, defined in a key/value table.

2. a registry API to register/unregister Publish callback functions

   Infrastructure thorns can provide callback functions for the Publish user API and register them with thorn PUBLISH at simulation startup. This thorn will then invoke all registered callbacks each time an application thorn calls any of the Publish user API functions.

   Publish callback functions are the actual worker routines behind the Publish API: they consume the published user-defined metadata and process/publish them in various ways.

All functions of both the Publish user and registry API are described in detail in appendix A.

2.2.2 Parameters of Thorn PUBLISH

The functionality of thorn PUBLISH can be controlled via parameter settings in a simulation's parameter file.

So far there is only a single integer parameter:

```
integer Publish::publish_every
How often to publish some example data using the Publish API
```

Setting this integer parameter to a positive value will activate the self-test of thorn PUBLISH where the iteration number and the current physical time of the ongoing simulation are published to all registered callback listeners.

2.3 Cactus Thorn HTTPS

The Cactus Computational Toolkit includes a Cactus thorn named HTTPD, written by Gabrielle Allen, Tom Goodale and Thomas Radke, which implements a web server integrated into a Cactus simulation. This web server provides full-fledged functionality for application-specific monitoring and steering capabilities. It uses the HTTP protocol for communication and can therefore be contacted from any standard web browser[6].

Within AstroGrid, a new thorn HTTPS was developed on top of the existing thorn HTTPD. In its first prototype it provides the same Cactus-specific monitoring and steering functionality as HTTPD, but was enhanced to using HTTPS (HTTP over OpenSSL TCP/IP socket connections) as the standard network protocol for client-server communication. This gives Cactus users a *secure* method to monitor and steer their Cactus simulations online, which was one of the requirements described in [7]. On the simulation startup page they can now log into the running simulation using a self-defined password which will be prompted for by the web browser and automatically transfered

[2] *C*actus *C*omputation *T*ool*K*it

to the web server for user authentication. After successful authentication, all further communication between the user (through the web browser) and the simulation – with thorn HTTPS acting as its web server frontend – will be encrypted using standard OpenSSL functionality, just like in other web and Grid services.

3 Cactus Metadata Management in the Portal

The Cactus Computational Toolkit comes with a built-in mechanism to test individual parts of the code and verify whether they are still functional; this mechanism – running a Cactus simulation with a known input (the testsuite parameter file) and known output (the expected data files and their contents) – is called a Cactus testsuite.

Within AstroGrid a specific Cactus use case scenario was developed to automate the procedure of regular Cactus tests and allow users to conveniently monitor the status and history of such test simulations. This scenario was realised using AstroGrid technology: (1) the information service developed in work package WP-II for storing and managing application-specific metadata, and (2) the GridSphere portal framework provided by work package WP-VII to build a Cactus User Portal as a standardised web-based user interface to access and query application-specific metadata. In work package WP-VI a Cactus Integration Tests module for generating the metadata and a CactusRDF portlet for presenting the metadata were deveoped. These two software components are described in the following.

3.1 Cactus Integration Tests Module

In order to automate the process of testing individual Cactus code modules, a software module wrapping the Cactus testsuite mechanism was developed. This Cactus Integration Tests module includes the following interdependent unit tests which are executed in the given order:

1. check out the Cactus flesh and all thorns listed in the given thornlist
2. create a Cactus configuration with the given configuration options
3. build the Cactus executable
4. build all utility programs associated with thorns
5. run all available Cactus testsuites

After running all unit tests, the module processes the corresponding logfiles, extracts a summary of test results, and generates an RDF/XML document which represents them equivalently in a machine-readable form. For the translation of human-readable textual metadata into RDF/XML, an RDF schema was developed describing the following items of information:

- a descriptive name identifying this test
- the exact date/time of the test
- the hostname of the machine the test was run, plus the total number of processors used
- the login name of the user who ran the test
- the configuration options and thornlist used to build a Cactus executable
- the status results (`succeeded/failed`) and logfiles for each individual unit test:

- for the testsuites, also the names and a summary of passed/failed tests

Finally, the RDF/XML document is uploaded by the CACTUS INTEGRATION TESTS module to an external AstroGrid information service to store the integration test results.

3.2 CACTUSRDF Portlet

Closely related to the generation of Cactus metadata on the application side is its presentation through a human-machine interface in the form of a web-based Cactus user portal. Such a portal, based on the GridSphere portal framework, has been deployed by work package WP-VII. In work group WP-VI the necessary Cactus metadata management portlet was developed which provides functionality to query Cactus integration test results from an external AstroGrid information service and present them in a flexible and user-friendly format. Its implementation follows the requirements specification on a Cactus user portal which have been described in [11]. Since it uses AstroGrid technology (the Cactus integration test metadata RDF scheme and the RDF/SPARQL API to interact with an external information service), the portlet was named CACTUSRDF.

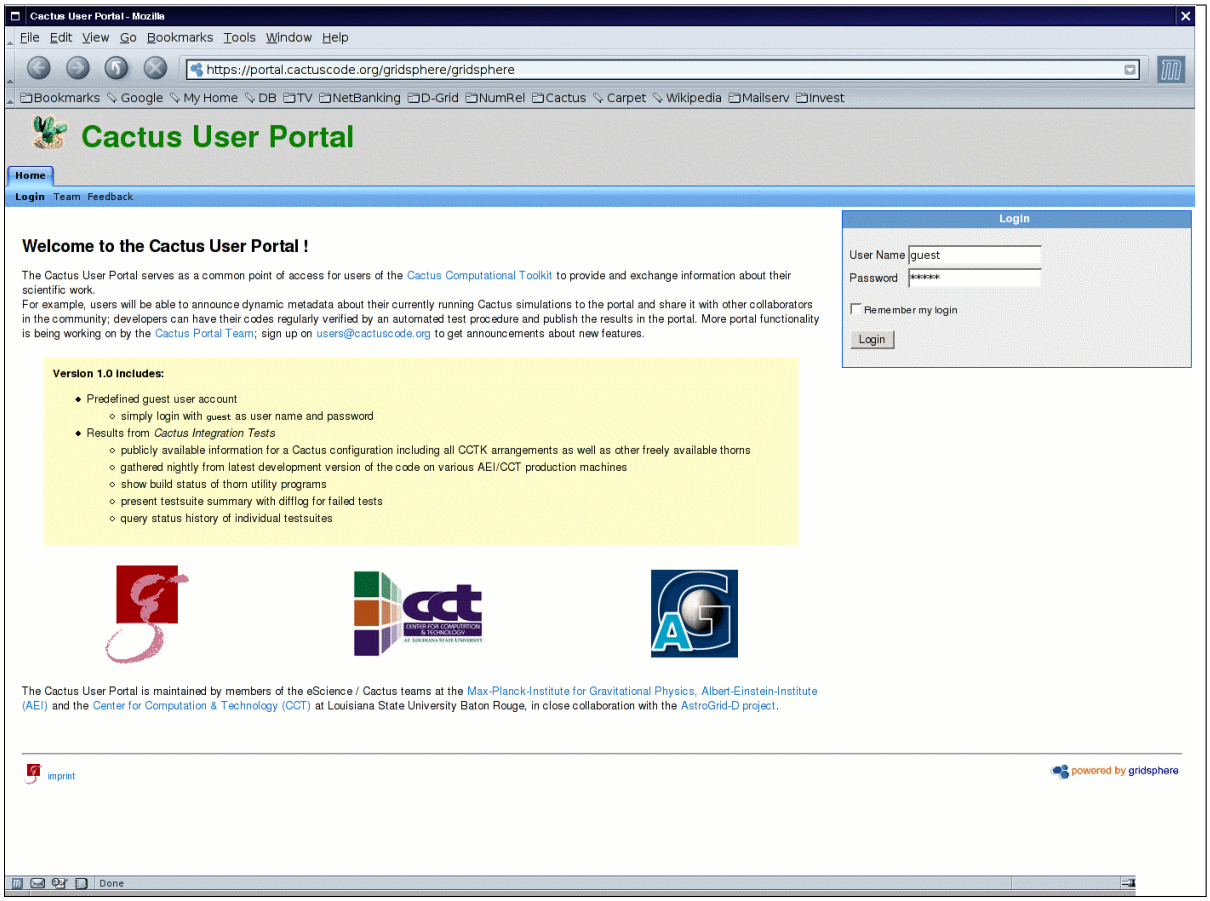

Figure 1: Login Page of the *Cactus User Portal*

When logged in, the user can then switch to the Cactus metadata page provided by the CACTUSRDF portlet and display Cactus integration test results in three different ways:

Version 1.0 of the Cactus User Portal, which was released as part of the D6.4 deliverable of work package VI, includes a predefined guest user account by which Cactus users can simply login with `guest` as user name and password. A snapshot of the Cactus User Portal login page can be seen in figure 1.

1. a summary view of all most recent integration tests from all test machines, showing the status of all unit tests
2. a detailed view of Cactus testsuites for an individual integration test, showing the status of all testsuites
3. a history view for an individual Cactus testsuite, queried over all available integration tests results on all test machines

For the queries, the user can also specify parameters to restrict size of the resulting metadata shown in the portal by binding the result set eg. to an individual Cactus integration test (identified by its name), to the user who ran the test (identified by the user's login name), or to a specific test machine where the test was run (identified by the hostname).

The *Cactus User Portal* is available on `https://portal.cactuscode.org`. Also available is a *Numerical Relativity Portal* with personalised user access for physicists of the Numerical Relativity community; this portal, online under `https://portal.aei.mpg.de`, provides Cactus Integration Test results for the majority of non-public Cactus thorns which are used by the numerical relativists for their daily Cactus production runs.

4 Code Dissemination

4.1 Cactus Thorn FORMALINE

The Cactus thorn FORMALINE is publicly downloadable via anonymous CVS from

```
export CVSROOT=:pserver:cvs_anon@cvs.aei.mpg.de:/numrelcvs  # for bash
setenv CVSROOT :pserver:cvs_anon@cvs.aei.mpg.de:/numrelcvs  # for (t)csh

cvs login                                          # password is 'anon'
cvs checkout AEIThorns/Formaline
```

Documentation for this thorn is contained in the CVS source module in the toplevel directory as a REAME file.

4.2 Cactus Thorns PUBLISH and HTTPS

The Cactus thorns PUBLISH and HTTPS are publicly downloadable via anonymous CVS from

```
export CVSROOT=:pserver:cvs_anon@cvs.aei.mpg.de:/eScienceCVS  # for bash
setenv CVSROOT :pserver:cvs_anon@cvs.aei.mpg.de:/eScienceCVS  # for (t)csh

cvs login                                          # password is 'anon'
cvs checkout AstroGrid/Cactus/Thorns/HTTPS
cvs checkout AstroGrid/Cactus/Thorns/Publish
```

The PUBLISH thorn also contains the RDF schema describing Cactus metadata defined by this thorn and the user-defined metadata generated throughtthe PUBLISH API.

Documentation for both thorns is contained in the CVS source modules in a subdirectory `doc/`; it comes in LaTeXformat so that it can be easily integrated in the standard Cactus thornguide documentation.

4.3 Cactus Integration Tests

The source code for the *Cactus Integration Tests* is publicly downloadable via anonymous CVS from

```
export CVSROOT=:pserver:cvs_anon@cvs.aei.mpg.de:/eScienceCVS  # for bash
setenv CVSROOT :pserver:cvs_anon@cvs.aei.mpg.de:/eScienceCVS  # for (t)csh

cvs login                                          # password is 'anon'
cvs checkout AstroGrid/Cactus/IntegrationTests
```

This CVS module contains both the RDF metadata schema used for Cactus integration tests, as well as the perl script to run the tests on a given machine, generate the RDF metadata, and send them off to the AstroGrid metadata information service.

4.4 Cactus RDF Portlet

The Cactus RDF portlet for GridSphere is available for AstroGrid users via the AstroGrid SVN:

```
svn checkout svn://svn.gac-grid.org/software/gridsphere/cactusrdf
```

5 Test Report Summary

The Cactus simulation monitoring thorns described in section 2 have been thoroughly tested so far on cluster production machines of the *Numerical Relativity* group at AEI and their collaborators at the Center for Computation and Technology (CCT) at Louisiana State University, Baton Rouge, USA. The extensions to the existing webserver thorn HTTPD were made available to the Cactus developers community in October 2006, and have been used since then by several Cactus users in test simulation runs. The code was also tested by the AEI eScience group on the SGI Altix machine at Leibniz Rechenzentrum (LRZ) München and on an IBM SP5 supercomputing facility at the MPG Rechenzentrum Garching (RZG) both of which are part of the AstroGrid computing resource testbed.

Since month 17 into the AstroGrid project, the Cactus Integration Tests module and the CactusRDF portlet are being used as production services in two user portals based on GridSphere: the publicly available *Cactus User Portal [9]* and the *Numerical Relativity Portal [10]* which is restricted to physicists in the *NumRel* community at AEI and CCT. Both portals are running on a production server machine at AEI (`portal.aei.mpg.de`) and are managed by AEI's eScience group. Integration tests have been running since November 2006 as nightly `cron` jobs on 4 different supercomputers and HPC clusters at AEI and CCT. The results are uploaded to an AstoGrid information service instance and accumulated there. The information service instance was deployed on a Grid server machine at AEI (`buran.aei.mpg.de`), along with proper firewall setup and periodic database backups.

For the *Cactus Metadata Management* services listed above, a prototype implementation of the AstroGrid information service in its versions 0.0.1 and 0.0.3 were used and heavily tested in the process of making them production-ready. While most of the required functionality to manage Cactus metadata has already been implemented in these early versions, practical experiences using the AstroGrid information service revealed certain performance problems when querying increasing amounts of metadata (as being produced regularly during nightly Cactus integration tests). Potential reasons for these inefficiency problems are to be found in the implementation of the internal query engine of the information service and in the formulation of SPARQL queries to query metadata from individual contexts. Together with working group WG-II, members of working group WG-VI will continue to work on these issues and provide improved an version in their next code releases.

F: References / Bibliography

References

[1] AstroGrid-D Use Case inquiry. AstroGrid public webpage;
http://www.gac-grid.org/project-documents/UseCases.html

[2] Thomas Radke: Architecture of generic grid-enabled Monitoring & Steering Methods in AstroGrid-D Applications. Architecture Specification, Work Group VI deliverable document D6.3, AstroGrid project;
http://www.gac-grid.org/project-documents/deliverables/wp6/WG6_D6_3.pdf

[3] Thomas Radke: Status WG6 and Informationservice for Cactus. Presentation at the 4th AstroGrid Project Meeting, 24./25. July 2006, ZAH Heidelberg;
http://www.gac-grid.org/project-overview/events-meetings/meetings/meetingzib-1/wg6-status-report.pdf

[4] Thomas Radke: Cactus Metadata Management. Presentation at the 5th AstroGrid Project Meeting, 14./15. November 2006, MPE Garching;
http://www.gac-grid.org/project-overview/events-meetings/meetings/meeting-MPE/cactus-metadata-management.pdf

[5] Thomas Radke: Status Report WP-VI: *Grid Monitoring and Steering*. Presentation at the 6th AstroGrid Project Meeting, 30. January 2007, AEI Golm;
http://www.gac-grid.org/project-overview/events-meetings/meetings/AEIMeeting/Presentations/wg6-status-report.pdf

[6] Thomas Radke: Existing Monitoring & Steering Functionality in AstroGrid-D Applications. Comparison Study, Work Group VI deliverable document D6.1, AstroGrid project;
http://www.gac-grid.org/project-documents/deliverables/wp6/WG6_D6_1.pdf

[7] Thomas Radke: Requirements on grid-enabled Monitoring & Steering Methods in AstroGrid-D Applications. Requirements Specification, Work Group VI deliverable document D6.2, AstroGrid project;
http://www.gac-grid.org/project-documents/deliverables/wp6/WG6_D6_2.pdf

[8] *Cactus Reference Guide* Manual describing the Cactus flesh and thorn programming interfaces.
http://www.cactuscode.org/old/Guides/Stable/ReferenceManual/ReferenceManualStable.pdf

[9] *Cactus User Portal* A public user portal for the Cactus community.
https://portal.cactuscode.org

[10] *Numerical Relativity Portal* A portal for members of the Numerical Relativity community.
https://portal.aei.mpg.de

[11] Oliver Wehrens: Requirement analysis for specific components and services of the Astro community for the GACG portal. Work Group VII deliverable document D7.2, AstroGrid project;
http://www.gac-grid.org/project-documents/deliverables/wp7/M2.pdf

G: Appendix

Appendix A: Thorn PUBLISH API Function Descriptions

`Publish{Boolean,Int,Real,String,Table}`

Publish user API functions to publish user-defined information as an entity of either a single scalar value of a given datatype, or a table of such scalar values

Synopsis

```
#include "Publish.h"

CCTK_INT istatus = PublishBoolean (CCTK_POINTER_TO_CONST cctkGH,
                                   CCTK_INT              level,
                                   CCTK_INT              value,
                                   CCTK_STRING           key,
                                   CCTK_STRING           name)

CCTK_INT istatus = PublishInt     (CCTK_POINTER_TO_CONST cctkGH,
                                   CCTK_INT              level,
                                   CCTK_INT              value,
                                   CCTK_STRING           key,
                                   CCTK_STRING           name)

CCTK_INT istatus = PublishReal    (CCTK_POINTER_TO_CONST cctkGH,
                                   CCTK_INT              level,
                                   CCTK_REAL             value,
                                   CCTK_STRING           key,
                                   CCTK_STRING           name)

CCTK_INT istatus = PublishString  (CCTK_POINTER_TO_CONST cctkGH,
                                   CCTK_INT              level,
                                   CCTK_STRING           value,
                                   CCTK_STRING           key,
                                   CCTK_STRING           name)

CCTK_INT istatus = PublishTable   (CCTK_POINTER_TO_CONST cctkGH,
                                   CCTK_INT              level,
                                   CCTK_INT              table,
                                   CCTK_STRING           key,
                                   CCTK_STRING           name)
```

Parameters

`cctkGH` optional pointer to a cGH structure, or NULL if not available

`level` the importance level for the entity to be published; this integer parameter should take as its value one of the following preprocessor constants defined in the `Publish.h` header file: `PUBLISH_LEVEL_ERROR`, `PUBLISH_LEVEL_WARNING`, `PUBLISH_LEVEL_NOTICE`, `PUBLISH_LEVEL_INFO`, `PUBLISH_LEVEL_DEBUG`

`value` the value of the entity to be published; this is either a scalar value of type `CCTK_INT`, `CCTK_REAL`, or `CCTK_STRING`, or a key/value table of one or more scalar values of that type (note that `PublishBoolean()` expects a `CCTK_INT` typed value which is then

interpreted internally as a boolean (*true* or *false*)

`key` the case-sensitive key to associate with the entity to be published (must be passsed as a pointer to a non-empty string)

`name` an optional case-sensitive identifier to be attached to the entity to be published (if passed as a pointer to a non-empty string)

Result

`istatus` (≥ 0)
how often this entity was published by registered Publish callbacks

Errors

`PUBLISH_ERROR_INVALID_KEY`	the `key` argument is a NULL pointer or points to an empty string
`PUBLISH_ERROR_INVALID_LEVEL`	the `level` argument is negative

Discussion

This set of Publish API functions can be used by application thorns to publish user-defined metadata: either as a single entity of a scalar value of one of Cactus's generic datatypes `CCTK_INT`, `CCKT_REAL`, or `CCTK_STRING`; or as a compound entity of multiple such scalar values, defined in a key/value table. For scalar entities it is also possible to publish a boolean value (*true* or *false*); since there doesn't exist a corresponding CCTK datatype for that in Cactus, such a value must be passed as a `CCTK_INT` (non-zero or zero).

Each published entity's value gets associated with it a case-sensitive string `key` which can be used as a unique identifier when querying for specific metadata. Additionally, an optional case-sensitive string `name` can be given which is then also attached to the published entity.

The `cctkGH` pointer argument is optional; when available in the calling routine it should be passed through the Publish API as a hint to the registered publish callback functions. If not available, a NULL pointer value should be passed instead.

The `level` positive integer argument may be used by registered Publish callback functions to decide whether this entity should be published or not. Its value may be set to one of the following preprocessor integer constants defined in the `Publish.h` header file:

`PUBLISH_LEVEL_ERROR`	($= 0$)	for error conditions
`PUBLISH_LEVEL_WARNING`	($= 1$)	for warning conditions
`PUBLISH_LEVEL_NOTICE`	($= 2$)	for normal, but important, conditions
`PUBLISH_LEVEL_INFO`	($= 3$)	for normal, but less important, conditions
`PUBLISH_LEVEL_DEBUG`	($= 4$)	for debugging purposes

Note that these predefined constants are similar, but not identical to, the `CCTK_VWarn()` warning levels. The total number of registered callbacks which did publish the given entity is returned as result of the Publish API functions. It can be zero if all registered callbacks decided (based on the `level` argument) not to publish the entity, or if no callbacks had been registered in the first place, eg. if no thorn providing Publish callbacks was activated, or – as is often the case in multiprocessor runs – callbacks were registered only on a single processor (eg. on processor 0).

See Also

`Publish{Boolean,Int,Real,String,Table}_Register()`
Register Publish API callback functions.

`Publish{Boolean,Int,Real,String,Table}_Unregister()`
Unregister Publish API callback functions.

Examples

C++

```
#include <iostream>

#include "cctk.h"
#include "cctk_Arguments.h"
#include "util_Table.h"

#include "Publish.h"

// we assume that the current routine uses the DECLARE_CCTK_ARGUMENTS macro
// to get access to cGH information
if (CCTK_IsFunctionAliased ("PublishTable"))
{
  std::ostringsteam buffer;
  buffer << "cctk_iteration = " << cctk_iteration << std::endl
         << "cctk_time      = " << cctk_time      << std::endl;
  const int table = Util_TableCreateFromString (buffer.str().c_str());
  PublishTable (NULL, PUBLISH_LEVEL_DEBUG, table,
                "Runtime Info", CCTK_THORNSTRING);
  Util_TableDestroy (table);
}
```

Fortran

```
#include "cctk.h"

#include "Publish.h"

integer      istatus
CCTK_POINTER cctkGH

call CCTK_IsFunctionAliased (istatus, "PublishString")
if (istatus .ne. 0) then
  cctkGH = CCTK_NullPointer ()
  call PublishString (cctkGH, PUBLISH_LEVEL_NOTICE, &
                      "Horizon found", "event", CCTK_THORNSTRING)
end if
```

`Publish{Boolean,Int,Real,String,Table}_Register`

Publish registry API: Register callback functions for the Publish API

Synopsis

```
CCTK_INT istatus =
        PublishBoolean_Register (CCTK_INT (cb) (CCTK_POINTER_TO_CONST cctkGH,
                                                CCTK_POINTER          cb_data,
                                                CCTK_INT              level,
                                                CCTK_INT              value,
                                                CCTK_STRING           key,
                                                CCTK_STRING           name),
                                 CCTK_POINTER cb_data,
                                 CCTK_STRING  name)

CCTK_INT istatus =
        PublishInt_Register     (CCTK_INT (cb) (CCTK_POINTER_TO_CONST cctkGH,
                                                CCTK_POINTER          cb_data,
                                                CCTK_INT              level,
                                                CCTK_INT              value,
                                                CCTK_STRING           key,
                                                CCTK_STRING           name),
                                 CCTK_POINTER cb_data,
                                 CCTK_STRING  name)

CCTK_INT istatus =
        PublishReal_Register    (CCTK_INT (cb) (CCTK_POINTER_TO_CONST cctkGH,
                                                CCTK_POINTER          cb_data,
                                                CCTK_INT              level,
                                                CCTK_REAL             value,
                                                CCTK_STRING           key,
                                                CCTK_STRING           name),
                                 CCTK_POINTER cb_data,
                                 CCTK_STRING  name)

CCTK_INT istatus =
        PublishString_Register  (CCTK_INT (cb) (CCTK_POINTER_TO_CONST cctkGH,
                                                CCTK_POINTER          cb_data,
                                                CCTK_INT              level,
                                                CCTK_STRING           value,
                                                CCTK_STRING           key,
                                                CCTK_STRING           name),
                                 CCTK_POINTER cb_data,
                                 CCTK_STRING  name)

CCTK_INT istatus =
        PublishTable_Register   (CCTK_INT (cb) (CCTK_POINTER_TO_CONST cctkGH,
```

```
                                        CCTK_POINTER          cb_data,
                                        CCTK_INT              level,
                                        CCTK_INT              value,
                                        CCTK_STRING           key,
                                        CCTK_STRING           name),
                          CCTK_POINTER cb_data,
                          CCTK_STRING  name)
```

Parameters

`cb` the function pointer of the callback function be registered

`cb_data` an optional user-defined data pointer to associate with the callback function to be registered (may be given as `NULL` pointer)

`name` a case-sensitive non-empty string uniquely identifying the callback function to be registered

Result

`istatus` All register functions return 0 (zero) for success, or a negative integer value in case of an error.

Errors

`PUBLISH_ERROR_INVALID_CALLBACK` the `cb` argument is a NULL pointer

`PUBLISH_ERROR_INVALID_CALLBACK_NAME`
the `name` argument is a NULL pointer or points to an empty string

`PUBLISH_ERROR_CALLBACK_ALREADY_REGISTERED`
a callback under the same `name` has already been registered

Discussion

Before application thorns can make practical use of the Publish API (as described on pages 5ff), publish callback functions must be registered; such functions will receive the information to be published and then do the actual work.

The Publish registry API provides a separate function for registering a callback to handle each of the supported scalar datatypes and for key/value tables. Each callback is registered under a unique name which distinguishes it from other callbacks of the same type. In order to unregister a callback, that name must be given as unique identifier.

Publish callbacks get passed as function arguments the information from the application routine invoking the Publish API: the value to be published, either as single scalar value entity or as a compound entity defined by a key/value table; a case-sensitive key to associate with that entity; an optional case-sensitive name to be attached to the entity; and an integer value to specify the importance level for the entity to be published). When available in the calling routine, a pointer to the current grid hierarchy structure should be passed by the user in the first argument (of type `CCTK_POINTER_TO_CONST` as a hint to the Publish callback function. Registered callback functions must not rely on the presence of such a hint provided by the user – if a NULL pointer value is passed instead, the callback should gracefully deal with this case (ie. not treat it as an error). Additionally, a `CCTK_POINTER` argument will be passed to each registered callback. This

argument is defined at registration time by the callback provider who can pass here a pointer to some user-defined data structure, to be used within the Publish callback function.

Preferably a register operation should be scheduled early in the process of simulation startup (eg. at `STARTUP` after `Driver_Startup`).

See Also

`Publish{Boolean,Int,Real,String,Table}()`
Publish API functions.

`Publish{Boolean,Int,Real,String,Table}_Unregister()`
Unregister Publish API callback functions.

Examples

C

```
#include <stdio.h>
#include <stdlib.h>

#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

/* the Publish logfile is open when registering callbacks */
static FILE* logfile = NULL;

/* define the Publish callback somewhere in your code */
static CCTK_INT PublishInt_ToStdout (CCTK_POINTER_TO_CONST cctkGH,
                                     CCTK_POINTER          cb_data,
                                     CCTK_INT              level,
                                     CCTK_INT              value,
                                     CCTK_STRING           key,
                                     CCTK_STRING           name)

  char* datatime = Util_CurrentDateTime ();
  fprintf (logfile, "%s", datetime);
  free (datetime);
  if (cctkGH)
  {
    fprintf (logfile, "[it=%d, time=%g]", cctkGH->cctk_iteration, cctkGH->cctk_time
  }
  fprintf (logfile, ": Publishing integer value %d with key '%s'\n", value, key);
  return (1);
}

/* this routine should be scheduled at simulation startup, eg. at CCTK_WRAGH */
void PublishToStdout_RegisterCallback (CCTK_ARGUMENTS)
{
  DECLARE_CCTK_PARAMETERS;

  if (CCTK_IsFunctionAliased ("PublishInt_Register"))
```

```
  {
    /* register only on processor 0 */
    if (CCTK_MyProc (cctkGH) == 0)
    {
      /* the logfilename parameter specifies the name for the Publish logfile */
      logfile = fopen (logfilename, "w");
      if (logfile)
      {
        PublishInt_Register (PublishInt_ToStdout, NULL, "Publish To Stdout");
      }
    }
  }
}
```

Fortran Since Publish callback functions have to process `CCTK_POINTER`, `CCTK_POINTER_TO_CONST` and `CCTK_STRING` arguments, it is unlikely that someone will code them in the Fortran language. Therefore no Fortran code example is given here.

`Publish{Boolean,Int,Real,String,Table}_Unregister`

Publish registry API: Unregister callback functions for the Publish API

Synopsis

```
CCTK_INT istatus = PublishBoolean_Unregister (CCTK_STRING name)

CCTK_INT istatus = PublishInt_Unregister     (CCTK_STRING name)

CCTK_INT istatus = PublishReal_Unregister    (CCTK_STRING name)

CCTK_INT istatus = PublishString_Unregister  (CCTK_STRING name)

CCTK_INT istatus = PublishTable_Unregister   (CCTK_STRING name)
```

Parameters

`name` a case-sensitive non-empty string uniquely identifying the callback to be unregistered

Result

`istatus` All unregister functions return 0 (zero) for success, or a negative value in case of an error.

Errors

`PUBLISH_ERROR_INVALID_CALLBACK_NAME`
: the `name` argument is a NULL pointer or points to an empty string

`PUBLISH_ERROR_CALLBACK_NOT_REGISTERED`
: no callback was registered under the given `name`

Discussion

Registered callback functions may need to be unregistered in order to safely shut down any underlying Publish services (eg. flush/close an open logfile or database, close the connection to external metadata information storage or publishing services such as a portal).

Preferably an unregister operation should be scheduled late in the process of simulation termination (eg. `at TERMINATE before Driver_Terminate`).

See Also

`Publish{Boolean,Int,Real,String,Table}()`
: Publish API functions.

`Publish{Boolean,Int,Real,String,Table}_Register()`
: Register Publish API callback functions.

Examples

C

```
#include "cctk.h"


if (CCTK_IsFunctionAliased ("PublishTable_Unregister"))
```

```
{
  PublishTable_Unregister ("Publish To Stdout");
}
```

Fortran

```
#include "cctk.h"

integer istatus

call CCTK_IsFunctionAliased (istatus, "PublishReal_Unregister")
if (istatus .ne. 0) then
  call PublishReal_Unregister ("Publish To Stdout")
end if
```