About This Document

This document is under construction. Author is David Standish. It should include all required documentaion for the Generation Wrapper which is a component of the Global Document Library (GDL) system. For description of terminnology used in this document refer to GDL Terminology.

Target Audience

The intended audience of this document are :

- GDL development team members
- GDL operation team members
- GDL deployment team members

Table of Contents

Chapter 1: Introduction

Global Document Library (GDL) is aimed to replace DVM, XDVM and EDAS (these three systems were responsible for generating, caching and distributing plot files from design files produced by various CAD tools including MCAD and ECAD which are submitted by users. Each system serves different user communities even though the main end results are similar. GDL does not merely consolidate the functionalities of DVM, XDVM and EDAS, it attempts to re-use the best features of the current systems and also it is designed from ground up with current industry standard technology to improve the performance, modularity, and ease of maintenance and support.

The Generation Wrapper, or wrapper for short, is one of the core services for GDL generations. Its task is to generate GDL derive file from newly submitted datasets detected by the master monitor and whose source datasets have been retrieved and are ready for generation.

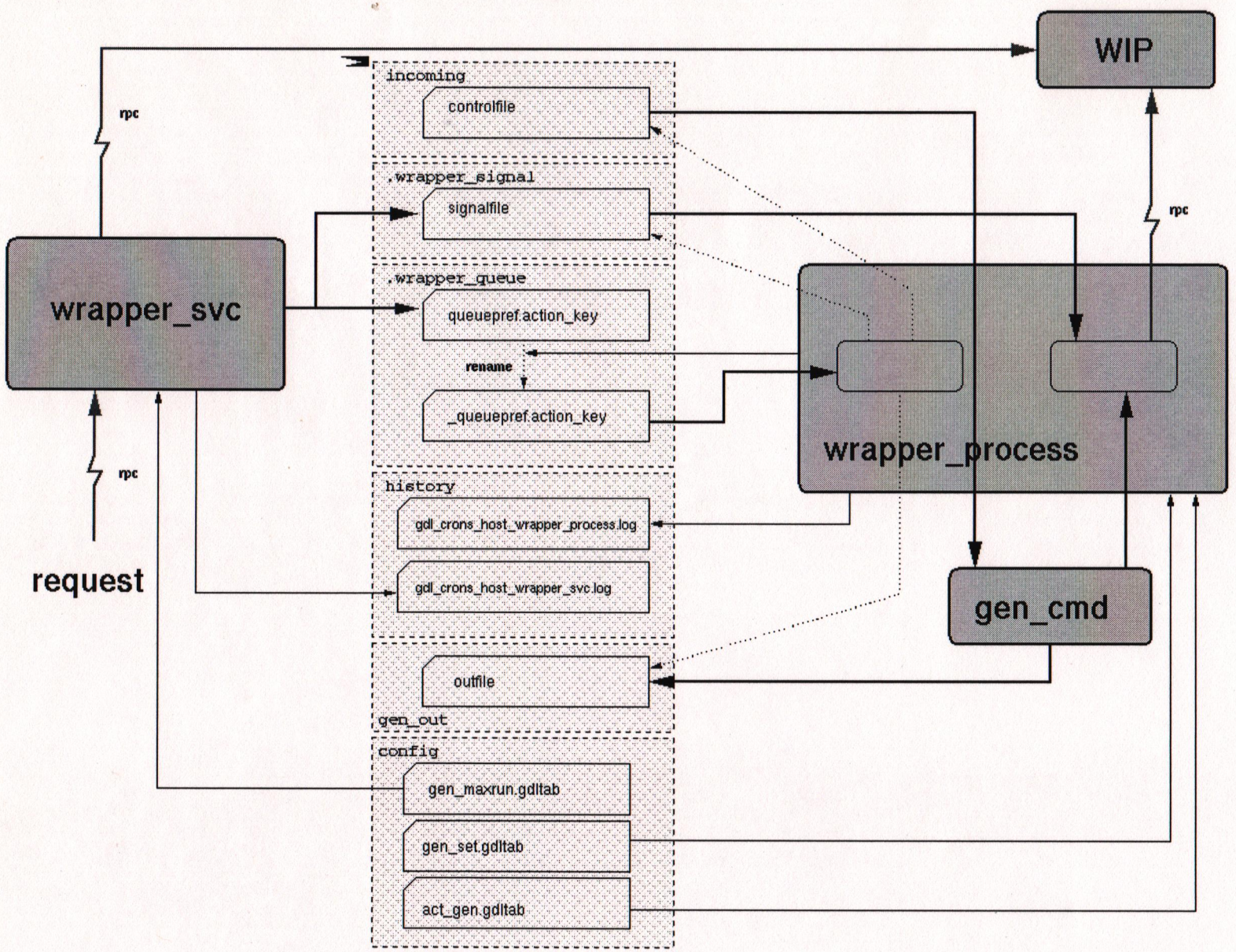
Chapter 2: Functional Requirements

1. The wrapper is responsible for generations requested on the local host and does not need to consider other hosts.
2. The Wrapper must queue generation upon receipt of a request via an RPC procedure call with one parameter, an stWrapperReq structure.
3. The szDsControlFile member of this structure is the name of the control file loacted in ~/incoming. This filename should consist of three parts separated with periods. The second part should be the unique ID identifying the generation. The third part should consist of the action_key of the base DDME dataset in lowercase plus the string "pack".
4. It must be able to queue generation requests independent of in progress generation requests.
5. It must determine how to generate the derives based on the dataset action key.
6. All generations must be invocable in the form "`gen_program packfile unique_id`"
7. This form must be invocable without prior interpretation from an interpreter (i.e. can launch via C exec()). A mapping from action key to gen_program is maintained on disk via configuration files.
8. The name of the *gen_program* invoked must be configurable.
9. The *gen_program* program is responsible for locating and disposing of the control file and any DDME source files provided, except in the case where the generation has been canceled.
10. The *gen_program* program shall write any log messages to standard output or standard error and shall not require any input from standard input.
11. If the wrapper is stopped and restarted no queued requests should be lost.
12. If shut down in an orderly manner it should complete any in progress generations before exiting. It should also refrain from starting any new generations.
13. More than one generation may be run simultaneously but the number of simultaneous runs using a particular tool may be limited. This may be because of load considierations, software licence limitations, or other. These limits should be settable via configuration files.
14. More than one action key may take part in a given limit as specified by the configuation files.
15. Latency between request and start of processing must be less than 5 minutes in the absence of ongoing generations.
16. It should not start generations if their WIP status is "canceled" but instead should update the WIP status to "Cancle complete" and invoke an external program to remove any retrieved datasets given the pack file.
17. When it starts the actual generation it should update the WIP status to `WIP_SENT2GENWRAP`.
18. It should update the WIP status upon successful or unsuccessful generation.
19. It should capture any output from the generation in a file in `~/gen_out` whose name is the name of the packfile, plus the extesion `.out`.
20. The overhead of queueing requests, launching, and monitoring generations should not consume a lot of CPU, disk, or IO resources.

Future enhancement: estimate machine load per action key or generation type so as to avoid exceeding machine capacity.

Chapter 3: Design

The general structure of the generation wrapper is shown in the figure below.



The generation wrapper is divided into two parts running as separate UNIX processes, *wrapper_svc* and *wrapper_process*.

wrapper_svc

The role of the *wrapper_svc* is to accept generation requests for jobs once their DDME source files are locally available. It is invoked via a remote procedure call (RPC). As per all RPC calls the service must be running and registered with the portmapper before it can be invoked. It is started as a daemon process via the `startserver` command and runs until stopped via `stopserver`.

When it receives a request, it stores the request data in a file in `~/.wrapper_signal`, named with the name of the control file (also called a packfile), suffixed with `.signal`. It also queues the request in the directory `~/.wrapper_queue` by storing the name of the control file in a queue file named after the current date and time and modified by the priority of the request and the weight configuration for the action_key in question and suffixed with a period followed by the action_key in uppercase.

In the present implementation the effectiveness of this strategy is doubtful as the resulting file names do not sort predictably due to insufficient zero padding. However, for a given action key the names will sort in the order they are queued assuming the requests have the same priority. (Note that currently all requests may ahve the same priority.)

wrapper_svc returns a "success" value to the requestor only after it has stored the request data and queued the request.

It also updates the WIP with the status WIP_IN_QUEUE_WRAP.

wrapper_svc Environment

Env Variable	Value	Description
gdl_HOME	$HOME	Server home directory
gdl_CONFIGDIR	$gdl_HOME/config	Location of configuration files

wrapper_svc Configuration Files

The wrapper_svc has only one configuration file, `$gdl_CONFIGDIR/gen_maxrun.gdltab`. This file is used to specify the value of *weight* for each supported action key. While originally envisaged to supply additional information, it has been supplanted by other configuration files used by wrapper_process.

`gen_maxrun.gdltab` is a gdltab format file. Unfortunately, wrapper_svc does not fully implement the gdltab format. As a result the field positions cannot be changed. From the viewpoint of the wrapper only the first and third fields, ActionKey and Weight, are used though the second field, MaxParallelRunAllowed, must be a positive integer. The ActionKey is the key and Weight is the value we want. wrapper_svc only reads this file once.

If the file is changed wrapper_svc must be stopped and restarted for the change to take effect. Below is a sample `gen_maxrun.gdltab`.

```
#########################################################################
# Name: $RCSfile: wrapper.html,v $
# This table is used by the generation server wrapper. The first field   #
# identifies the action key for a dataset(s) to be generated. The second #
# field specifies how many child processes can be spawned by the wrapper #
# (each child process runs a generation/derivation script for that parti #
# cular action key). The weight is used for the wrapper queue; the big-  #
# ger the weight, the quicker the dataset will be sent to the generation #
# script (if it's queued and for the same type of action key).           #
#
# Rcsid = $Id: wrapper.html,v 1.4 1999/05/10 22:38:14 daves Exp $
#
#f ActionKey | MaxParallelRunAllowed  | Weight | Comments
```

```
#  =========   ====================   ======   ========
#
   ARC         |   1                  |  1     | ARC file.
   PROP        |   1                  |  1     | ProEng Piece part.
   PE00        |   1                  |  1     | ProEng Piece part.
   PEAM        |   1                  |  1     | ProEng Assembly.
   CDRA        |   1                  |  1     | Cadra design file.
   MIF         |   1                  |  1     | MIF file (frame).
   FMP         |   1                  |  1     | FrameMaker Pack file.
   FM          |   1                  |  1     | FrameMaker file.
   FM00        |   1                  |  1     | FrameMaker file.
   DXF         |   1                  |  1     | AutoCAD design file.
   ACAD        |   1                  |  1     | AutoCAD design file.
   CDS         |   1                  |  1     | FCBDS (Unicad) design file.
   MSL         |   1                  |  1     | Manuf. Stock List.
   RCR         |   1                  |  1     | Report.
   PRHC        |   1                  |  1     | Report.
   EC          |   1                  |  1     | Engineering Change.
   NFAC        |   1                  |  1     | ICODAS' ARC (Unicad) file.
   NFPE        |   1                  |  1     | ICODAS' ProEng file.
   NFFM        |   1                  |  1     | ICODAS' FrameMaker file.
   NFRC        |   1                  |  1     | ICODAS' User Maintained report.
   NFSL        |   1                  |  1     | ICODAS' SAM Stocklist report.
```

wrapper_svc Initialization

The wrapper_svc does not do any initialization until it recieves its first request. At that time it loads the configuration information in gen_maxrun.gdltab and verifies that the following directories exist:

- `~/gen_out`
- `~/workdir`
- `~/.wrapper_signal`
- `~/Send2Caches`
- `~/librarydisk`

It also opens its error file, `~/history/wrapper.errlog` at this time. Error logging to this file is not fully implemented and does not conform to GDL standards. It should be removed in a future design.

A future enhancement would be to perform initialization at start-up.

wrapper_svc Queueing Formula

weigth
: as per gen_maxrun.gdltab (defulats to 1.0).

priority
: speicified with the request

timeofday
: time of day to microsecond resolution (from Jan 1, 1970)

queueval
: time based part of queue file name

```
queueval = timeofday * (priority/3) / (1000 * weight)
```

The name of the queue file is queueval zero padded to at least six digits with 7 decimals suffixed by a period followed by the action key in uppercase.

wrapper_process

wrapper_process is basically a job control program. It launches generation programs and montitors their exit status while limiting server load based on a simple model.

Generations are grouped into sets or classes. Each class has a configured limit which determines the maximum number of jobs that can be running simultaneously for that class.

wrapper_process does not impose any combined limits that would take into account the combined resource demands of all active generations.

Generation requests are mapped to classes based on their action keys.

wrapper_process works as follows.

1. It scans its queue directory ignoring hidden and in progress files. (In progress files are marked with a leading underscore in their file name. It also ignores bad files.) The queue directory should not contain anything but normal files, no directories.
2. For each file, in ascii order of name, it examines the action key suffix of the file name.
3. If the file name doesn't have an action key or the action key is unknown, then the file is considered bad and renamed to have a '_BAD_' prefix.
4. Next it checks if the generaion set to which the action key belongs is already running at its limit. If it is at its limit then the file is ignored until the next time the program scans the queue directory. Otherwise, wrapper_process services the request as follows:
    1. If it isn't at its limit the program marks the file as being in progress by prepending an underscore to the file name.
    2. Read the name of the control file in `~/incoming` from the queue file. Rename the queue file with _BAD_ prefix on error and go on to next queue file.
    3. Verify that: the control exists in `~/incoming`; that the unique id can be determined from the name of the control file; and that the signal file exists in `~/.wrapper_signal`. If any of these conditions are not satisfied rename the queue file with _BAD_ prefix and go on to next queue file.
    4. Read the signal file. Rename the queue file with _BAD_ prefix on error and go on to next queue file.
    5. Set first free job slot for corresponding generation set to this job.
    6. Note current time as start time for job (seconds since Jan 1, 1970).
    7. Check with the WIP if someone has requested that this job be canceled. If it has been canceled, then set the job status to canceled and run the `wrap_cancel` pseudo generation script to clean up any DDME source files that would normally be consumed by a real generation. Once `wrap_cancel` finishes wrapper_process contacts the WIP and lets it know that the cancelation is complete.
    8. Fork a new process. If fork fails the job is considered failed but the queue file is left in place for possible operator restart.
    9. Create output log file in `~/gen_out` named after control file plus ".out". If cannot create output file the job is considered failed but the queue file is left in place for possible operator restart.
    10. Exec the generation program using the gen_prog appropriate for the action key.

        ```
        gen_cmd control_file unique_id
        ```

        If the generation program could not be execed, perhaps due to a configuration or install error, the forked process exits with exit code 127. The job is considered failed but the queue file is left in place for possible operator restart.
5. Once it has checked all queued requests and started any it can, `wrapper_process` sleeps for approximately but not exactly 1 min.
6. Finally, `wrapper_process` polls the current active generation jobs to see if any have completed.
7. For any completed jobs `wrapper_process` performs the following tasks:

1. Set end time to current time.
2. Examine wait(2) status of completed job. The following conditions are possible:

exit code 127	special exit indicating the forked process could not exec the generation script. This is considered and abnormal exit.
died from signal	either an operator killed the generation script or the generation script failed catastrophicly due to a memory access violation or similar problem. This is considered and abnormal exit.
stopped	This rare condition could be caused by someone doing a KILL -STOP on the generation script or some other bizare condition. This is considered and abnormal exit.
exit code 0	Successful generation.
exit code not 0, not 127	Failed generation but not considered abnormal.

3. Unless `wrapper_process` was unable to run the generation script or some other abnormal exit occured, `wrapper_process` removes the queue file, the signal file, and if it still exists, the control file.
4. Otherwise the files are left as they are including the leading underscore in the name of the queue file. If the problem was due to some correctiable situation on the server an operator could conceivably requeue the qeneration by renaming the queue file to remove the leading underscore from its name.
5. Finally, `wrapper_process` updates the WIP with the result.

wrapper_process Environment

Env Variable	Value	Description
gdl_HOME	$HOME	Server home directory
gdl_CONFIGDIR	$gdl_HOME/config	Location of configuration files

wrapper_process Configuration Files

`wrapper_process` uses two configuration files:

- `$gdl_CONFIGDIR/gen_set.gdltab`
- `$gdl_CONFIGDIR/act_gen.gdltab`

`$gdl_CONFIGDIR/gen_set.gdltab` speicifies the maximum number of simultaneous running instances for each generation set or class. Like the `gen_maxrun.gdltab` used by wrapper_svc, wrapper_process does not support full gdltab functionality for this file. As a result the fields should not contain whitespace and field one must be gen_class while field two must be max_job. Below is a sample `gen_set.gdltab`.

```
#
# $Id: wrapper.html,v 1.4 1999/05/10 22:38:14 daves Exp $
#
#f gen_class | max_job
PROP|1
PEAM|1
UNICAD|1
FRAME|1
CDRA|1
```

`$gdl_CONFIGDIR/act_gen.gdltab` specifies for each action key the generation class to which the action key belongs plus the generation program that should be invoked to

process requests. Again wrapper_process does not support full gdltab functionality for this file. As a result the fields should not contain whitespace and field one must be action_key, field two must be gen_cmd, and field three must be gen_class. Note that case is significant, Aciont_key and gen_class values should be uppercase, Below is a sample `act_gen.gdltab`.

```
#
# $Id: wrapper.html,v 1.4 1999/05/10 22:38:14 daves Exp $
#
#f action_key | gen_cmd | gen_class
PROP|launch_generation_make|PROP
PEAM|launch_generation_make|PEAM
ARC|gdl_UNICAD_gen|UNICAD
CDS|gdl_UNICAD_gen|UNICAD
WIDF|gdl_UNICAD_gen|UNICAD
MIF|launch_generation_make|FRAME
CDRA|launch_generation_make|CDRA
```

wrapper_process Initialization

wrapper_process performs its initialization at start up. As an aid to debugging, it dumps the loaded configuration to standard output, which is normaly redirected to its logfile `~/history/gdl_crons_pbrwh00c_wrapper_process.log`. This would normally look something like:

```
*** BEGIN *** : wrapper_process  : 19990507 17:09:18
gen_set: ngen 5, nfree 5
gen[0]================
gen: gen_name 'PROP' max_job 1, njob 0
job[0]--------
gen_job: pid 0, gen_cmd '(null)', gen_state 0
job_status: status 0, canceled 0 could_run 0, ab_exit 0, signal 0, core 0, exit_code 0, begin 0, end 0, message 'not_started'
gen_request: null pointer
gen[1]================
gen: gen_name 'PEAM' max_job 1, njob 0
job[0]--------
gen_job: pid 0, gen_cmd '(null)', gen_state 0
job_status: status 0, canceled 0 could_run 0, ab_exit 0, signal 0, core 0, exit_code 0, begin 0, end 0, message 'not_started'
gen_request: null pointer
gen[2]================
gen: gen_name 'UNICAD' max_job 1, njob 0
job[0]--------
gen_job: pid 0, gen_cmd '(null)', gen_state 0
job_status: status 0, canceled 0 could_run 0, ab_exit 0, signal 0, core 0, exit_code 0, begin 0, end 0, message 'not_started'
gen_request: null pointer
gen[3]================
gen: gen_name 'FRAME' max_job 1, njob 0
job[0]--------
gen_job: pid 0, gen_cmd '(null)', gen_state 0
job_status: status 0, canceled 0 could_run 0, ab_exit 0, signal 0, core 0, exit_code 0, begin 0, end 0, message 'not_started'
gen_request: null pointer
gen[4]================
gen: gen_name 'CDRA' max_job 1, njob 0
job[0]--------
gen_job: pid 0, gen_cmd '(null)', gen_state 0
job_status: status 0, canceled 0 could_run 0, ab_exit 0, signal 0, core 0, exit_code 0, begin 0, end 0, message 'not_started'
gen_request: null pointer

act_gen_table: naction=7
        0 act_gen_entry: action_key='PROP' gen_name='PROP' gen_cmd='launch_generation_make'
        1 act_gen_entry: action_key='PEAM' gen_name='PEAM' gen_cmd='launch_generation_make'
```

```
2 act_gen_entry: action_key='ARC' gen_name='UNICAD' gen_cmd='gdl_UNICAD_gen'
3 act_gen_entry: action_key='CDS' gen_name='UNICAD' gen_cmd='gdl_UNICAD_gen'
4 act_gen_entry: action_key='WIDF' gen_name='UNICAD' gen_cmd='gdl_UNICAD_gen'
5 act_gen_entry: action_key='MIF' gen_name='FRAME' gen_cmd='launch_generation_make'
6 act_gen_entry: action_key='CDRA' gen_name='CDRA' gen_cmd='launch_generation_make'
```

wrapper_process directories and files

wrapper process requires the following directories:

$gdl_HOME/incoming	location of control files
$gdl_HOME/gen_out	where log files are created
$gdl_HOME/.wrapper_queue	location of queued requests
$gdl_HOME/.wrapper_signal	location of stored signal files

The queue file contents must be exactly the the name of the control file, no trailing blanks or newline, no leading path.

The stored signal file must be a serialized stWrapperReq structure. The libraries currently used to load the information from this file for the purposes of WIP updates is unstable if the file doesn't contain correct data in the correct format and as a result wrapper_process may crash.

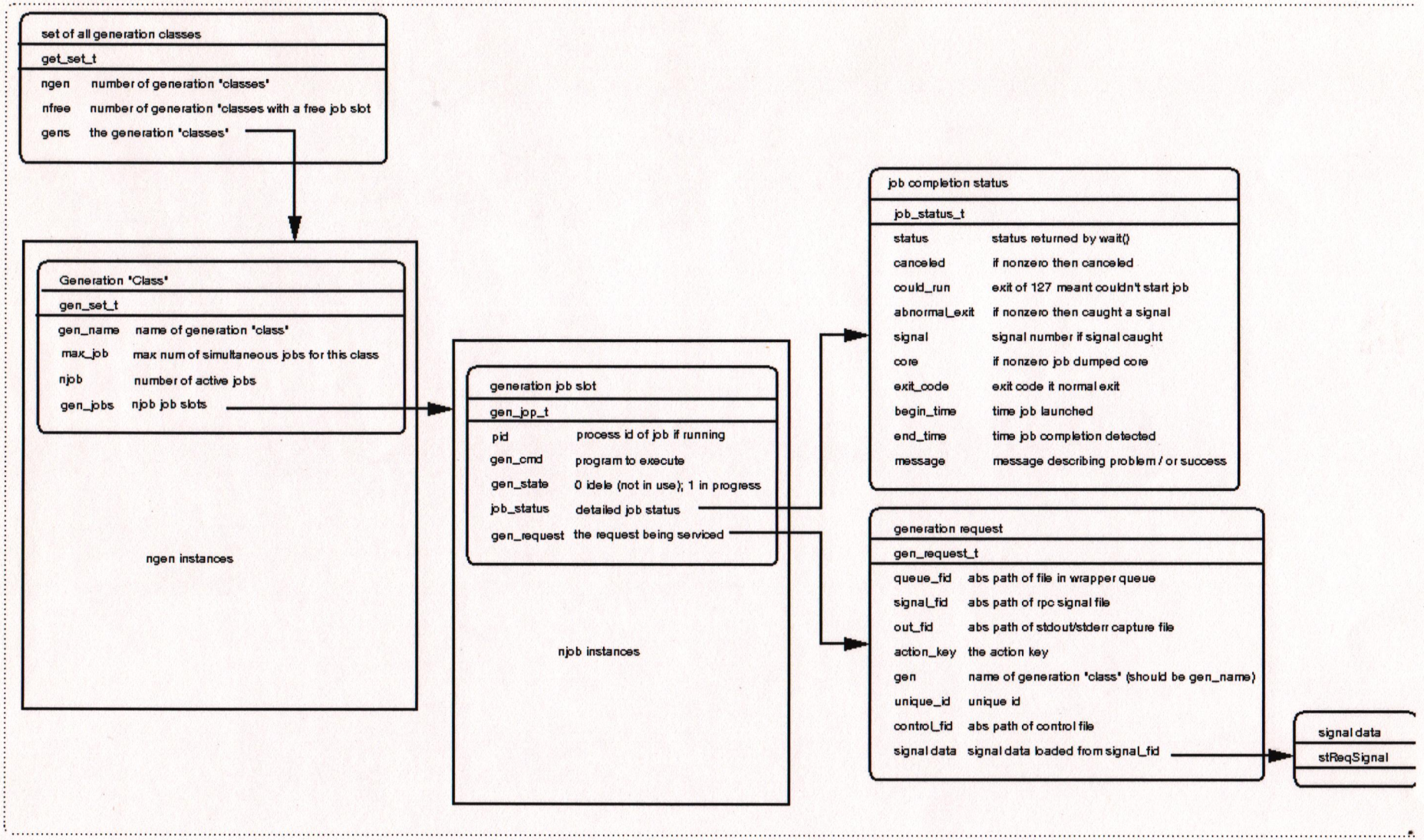
wrapper_process sample log

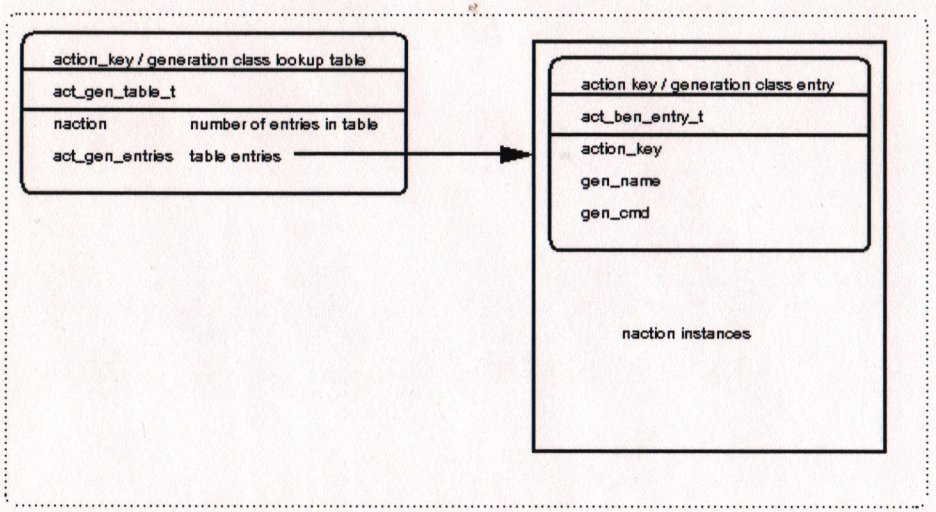
```
19990507 17:13:48 (I) wrapper_process:0928[20754]: Started '/u/gdlsrvt/incoming/ADAK11BC_T5X_CDRA+00+01.00001826.cdrapack' pid 22843
19990507 17:14:55 (I) wrapper_process:1192[20754]: *** cleanup_job stub *** pid 22843
job_status: status c00, canceled 0 could_run 1, ab_exit 0, signal 0, core 0, exit_code 12, begin 926097227, end 926097295, message 'failure'
gen_request: queue_fid '/u/gdlsrvt/.wrapper_queue/_617397.7746667.CDRA', signal_fid '/u/gdlsrvt/.wrapper_signal/ADAK11BC_T5X_CDRA+00+01.00001826.
19990507 17:14:55 (I) wrapper_process:0977[20754]: Completed '/u/gdlsrvt/incoming/ADAK11BC_T5X_CDRA+00+01.00001826.cdrapack' failure rc=12
```

wrapper_process shutdown

wrapper_process exits gracefully upon receipt of a SIGTERM (i.e. default kill signal - 15). It refrains from launching any new generations and finishes up any in progress generations before exiting. This means that the standard GDL stopserver will not cause any generations to be lost by wrapper_process.

wrapper_process data structures


set of all generation classes
get_set_t
ngen number of generation "classes"
nfree number of generation "classes with a free job slot
gens the generation "classes"
Generation "Class"
gen_set_t
gen_name name of generation "class"
max_job max num of simultaneous jobs for this class
njob number of active jobs
gen_jobs njob job slots
ngen instances
generation job slot
gen_jop_t
pid process id of job if running
gen_cmd program to execute
gen_state 0 idele (not in use); 1 in progress
job_status detailed job status
gen_request the request being serviced
njob instances
job completion status
job_status_t
status status returned by wait()
canceled if nonzero then canceled
could_run exit of 127 meant couldn't start job
abnormal_exit if nonzero then caught a signal
signal signal number if signal caught
core if nonzero job dumped core
exit_code exit code it normal exit
begin_time time job launched
end_time time job completion detected
message message describing problem / or success
generation request
gen_request_t
queue_fid abs path of file in wrapper queue
signal_fid abs path of rpc signal file
out_fid abs path of stdout/stderr capture file
action_key the action key
gen name of generation "class" (should be gen_name)
unique_id unique id
control_fid abs path of control file
signal data signal data loaded from signal_fid
signal data
stReqSignal


action_key / generation class lookup table
act_gen_table_t
naction
number of entries in table
act_gen_entries
table entries
action key / generation class entry
act_ben_entry_t
action_key
gen_name
gen_cmd
naction instances