The Process Manager in the ATLAS DAQ System

Giuseppe Avolio, Marc Dobson, Giovanna Lehmann Miotto, and Matthias Wiesmann

***Abstract*—This paper describes the Process Manager in the ATLAS DAQ system. The purpose of the Process Manager is to perform basic process control on behalf of the software components of the DAQ system. It is able to create, destroy and monitor the basic status (e.g., running, exited, killed) of software components on the DAQ workstations and front-end processors.**

Section I gives a brief overview of the Process Manager functionalities. Section II focuses on the requirements the Process Manager system has to fulfil to be fully integrated in the DAQ system. Section III shows how the requirements are met by the current implementation. The communication schema between the different parts of the Process Manager system, the procedure to launch a process and the possible states in which a process can be are described in Sections IV ,V and VI. Section VII deals with some consideration of the Process Manager performance while some conclusions are given in Section VIII.

***Index Terms*—Data acquisition, process control, process monitoring.**

I. Introduction

THE Process Manager (PM) [1] system is part of the distributed architecture upon which the ATLAS DAQ [2] system is built. It offers a service to manage processes in the DAQ context. The tasks of the Process Manager can be coarsely grouped in three categories:

- *Process creation*: It does not only imply the creation of the actual process, but also the preparation of all the resources and data needed to actually start the process. This includes setting of the process environment variables, resource acquisition and general checking.
- *Process control*: It includes mostly process termination. At a lower level this includes the sending of *POSIX* signals to the process. In case of process termination, resources need to be released and general cleanup operations[1] initiated.
- *Process monitoring*: It implies both giving state information on request (*pull* mode), but also dispatching *call-backs* to notify clients that a process has changed its status (*push* mode).

II. Requirements and Constraints

A. Independence

Since the Process Manager is responsible for launching and controlling almost all the servers that build up the general infrastructure of the system, it cannot rely on any service that it has to launch and terminate. Moreover a failure of the Process Manager system means the loss of control of the system (about 3000 machines and more than 15000 concurrent processes).

Because of this, the Process Manager should be autonomous and handle the failure of other services gracefully, be robust, offer some fault tolerance features and be easy to control using simple command line tools.

B. Functional Requirements

- *Process creation:* The Process Manager shall be able to start processes on behalf of the user, either on the user's local host or on a remote machine.
- *Process ownership*: a process created by the Process Manager will run on the host operating system under the identity of the user that initiates the launch request[2];
- *Process differentiation*: every process the Process Manager is requested to start, shall be assigned a unique identifier across all the previous and current process requests (i.e., the process *Handle*);
- *Process creation notification*: after a process creation attempt, success or failure shall be reported;
- *Process creation verification*: the user should have the capability to verify that the created process is running;
- *Process termination*: the user should have the capability to terminate a process launched by the Process Manager;
- *Process signaling*: the Process Manager shall be able to send signals to a created process on user request;
- *Process parameters*: the Process Manager shall provide the user application with the means to specify any relevant process creation parameters;
- *Process monitoring*: the Process Manager shall detect when a created process dies;
- *Process termination notification*: users shall be able to subscribe to information regarding the state of any managed process using the process *Handle*. The user shall also be notified if the process terminates;
- *HW and SW resources*: the Process Manager shall take into account any specified constraint on the use of hardware resources by software modules and any specified limit on the number of copies of a given application that can be run simultaneously. The resources available or used are specified in the DAQ configuration;
- *Access control*: requests to start a process, terminate it or to send signals to it shall be subject to access control;
- *Publishing information*: the Process Manager shall make publicly available information about the managed processes and the Process Manager system itself. The

Manuscript received May 2, 2007; revised August 9, 2007.

The authors are with CERN, CH-1211 Genève 23, Switzerland (e-mail: giuseppe.avolio@cern.ch).

Digital Object Identifier 10.1109/TNS.2007.910507

[1]All the files used to start and control the process (see Section IV) are removed. The published (see Section III.B) information regarding the process stop time, exit code and, eventually, the signal which caused the process to exit, is updated.

[2]This requires the user accounts to be distributed to all the machines in the cluster.

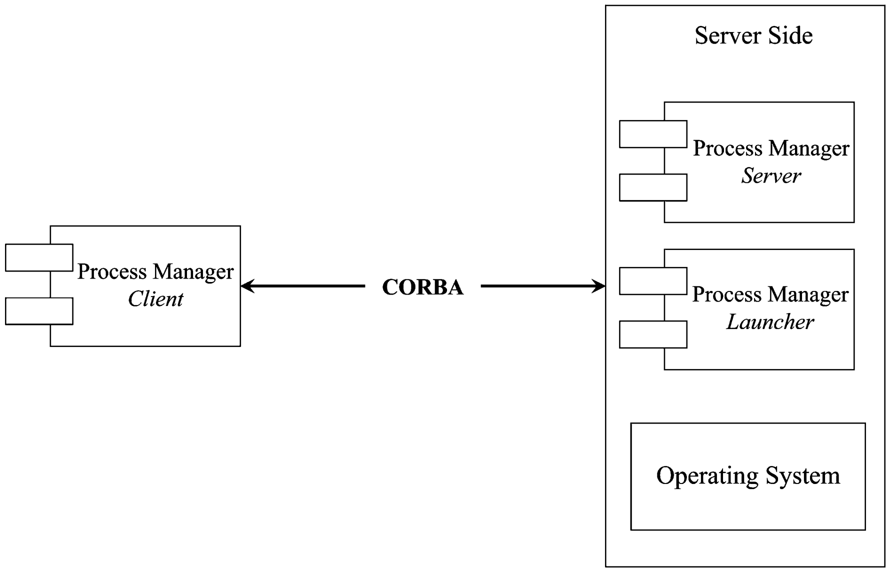

Fig. 1. The Process Manager general architecture. The *Client* and the *Server* communicate using *CORBA* [3].

published information shall be updated whenever a change occurs;
- *Error recovery*: if the Process Manager fails for any reason then it shall be able to inherit, when it is restarted, any alive process created by its previous instance. It shall be able to recover information about the running processes and manage them as if they were created by it;
- The Process Manager shall have no requirement for the kind of processes it is required to start and each managed process shall not be required to have any knowledge of the Process Manager system.

C. Operating System Constraints

The Process Manager system needs to interact with the underlying operating system to launch and control processes. As the Process Manager system is designed to run on *Linux* systems it has to adapt to constraints imposed by the system call semantics of the platform. In order to be able to start any process belonging to the DAQ infrastructure, the Process Manager needs to be started automatically and be available before any other DAQ service. In order to be able to run processes with any user identity, part of the Process Manager needs to have supervisor permissions (*super user*).

III. Implementation

The Process Manager architecture divides the system into three main components (Fig. 1): the Process Manager *Client*, the Process Manager *Server* and the Process Manager *Launcher*. Together the *Server* and the *Launcher* represent the *server part* of the Process Manager.

A. The Client Interface

The *Client* resides on the host where Process Manager requests are initiated and offers the user level interface to the Process Manager system. This interface gives tools to create and manipulate processes in the whole system and its functionalities can be grouped in two main categories: *Process Control* and *Process Information*.

1) Process Control: This category contains all methods that force the process to change its status (i.e., methods to request the termination of the process, or to send it a *POSIX* signal). These actions have strong authentication needs because the system should not allow any user to shutdown arbitrary processes. This requirement is fulfilled by the *Server* which checks the user authorizations before sending any signal to the process (see Section III.B).

2) Process Information: This category contains all methods requesting process information. It includes the following.
- General process information (process lifetime, standard output and error destination, the full path of the executable, parameters, environment variables, process id, process owner, host, working directory and *Handle*);
- Information reflecting the state of the process (typically if the process is running or has exited). This information is updated constantly and can be retrieved either by accessing local cached data or by registering a *call-back*;
- Information about usage statistics from the process (typically consumed resources on the host machine).

The *Client* interface defines two kinds of *Processes*:[3] *linked* and *unlinked*. A linked *Process* is tightly coupled with the *actual process* and user defined *call-back* functions (executed whenever the *process* changes its status) are associated with it. An unlinked *Process* has not such a direct coupling but access to the *actual process* information is granted using local cached data. One or the other mode can be used depending on the needs of the client application and the kind of a *Process* can be changed (i.e., a linked *Process* can be unlinked and vice versa).

Table I summarizes the allowed *Process* operations for different states (linked, unlinked and terminated) and actions. For the *Process Info* category the table specifies if the information

[3]*Process* refers to the *Client* classification, while *process* always identifiies the actual launched application.

TABLE I
ALLOWED PROCESS OPERATIONS

	Unlinked	**Linked**	**Terminated**
Control	Allowed	Allowed	Disallowed
Process Info	Allowed (local data)	Allowed (local data)	Allowed (local data)
Link	Allowed	Ignore	Disallowed
Unlink	Ignore	Allowed	Ignore

should be retrieved using local data or invoking remotely the server. As it can be noticed the *Client* makes an intensive use of caching to increase performance and minimize the number of queries to the Process Manager *Server* (see Section III.B).

B. The Server

Each host where processes need to be managed by the Process Manager runs one instance of the Process Manager *Server*. The *Server* acts as an information hub and dispatches request and stores important data structures. It is responsible of the following tasks.

- Manage process hierarchy (partition, application name, etc.);
- Manage *call-back* lists;
- Handle resource allocation using the Resource Manager (RM)[4] [4] service provided by the DAQ infrastructure;
- Handle user authorizations using the Access Manager (AM)[5] [5] service provided by the DAQ infrastructure;
- Interact with *Launchers* (see Section III.C);
- Interact with *Clients*;
- Publish information about itself and all the launched processes using the Information Service (IS)[6] [6] service provided by the DAQ infrastructure.

Because of its complexity provisions are taken to insure that the Process Manager *Server* can restart in case of crash. This implies storing state information in the file system. In case of any *Server* failure, its new instance uses this state information (contained in the *Manifest* file, see Section IV.A) to inherit already running processes.

One important Process Manager requirement concerns the properties of process references to be used for control and monitoring operations. The *Server* takes care of building process *Handles* which have the following properties.

- Each *Handle* uniquely describes one process in the system;
- The *Handle* exists as soon as the process creation is requested;
- The *Handle* is usable with a minimal number of lookups;
- In order to be usable with command line tools, the *Handle* is in text format.

[4]The Resource Manager service task is to marshal multiple access to DAQ limited resources to avoid conflicts.

[5]The Access Manager is an online software security service responsible for DAQ users authorization. It implements the ATLAS DAQ access policy in order to prevent corruption of the system functionality by actions performed by non-authorized persons.

[6]The ATLAS DAQ Information Service is generally used to share information between applications in a distributed environment.

The *Handle* format is built upon the structure of Uniform Resource Locators (*URL*) and its format is

$$pmg :: //server\_name/partition/application/id$$

where *pmg* is simply the protocol identifier, *server_name* is the fully qualified host name of the machine running the process, *partition* is the name of the partition the process is running in, *application* is the name of the application as defined by the *Client* and *id* is a unique identity number that is generated by the *Server* when the start of a process is requested. Within the scope of a given partition and given application name, those identities are monotonically increasing.

C. The Launcher

Each process managed by the Process Manager system is handled by a *Launcher*. This component is responsible for starting, monitoring and terminating a single process. The goal of the *Launcher* is to have one single component that handles low level process management. The decision to have the *Launcher* as a separate process is motivated by the following reasons.

- In order to launch processes under the identity of arbitrary users, the program that does the actual launching needs to run as *root*. Running a large *CORBA*-based [3] program as *root* would represent a large security risk.
- One requirement on the Process Manager system is to be able to restart the *Server* process in case of crash. Reattaching the *Server* to all launched processes would have been a complicated task, and would imply running the *Server* as *root* because only the *root* user can wait for an arbitrary process to exit.
- A small launcher component has less chances of crashing and offers a better abstraction level. Additionally, the *Launcher* is designed to be controlled using simple command line tools; this means that in case of problems some cleanup tasks can be done by hand if required.

Each *Launcher* is responsible for the following tasks:

- start one process with the correct parameters and permissions;
- monitor this process for termination;
- send control signals to the process;
- transmit process status information to the *Server*.

In order to accomplish its tasks the *Launcher* binary is installed in a well known place on the target machine, the binary is owned by *root* and has the *setuid* bit.

IV. COMMUNICATION SCHEMA

Fig. 2 illustrates the process view of the Process Manager system. The *Client* sends requests to the *Server* using *CORBA*. The *Server* and each *Launcher* communicate using a pair of named pipes[7] (one used by the *Server* to control the *Launcher*, the other one used by the *Launcher* to notify the *Server* of process state updates) and a memory mapped file (the *Manifest*). Each *Launcher* is the parent of one process and controls it using *POSIX* system calls. The *Server* interacts with other

[7]An operating system pseudo-file that offers first-in, first-out semantics.

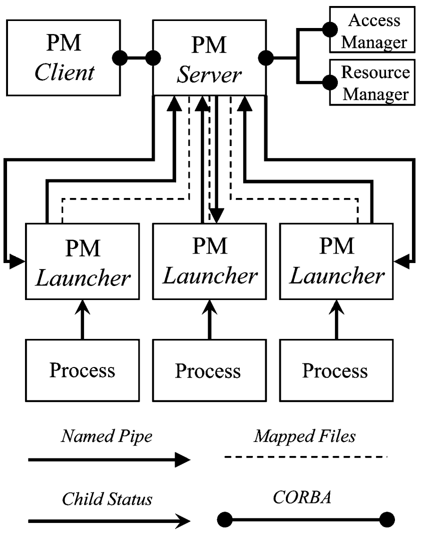

Fig. 2. The Process Manager communication schema.

server processes that implement different online services, like the Resource Manager and the Access Manager servers. Each instance of the *Launcher* contains two threads: one waiting for the child process to exit, the other one waiting for instructions from the *Server* with a blocking read on a named pipe. For each *Launcher* the *Server* has one thread doing a blocking read on a named pipe waiting for process status updates.

Manifest and named pipes data structures are resident in the local file system of the host machine.

A. The Manifest

The *Manifest* file is used both as a means of communication between the *Launcher* and the *Server* and to store information about a process (i.e., process and *Launcher PID*, binary name, process environment, *Handle*, status, etc.). Since the file is mapped into memory it allows the Process Manager system to share data between the *Launcher* and the *Server* in an efficient way. The information stored in the *Manifest* is used by the *Server* to reconnect to running *Launchers* after a failure.

V. Process Starting Procedure

In order to create a process, the Process Manager has to pass via a certain number of phases. Fig. 3 illustrates those phases:

- The *Server* receives a request from a *Client* to start a process;
- The right to launch the process is checked with the Access Manager;
- Available resources are acquired asking the Resource Manager;
- The *Manifest* is created and initialized;
- The *Server* builds the *Handle* for the process and sends it to the *Client*;
- The *Client* uses the received *Handle* to create its own data structures;
- If needed a *call-back* is registered for the process;
- The *Client* asks the *Server* to really start the process;
- The *Launcher* process is started.

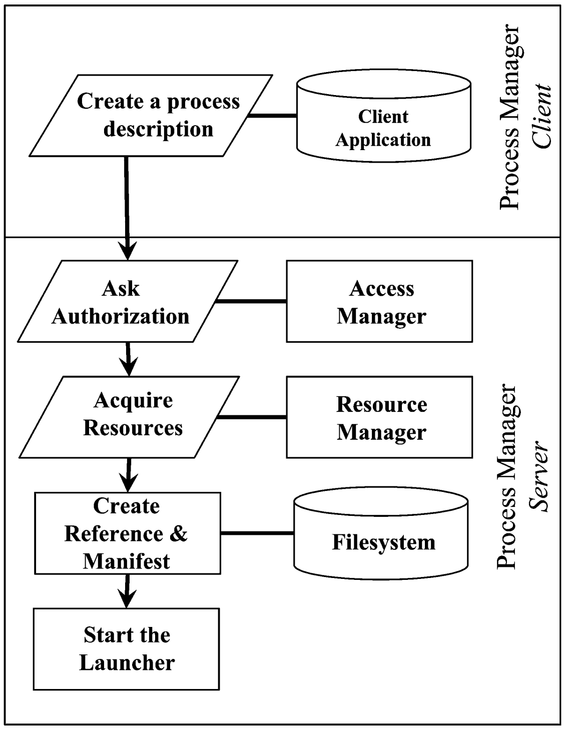

Fig. 3. Flow diagram showing all the phases to launch a process.

VI. Process States

A process is represented by the Process Manager as a state machine (Fig. 4). When a client application requests the start of a process the process is in the *REQUESTED* state. If the process is properly launched then it is in the *RUNNING* state, otherwise it goes into the *FAIL* state. Processes configured to notify the Process Manager system when they are actively running have a different path. They first go into the *CREATED* state. Once they confirm they are running, they go into the *RUNNING* state. If they fail to do so within a certain amount of time, they are first terminated and then go into the *SYNC ERROR* state. A running process can do two state changes. First it can terminate and go to the *EXITED* state. Some clients can also request termination for the process in which case the Process Manager will then send this process signals to terminate it. When a process terminates because of a signal (either an internal one, like segmentation fault, or an external one) it goes into the *SIGNALED* state. The Process Manager can terminate a process both by sending signals that the process can ignore and by sending signals that cannot be masked (like *KILL*). It is also possible to combine the former two actions: first sending a *TERM* signal, waiting for a defined amount of time for the process to exit and then eventually sending a *KILL* signal.

The states represented in Fig. 4 with a gray background are states that are reported to the client. All states on the bottom line (*FAIL*, *SIGNALED*, *EXITED* and *SYNC ERROR*) are "pit" states that a process cannot leave.

VII. Performance and Quality

The process management activity will mostly be high during state transitions. The design is done in such a way that no component needs to do polling. So while there will be many in-

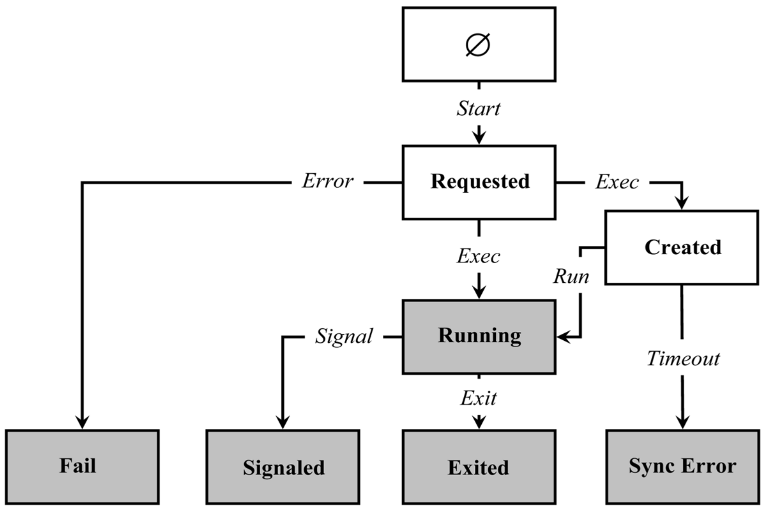

Fig. 4. The process state machine.

stances of the *Launcher*, they will all be blocked waiting for either their child process to terminate, or some command to be sent to their control named pipe. The *Manifest* file will stay small (32 KB) and data transfer between the *Server* and the *Launcher* will have a negligible impact on performances because of the shared memory area.

The Process Manager architecture is built to be able to tolerate the crash of certain elements of the system. If the *Server* crashes, it can be restarted. Upon restart, it reads the *Manifest* files and resumes processing. *Client* crashing have a small impact on the overall system. The only vulnerable element is the *Launcher* and for this reason it is designed to be small and simple in order to minimize the probability of bugs.[8]

A. Single Host Tests

In order to prove the efficiency and performances of the Process Manager design implementation some tests have been accomplished using a single host configuration and disabling all the connections to external services (i.e., AM and RM). Such a configuration is suitable for removing any bottleneck due to the latency of the network connection between the *Server* and the *Client* application and for decoupling the Process Manager from any other software module. Tests consist of a certain number of concurrent *Client* applications (up to 6) asking the local Process Manager *Server* to start and terminate very simple processes.[9] To pass the test the *Server* has to be able to deal with about 200000 start/kill steady requests without any failure. The prompt notification to the client applications of any change in the status of the controlled processes is mandatory. This kind of test has been successfully passed and an average time of 0.05 s to start a process, terminate it and dispatch all its status updates was measured.[10] No dependence on the number of already started processes has been noticed. A single *Server* did also succeed in managing a large number of concurrent processes (up to 50).

B. Distributed Tests

This category of tests includes the case of both single and multiple *Clients* sending requests to several *Servers* running on different hosts. The connections to all the external services are enabled. The aim of this test procedure is to check the Process Manager system capability to interact with all the DAQ infrastructure and its ability to deal with any external service failure. The Process Manager is requested to distinguish between the missing availability of a fundamental or accessory infrastructure component[11] and take opportune actions after having notified the client applications. Such tests were performed during some DAQ system technical runs and detector commissioning activities exploiting up to 100 different hosts. During all the test period the Process Manager was able to carry out its tasks in a proper way.

C. Recovery Tests

To simulate a Process Manager failure the *Server* is killed after launching a number of processes, causing the new *Server* instance to try and reattach to all the existing *Launchers*. This test has always been successful and the new *Server* has shown its ability to treat the running processes as if started by itself.

D. System Resource Usage

During all the tests the Process Manager has shown a small system resource usage. In *real life* operations the average CPU

[8]Anyway the Process Manager system is able to notify the user when, for any reason, the *Launcher* terminates while the launched process is still running. In this case some very low level utilities provided by the DAQ infrastructure can be used to avoid having dangling processes in the system.

[9]Here *simple* refers to processes linking a small number of shared libraries, so as to minimize their loading time when the process is launched. A basic test process just sleeping for a given amount of time was frequently used.

[10]Tests were hosted by a machine featuring two Intel Xeon 5150 CPUs (4 MB internal cache, 2.66 GHz clock and 1066 MHz system bus clock provided by the Intel 5000P chipset) running Scientific Linux CERN 4.5 (2.6.9 Linux kernel version).

[11]Is considered *fundamental* any service whose failure will not permit to launch a process (i.e., an *RM* unavailability could cause the wrong starting of a process whose used resources could conflict with other running processes). On the contrary an *IS* failure does not prevent a process to be safely started.

utilization is less than 15% when launching processes[12] and is virtually zero during the running period. The average memory usage is around 10 MB for the *Server* and less than 4 MB for the *Launcher*.

During the development phase some reliable memory checker tools (i.e., *valgrind* [7]) were used to detect memory leaks, accesses to uninitialised memory and misuse of allocated memory (e.g., double frees, access after free). All the tests in this area did not show any memory leak or corruption.

VIII. Conclusion

In this paper, the design, implementation and performances of the Process Manager for the ATLAS DAQ system have been presented. The aim of the software design has been to realize a component able to start, monitor and terminate processes without any restriction on the process type. Particular attention has been given to ensure a small usage of the host system resources in the monitoring phase, when all the DAQ infrastructure is running and engaged in data taking activities. This is crucial to avoid performance deterioration of the whole DAQ system (one *Server* instance runs on each host). The severe requirements to fulfil and guarantee stable and reliable operations are justified by the extreme and basic importance of the process management: loosing the control of the running processes will mean loosing the control of the system. All the performed tests have contributed to and helped verify the robustness of the current implementation of the Process Manager: even in case of failure the arranged procedures allow to recover the control of all the started processes.

[12]Value measured running one *Client* and one *Server* on two separte hosts (both machine with the same hardware and software configuration as for the *single host* tests). The client application continuosly asks the *Server* to start a simple process and receives status updates.

References

[1] M. Dobson, M. Wiesmann, and G. Lehmann, *Process Manager Requirements*, Feb. 7, 2004 [Online]. Available: https://edms.cern.ch/file/528906/2.4/Requirements2.4.doc

[2] "The atlas collaboration," ATLAS High Level Trigger, Data Acquisition and Controls Technical Design Rep. CERN-LHCC-2003-022, CERN,. Geneva, Switzerland, Jun. 2003, .

[3] *Object Management Group, Inc.*, CORBA/IIOP 3.0.3 Specification, Mar. 2004 [Online]. Available: http://www.omg.org/docs/formal/04-03-12.pdf

[4] I. Alexandrov, V. Kotov, and R. Roumiantsev, *User Guide and Implementation of the Resource Manager*, Jun. 21, 1999 [Online]. Available: http://atddoc.cern.ch/Atlas/postscript/Note130.ps

[5] J. E. Sloper *et al., Access Management in the ATLAS TDAQ*, Nov. 9, 2006 [Online]. Available: http://doc.cern.ch//archive/electronic/cern/others/atlnot/CONF/daq/daq-conf-2006-015.pdf

[6] S. Kolos, *Information Service User Guide*, Nov. 2005 [Online]. Available: http://atlas-onlsw.web.cern.ch/Atlas-onlsw/components/is/doc/userguide/is-usersguide.pdf

[7] "The Valgrind developers," The Valgrind User Manual. Jun. 2006 [Online]. Available: http://www.valgrind.org/docs/manual/manual.html