Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2007-27

Masterarbeit

im Studiengang "Angewandte Informatik"

Testing Grid Applications Using TTCN-3

Thomas Rings

am Institut für

Informatik

Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

11. September 2007

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 11. September 2007

Master's thesis

Testing Grid Applications Using TTCN-3

Thomas Rings

September 11, 2007

Supervised by
Dr. Helmut Neukirchen
Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Georg-August-University Göttingen

Abstract

The collective and coordinated usage of resources for joint problem solutions within dynamic virtual organizations across different institutes is realized with the Grid computing technology. Since the deployment of Grid computing grows, the quality of Grid computing environments and Grid applications has to be assured. An important constituent of quality assurance is testing.

This thesis investigates the applicability of the Testing and Test Control Notation version 3 (TTCN-3) for testing Grid applications. As case study of this thesis, test cases implemented in TTCN-3 have been applied to an application running in a Grid computing environment.

A description of the implemented example Grid application that is tested with TTCN-3 is covered by this thesis. The main focus of the thesis is on the realization of a basic TTCN-3 test system and its test harness in order to check the correctness of the application that runs in a Grid environment. The realized tests demonstrate that TTCN-3 is applicable for distributed testing in Grid environments. The Grid environment used in the case study is the Instant-Grid. It is based on the Grid middleware Globus Toolkit.

Contents

List of Figures

Listings

1 Introduction

Grid computing is an infrastructure that allows an integrated and joint usage of different autonomous resources, which may be located in different geographical locations. The concept of Grid computing has been introduced by Ian Foster in the end of the year 1998 [11].

The motivation that led to the establishment of Grid computing, or computer Grids, was the collective solving of problems within dynamic virtual organizations across different institutes [13]. This allows, after the determination of billing data and authorizations, direct collective access to computing power, applications, data, instruments and so on. A Virtual Organization (VO) in this context is a dynamic union of individuals and/or institutes that pursue common purposes with their established Grid. The focus of Grid computing is on tasks that run in a distributed manner. But the main goal is analog to the establishment of the Internet, the realization of a uniform, consistent and global computer Grid.

Since Grid computing is used in many different areas, for example in medical, physical, biological, or automotive industries, it has to be assured that the services provided by the Grid work in a proper way. Applications running inside a Grid computing environment have to trust the provided and used services of the Grid in order to perform their own tasks. These applications are typically very computing intensive. Applications running in a Grid environment are allowed to use applicable resources of the Grid in order to decrease their computational times. Typically, these applications are parallelizable in order to allow a breakdown of the main task into smaller tasks. These tasks use the resources of the Grid in a parallel manner. The determination, distribution and synchronization of the smaller tasks and even of different applications are typically done by a management software that includes a scheduler. All applications and their tasks have to be managed. If more than one application runs in the Grid, the management has to determine which resources in the Grid are the most applicable for the tasks of the applications. Each application has a sub-management that has to follow the main management. The main management can assign resources to the sub-managements, which themselves can assign these resources to the determined tasks of their managed application.

Grid computing is still in a beginning phase, but with recent developments, this technology gets more and more complex as the above example with the sub-managements shows. The quality assurance of Grid computing is an important role in its development. An important constituent of quality assurance is testing. Testing reveals defects through executing a System Under Test (SUT). The actual behavior is compared with the expected behavior that can be, for example, extracted from the specification of the software that has been tested. The computer Grid itself has to act as expected in order to be accepted by user communities.

The realization of functional and non-functional tests can be done with the Testing and Test Control Notation version 3 (TTCN-3), which is widely used for test specifications and test implementations for distributed systems. This standardized language is well supported by several vendor tools, that provide beside a TTCN-3 compiler also test harness development and execution environments.

1.1 Scope of this thesis

This thesis concentrates on a case study for TTCN-3 based testing of Grid applications. It is investigated if it is possible or applicable to test Grid computing applications and their Grid computing managements using TTCN-3.

In the used Grid only one application and its sub-management, in the following called management, is executed in order to keep the description straightforward. The application itself is parallelizable and, therefore, can be divided into smaller tasks by its management. Hereby, it has to be assured that the composed output of the smaller tasks is the same as the output of one main task. Therefore, as case study of this thesis, such an application and its management are implemented in order to realize test scenarios based on TTCN-3 in a Grid computing environment.

In this thesis two main test items have been determined:

1. Testing the application executed on a Grid node,
2. Testing the application management.

The first test item examines if a single task has been executed in the Grid computing environment as expected. The second test item checks if the management of the application determines and assigns the tasks to resources in the Grid correctly and composes the final result as expected. The test cases related to these test items are realized with TTCN-3.

1.2 Structure of this thesis

The structure of this thesis is as follows: After this introduction, in Chapter 2, the Grid computing technology and its concepts are described. The history and basics of Grid computing are explained, followed by an overview of Grid computing systems and a Grid computing architecture. This Chapter includes the description of the used Grid computing middleware Globus Toolkit 4 and the used Grid environment Instant-Grid. Furthermore, the gridification of an application is described.

Afterwards in Chapter 3, the foundations of software testing are explained. This includes testing fundamentals and the test process. Dynamic testing is explained in particular.

Following is a description of TTCN-3 in Chapter 4 that introduces main concepts of this software testing specification language. This includes the realization of a distributed test system architecture based on TTCN-3.

Chapter 5 describes the development of an example Grid application and its management that are used for the case study of this thesis. The example application is the calculation of the Mandelbrot set, which is executed distributedly by its management.

Chapter 6 describes how this example Grid application can be tested using TTCN-3. First of all, test purposes, test architectures and test behaviors are explained. Afterwards, the TTCN-3 specifications and the test adapter implementations are described. This chapter also covers a description of the executions of the implemented tests.

Finally, Chapter 7 concludes with a summary, an outlook and a discussion of related work. The Master's thesis is completed by a list of acronyms and the bibliography.

2 Grid computing

A Grid computing environment that is also known as a computer Grid is created with the purpose of providing users easy access to a collection of heterogeneous computers and resources. These are usually spread across multiple administrative domains [21]. This Chapter describes Grid computing in detail: Section 2.1 gives an insight of the historical origin. In Section 2.2, the basics of Grid computing are described, followed by an overview of the different kinds of Grid systems in Section 2.3. A description of a common Grid architecture is given in Section 2.4. The focus of Section 2.5 is the Globus Toolkit 4 (GT4). Globus Toolkit 4 is a common and open source middleware for Grid technologies and provides communication and security services. This Grid middleware is used by Instant-Grid that is described in Section 2.6. Instant-Grid provides a computer Grid environment and is a primary base point of the case study used in this thesis. General instructions about the integration of an application into a computer Grid are given in Section 2.7.

2.1 The history of Grid computing

The term "Grid" has been adapted from electric power grids, since a computer Grid follows the same concept. In 1910, electric power generation became possible and everybody could use a generator to produce electricity [83]. By this time, the electric power grid has been developed and changed the situation for associated transmission and distribution technologies of power. These developments had made reliable low-cost access and standardized services possible. As a result, they have provided universally accessible power [3].

For performing computationally intensive and complex tasks, a Grid computing environment provides an efficient sharing and management of computing resources. Since the early 1990s, Grid computing has emerged in the scientific and defense community. However, in the late 1990s the drivers behind the Grid technology were mostly businesses with a limited fund of supply, which needed powerful, inexpensive, flexible computational power. Also university and research facilities had developed more and more computational intensive applications, which needed the processing power of a supercomputer. A big problem was that many users could not afford a supercomputer. The solution to this problem was to build a computational environment that uses the computational power of every connected computer, e.g. on the whole campus or enterprise, to execute complex calculations [36].

2.2 The basics of Grid computing

Grids are a newer technology with huge developing potential. That is the reason a discussion started about what exactly a computer Grid is [36]. Ian Foster suggests a three point checklist for the definition of a Grid:
"A Grid is a system:

1. that coordinates resources that are not subject to centralized control...
2. ... using standard, open, general-purpose protocols and interfaces...
3. ... to deliver nontrivial qualities of service." [8]

The three point checklist signifies that a Grid integrates and coordinates resources and several users living within different control domains, where the issues of security, policy, payment, membership and so forth are solved by the Grid. Related to these issues, resources can be heterogeneous and include clusters, mass storages, databases, applications and sensors. The communication and data exchange between these domains are built from multi-purpose protocols and interfaces that allow the handling of such fundamental issues as authentication, authorization, resource discovery and resource access.

It is important that standard and open protocols and interfaces are used. The reason is the need of establishing resource-sharing arrangements dynamically with any interested party. Standards are also relevant for the use of general-purpose services and tools development. A Grid has to be clearly distinguished from local managements and application specific systems.

To obtain various qualities of service, which are measured with, for example response time, throughput, availability and security, a Grid has to allow its constituent resources to be used in a coordinated way to meet complex user demands. Therefore, the utility of the combined system has to be significantly greater than the utility of the sum of its parts [8].

The definition of Ian Foster is helpful in classifying a Grid compared to other technologies, like distributed computing, cluster computing, peer-to-peer computing, or meta computing, which are similar to Grids. A more convenient description of a Grid can be found in [11]:

> "A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high end computational capabilities."

In this description, the Grid is referred to as an **infrastructure**, because it is related to a large-scale pooling of resources, whether to reckon cycles, data, sensors, or people. A pooling of this caliber needs an outstanding hardware infrastructure to keep all the necessary interconnections up-to-date. A complex software infrastructure is required for checking and controlling all interacting applications and their results [11].

Providing a **dependable** service is fundamental, because users need to be assured that they will receive expected and persistent performance to generate outputs from a diversity of components, of which the Grid consists. If such conditions will not be ensured, applications should not be executed, written, or used inside the Grid. Due to different applications, the performance characteristics, which indicate the efficiency of a system, are different, depending on the components and the submitted jobs. The performance characteristics may also include network bandwidth, latency, jitter, computer power, software services, security and reliability to ensure a fail-safe computation of jobs in the Grid [11].

The second critical concern is the need for **consistency** of service, like standard services, accessible via a standard interface and operating with standard parameters. Since the Grid consists of heterogeneous resources, without designing standardized applications and pervasive use of these, a Grid computing environment is inefficient [11].

The infrastructure should provide a **pervasive** and universal access in order to make services available that are independent of the environment it is being used in. The access fee to the Grid should be offered at a reasonable price (**inexpensive**) if it is needed to capture the greatest market share [11].

However, dependability, consistency and pervasiveness are important reasons for creating the Grid to get a transforming effect on how computation is performed and used [11].

By enhancing the set of capabilities and ensuring it to the resources and users of the Grid, Grids allow new tools to be developed and widely deployed. Increasing the number of capabilities with **transparency** and **utility** would lead to a more user friendly system. If the Grid is transparent, users do not have to take care where their jobs will be executed or where temporary data is being stored. The management of the middleware has to assign resources, which are best suited to perform the submitted task and has to locate and retrieve the output data. Utility assures that demanded computer power or storage capacity is being allocated efficiently for the submitted task, like electricity from the outlet [54].

The base development of application interfaces needs to be modified in order to allow pervasive access to applications in the applied Grid. The change is required, because most of today's applications are designed for single computing and are not suitable for a computer Grid. Furthermore, a Grid has to provide an environment that allows the use of different resources that are seen as one resource by the applications. Therefore, computer Grids can change our thinking about computation and resources constitutively [11].

2.3 Grid systems

A Grid system can be distinguished in a computational, data, or service Grid as shown in Figure 2.1. A computational Grid system has a significant higher combined computational capacity, which is available for single applications, than the capacity of any resource in this Grid. Depending on the capacity utilization, such systems can be differentiated into distributed supercomputing and high throughput. Whereas distributed supercomputing Grids minimize the completion time of jobs by executing these in a parallel manner on several resources, high throughput Grids increase the

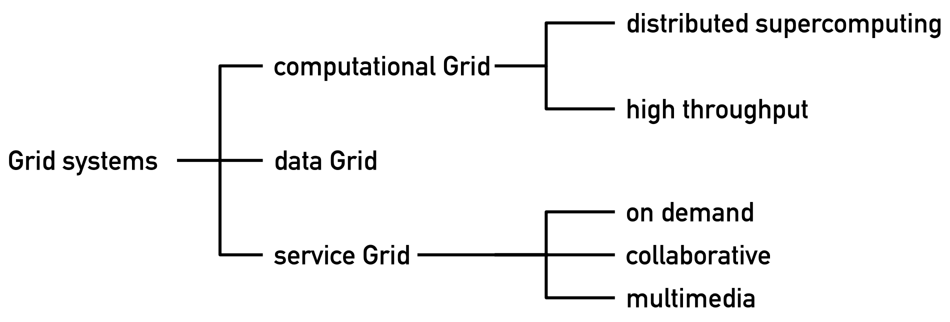


Figure 2.1: A Grid system's taxonomy [24]

completion rate of a stream of jobs. Distributed supercomputing Grids are mainly required by applications for complex computational intensive analysis like climate modeling, astronomy applications and nuclear simulations, whilst jobs that can be divided into independent jobs through 'parameter swap' mainly run on high throughput Grids as for example Monte Carlo simulations [25] or SETI@Home [79].

Data Grid systems provide an infrastructure in order to allow a merged view of already existing data, which are distributed in a wide area network. These are made available as new information, such as digital libraries or data warehouses.

A computational Grid has to offer services for data operations to the applications, too. The main difference to a data Grid are the infrastructure and services, which are provided to the applications for storage and data management. In a computational Grid, the applications use their own implemented storage management schemes, whereas in a data Grid, applications use these services as provided by the Grid. Several data Grid initiatives, such as the European DataGrid Project [78] and Globus [42], design large scale data organizations, catalog, management and access technologies.

Service Grids provide services that are not provided by any single resource. They can be subdivided in on-demand, collaborative and multimedia Grid systems. An on-demand Grid provides new services by aggregating several resources dynamically to satisfy short term requirements. A collaborative computing Grid provides a shared workspace for enabling real time interactions between users and applications in a collaborative workgroup. A multimedia Grid makes an infrastructure for real-time multimedia applications available.

Most of the present research in developing Grid systems can be classified in one of the above categories. There still remains the challenge to develop a truly general-purpose Grid system, which can support multiple or even all of these categories [24].

Today, it is not yet possible to convert every application for allowing parallel runs on a Grid. There is only the possibility to use parallelizable applications on a Grid. Practical tools for transforming arbitrary applications to exploit the parallel capabilities of a Grid are not available, yet. Automated transformation of applications is a science in its infancy, since there are practical tools that can only be used by skilled developers for writing parallel Grid applications [3].

2.4 Grid architecture

After about one decade of intensive research, development and experiments, a general specification of the architecture for computer Grid technologies has been defined. Additionally, open and standard protocols for communication and controlling have been developed in order to create a foundation for further interoperability that Grid systems depend on. Another important issue is the definition of standard interfaces to provide libraries through standard Application Programming Interfaces (API) and ease the construction of Grid components by allowing code components to be reused [3].

According to the role in a Grid system, protocols and APIs can be categorized schematically as shown in Figure 2.2. A Grid architecture can be seen as several constitutive layers with different width as in the “hourglass model”. Components of a layer have similar characteristics and normally are dependent on abilities and behavior of deeper layers. A small set of core abstractions and protocols, e.g. TCP and HTTP on the Internet, are defined in the narrow neck of the hourglass. Onto this set many high-level behaviors can be mapped (the top of the hourglass) and the set itself can be mapped on several underlying technologies (the bottom of the hourglass). By definition, the number of protocols defined at the neck must be small [7,13]. Each layer is described in the following[1].

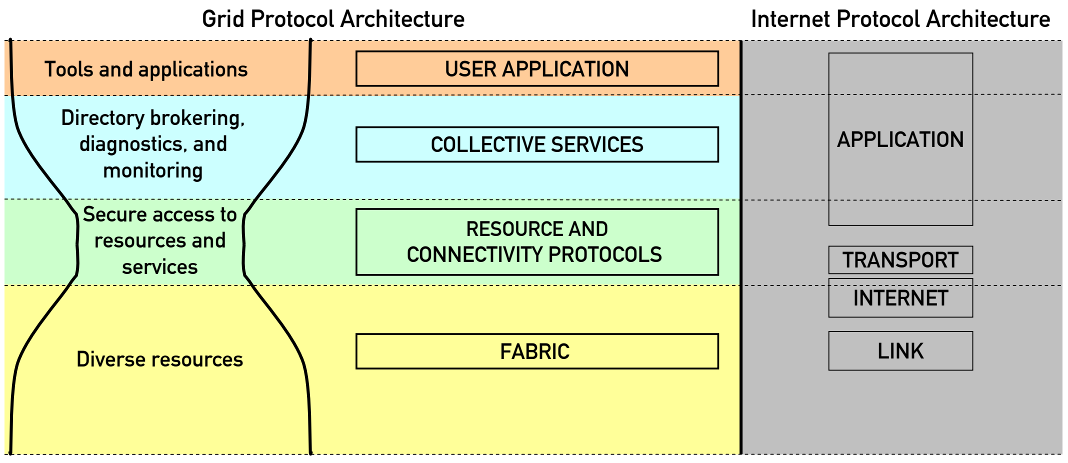

Grid Protocol Architecture
Internet Protocol Architecture
Tools and applications
Directory brokering, diagnostics, and monitoring
Secure access to resources and services
Diverse resources
USER APPLICATION
COLLECTIVE SERVICES
RESOURCE AND CONNECTIVITY PROTOCOLS
FABRIC
APPLICATION
TRANSPORT
INTERNET
LINK

Figure 2.2: Architecture concept of a Grid (based on [11] and [13])

Fabric: The lowest level provides a basis as a common interface for all kinds of physical devices or resources. Higher layers access the fabric layer through standardized methods. All resources, to which this uniform layer is applicable, can be integrated in the Grid concept. The resources can include computers, storage systems, networks and various forms of sensors [11,13].

Resource and connectivity protocols: The resource and connectivity protocols layer is located above the fabric level and defines the core communication and

[1]A more detailed description about Grid architecture is given by Foster, Kesselman and Tuecke [11].

authentication protocols, which are required by the Grid. The communication protocols allow the transmission of data between different resources, connected through the fabric layer whereas the authentication protocols have to verify the identity of users and resources by providing communication services with cryptographically secure mechanisms. The delegation of authorizations and methods for a single sign-on are also essential for this layer. In the resource protocols, the common access to several resources is organized in order to enable secure initiation, monitoring and control of resource-sharing operations, like assignment or reservation [3].

The Open Grid Service Architecture (OGSA) [14] is used in many Grid projects for the realization of this layer [30]. The Globus Toolkit 4 (GT4), which is explained in Section 2.5, is a common implementation of the OGSA specification and provides software services and libraries in order to realize an OGSA as resource and connectivity layer [11, 13, 14].

Collective services: The role of this layer is the coordination between different resources. Access to these resources does not happen directly, but merely by the underlying protocols and interfaces. Because this layer analyzes and consolidates components from the relatively narrow resource and connectivity layers, the components of the collective layers can implement a wide variety of tasks without requiring new resource-layer components. Responsibilities of the collective services layer include directory and brokering services for discovery, allocation, monitoring and diagnostic of resources. Furthermore, this layer provides Data Replication Services (DRS) as well as membership and policy services to observe and handle the accreditation of members to access resources [3, 11, 13].

User applications: In this layer, all user applications are implemented and supported from the components of the other layers. The user applications can call services from lower layers and use resources transparently [11, 13].

2.5 Globus Toolkit

Globus Toolkit 4 (GT4) of the Globus Alliance [42] is a middleware for Grid systems and provides all required components for a realization of a computer Grid, as a community-based, open architecture, open-source set of services and software libraries. These components include software for security, information infrastructure, resource management, data management, communication, fault detection and portability. It is packaged as a set of components that can be used either independently or together to develop applications [12].

GT4 defines protocols as well as APIs for each component. In addition, it provides open-source reference implementations in C and Java (for client-side APIs). In terms of these basic components, a wide variety of higher-level services, tools and applications have been developed. Several of these services and tools are integrated in the components of the GT4, while others are distributed through other sources [10].

2.5.1 Architecture

Ian Foster [10] describes the Globus Toolkit architecture with various aspects. A schematic view of GT4.0 components is depicted in Figure 2.3. The components of GT4.0 can be divided in three sets as described in the following.

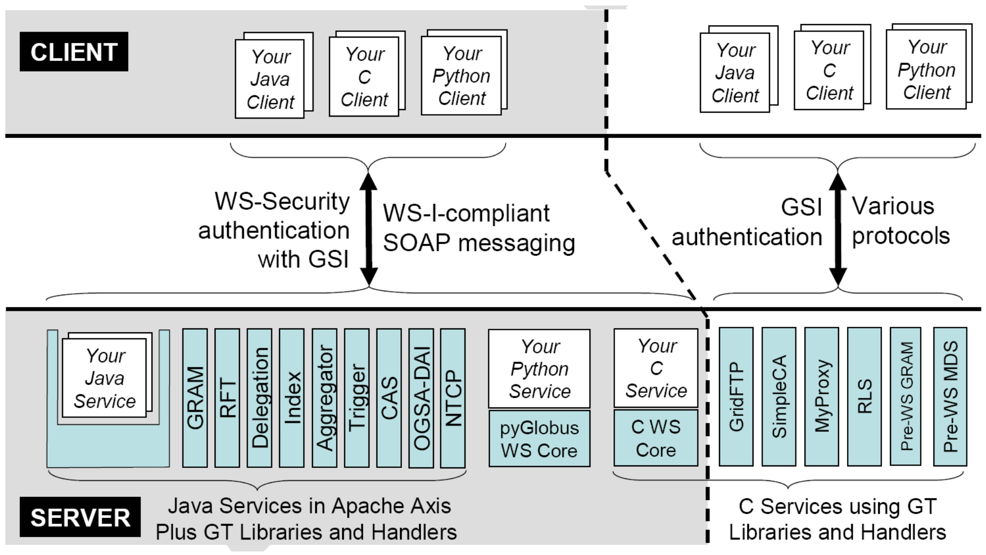


Figure 2.3: Schematic view of GT4.0 components [9]

- At the bottom half of the figure a set of **service implementations** implements infrastructure services. These services address such concerns as execution management (GRAM), data access and movement (GridFTP, RFT, OGSA-DAI), replica management (RLS, DRS), monitoring and discovery (Index, Trigger, WebMDS), credential management (MyProxy, Delegation, SimpleCA) and instrument management (NTCP) [37]. Whereas most of them are Java Web Services, the others on the bottom right are implemented in other languages and/or use other protocols [9, 10].
- User developed services written in Java, Python and C are hosted respectively in one of the three **containers**, which provide implementations of security, management, discovery, state management and other mechanisms frequently required when building services [9, 10].
- Invocation operations of GT4 and user-developed services are possible with client programs in Java, C and Python that are provided by a set of **client libraries** [9, 10].

GT4 provides components for Web Services (WS), which are depicted on the left of Figure 2.3 and for non-WS on the right of Figure 2.3 [10]. Another perspective on the

GT4 structure is provided in Figure 2.4 showing the components provided for security, data management, execution management, information services and basic runtime.

Security tools are responsible for authentication, which means creating identities of users or services, protecting communication and authorization that is the regulation of actions that a user is allowed to perform. Supporting functions such as managing user credentials and maintaining group membership information are provided, too [9].

Data Management tools allow the location, transfer and management of distributed data. Various basic tools, e.g. GridFTP for high-performance and reliable data transport, RFT for managing multiple transfers, RLS for maintaining location information for replicated files and OGSA-DAI for accessing integrated structured and semi structured data, are provided by the GT4 [9].

Execution Management tools are responsible for initiation, monitoring, management, scheduling and/or coordination of remote computations. GT4 supports the Grid Resource Allocation and Management (GRAM) interface as a basic mechanism for these purposes. The GT4 GRAM server is typically deployed in association with delegation and RFT servers to handle data staging, delegation of proxy credentials and computation monitoring and management in an integrated manner [9].

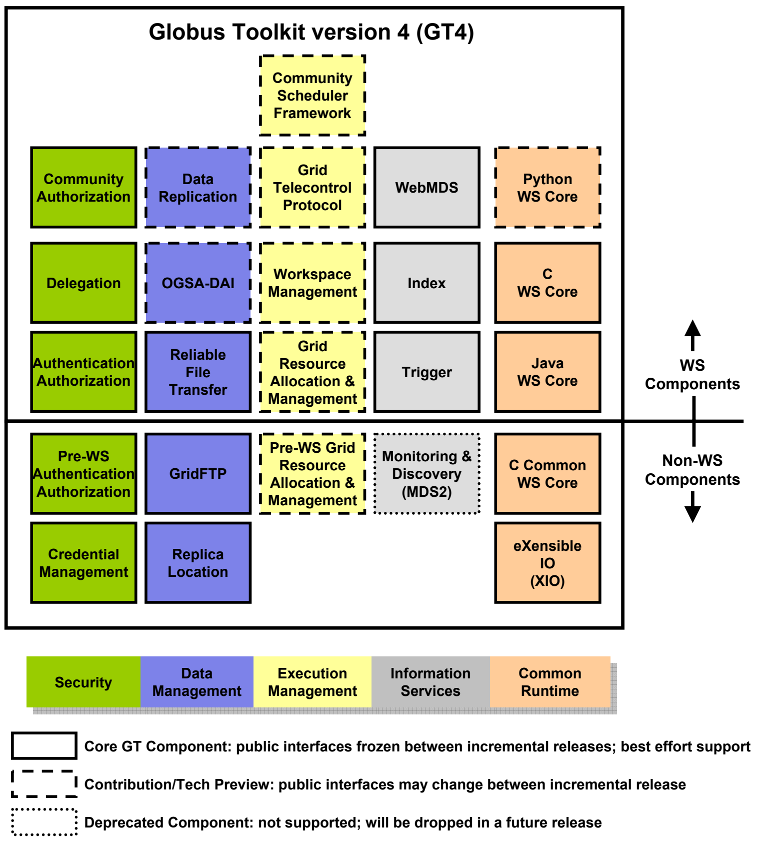

Globus Toolkit version 4 (GT4)
Community Scheduler Framework
Community Authorization
Data Replication
Grid Telecontrol Protocol
WebMDS
Python WS Core
Delegation
OGSA-DAI
Workspace Management
Index
C WS Core
Authentication Authorization
Reliable File Transfer
Grid Resource Allocation & Management
Trigger
Java WS Core
WS Components
Non-WS Components
Pre-WS Authentication Authorization
GridFTP
Pre-WS Grid Resource Allocation & Management
Monitoring & Discovery (MDS2)
C Common WS Core
Credential Management
Replica Location
eXensible IO (XIO)
Security
Data Management
Execution Management
Information Services
Common Runtime
Core GT Component: public interfaces frozen between incremental releases; best effort support
Contribution/Tech Preview: public interfaces may change between incremental release
Deprecated Component: not supported; will be dropped in a future release

Figure 2.4: Globus Toolkit 4 components [43]

Information Services provide monitoring and discovery mechanisms that are responsible for obtaining, distributing, indexing, archiving and handling information about the configuration and status of services and resources. The motivation for collecting this kind of information is on one hand the discovery of services or resources and on the other, the monitoring of the system status [9].

The **Common Runtime** provides a basis for platform independence of the services named and explained above. GT4 provides libraries to facilitate the development of abstract layers and support a comfortable realization of functionalities on the base of WS [9].

The installation of GT4 requires a special expertise and needs an expenditure of time due to a high measure of complexity. A detailed description of the installation and configuration progress can be found in [45, 67].

2.5.2 Alternative Grid middleware systems

Several Grid middleware technologies have been analyzed by Buyya et al. [1]. The functionality, differences and similarities between chosen Grid middleware systems including UNICORE [82], Gridbus toolkit [81] and Legion [65] are briefly described in the following.

UNICORE is a multi-tiered Grid computing system implemented in Java. It focuses on high level programming models providing a seamless and secure access to distributed resources. UNICORE is an open source implementation and is used in several projects including EUROGRID [50], OpenMolGrid [73] and Grid Interoperability Project (GRIP) [77].

The open source Gridbus toolkit extensively leverages related software technologies. It provides an abstraction layer to hide idiosyncrasies of heterogeneous resources and low level middleware technologies from application development. The focus is on the implementation of utility computing models including clusters, Grids and peer-to-peer computing systems. The Gridbus toolkit is included in projects like ePhysics Portal [2] [72] and Belle Analysis Data Grid [59].

Legion is an object-based middleware system. It helps to combine heterogeneous domains, storage systems, database legacy codes and user objects distributed over wide-area-networks into a single, object-based metacomputer. This metacomputer provides high degrees of flexibility and site autonomy. Legion is implemented in projects like the NPACI Testbed [71]. Ultimately Buyya et al. describes that the analyzed Grid middleware technologies all have similar functionalities.

2.6 Instant-Grid

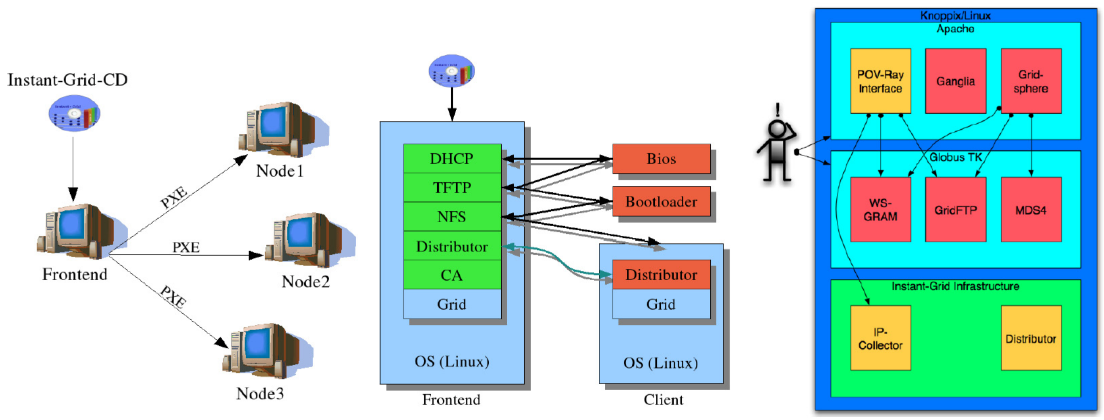
Instant-Grid [60] is a project of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) and is financed by the German Federal Ministry of Research and Technology (BMFT). Instant-Grid is mainly developed by the GWDG, but also supported by partners like ed-media, the FernUniversität of Hagen, FIZ Chemie and Fraunhofer FIRST.

The main purpose of Instant-Grid is to provide a computer Grid environment for users without special expertise in Grid technologies. It includes an automated configuration mechanism, which installs the Instant-Grid on common local connected computers without user interaction, to address a wide variety of users. Instant-Grid demonstrates basic Grid technologies and makes a development of user-specific Grid applications possible. In particular, Instant-Grid shows increases of performance in a computer Grid environment compared with a standard single computer environment [41].

Instant-Grid uses the software services and libraries provided by GT4, which are integrated in the Instant-Grid environment. GT4 realizes the “Resource and connectivity protocols” layer, while Instant-Grid represents the layers “Collective services” and “User applications” of the Grid architecture (see Section 2.4).

2.6.1 Architecture

Instant-Grid provides a complete independent computer Grid environment based on a Linux Knoppix live CD-ROM, which includes all required services for establishing a Grid for x86 computers. As depicted in Figure 2.5, the front-end computer boots from the Instant-Grid CD-ROM and provides mechanisms to initialize several nodes over an Ethernet by the use of a Preboot Execution Environment (PXE) (Figure 2.5(a)). Therefore, services including Dynamic Host Configuration Protocol (DHCP), Trivial File Transfer Protocol (TFTP) and Network File System (NFS) server, are initialized and started in the booting sequence of the front-end. An additional service, the Distributor, is needed to distribute further information and configurations to the clients. Furthermore, all components of the Grid middleware GT4 and a service for certification, the CA, will be initialized (Figure 2.5(b)). The Instant-Grid consists of different components that are provided in a GNU/Linux environment (Figure 2.5(c)) [41]. These components are described in the following.



(a) Distribution (b) Boot process (c) Components

Figure 2.5: Architecture of Instant-Grid [41]

The **Instant-Grid Infrastructure** mainly consists of two categories. The IP-Collector is concerned to refresh basic information about resources in the Grid. The Distributor distributes information about configurations within the Grid through a file structure on the NFS [41].

The **GT4** and its belonging components are responsible for data transfer (GridFTP), job submitting (WS-GRAM) and identification of resources and services of the Grid (MDS4). This behavior can be controlled by the user in the command line or via a web based user interface [41].

Besides the command line, the web based user interface that is provided by the **Apache** HTTP server and the Apache servlet container Tomcat is an important interface for the user. Most of the functionalities are provided by the GridSphere [62] portal. Additional information about resource load can be displayed with Ganglia [56], which is a scalable and distributed system for monitoring. The distributed rendering tool "POV-Ray" [76] is an application that demonstrates the distributed computation in a Grid and can be controlled through a web interface [41].

The distribution and administration of tasks within the Grid are controlled by the interface WS-GRAM of GT4. The applications for demonstration have to choose appropriate resources and have to compose the results of the resources on their own [60].

2.6.2 Alternative Grid environments

There are several initiatives and organizations, which provide implementations for Grid software, e.g. Sun Grid [68], NorduGrid [70], Open Science Grid [61], OurGrid [74], Enabling Grids for E-scienceE [53].

Sun Grid is an on-demand Grid computing service operated by Sun Microsystems. Sun Grid delivers computing power and resources over the Internet and allows developers, researchers, scientists and businesses to optimize performance, speed up the time to results and accelerate innovation without investment in their IT infrastructure. The Sun Grid Compute Utility provides paid access to a computing resource over the Internet. It supports open-source technologies such as Solaris 10, Grid Engine and the Java platform, which is based on the open-source Sun Grid Engine. [69]

NorduGrid is a Grid Research and Development collaboration with the purpose of developing, maintaining and supporting the Advanced Resource Connector, a free Grid middleware. [70]

The Open Science Grid is a production-quality Grid computing infrastructure for large scale scientific research, built and operated by a consortium of universities and national laboratories, scientific collaborations and software developers. The Open Science Grid Consortium was established in 2004 to enable diverse communities of scientists access to a common Grid infrastructure and shared resources. Groups that choose to join the Consortium contribute effort and resources to the common infrastructure. [61]

OurGrid is a peer-to-peer grid founded in 2004. Users can access a large amount of computational power that is provided as a Grid by the idle resources of all participants [74].

Enabling Grids for E-sciencE (EGEE) is a project that provides researchers in academia and industry access to a production level Grid infrastructure, independent of their geographic location. [53]

The implementations discussed above might not be Grids according to the theory that is depicted in Section 2.2. Nevertheless, most of the functionalities of the Grids above are the same, whereas the realization can differ.

The decision on using Instant-Grid for the research of this thesis was mainly driven by its use of the middleware GT4, because Globus Toolkit is one of the most widely used low-level middlewares today [1]. Besides that, Instant-Grid provides a user-friendly establishment and hosting of a Grid computing environment.

2.7 Gridification of an application

Gridification means the integration of an application into a computer Grid in order to use all advantages of Grid computing. Hereby, the utility of the Grid integrated application has to be significantly greater than the utility of the monolithic application, as mentioned in Section 2.2 [8].

A task that is solved by an application, which will be integrated into a Grid, has to be parallelizable in order to allow a division into several job parts. The division should be handled by a management system that includes a scheduler. The management system is mainly responsible for the determination of all jobs and their descriptions. Another task of the management system is the synchronization of the jobs. It assures that dependent jobs are executed in the correct order. The job dependencies have to be entered by the user, for instance with the Grid Workflow Execution Service (GWES). The user has to enter the rules of the division of the application into jobs. This can be difficult and time intensive, since the management system has to be readjusted for similar applications or new implemented.

The core point of the management system is the determination of the jobs and their synchronized distribution. The management system itself uses the services of the Grid middleware. If the jobs and their order have been determined, the jobs are distributed to the resources in the Grid. The management system should assign the jobs to the resources, when it is necessary in order to allow a maximum flexibility. For the assignment, the execution service of the Grid middleware is used. This service is usually a command line call with parameters, which include the job and its description, e.g. the application name with the determined application parameters. It is possible that assigned resources disappear and the jobs have to be reassigned. The management system uses the service of the Grid middleware in order to determine if jobs are running, finished, or failed. The collection of the output data from the jobs is done with services of the Grid. That includes copying of files or transforming data from the assigned nodes. If the application is integrated into the Grid and using the Grid mechanism for its advantage, the application is gridificated. The realization and integration of an application into a Grid environment is described in Chapter 5.

3 Software testing

This Chapter gives an overview of software testing. Software testing is an essential part of Software Engineering. It checks the developed fragments in every phase of the development process.

Section 3.1 describes the fundamentals of testing. The dynamic testing technique black box testing is explained in Section 3.2, whereas Section 3.3 gives a detailed explanation of the fundamental test process.

3.1 Testing fundamentals

Software testing is an analytic activity for evaluating the quality of software [39], which is included in the activities of Software Quality Assurance (SQA) [31]. SQA contains additionally activities for organizing examination, which comprises software project management, and constructive activities, which includes software engineering, in order to avoid errors [26].

Testing is the examination of test objects by their execution. A test object is a part of a software system or the software system itself that has to be tested. Testing should not be confounded with debugging, which is the localization and correction of defects, bugs and not fulfilled specifications of software systems. The task of debugging is to localize and correct defects in a software system in order to increase the quality of a product with the assumption that no new defects are caused by the correction. Testing itself provides a base for debugging, since the goal of testing is the detection of failures, which indicate defects of the tested software. Besides this, testing can increase the confidence in a software product, measure quality and avoid defects through analyzing programs or their documentations [23, 35].

A test is defined as the whole process of systematic execution of a program in order to find out whether the software realization matches the requirements. For a efficient realization of tests, it is important to characterize a test process that includes the test object execution with test data as well as the planning, design, implementation and analysis, which are also specified by the test management. The fundamental test process is described in detail in Section 3.3. The defined test conditions, the inputs and the expected outputs or the expected behavior of the test object are specified in test cases. A test case should be designed with the intent to detect undiscovered faults with a high probability [29]. The execution of one or more test cases is part of a test run or test suite. In a test suite detailed instructions, goals and the system configurations used during testing are specified for each collection of test cases [35].

The ability of proving the absence of defects is not possible through testing. Testing can only discover present defects that a software product has. For the detection of different classes of faults, it is necessary to construct testing as systematical as possible and, in addition, with the constraint of having a minimum amount of time and effort [66].

3.2 Dynamic Testing

Software testing comprises dynamic and static testing. Static testing includes methods like code reviews, coding guidelines and inspection, whereas dynamic testing is the execution of test objects on a computer for testing a software product. The test object has to be executed with input data. If the test object is not executable, like in an early software development phase, it has to be embedded in a test bed as shown in Figure 3.1 in order to obtain an executable program. A test object usually calls different parts of a software product through predefined interfaces. If these parts are not implemented or if they should be simulated for the test, placeholders, called stubs that substitute parts, are required. Stubs simulate the input and/or output behavior of programs or their parts which are called by the test object [35].

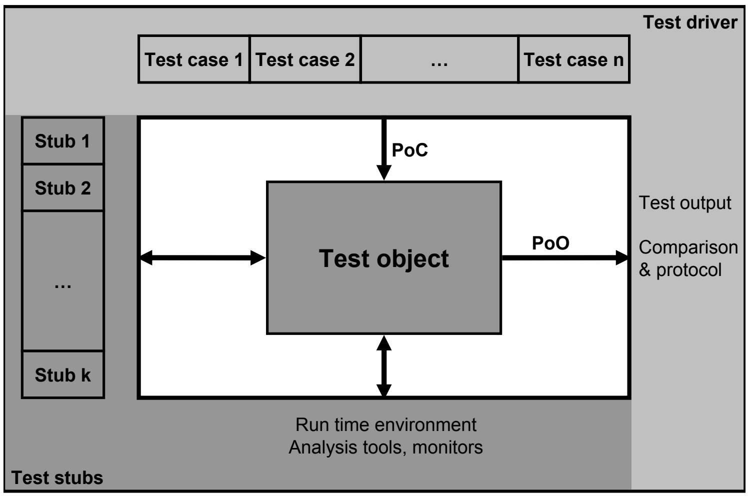


Figure 3.1: Test Bed [35]

The test bed has to provide the input data realized through the test driver. The test driver simulates programs that are supposed to call the test object with test data specified in test cases. The test bed comprises the driver and the stubs and constitutes the executable program with the test object. Test beds have either to be created, adjusted, expanded, or modified by the tester [35].

There exist several different approaches for testing the test object. Dynamic testing is divided into two groups: black box and white box testing. These test case design techniques provide a base for creating test cases [35]. The black box testing technique is described in the following, since it is applied in the case study of this thesis.

If the black box testing technique is used, the test object is seen as a black box, i.e., no knowledge of internal logic or code structure is required. As depicted in Figure 3.2, the focus of black box testing lies on the input and output data. For a given input, an input related behavior is expected and, hence, the expected outputs have to be determined before the execution of the test object. The test cases are built from the specification of the test object. The Point of Control (PoC) is outside the test object, which means the test object can only be controlled through inputs. The Point of Observation (PoO) is outside the test object, too. That is the reason why the test can only be evaluated through the outputs. There is no direct access to the structure of the test object. The base of the black box testing strategy lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system. The generic type of functional testing includes black box testing. Test cases can be developed from equivalence classes, boundary value analysis, state transition testing and/or cause-effect graphing and decision table techniques [35].

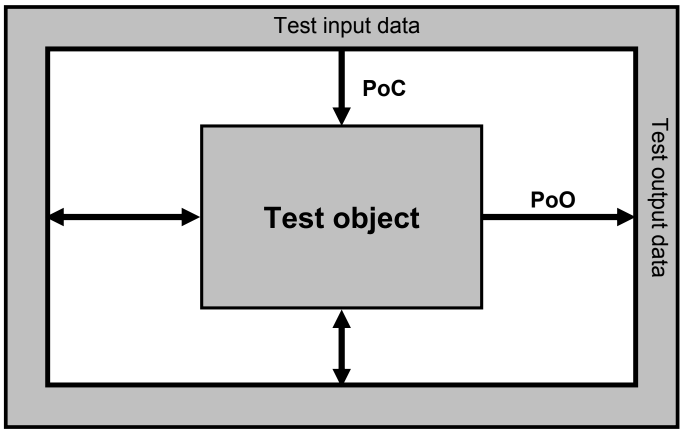


Figure 3.2: PoC and PoO at black box technique [35]

3.3 The fundamental test process

For structured and controllable testing, it is necessary to define a test process. Each phase of a development process, e.g. of the V-Model, should have a corresponding testing phase. Each testing phase of the development can be divided into the fundamental test process. As shown in Figure 3.3, the fundamental test process consists of five phases; test planning and control, test analysis and design, test implementation and execution, evaluation of test exit criteria and reporting and post testing activities [35].

1. **Planning and control**

   A controlled realization of tests cannot be done without planning. Planning of tests is at the beginning of the test process. Test planning may include planning of resources, determination of test strategies, test intensity, prioritization of tests and

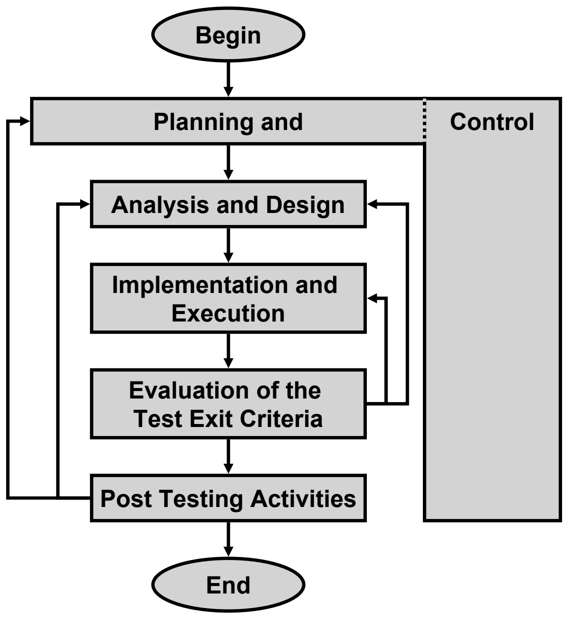


Figure 3.3: Fundamental test process [35]

tool support. Tasks and goals of the test have to be specified in a test plan that may also include the coordination of required resources, like employees, equipments, utilities and time. The control comprises test progress monitoring where test activities are compared with the test plan in order to reach goals as specified in the test plan or to reconfigure resources as demanded from the situation to achieve the goal specified in the test plan [35].

A main point of test planning is the determination of the test strategy. This includes a prioritization of subsystems of the software product depending on the expected risk and severity of failure effects in order to assess the test effort for each of them. The intensity of testing depends mostly on the used test methods and the test coverage. The test coverage is a test exit criterion. Besides this, the completion of certain specifications given by the customer can be a test exit criterion, too. All test exit criteria should be defined in order to decide, when tests are completed. Additionally, testing tools that will be used should be acquired or updated [35].

2. **Analysis and design**

   Based on the analysis of the specifications, the expected behavior and the structure of the test object, the preconditions and requirements for the test case design can be determined. The specifications have to be reworked if they are not concrete enough. If the specifications are imprecise, the testability would be insufficient. The testability is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [91]. The most important task is to define logical test cases in order to design concrete test cases. The test techniques are adapted from the test plan and also determined in the analysis corresponding to the complexity of the test object. Therefore, test planning, test analyzing and test designing can be considered simultaneously. For every test case, the input and the expected result have to be defined as well as the expected behavior, changes to global data and states and any other consequence of the test case [35].

3. **Implementation and execution**

   In this phase, concrete physical test cases are developed from logical test cases. Additionally, it has to be defined how and in which order the test cases will be executed. For this issue, the priority that was defined in the test planning phase has to be considered. To conduct a test, its environment includes test harnesses, which comprise stubs and drivers have to be programmed, built, acquired or set up as part of the test implementation. The correct functioning of the test environment has to be ensured, since faults can also be caused by added test harnesses. The test execution technique is used to perform the actual test execution, either manually or automated [35].

   If a failure is found, it has to be exactly determined how it was caused. It can be caused by an inexact test specification as well as by a wrong software implementation. Real failures have to be documented. If more than one failure has been found, the priority of the fault correction has to be determined. For later evaluation the test coverage should be measured [35].

4. **Evaluation of the test exit criteria and reporting**

   The test objectives are compared with the test protocols, i.e., it is evaluated if the test criteria are fulfilled. This can either mean that the tests are finished because every criterion has been reached or it may be decided that additional tests should be run or the test criteria had a too high level. When the test criteria are achieved or not achieved criteria are clarified, a test summary report should be written. Traceability, which is the ability to identify related items in documentation and software, such as requirements with associated tests, should be assured in a high level in order to improve reproducibility [35].

5. **Post testing activities**

   In this final phase experiences of the test work should be analyzed and conserved for future projects. Especially discrepancies between planning and execution of various behaviors and supposed causes should be considered and reported. In addition, software system release date, test finishing date, achievement of milestones, etc. should be documented. The test evaluation report is a document that is produced in the end of the test process summarizing all testing activities and results. It also contains an evaluation of the test process and lessons learned [35].

4 Testing and Test Control Notation Version 3 (TTCN-3)

The Testing and Test Control Notation Version 3 (TTCN-3) was specifically designed for testing and certification [55]. The language comprises many constructs similar to those in other programming languages. TTCN-3 additionally applies concepts like a rich type system including native list types and support for sub-typing. It embodies a powerful built-in data matching mechanism and distributed test system architecture [40]. It provides a snapshot semantics, i.e., well defined handling of port and timeout queues during their access. Furthermore, it introduces the concept of verdicts and a verdict resolution mechanism. In addition, specifications of concurrent test behavior and timers are supported by TTCN-3. It allows test configuration at run-time and the tests focus only on the implementation that is to be tested [55].

TTCN-3 supports test specifications using black box testing techniques that are applicable for a diversity of tests for reactive and distributed systems [17], e.g. telecom systems (ISDN, ATM), mobile systems (GSM, UMTS), Internet (IPv6, SIP) and CORBA based systems [18].

The TTCN-3 language is standardized by the European Telecommunications Standards Institute [64]. The standard is divided into several parts, whereas each covers a different aspect of the language. Examples are listed in the following:

- Part 1: TTCN-3 Core Language [85],
- Part 2: TTCN-3 Tabular Presentation Format [86],
- Part 3: TTCN-3 Graphical Presentation Format [87],
- Part 4: TTCN-3 Operational Semantics [88],
- Part 5: TTCN-3 Runtime Interface [89],
- Part 6: TTCN-3 Control Interface [90].

The textual syntax of the TTCN-3 language is specified in Part 1. This part describes the development of TTCN-3 test suites. Additional, the tabular presentation format is specified in Part 2 and the graphical presentation format in Part 3. The semantics of the TTCN-3 core language is specified in detail by using a flow graph notation in Part 4 [22]. Part 5 describes the runtime interface to the system and platform adapters and Part 6 the control interface, for example to the test management and component handling [58].

TTCN-3 is not tied to a particular application or its interface(s) and neither to any specific test execution environment, compiler or Operating System (OS). TTCN-3 itself is not executable and requires a compiler/interpreter as well as adapter and codec implementations [55].

Concepts of TTCN-3 are described in Section 4.1. Basic elements of the language are briefly explained in Section 4.2 and a typical TTCN-3 test architecture is explained in Section 4.3, which is adapted for distributed systems in Section 4.5. Section 4.4 explains the process of how to build tests in TTCN-3 and Section 4.6 gives an overview of ongoing TTCN-3 tools.

4.1 Concepts

A test environment in TTCN-3 contains basically one or more test cases that are communicating with the System Under Test (SUT) through ports. First of all, the SUT has to be stimulated through a stimulus that is sent over a TTCN-3 port, which is mapped to the SUT. After sending the stimulus, a related response will be received by the TTCN-3 port and evaluated by the test case [20].

Due to the fact that TTCN-3 allows the specification of dynamic and concurrent test systems, several scenarios can be realized. A test system in TTCN-3 includes a set of interconnected test components with well-defined communication ports. In addition, an explicit Test System Interface (TSI) defines the boundaries of the test system. Figure 4.1 shows a possible TTCN-3 test system [19].

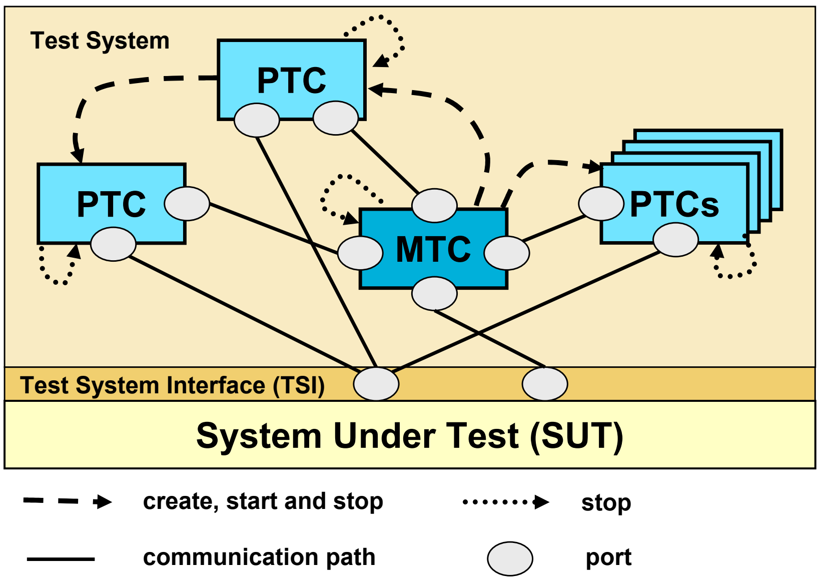


Figure 4.1: Dynamic configuration in TTCN-3 [19]

The Main Test Component (MTC) is the base of the test case and is started automatically at the beginning of each test case execution. A test case terminates when the MTC terminates. The MTC manages the test execution and is usually responsible for the main workflow of the test system. The behavior of the MTC is specified in the body of the test case definition. Parallel Test Components (PTCs) can be created, started and stopped dynamically during test case execution. A Test Component (TC), which is either a PTC or the MTC, may stop itself or can be stopped by another TC [19].

For communication purpose between TCs, TTCN-3 provides the port concepts that is shown in Figure 4.2. Each TC can be mapped to the SUT or connected to another TC in order to send and receive messages either from the SUT or a TC. Ports have directions, which should be `in` for receiving messages, `out` for sending messages or `inout` for both. A test port with the direction `in` is modeled as an infinite FIFO queue. Incoming messages are stored by the FIFO queue until they are processed by the receiving TC. Outgoing information is not buffered, because message sent through a port with the `out` direction are directly linked to the communication partner [19].

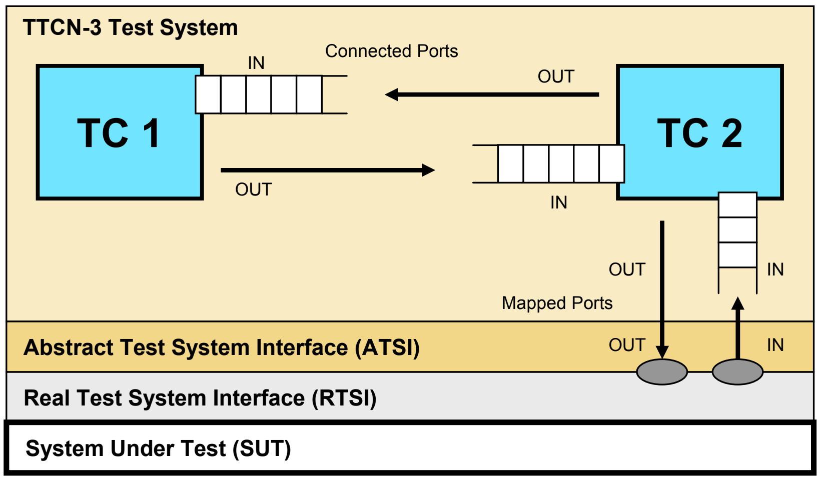


Figure 4.2: Illustration of the connect and map operations [85]

Ports can be defined as message-based, procedure-based, or mixed. Message-based communication adapts the principle of asynchronous message exchange. Procedure-based communication allows calling procedures in remote entities [19].

TTCN-3 distinguishes between the Abstract Test System Interface (ATSI) and the Real Test System Interface (RTSI). The ATSI that is modeled as a collection of ports defines the abstract interface to the SUT. That means, the communication endpoints to the SUT are specified by the ATSI in an implementation independent manner. The application specific part of a TTCN-3-based test environment is the RTSI, because it implements the real interface of the SUT. That means, the RTSI defines the implementation dependent adaptation between ATSI and SUT [6, 19].

4.2 Basic elements

Modules are the building blocks of all TTCN-3 test specifications. A module consists of a definition part and an optional control part. In the definition part, definitions, e.g. for test components, their communication interfaces, types, test data templates, functions and test cases, are defined. The structure of a TTCN-3 module is depicted in Figure 4.3.

The definition part covers constant and data type definitions that are based on TTCN-3 predefined and structured types. TTCN-3 types are grouped in basic (e.g. integer, float), structured (e.g. record, set), any type (i.e., every type which is known in the module) and configuration types (e.g. port type, component type) [34].

The test data description is defined in templates to either transmit a set of distinct values or to test whether a set of received values matches the template specification. Templates support matching mechanisms for the denotation of a variety of expected data [34].

For the test configuration, it is required to define test components and their communication interfaces. The test components communicate through ports. Timers can be used for guarding the behavior of test components [34].

Furthermore, the test behavior has to be defined. This includes definitions of test cases, functions and altsteps. A test case runs on a component type and can be called through the TTCN-3 `execute` operation in the control part of a module. Hence, in the module control part, the execution can be controlled [34].

Modules can be imported into other modules. Hence, this mechanism allows a fine granular reuse of all TTCN-3 objects. Therefore, new test scenarios can be developed on the basis of existing tests.

For evaluation of test runs, TTCN-3 provides a special test verdict mechanism. The base for this mechanism is a set of predefined verdicts that are evaluated in local and

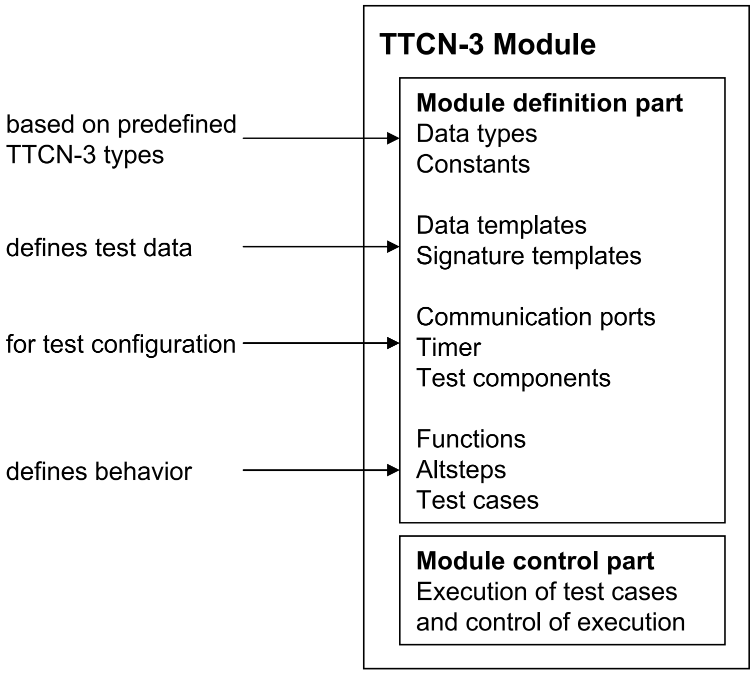


Figure 4.3: A TTCN-3 module [34]

global test verdicts. The local test verdict can be affected by the `set` operation and read by the `read` operation. The predefined verdicts are `pass`, `inconc`, `fail`, `error` and `none`. With help of the verdict test runs can be judged [19].

4.3 Test system architecture

A TTCN-3 test system conceptually consists of a set of interacting entities where each entity implements specific test system functionalities. These entities manage test execution, interpret or execute compiled TTCN-3 code, realize the communication with the SUT, administer types, values and test components, implement external functions and handle timer operations [89]. The test system architecture in TTCN-3 is shown in Figure 4.4 [90].

The TTCN-3 Executable (TE) entity is a component of the test system. It implements TTCN-3 modules on an abstract level. This abstract concept is made concrete through the other entities [33]. The TE entity has to realize three tasks, which are the control of test case execution, the proper execution of TTCN-3 behavior and queuing of events. The TE communicates with the Test Management and Control (TMC), which includes the Test Management (TM), the Component Handling (CH), the Test Logging (TL) and the CoDec (CD), via the TTCN-3 Control Interfaces (TCI). The interfaces between the TE, the System Adapter (SA) and the Platform Adapter (PA) are defined by the TTCN-3 Runtime Interfaces (TRI) that are used for the realization of the communication with the SUT [4, 89].

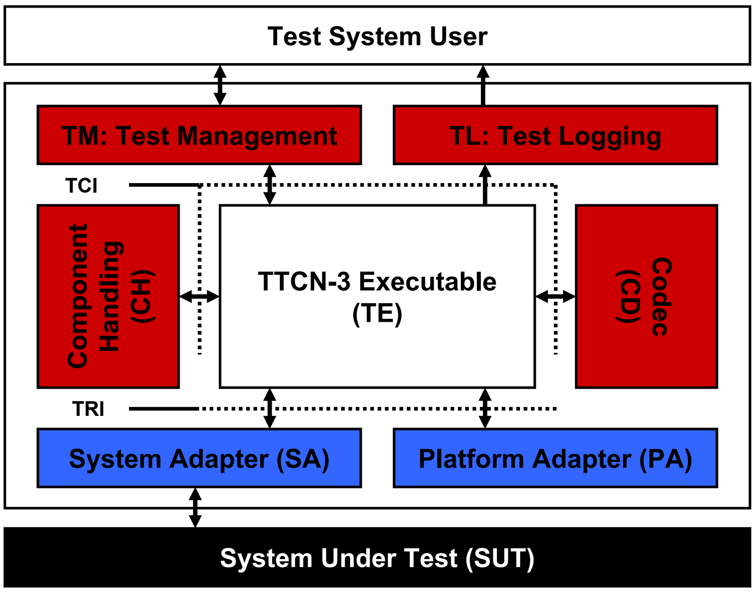


Figure 4.4: General Structure of a TTCN-3 Test System [89]

4.4 TTCN-3 based test development process

The process of the base development of tests in TTCN-3 is depicted in Figure 4.5. Test purposes are defined from the SUT specifications through test definitions. After the tests have been designed, they can be specified in TTCN-3 modules. These modules can be divided into data and behavior modules. It is possible to define arbitrary finer granularity of modules for possible reuse. The Abstract Test Suite (ATS) represents the test specification and includes these TTCN-3 modules. These steps belong to the test planning, test analyzing and test designing phases related to the fundamental test process that is described in Section 3.3 [16, 38].

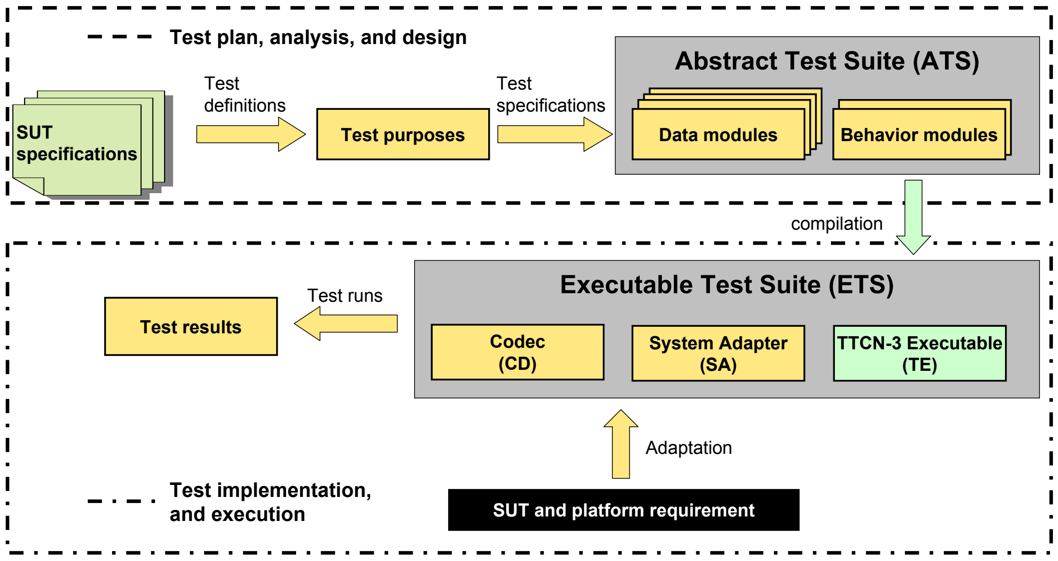


Figure 4.5: TTCN-3 based test development process [38]

The ATS is not executable on its own and must be compiled to the TTCN-3 test executable (TE). The TTCN-3 code has to be compiled to Java (or depending on the TTCN-3 environment C#, C, or C++) code. This compiled code is part of the Executable Test Suite (ETS). To obtain a complete ETS, elements related to the SUT and the test platform, such as the CD and the SA must be developed and combined with the TE. The SA mediates between the SUT and the TE. Messages sent to the SUT have to be encoded by the encoder and messages received from the SUT have to be decoded by the decoder. The encoder and decoder are part of the CD. The SA and the CD have to adapt concepts of the SUT and its platform requirements in order to communicate with it. The tests of the ETS can be run to provide the required test results in order to evaluate them [38]. The process described in this paragraph belongs to the test implementation and test execution phase of the fundamental test process that is described in Section 3.3.

4.5 Distributed testing

For testing in a Grid computing environment, a support for distributed testing is required. Concepts like component-based distributed test systems in dynamic test configurations are available in TTCN-3 [33]. The general structure of a distributed TTCN-3 test system is illustrated in Figure 4.6. The TE is instantiated on each test device. The communication between distributed test system entities is realized through the CH. The CH entity provides the structure for synchronizing test system entities, which might be distributed onto several nodes. The CH entity allows the test management to create and control distributed test systems in a manner which is transparent and independent from the TE [90].

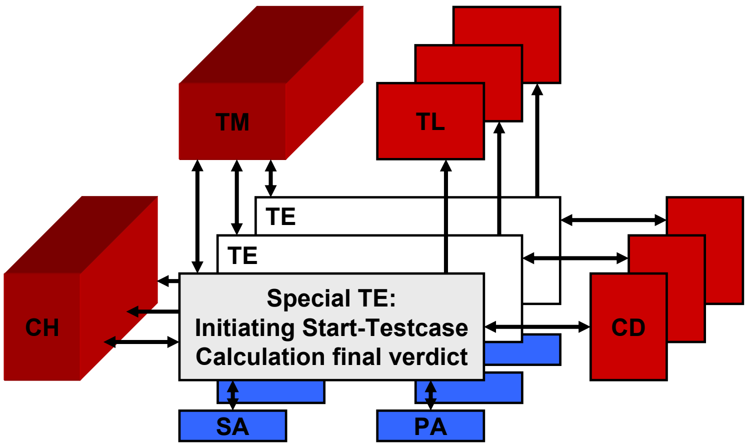


Figure 4.6: General structure of a distributed TTCN-3 test system [90]

Each node within a test system includes the TE, SA, PA, CD and TL entities. The entities Component Handler (CH) and TM intercede between the TEs on each node. The TE which starts a test case is a special TE. It calculates the final test case verdict. Besides this, all TEs are treated equally [90].

For deploying and executing distributed tests, Din et al. [5] designed the architecture shown in Figure 4.7. The test console is the control point of this platform. It handles the management operations to create test sessions, deploys test components into containers and controls the test execution. In the context of a test session, the tests are deployed, configured and executed. An important functionality of the session manager is the load balancing, which coordinates the distribution algorithms [5].

A test daemon is installed on each host as a standalone process and manages the container, which belongs to a specified session. A container includes the specific test system's entities: TM, CH, TE, CD, TL, SA and PA. For the communication, the containers provide services to the test console and the test components, including transaction support and resource pooling. Hence the containers intercede between the test console and the test components. The containers manage the installation, configuration and removal of PTCs. The containers are the target operational

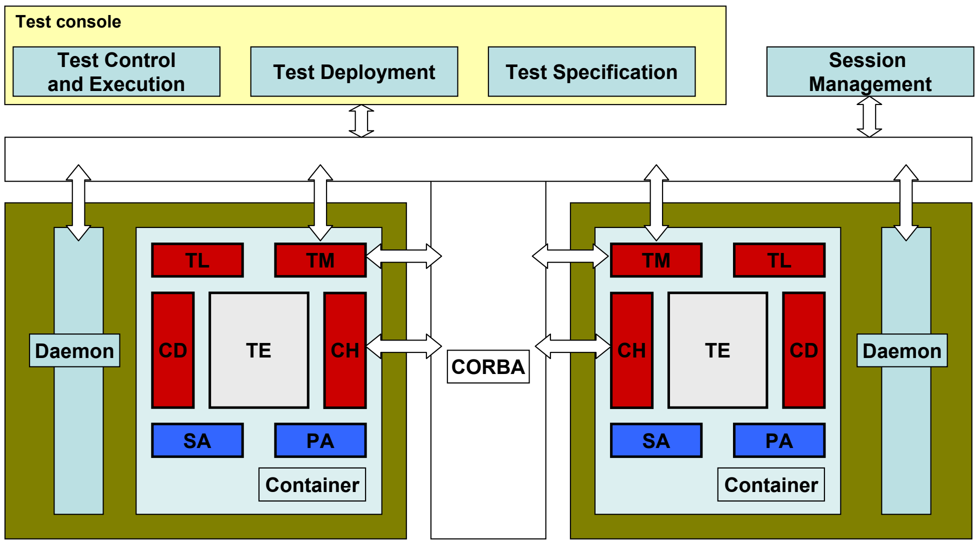


Figure 4.7: Distributed test platform architecture

environment and comply with the TCI standard [90] for TTCN-3 test execution environments. The containers themselves are connected through a CORBA [32] platform in oder to communicate with each other [5].

The CH realizes the distributed handling of test components. It is responsible for the distribution of TTCN-3 configuration operations like `create`, `start` and `stop` of test components, the connection between test components (`connect` and `map`) and inter-component communication like `send`, `call` and `reply` among two TEs participating in the test session [5].

4.6 Tools

Various vendors provide TTCN-3 tools in order to compile TTCN-3 code into Java, C#, C, or C++ to build executables. Some of these vendors and their tools are listed in the following:

- Danet Group - TTCN-3 Toolbox [63],
- Elvior - MessageMagic [49],
- Fraunhofer FIRST and Metarga GmbH - TTCN-3 Express [51],
- Métodos y Tecnología - ExhaustiF/TTCN [84],
- OpenTTCN - OpenTTCN Tester [75],
- Telelogic - Tau Tester [80],
- Testing Technologies - TTworkbench [57].

A discussion or comparison of these tools is not in the scope of this thesis. The best description of each tool is given by the supporting vendor on their websites, which are referenced in the bibliography.

The realization of the case study of this thesis has been implemented with the TTworkbench Enterprise provided by Testing Technologies. TTworkbench Enterprise is a graphical test development and execution environment for TTCN-3. It includes the full range of features needed for test specification, execution and analysis [57].

The main point for this decision was that TTworkbench Enterprise allows executing test scenarios in a distributed manner, which is an important requirement for executing tests in a Grid computing environment. The TTmex feature of the TTworkbench Enterprise allows managing, executing and analyzing of distributed TTCN-3 test scenarios. The TTworkbench series compile TTCN-3 code into Java code. Hence the test harnesses are implemented in Java [57].

5 A Grid application

This Chapter gives an overview about the Grid application that is used in the case study of this thesis. Section 5.1 gives an insight into this Grid application, whereas Section 5.2 gives an explanation on how this application has been integrated into the Grid. Section 5.3 describes the realization of the application management used in the Grid, whereas Section 5.4 gives a description about its technical specification .

5.1 Description of the application

The algorithm of an application that is supposed to run in a Grid computing environment should be parallelizable in order to tap the full potential of the advantages of Grid computing. The reason behind the parallelization is that the Grid has to compute a task distributed with the power of more than one resource. The main task has to be divided into smaller tasks in order to allow concurrent runs in the Grid.

The application used in this case study is the calculation of Mandelbrot sets [27]. This algorithm is parallelizable and allows a spreading in different independent jobs. The Mandelbrot set is a set of points in the complex plane that forms a fractal. Mathematically, the Mandelbrot set can be defined as the set of complex c-values for which the orbit of zero under iteration of the quadratic map x^2+c remains bounded [46]. Each point of the complex plane has to be inserted in the iteration formula $z_{n+1} := z_n^2+c$. A point of the plane belongs to the Mandelbrot set if it does not leave a predefined radius. It is possible to calculate the points parallel, because their calculation is independent from each other.

The application used in the case study of this thesis is implemented in the programming language Python, which allows a direct calculation of complex numbers. A function `Mandelbrot` that includes the Mandelbrot set algorithm is shown in Listing 5.1. The parameters of this function allow the calculation of a specified part of the set. Multiple independent executions of this function are possible through different parameter specification in order to allow a parallelization. The most important parameters are `left`, `right`, `top` and `bottom`, which specify the part, i.e., the local set, of the global Mandelbrot set. The parts can be put together to the global Mandelbrot set after their calculation. The other parameters cover the color of the set, the output file and the resolution of the output image (line 1). Variables are initialized in lines 2 to 6 as described in the following. The predefined radius of the Mandelbrot set is set to 2.0 and shown in Listing 5.1 in line 2. The maximum iteration of the formula is set to 20 (line 3). If the iteration is set to a higher number, the picture would be more detailed but more computationally intensive. Afterwards an image is initialized, the main algorithm begins. The first two `for` loops iterate through each pixel of the image and determine the complex number, which is

```
1  def Mandelbrot(left, right, top, bottom, red, green, blue, file, width, height):
2    limit  = 2.0 # radius
3    maxiteration = 20 # detailed, number of iterations
4    im     = Image.new("RGB", (width, height), (255, 0, 0)) # set up color
5    draw   = ImageDraw.ImageDraw(im) # set up an image
6    for x in range(width):
7      creal = left + x*(right - left)/width
8      for y in range(height):
9        cimaginary = bottom + y*(top - bottom)/height
10       c = complex(creal, cimaginary)
11       z = 0.0
12       i = 0
13       for i in range(maxiteration):
14         if abs(z) > limit:
15           break
16         z = (z**2) + c
17         if i == (maxiteration - 1):
18           draw.point((x, y), (red, green, blue))
19         else:
20           draw.point((x, y), (i*10, i*15, i*30))
21   im.save(file)
```

Listing 5.1: Mandelbrot set algorithm

inserted in the iteration formula $z_{n+1} := z_n^2 + c$ in line 16. If it is part of the Mandelbrot set, the pixel gets the basic color (line 18), if not, it gets a color that depends on the number of iterations (line 20). After the computation is terminated, the calculated data is written into an image `file` (line 21) that is specified by the parameter file (line 1). Figure 5.1 shows a typical visualization of the Mandelbrot set.

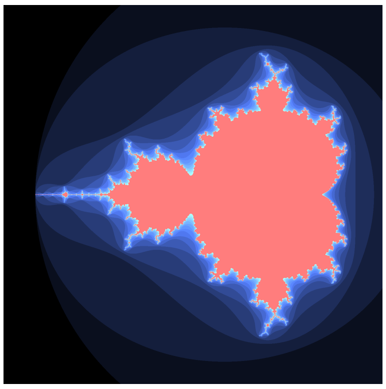


Figure 5.1: Mandelbrot set of the implemented Grid application

5.2 Integration into the Grid

The parallelization of the application explained in Section 5.1 is handled by an application management system that has been integrated into the Instant-Grid environment. The Instant-Grid environment uses most of the services provided by the Grid middleware GT4 automatically. These services include services for security, authentication and authorization. Services like distributing tasks or copying files to another node have to be handled by a management system that includes a scheduler. The management system controls and monitors the distribution and execution of tasks on Grid nodes by using the services of GT4. Figure 5.2 shows how the Mandelbrot set application and its management have been integrated into Instant-Grid.

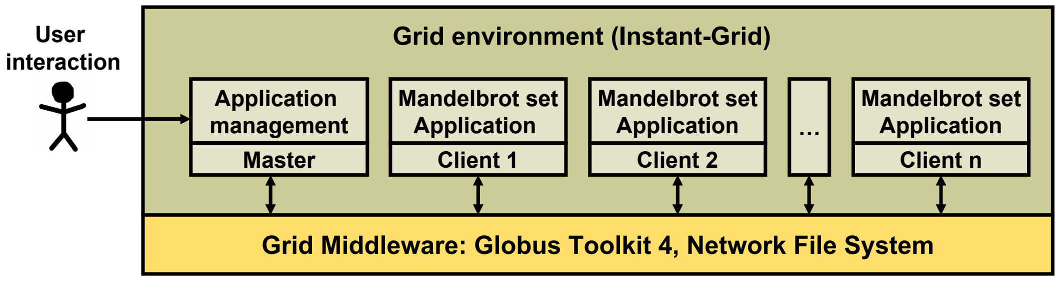
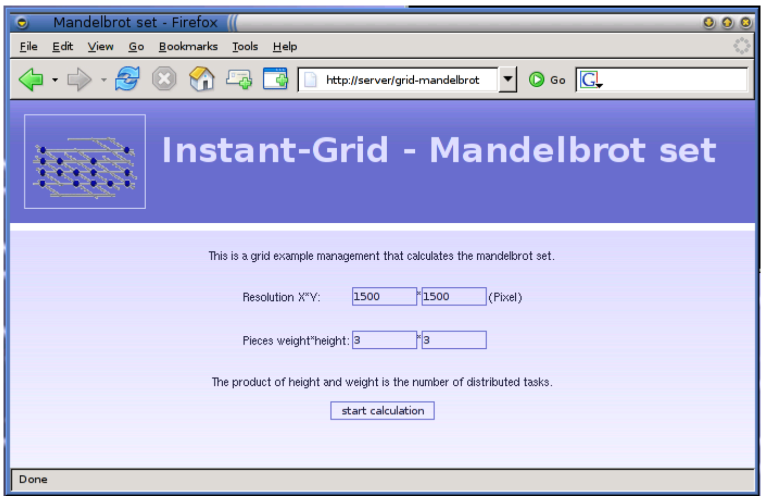

User interaction
Grid environment (Instant-Grid)
Application management
Master
Mandelbrot set Application
Client 1
Mandelbrot set Application
Client 2
...
Mandelbrot set Application
Client n
Grid Middleware: Globus Toolkit 4, Network File System

Figure 5.2: Application integration into Instant-Grid

The user interacts with the application management system through a web interface that is shown in Figure 5.3. The user enters service parameters, including the resolution of the output image and the number of tasks and requests the application management to start the calculation. The application management is mainly responsible for computing the task parameters, assigning tasks to free hosts, synchronizing the tasks, monitoring


Mandelbrot set - Firefox
File Edit View Go Bookmarks Tools Help
http://server/grid-mandelbrot
Go
start calculation
Done

Instant-Grid - Mandelbrot set

This is a grid example management that calculates the mandelbrot set.

Resolution X*Y: 1500 * 1500 (Pixel)

Pieces weight*height: 3 * 3

The product of height and weight is the number of distributed tasks.

Figure 5.3: Web interface

the execution of tasks and collecting and composing the results of the tasks.

After the user has sent a request, the management divides this request into tasks and submits them as jobs to the GT4 in order to execute them on an idle client. For the submission of the tasks a call to the GT4 service `globusrun-ws` is used. This call requires as parameters, besides GT4 options, the name of an idle host and the task options including the task name, its location and parameters. The task parameters have been determined by the application management system and are equal to those of the application explained in Section 5.1.

After a task has been finished, a file is written in the NFS directory by the responsible node. Alternatively, it is possible to use the GT4 service GridFTP instead of the NFS. The GridFTP is a service for copying files between the nodes of the Grid. However, since the Instant-Grid provides an NFS automatically, the NFS infrastructure is used by the application management.

Once all tasks are completed, the application management initiates the composition of the job results from the NFS. The composed overall result is presented to the user through the web interface. The interaction can be described as a grid application scenario, which is depicted as a sequence diagram in Figure 5.4. In this example the user starts the calculation through the web interface as explained above. In this scenario the Grid computing environment consists of a master host, on which the management system is running and two clients. Two jobs are calculated from the user's request by the

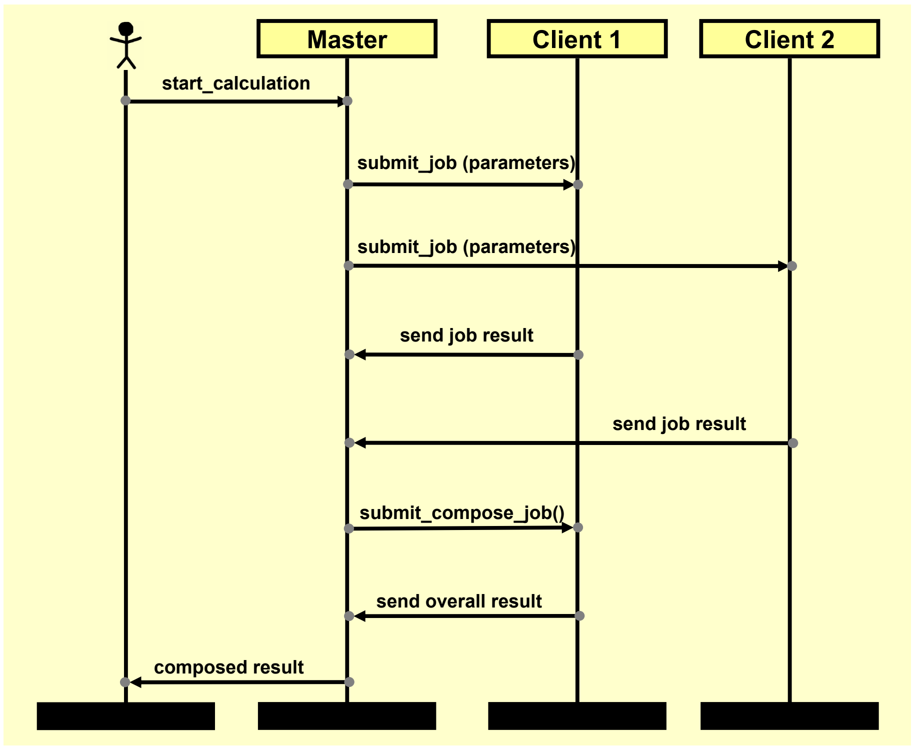
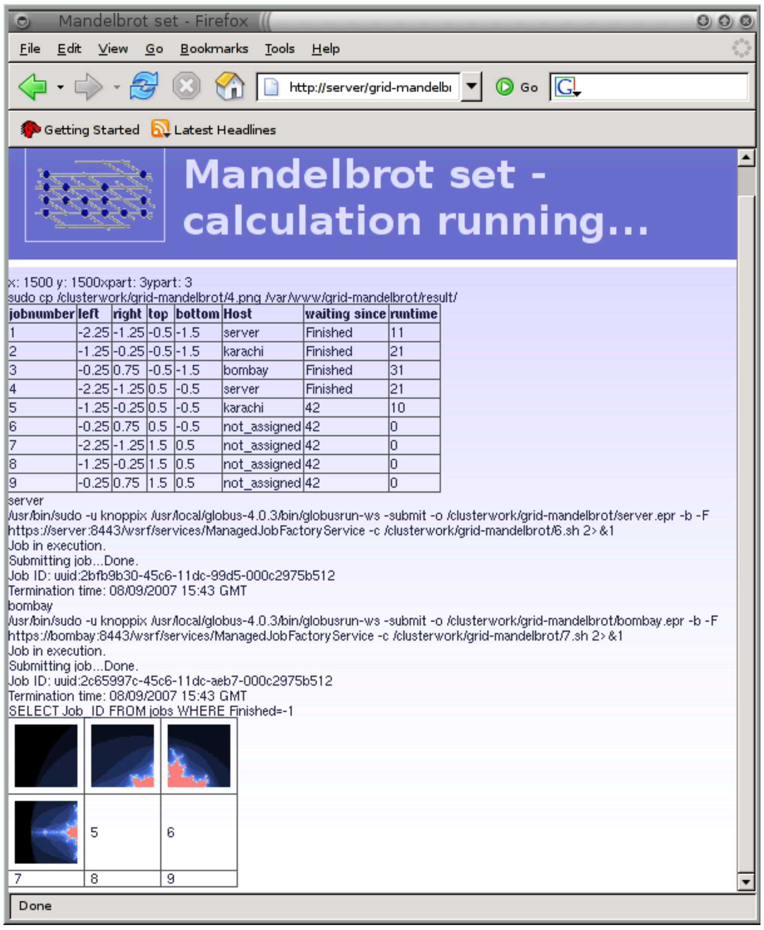


Figure 5.4: Grid application scenario

application management. The first task is submitted to client one and the second to client two. After their completion, the final job, which composes both results, is assigned to client one and its output is presented to the user at the web interface on the master host. The user can download the composed result.

While the application management distributes the tasks, monitoring is provided to the user through the web interface. It monitors task completion and displays the progress to the user. The related web interface is shown in Figure 5.5. In this example nine jobs with different parameters have been created as it can be seen in the shown table. In that table, the user is informed about the progress of the determined jobs. For example, information, on which host the job ran or is running, whether the job is finished and since when an unfinished job is waiting, is shown. The last column provides the runtime of finished jobs. In this case, four jobs have already been completed. They are also shown in the image preview on the bottom of Figure 5.5. Between the table and the image preview, the command of the actual job submissions and their GT4 status can be evaluated.


Mandelbrot set - Firefox
File Edit View Go Bookmarks Tools Help
http://server/grid-mandelbı
Go
Getting Started
Latest Headlines
Done

Mandelbrot set - calculation running...

```
x: 1500 y: 1500xpart: 3ypart: 3
sudo cp /clusterwork/grid-mandelbrot/4.png /var/www/grid-mandelbrot/result/
```

jobnumber	left	right	top	bottom	Host	waiting since	runtime
1	-2.25	-1.25	-0.5	-1.5	server	Finished	11
2	-1.25	-0.25	-0.5	-1.5	karachi	Finished	21
3	-0.25	0.75	-0.5	-1.5	bombay	Finished	31
4	-2.25	-1.25	0.5	-0.5	server	Finished	21
5	-1.25	-0.25	0.5	-0.5	karachi	42	10
6	-0.25	0.75	0.5	-0.5	not_assigned	42	0
7	-2.25	-1.25	1.5	0.5	not_assigned	42	0
8	-1.25	-0.25	1.5	0.5	not_assigned	42	0
9	-0.25	0.75	1.5	0.5	not_assigned	42	0

```
server
/usr/bin/sudo -u knoppix /usr/local/globus-4.0.3/bin/globusrun-ws -submit -o /clusterwork/grid-mandelbrot/server.epr -b -F
https://server:8443/wsrf/services/ManagedJobFactoryService -c /clusterwork/grid-mandelbrot/6.sh 2>&1
Job in execution.
Submitting job...Done.
Job ID: uuid:2bfb9b30-45c6-11dc-99d5-000c2975b512
Termination time: 08/09/2007 15:43 GMT
bombay
/usr/bin/sudo -u knoppix /usr/local/globus-4.0.3/bin/globusrun-ws -submit -o /clusterwork/grid-mandelbrot/bombay.epr -b -F
https://bombay:8443/wsrf/services/ManagedJobFactoryService -c /clusterwork/grid-mandelbrot/7.sh 2>&1
Job in execution.
Submitting job...Done.
Job ID: uuid:2c65997c-45c6-11dc-aeb7-000c2975b512
Termination time: 08/09/2007 15:43 GMT
SELECT Job_ID FROM jobs WHERE Finished=-1
```

	5	6
7	8	9

Figure 5.5: *Web interface: Assigning and monitoring tasks*

5.3 Realization of the application management

The application management has been implemented in PHP in order to provide a web interface to the user. To control concurrent tasks, the management periodically reloads itself to keep the user up-to-date. After the user has requested a calculation, parameter related jobs will be determined and inserted into a database as depicted in Figure 5.6. If a node is idle in the Grid, a job will be assigned to this node by the management and executed on it through the GT4 service `globusrun-ws`. This service call includes the idle client and the application with its belonging and previously determined parameters. GT4 starts the submitted jobs as standard executables on the specified node. GT4 is allowed to execute tasks on other nodes in the Grid, because they have been certified in the Grid through GT4 during the boot process (see Section 2.4). Before starting the tasks on the nodes, the application itself has to be distributed to them. In this case study the application has been integrated in the Knoppix boot image in order to make it available on each client file system. Another possibility the copying of the application files to each node before the tasks have been assigned using the GridFTP service that is provided by GT4.

The management evaluates the file "/etc/bin/machines" to distinguish which clients are currently belonging to the Grid. After a node has been registered within the Grid, the file will be updated with the new node. This is possible, because a DHCP server and the IP-Collector are running on the master host, which are polling for new nodes and

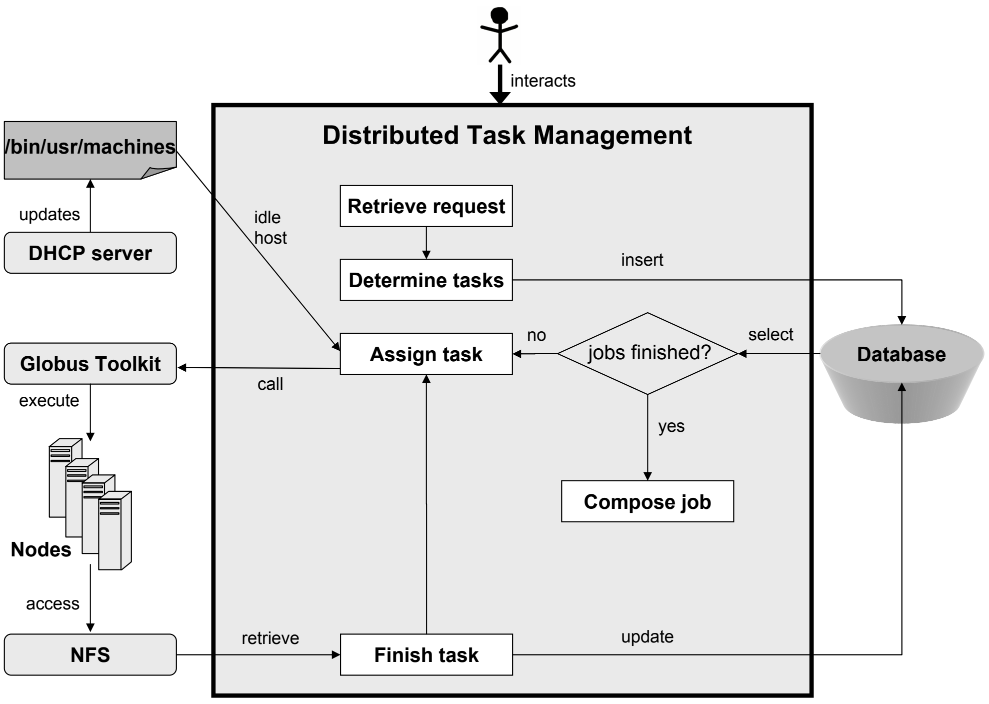


Figure 5.6: Distributed task management

node losses repeatedly. After the management has assigned a task, the task is marked as running in a database with the node identification. If the task has been completed, an output image file is written on an NFS folder, which every node can access. After the output has been recognized by the management, the related task will be marked as finished. If there is an unfinished task, this task will be submitted to an idle node.

The application management has to make sure that on each node only one task is running. Synchronization of the task is not important in that case, since every task is independent from the other tasks. A restrictive condition is that the final compose job, which is dependent on all previous jobs, has to be executed after all the other task have been finished. Since the management waits for all outputs before it submits the final compose job, that condition has been fulfilled.

While the application itself belongs to the layer “User Application”, the application management is part of the “Collective Services” layer as it is depicted in Figure 2.2. In addition to the “Collective Services” layer, services of the Instant-Grid, like directory brokering and diagnostics, which are provided automatically by the Instant-Grid, also have to be included. The services provided by the GT4 are included by the “Collective Services” layer. GT4 services also cover the “Resource and Connectivity Protocols” layer, since it is responsible for secure access to resources and services.

5.4 Technical specification

For determination of test data and test behavior for a software product, its specification has to be analyzed (see Chapter 3). The events that are handled by the Mandelbrot set application management are depicted in Figure 5.7 as an event-driven process chain. The user calls the web interface `http://server/grid-mandelbrot` as depicted in Figure 5.3, which is the first step in this process. After this web interface is loaded, the user has to enter the service parameters for the output image of the Mandelbrot set. These parameters include the resolution of the output image, i.e., the number of pixels in horizontal and vertical directions, and the number of parts in vertical and horizontal directions, which signify how the image will be divided by the management.

The management has to determine the parameters for each task for the Mandelbrot set function, as shown in Listing 5.1, from the service parameters that are provided by the user through the web interface. This determination is a prerequisite for the distribution and synchronization of the tasks. For controlling the distribution, each task with its belonging parameters and preferences is inserted into an SQL database. Before a task is distributed, the management builds a shell script that includes the application location and the belonging determined parameter. The application itself is located on each node in the folder `/usr/local/mandelbrot/` with the name `mandelbrot.py`. Each task receives an ID from the management in order to control the different tasks. The ID is included in the file name of the output of each task. After execution and correct termination of a task, its output is stored in the NFS folder `/clusterwork/grid-mandelbrot`, which is in the following called the working path or directory.

Technically, these shell scripts are submitted to the nodes for execution as tasks

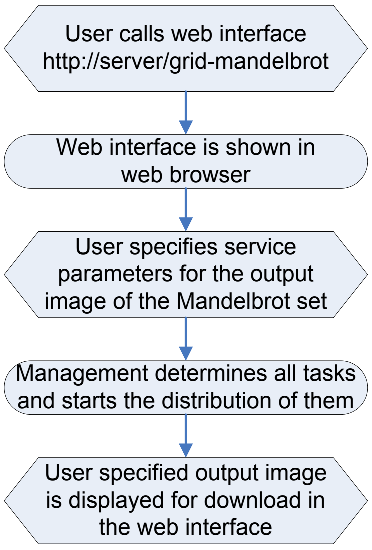

User calls web interface http://server/grid-mandelbrot
Web interface is shown in web browser
User specifies service parameters for the output image of the Mandelbrot set
Management determines all tasks and starts the distribution of them
User specified output image is displayed for download in the web interface

Figure 5.7: Event-driven process chain for the application management

by the management with help of the GT4 execution service. The shell scripts are also located in the working directory. The assigned nodes, wait times and run times of the tasks are inserted into the SQL database in order to handle timeouts and to provide a monitoring of the distribution progress to the user through the web interface. The image parts for the preview in the web interface are collected in the directory `/var/www/grid-mandelbrot/result/`.

After all jobs have been distributed, executed and terminated correctly on the nodes, an overall image is created from the results of all tasks that have been collected in the working directory. The file names of the task results consist of the task number and the file extension, for example `4.png`. The management considers this file as the output of task number four and it is, therefore, related to the fourth entry in the database. The composed image is saved in the folder `/var/www/grid-mandelbrot/result/` with the file name `mandelbrot.png`. Afterwards, a file named `finished` is created by the application management in the working directory in order to signalize that all jobs have been terminated correctly. The composed image is the final result and provided to the user by the web interface. To keep the management up-to-date, it reloads itself every 10 seconds in order to poll for new information about the determined and executed tasks.

Concluding, this application and its management can be realized more efficiently. The application and its management are kept simple, because this thesis concentrates on testing a Grid application using TTCN-3. For example, it is not possible to submit parallel user requests. For further developments of this Grid application many aspects can be improved.

6 Grid application testing

This chapter describes the development of test cases based on TTCN-3 for applications running in a Grid environment. In Section 6.1, the determination of the test items is described. Afterwards, in Section 6.2, the test purposes are determined, whereas in Section 6.3 their related test architectures and test behaviors are described. The TTCN-3 specifications of the test cases related to the determined test purposes are given in Section 6.4. Afterwards, the implemented test adapters are explained in Section 6.5, followed by a description of the execution of test cases in Section 6.6.

6.1 Determination of test items

The aim of this case study is to find out if TTCN-3 based tests are applicable for testing applications running in a Grid environment. The tests of the case study focus on the application output as well as the determination and the distribution of the tasks in order to make sure that the main aspects related to Grid computing are tested and covered through tests. This means the tests have to ensure that the Grid functionalities of the application work in the same way as it has been described in the specification in Chapter 5. This case study concentrates on functional testing that is realized in TTCN-3 using environments of TTworkbench Enterprise provided by Testing Technologies IST GmbH [57].

The tests in the scope of this thesis assume that GT4 runs in a proper way and that occurred failures are not related to it, but to the Grid application or its distribution management themselves. Hence, the tests applied in this case study include tests of the layers "User Application" and "Collective Services" and do not include the "Resources and Connectivity Protocols" and "Fabric" layers. Testing the Grid middleware GT4 itself will not be discussed, developed, or evaluated.

Several functional tests are thinkable and have to be evaluated in detail. These tests should include the check of the correctness of the distributed execution of the application in the Grid through the application management as well as the creation of an expected output. In addition, the tests have to check if the application produces the expected output while running on a specified node by using the distributed execution service of the Grid middleware GT4. Furthermore, tests of the correct determination of tasks and their distribution through the application management should be included.

The following test items have been determined:

1. Testing the application executed on a Grid node,

2. Testing the application management.

The test purposes and their related test cases are described in the following.

6.2 Test purposes

In this section, test purposes belonging to the above mentioned test items are determined. From the item “Testing the application executed on a Grid node” the test purpose TAG and from the test item “Testing the application management” the test purpose TAM are derived. Both test purposes are explained in the following, starting with test purpose TAG.

Test purpose TAG includes testing the Mandelbrot set application, which is executed on a Grid node. It checks if this application creates the correct output. This is important, because the user expects an accurate result. If the application itself behaves incorrectly, the application management would deliver a wrong output independent of the distribution.

The Implementation Under Test (IUT) of this test purpose is the Mandelbrot set application itself. The SUT includes besides the IUT, the Grid infrastructure provided by GT4 and NFS provided by Instant-Grid. The SUT can be stimulated through calling services of GT4, for example `globusrun-ws`, which should result in an expected output. The expected behavior of the SUT has to be determined. According to the Grid application scenario shown in Figure 5.4, a simple test sequence diagram, which shows the test purpose TAG, is depicted in Figure 6.1. The test system stimulates the SUT by submitting a job, whereupon the SUT returns the image as the result.

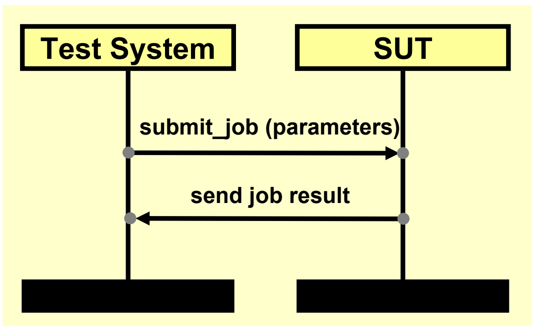


Figure 6.1: Test purpose TAG illustrated in a sequence diagram

For this test purpose, it is important that the application runs through the execution service of GT4 and produces a predefined result. It is not tested specifically if the job is executed on the specified node. The tests assume that the GT4 execution service runs properly as it is specified in the GT4 documentation [44]. Hence, if an expected output has been created, it is assumed that it was created by the specified node. More information about the execution service can be found in the Globus Toolkit documentation [44].

In the following, the test purpose TAM is described. Test purpose TAM includes testing the management of a Grid application. It checks if the management determines the tasks correctly and distributes all of them to the nodes. After the execution of all tasks on the nodes, a final output has to be created as expected. Testing the application management focuses on Grid functionalities, i.e., distributing all tasks and merging their outputs. Executing the application in a Grid computing environment without the application management would be no advantage against non-distributed computing. Therefore, the interaction of the management and the application is also tested. The test purpose comprises tests that check if the Grid user obtains a correct output after it is composed from all partial outputs of the tasks.

The application management, which runs on the master node, is the focus of this test purpose. The functionalities of the task distribution management have to be tested. The responsibilities of this management are: the retrieval of user request, the determination of tasks from that request, the assignment of the tasks to idling nodes in the Grid and the start of the final compose job after all tasks have been executed and terminated correctly. Hence, the application management is the IUT that has to be investigated. Figure 6.2 shows the determination of the IUT and the SUT based on Figure 5.6.

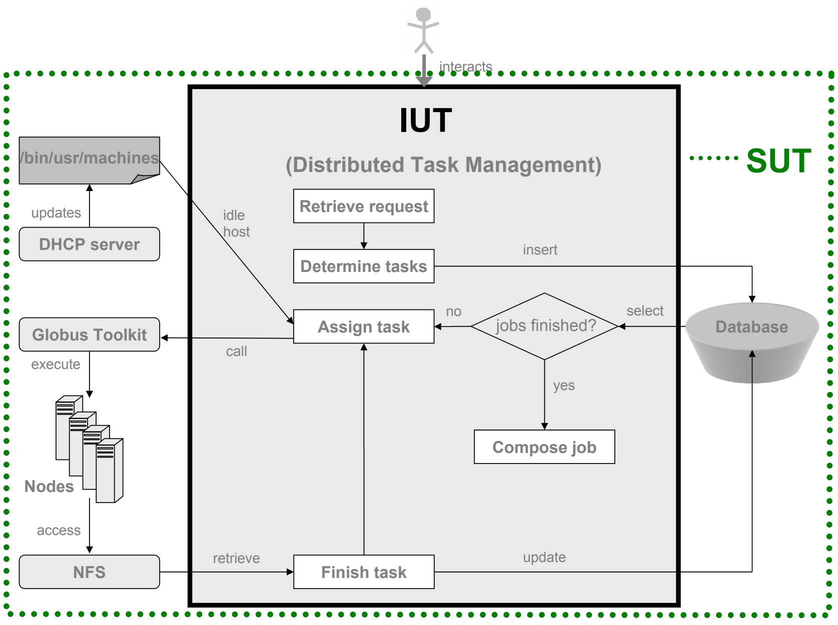


Figure 6.2: IUT and SUT specifications used in test purpose TAM

In addition to the IUT, the SUT comprises the Grid infrastructure and certain services. These include an SQL database, DHCP services, which update the file containing the node list, the GT4 infrastructure, which is provided to each node in the Grid and the NFS that each node can access.

The test purpose TAM is depicted in the sequence diagram in Figure 6.3. Initially, the test system sends a message for stimulation of the SUT (interaction 1). This stimulus includes the data, which is supposed to be entered by the user. The IUT requires this provided data in order to determine the task's options and parameters. After this determination, the IUT assigns the tasks to the nodes. In the test purpose that is depicted in the sequence diagram, the simulated users request results in the determination of two tasks and the additional final task that composes the result. After the submission, the

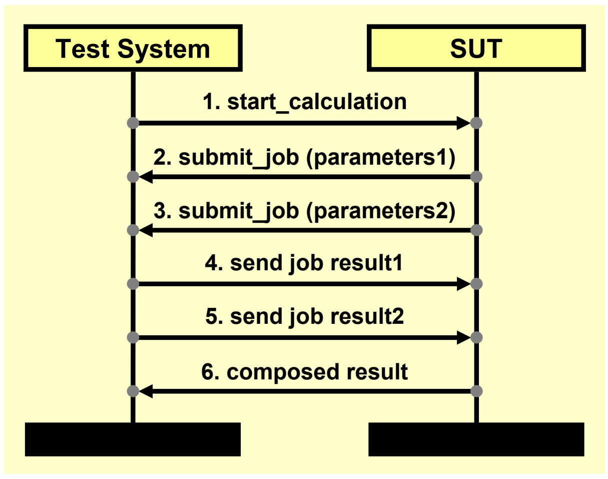


Figure 6.3: Test purpose TAM illustrated in a sequence diagram

tasks are intercepted by the test system in order to evaluate the task commands and their parameters (interaction 2 and 3). The test system creates an image file, which is sent back to the SUT, because the it expects a result from the task's assigned nodes (interaction 4 and 5). After all expected results have been received by the SUT, the final compose job is called by the SUT. The composed result is sent by the SUT to the test system (interaction 6). This test purpose is derived from the application behavior, which is shown in the sequence diagram of Figure 5.4. This test purpose has to be transformed into a test case as described in the next sections.

6.3 Test architectures and behaviors

The test architectures and test behaviors of the test purposes TAG and TAM are described in the following. The test architecture related to the test purpose TAG is shown in Figure 6.4, which is based on Figure 5.2. The SUT consists of the application itself, which is the IUT, the provided GT4 services and the NFS provided by the Instant-Grid.

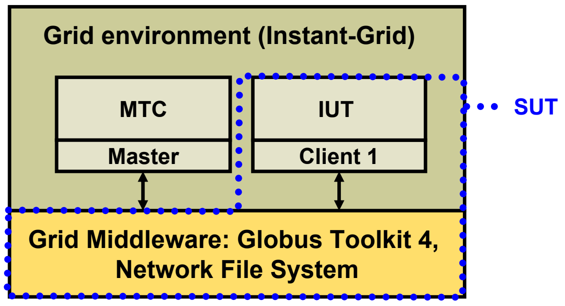

Grid environment (Instant-Grid)
MTC
Master
IUT
Client 1
SUT
Grid Middleware: Globus Toolkit 4, Network File System

Figure 6.4: Test architecture of test purpose TAG

The stimulation of the SUT occurs, when the submission of a job, which includes calling the execution service of GT4, is invoked through the MTC. This stimulation is the PoC of test purpose TAG, because the behavior and the result can be influenced by changing the input data. After sending that stimulation message, an output can be read from a specified folder by the MTC on the NFS. This is the PoO, since through the output, the behavior and results can be acquired and evaluated. The MTC has to poll for the output of the IUT, since it does not get a direct response from the IUT. The IUT is started with specified parameters by the execution service of GT4 and writes an output to the NFS. As described in Section 5.1, the output of the Mandelbrot set application is an image, which has to be matched with the data of the expected image. Keeping the match overhead small, an MD5 hash sum will be used to match the actual output image with the expected image. Since the MD5 sum is not collision free, an alternative for the matching is a comparison of the byte streams of both images. This comparison leads to test cases, which involve huge amounts of binary data.

As recapitulation, the stimulus of the SUT is the call of the GT4 execution service including the path, name and parameters of the application. The expected result is an image related to these parameters. The test case passes if the correct image is produced and stored into the NFS.

It is possible to add a PTC on client 1 in order to have a more precise test purpose that includes a testing of the Grid middleware GT4. This PTC could guard the executions on its installed client and send a message to the MTC if a certain application has been executed in order to determine if the application has been submitted to the correct node by the GT4 execution service. Since it is assumed that GT4 runs correctly, this issue is not considered in this test purpose.

For the test purpose TAM, a prerequisite is that at least two nodes are running in the Grid in order to test the distributing functionalities of Grid computing. The IUT, which in this case is the application management, is going to be stimulated with the necessary parameters by the MTC, which substitutes the user, as it is shown in Figure 6.5. The base of this figure is related to Figure 5.2 and shows how the test environment has been integrated into the Grid environment. The SUT comprises the IUT itself, the infrastructure provided by GT4 and the NFS.

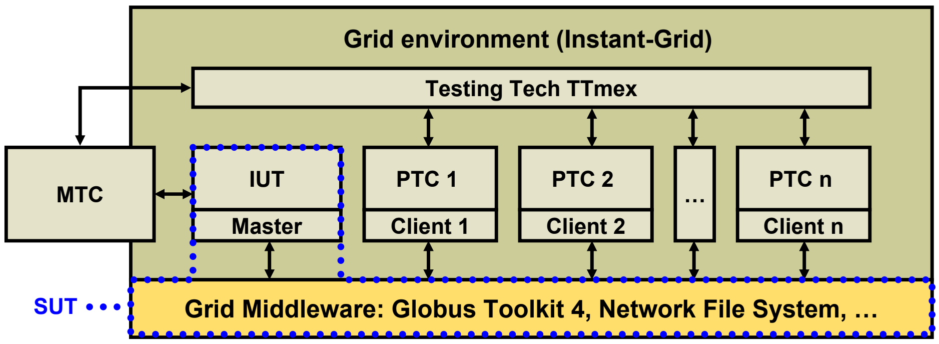

Grid environment (Instant-Grid)
Testing Tech TTmex
MTC
IUT
Master
PTC 1
Client 1
PTC 2
Client 2
...
PTC n
Client n
SUT
Grid Middleware: Globus Toolkit 4, Network File System, ...

Figure 6.5: Test architecture of test purpose TAM

After a stimulation by the MTC, the IUT is supposed to determine the tasks from the transmitted parameters. The determination is the premise for the distribution of the tasks by the IUT using the execution service of the Grid middleware GT4. In the tests, it has to be evaluated if the correct parameters for the tasks have been determined and submitted by the IUT.

Therefore, the application itself has been substituted through a stub in order to intercept the parameters that have been transmitted through the execution service of GT4. One stub runs on each node in order to replace the application that is located on each node. Additionally, a PTC runs on every node in order to receive the parameters from the stub. The PTC itself has to send these parameters to the MTC through the distributed TTworkbench functionality TTmex, which is based on CORBA [15, 32], because the PTC does not have the knowledge whether the received parameters are correct. TTmex provides its own infrastructure for the test system in order to allow a realization of SUT independent communication between test components. The integration of the TTmex functionality in the Grid environment is also shown in Figure 6.5.

The distribution of the tasks does not follow any restrictions, since the Mandelbrot set algorithm is parallelizable. It is not possible to predict which node executes which tasks distributed by the SUT. The reason behind this is that a node could have much more computing power than another one and, therefore, calculates its assigned task faster than a slow one. Therefore, the MTC has to collect all messages with the parameters received from the PTCs and has to evaluate if the correct parameters for each job have been determined by the SUT. It is also important that the MTC checks if the correct number of jobs has been executed. Additionally, the MTC has to check if the outputs of the tasks have been put together correctly through executing the final compose job.

The interactions between the SA and the TTCN-3 test system of test purpose TAM are depicted in Figure 6.6. The messages intercepted by the stubs are forwarded to the SA via a socket connection, whereas the SA sends these messages that include the application parameters further to the TTCN-3 test system. These messages are received by a port of the TTCN-3 test system. As responses of such messages, the SA gets image specifications that are sent via a port of the TTCN-3 test system in order to create image files. These image files are stored in the NFS working directory, which is used by the IUT. The image file creations are required, because the IUT waits for results of the assigned tasks.

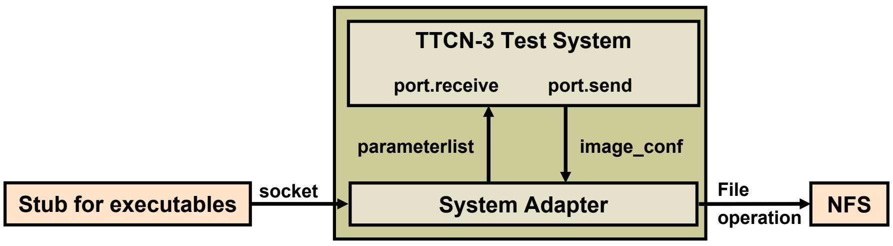


Figure 6.6: Communications in test test purpose TAM

According to the test purpose TAM in Figure 6.3, a simple graphical test case representation, which shows the expected behavior of the test purpose TAM, is depicted in Figure 6.7. This includes a simple test trace for an exemplified straightforward view on the behavior of test purpose TAM. This is the reason that the computer Grid comprises only 2 PTCs, where each runs on a client node. An arbitrary number of PTCs can be added. Additionally, this computer Grid contains the master node, on which the MTC and the IUT run.

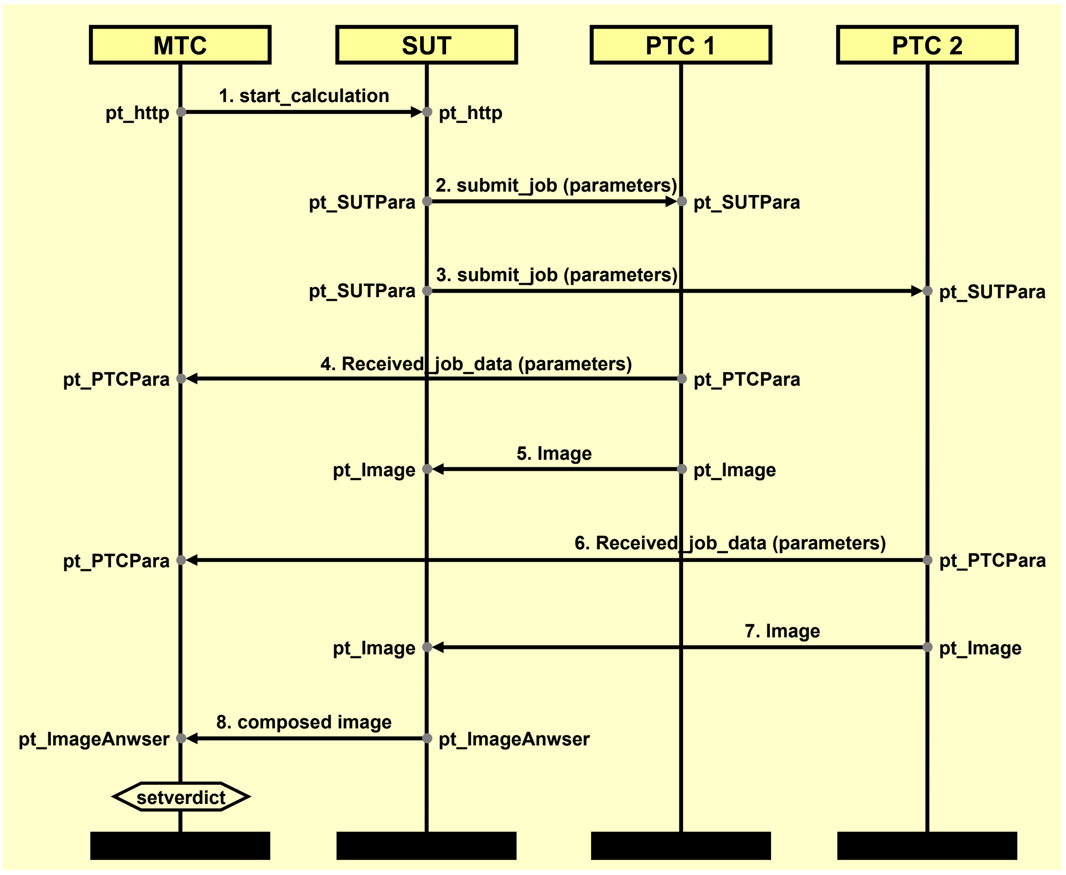


Figure 6.7: Expected behavior of test purpose TAM illustrated as a sequence diagram

In order to start a test, the MTC stimulates the SUT with a simulated user request that results in the determination of two jobs through the management (interaction 1). The stimulus is sent through the `pt_http` port of the MTC to the `pt_http` port of the SUT. After the subdivision of this imitated request through the SUT, the first job is submitted with its belonging parameters to PTC 1 (interaction 2) and the second with its calculated parameters to PTC 2 (interaction 3). The parameters observed by the stub are received as a message by the `pt_SUTParameters` port of a PTC and forwarded by the `pt_SUTParameters` port of the SUT. This message is further forwarded to the MTC by PTC 1 (interaction 4) and later by PTC 2 (interaction 6). After such a message has been forwarded, an arbitrary image is created by PTC 1 (interaction 5) and later by PTC 2 (interaction 7) using the parameters of the received message in order to feign an image creation to the SUT so that the SUT continues the task distribution. The image handling takes place through the `pt_Image` ports of the SUT and the PTCs. When all tasks are executed and terminated correctly, the SUT submits the final task to a node in the Grid, in order that the final output from all task results is going to be composed. Hence, the overall output is received by the `pt_ImageAnswer` port of the MTC (interaction 8). The MTC has to check if the correct number of tasks has been executed, if the job determinations match the expected ones in relation to the stimulus and if the output has been composed correctly.

6.4 TTCN-3 specification

This section describes the TTCN-3 implementation of the test cases. Hereby, test case `tc_TAG` is related to the test purpose TAG and test case `tc_TAM` is related to the test purpose TAM. The TTCN-3 data definitions are described in Subsection 6.4.1, whereas the TTCN-3 behaviors are explained in Subsection 6.4.2.

6.4.1 TTCN-3 data definitions

In Listing 6.1, the module parameter definition in TTCN-3 belonging to a test suite that includes test case `tc_TAG` is shown. The definition takes place within the curly braces of the TTCN-3 statement `modulepar`. The module parameter defined here has the type `charstring` and the name `HOST` with default content “bombay”, which is the host name of a node in the Grid (line 2). Through the command `with` (line 3), a description can be added to the module parameter (line 4). The module parameter can be changed after the TE has been compiled and can also be found in the Campaign Loader File (CLF), which is explained in Section 6.6 and depicted in Listing 6.23 (lines 7-24).

```
1  modulepar {
2      charstring HOST := "bombay";
3    } with {
4      extension (HOST) "Host on which the application runs." ;
5    }
```

Listing 6.1: Module parameter definitions used in test case `tc_TAG`

Listing 6.2 shows the definition of the module parameters that are used in test case `tc_TAM`. The seven module parameters are described in the following. The `NUMBER_OF_PTCS` signifies how many PTCs have to be started (line 2). This number has to be equal to the number of nodes registered in the Grid. Starting a PTC on each node is necessary, because the management can distribute the tasks to any of them. Therefore, it has to be ensured that every task that can be called on any node in the

Grid is observed by a PTC. In this example, the Grid consists of three nodes, whereas even on the master node a PTC runs, because tasks can also be executed on the master node itself.

```
1  modulepar {
2    integer NUMBER_OF_PTCS := 3;
3    integer HORIZONTAL := 5;
4    integer VERTICAL := 4;
5    float   SPEED_PTC1 := 2.0;
6    float   SPEED_PTC2 := 2.0;
7    float   SPEED_PTC3 := 2.0;
8    boolean COLORED := true;
9  } with {
10   extension (NUMBER_OF_PTCS) "Default number of PTCs";
11   extension (HORIZONTAL) "Number of horizontal pieces for the output image";
12   extension (VERTICAL) "Number of vertical pieces for the output image";
13   extension (SPEED_PTC1) "Reaction time of PTC 1";
14   extension (SPEED_PTC2) "Reaction time of PTC 2";
15   extension (SPEED_PTC3) "Reaction time of PTC 3";
16   extension (COLORED) "If true then each job is in the color of the ptc";
17 }
```

Listing 6.2: Module parameters definitions used in test case `tc_TAM`

The next two module parameters signify the two application management parameters that are entered by the user in the web interface. The number of parts, in which the image is going to be divided, is set in this case to 5 x 4. This means that the SUT calculates 20 tasks for the distribution (lines 3, 4).

The next three module parameters are defined for handling the reaction time of each PTC (lines 5-7). In this test suite the speeds of the PTCs are uniform. Through the PTC reaction times, nodes with different computing power can be simulated.

The `COLORED` module parameter is related to the color of a partial image created by a node through a task execution. If the `COLORED` module parameter is set to `true`, every image created by the same PTC is colored in the same predefined color. If this parameter is set to `false`, the partial image will be colored related to the job number and its place in the complete image (line 8). For the case that `COLORED` is set to `true`, it is not possible to predict the complete image with a hexadecimal representation, because the distribution of the tasks by the SUT cannot be determined exactly. If this option is set to `false`, this representation can be predicted and be matched with the one of the expected image. In lines 9 to 17, a short description of each module parameter is given in order to determine their meanings easily.

Template definitions used in test case `tc_TAG` are shown in Listing 6.3. For a better reusability, the templates are defined parameterized. The choice of parameterized templates was taken because in parameterized templates static content can be added easily.

The first template `a_GridClientAssign` belongs to the type `GridType` (lines 1-13).

```
1  template GridType a_GridClientAssign( Identification p_identification ,
2                                        charstring p_host ,
3                                        charstring p_path ,
4                                        charstring p_application ,
5                                        charstring p_parameters ,
6                                        charstring p_execType ) := {
7    identification  :=  p_identification ,
8    host            :=  p_host ,
9    path            :=  p_path ,
10   application     :=  p_application ,
11   parameters      :=  p_parameters ,
12   execType        :=  p_execType // hash or byte
13 }
14
15 template GridAnswerType a_GridClientAnswer( Identification p_identification ,
16                                             charstring p_Checksum ) := {
17   identification  :=  p_identification ,
18   Checksum        :=  p_Checksum
19 }
```

Listing 6.3: Template definitions used in test case `tc_TAG`

This template is used for sending messages to the SUT. This means a message contains a host that lives within the Grid (line 8), on which the specified application (line 10), which is located in the specified path (line 9) has to be invoked with certain parameters (line 11). A message includes, additionally, the identification number (line 7) in order to assure the correct synchronization of messages. The `execType` can be hash or byte, which signifies the method that is used for the image matching process (line 12). Depending on this, the output of the SUT is treated differently by the SA as further described in Section 6.5.

The second defined template `a_GridClientAnswer` is required for matching responses from the SUT (lines 15-19). This template includes the identification number that should be the same as the one that has been sent (line 17). The SUT requires a correctly processed identification number in order to differentiate the outputs of the tasks. The checksum that is used in this test case is an MD5 checksum and is defined as type `charstring` (line 18). Alternatively, for the byte-type option, the type `octetstring` has to be used.

The template definitions required for test case `tc_TAM` are shown in Listing 6.4. First, a template for the stimulus of the SUT is defined (lines 1-16). The template `a_httpStimulus` has two parameters that signify the number of parts of the image in vertical and horizontal direction (lines 1 and 2). Their product is the number of all parts in the image. The same number of jobs will be determined. The third parameter defines the working directory, where the IUT stores temporary data (line 3). This parameter is assigned in line 15. The protocol is specified as `http` (line 4), the host, in this case the web server, is a node inside the Grid (line 5) and the script, which has to be called, is the PHP script that computes the tasks inside the IUT (line 6).

```
 1 template StimulusType a_httpStimulus( integer p_horizontal,
 2                                         integer p_vertical,
 3                                         charstring p_workingpath) := {
 4     protocol             := "http",
 5     host                 := "server",
 6     file                 := "grid-mandelbrot/calculate.php",
 7     ResolutionXKey       := "xpixel",
 8     ResolutionXValue     := 1500,
 9     ResolutionYKey       := "ypixel",
10     ResolutionYValue     := 1500,
11     pcsHorizontalKey     := "xpart",
12     pcsHorizontalValue   := p_horizontal,
13     pcsVerticalKey       := "ypart",
14     pcsVerticalValue     := p_vertical,
15     workingpath          := p_workingpath
16   }
17
18 template ImageType a_Image (  charstring p_outputfile, integer p_red,
19                               integer p_green, integer p_blue,
20                               integer p_width, integer p_height) := {
21   outputfile  := p_outputfile,
22   red         := p_red,
23   green       := p_green,
24   blue        := p_blue,
25   width       := p_width,
26   height      := p_height
27 }
28
29 template ImageAnswerType a_ImageAnswer (charstring p_Checksum):= {
30   Checksum := p_Checksum
31 }
32
33 template ReactionType a_Reaction ( charstring p_application, float p_right,
34                                    float p_left, float p_top, float p_bottom,
35                                    integer p_red, integer p_green, integer p_blue,
36                                    charstring p_outputfile, integer p_resolutionX,
37                                    integer p_resolutionY) := {
38   application := p_application,
39   right       := p_right,
40   left        := p_left,
41   top         := p_top,
42   bottom      := p_bottom,
43   red         := p_red,
44   green       := p_green,
45   blue        := p_blue,
46   outputfile  := p_outputfile,
47   resolutionX := p_resolutionX,
48   resolutionY := p_resolutionY
49 }
```

Listing 6.4: Template definitions used in test case `tc_TAM`

Since the application management, i.e., the IUT, uses a web interface, CGI [36,92] variables have to be sent with the request. The CGI variables are defined in lines 7 to 14. Variables ending with `Key` in their names indicate that their contents are the names of CGI variables and the belonging variables ending with `Value` comprise the values of these CGI variables. For instance, the value of `ResolutionXKey` is `xpixel` (line 7) that is a CGI variable in the `calculate.php`. The content of the CGI variable xpixel is assigned to the value of ResolutionXValue that is 1500 (line 8). Typically, these variables are filled in the web interface by the user. But in this test case, the MTC has to send them. Therefore, the script calculate.php is called with these four CGI variables on the specified host, which is in this case the node with the host name server.

The SUT expects four inputs from the user's request, which is simulated through the call of the MTC. The record type StimulusType has to be adjusted or a new record type has to be defined if the management expects more than four CGI parameters.

Second, the template a_Image that includes the characteristics of an image is specified (lines 18-27). It includes attributes for defining a simplified image. These attributes are the file, in which the image is written (line 23), the color of the image (lines 24-26) and the resolution of the image (lines 27, 28). This template is completely parameterized in order to allow a maximum flexibility of image creations. The template is used for sending messages to the SUT.

Third, the template a_ImageAnswer of type ImageAnswerType is required for matching a checksum of a file. The checksum in this case is an MD5 checksum and is provided through a parameter (lines 29-31). The type record ImageAnswerType is used in order to allow adding other attributes of a file. Hence, an addition of attributes, like location, size, file type, or file name, is possible.

Fourth, the template a_Reaction for matching received messages from the SUT is defined (lines 33-49). This template is based on the parameter list for the Grid application. It includes the application name and location (line 38), the specification for the part of the Mandelbrot set (right, left, top, bottom, lines 39-42), the color base for the Mandelbrot set in RGB model representation (lines 43-45), the output file (line 46) and the resolution of the local image (lines 47, 48). This template is parameterized because more than one message is expected from the SUT. Hence, a parameterized template allows a better structure in the test behavior specification because of its reusability.

The port type defined for test case tc_TAG is listed in Listing 6.5. This port type is called GridPort and uses message-based communication (line 1). The types of messages that can be sent or received by a port that is declared in this type are defined in lines 2 and 3. A port instance of this port type would have a direction out for sending messages of type GridType and a direction in, which means that possible received messages should match the type GridAnswerType.

```
1  type port GridPort message {
2    out GridType;
3    in GridAnswerType;
4  }
```

Listing 6.5: Port type definition used in test case tc_TAG

The port type definitions required for test case tc_TAM are shown in Listing 6.6. Four message based port types are declared. The port type httpType with the direction out is defined in lines 1 to 3. A port of this port type can send messages of type StimulusType.

The port type `AppReactionType` is defined with an `inout` direction in order to allow a port of this type receiving and sending messages of type `ReactionType` (lines 5-7). A port defined with the type `CreateImageType` can send messages of type `ImageType`. Therefore, for this type, a direction `out` is assigned (lines 9-11). A port of port type `CreateImageAnswerType` evaluates received messages of type `ImageAnswerType` (lines 13-15).

```
 1  type port httpType message {
 2    out StimulusType;
 3  }
 4
 5  type port AppReactionType message {
 6    inout ReactionType;
 7  }
 8
 9  type port CreateImageType message{
10    out ImageType;
11  }
12
13  type port CreateImageAnswerType message {
14    in   ImageAnswerType;
15  }
```

Listing 6.6: Port type definitions used in test case `tc_TAM`

The TTCN-3 definition of components, including their belonging ports used in test case `tc_TAG`, are shown in Listing 6.7. Port `pt_mtc` of type `GridPort` belongs to the component type `GridClient` (lines 1-3), which is the MTC type of the following described test case. Port `pt_system`, also of type `GridPort`, belongs to the component type `systemType` (lines 5-7), which can be seen as the abstract interface to the SUT. These components are going to communicate with each other through these ports using the types described above.

```
1  type component GridClient {
2    port GridPort pt_mtc;
3  }
4
5  type component systemType {
6    port GridPort pt_system;
7  }
```

Listing 6.7: Component and port definitions used in test case `tc_TAG`

Figure 6.8 shows the abstract realization of the port connection in test case `tc_TAG`. The `pt_system` port can be seen as the interface between the ATSI and the RTSI that includes the SA, as described in Section 4.1. The RTSI interacts through the ATSI with the TTCN-3 test system in order to realize a communication between the SUT and the TTCN-3 test systems.

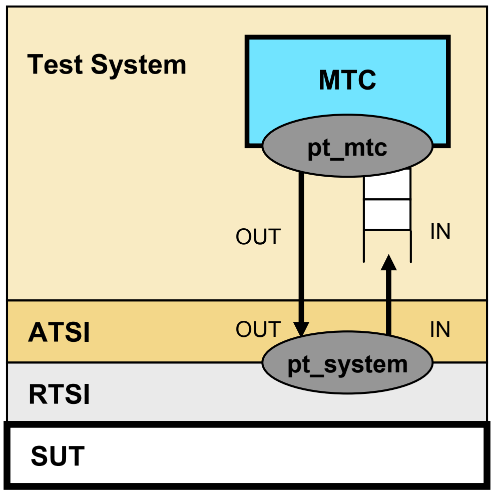


Figure 6.8: Port realization of test case `tc_TAG`

The test component types and their ports defined for test case `tc_TAM` are shown in Listing 6.8. The definition of component type `MTCType` includes three port and two timer definitions (lines 1-7). The defined ports of the `MTCType` component type are the `pt_http` port that has to send the stimulus to the web interface of the IUT (line 2), the `pt_PTCParameters` port in order to receive messages with the task parameters from the PTCs (line 3) and the `pt_ImageAnswer` port in order to receive a message with the final composed image via the file system from the IUT (line 4). The `MTCTimer` is defined with a timeout of 50 seconds in order to guard `alt` constructs in the test behavior (line 5). The `WaitForCompletionTimer` is used in the MTC behavior to wait 40 seconds for further messages, after all expected messages have been received (line 6). This timer is required because it has to be ensured that the correct number of jobs has been submitted.

The definition of the component type `PTCType` is used for executing parallel test behavior in order to observe the task submissions on several nodes in the Grid. The definition of component type `PTCType` includes three port and two timer definitions (lines 9-15). The defined ports of the `PTCType` component type are the `pt_SUTParameters` port in order to receive the task parameters from the system component (line 10), the `pt_PTCParameters` port that sends the received task parameters further to the MTC (line 11) and the `pt_Image` port that initializes a creation of an image for the SUT (line 12). The `PTCTimer` is defined with a timeout of 50 seconds in order to guard `alt` constructs in the PTC test behavior (line 13). The `PTCApplicationTimer` is used in the PTC behavior to simulate the response time of the node on which the PTC runs.

```
 1 type component MTCType {
 2   port httpType pt_http;
 3   port AppReactionType pt_PTCParameters;
 4   port CreateImageAnswerType pt_ImageAnswer;
 5   timer MTCTimer := 50.0;
 6   timer WaitForCompletionTimer := 40.0;
 7 }
 8
 9 type component PTCType {
10   port AppReactionType pt_SUTParameters;
11   port AppReactionType pt_PTCParameters;
12   port CreateImageType pt_Image;
13   timer PTCTimer := 50.0;
14   timer PTCApplicationTimer;
15 }
16
17 type component SystemType {
18   port httpType pt_http;
19   port AppReactionType pt_SUTParametersArray[NUMBER_OF_PTCS];
20   port CreateImageType pt_ImageArray[NUMBER_OF_PTCS];
21   port CreateImageAnswerType pt_ImageAnswer;
22 }
```

Listing 6.8: Component and port definitions used in test case `tc_TAM`

The component type `SystemType` is the abstract interface to the SUT. This component type comprises four ports (lines 17-22). The `pt_http` port is supposed to receives the HTTP request as stimulus for the SUT (line 18). The `pt_SUTParametersArray[NUMBER_OF_PTCS]` port sends a message with the intercepted task parameters to the PTC (line 19). The `pt_ImageArray[NUMBER_OF_PTCS]` port receives attributes from the PTC for creating an image (line 20). The `pt_ImageAnwserType` port sends a message with the specifications of the final composed image to the MTC (line 21).

An overview of the communication flow between the components and their defined ports used in test case `tc_TAM` is given in Figure 6.9. The SUT is stimulated through a message sent by the `pt_http` port of the MTC. This stimulation message is of type `StimulusType` and is sent `out` of the TTCN-3 test system to the SUT. After the stimulation, the IUT determines all tasks. Once the tasks have been submitted to the nodes, i.e., in the test system to the PTCs, the PTCs receive messages with the parameters of the task submissions from the system component port `pt_SUTParametersArray[n]` by their port `pt_SUTParameters`. This message of type `ReactionType` is forwarded to the MTC port `pt_PTCParameters` by the PTC ports `pt_PTCParameters`. After a specified amount of time, an image creation is initialized by the PTC port `pt_Image` through sending an image specification message to the system component port `pt_ImageArray[n]`. The message for the image creation is of type `ImageType` and is sent `out` of the TTCN-3 test system. When the MTC port `pt_PTCParameters` has received all expected messages from the PTCs, the MTC typically receives a message with attributes of the composed image through its port `pt_ImageAnswer` from the system component port `pt_ImageAnswer`. This message is received from outside the test system and is of type `ImageAnswerType`.

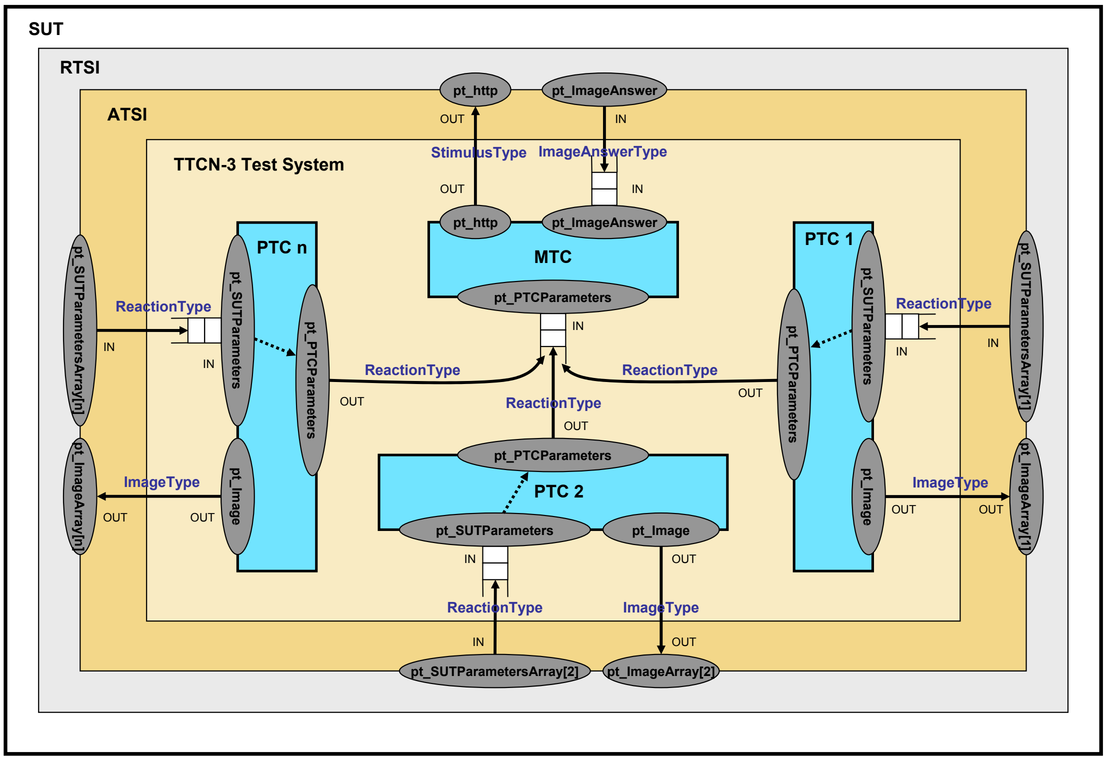


Figure 6.9: Ports realization in test case `tc_TAM`

6.4.2 TTCN-3 behavior definitions

The behavior of test case `tc_TAG` implemented in TTCN-3 is shown in Listing 6.9. This test case uses an MTC of component type `GridClient` that is specified by the TTCN-3 **runs on** clause (line 1). Additionally, a system component of component type `systemType` is used and indicated by the TTCN-3 operation `system` (line 1).

First, timer and certain variables are declared (lines 2-15). The variables are the abstract parameters for the GT4 execution service. In line 5, an arbitrary identification is assigned. Afterwards, the working path, where temporary data is stored by the IUT (line 7), the application name that will be executed (line 9) and its belonging parameters (lines 11, 12) are defined. Also, one part of the expected response, which is the expected MD5 hash code, is assigned (line 15). After these initializations, port `pt_mtc` is mapped to the port `pt_system` (line 17) in order to receive or send messages between the MTC and the SUT. The technical realization of the communication is given in Section 6.5.

The message that is sent in line 20 includes the variables assigned above and the host identification. The host identification is provided as a TTCN-3 module parameter,

```
 1  testcase tc_TAG() runs on GridClient system systemType {
 2    timer replyTimer;
 3
 4    // variables for the SUT stimulus
 5    var Identification v_id := 1234;
 6    // working path
 7    var charstring v_path := "/clusterwork/grid-mandelbrot/";
 8    // application for execution
 9    var charstring v_application := "/usr/local/mandelbrot/mandelbrot.py ";
10    // parameters for the application
11    var charstring v_parameters := "-2.25 0.75 -1.5 1.5 255 125 125 "
12      & v_path & int2str(v_id) & ".png 500 500";
13
14    // expected part of the response
15    var charstring v_hashsum := "8dbdbdb61381636c5897347f6888366b";
16
17    map(mtc:pt_mtc, system:pt_system);
18
19    // send the Stimulus
20    pt_mtc.send( a_GridClientAssign(  v_id,  HOST, v_path, v_application,
21                                      v_parameters, "hash" ) );
22
23    replyTimer.start( 200.0 );
24    alt {
25      // handle the case when the expected answer is received.
26      []  pt_mtc.receive( a_GridClientAnswer( v_id, v_hashsum ) ) {
27          replyTimer.stop;
28          setverdict( pass );
29      }
30      // Handle the case when unexpected answer is received.
31      []  pt_mtc.receive {
32          replyTimer.stop;
33          setverdict ( fail );
34      }
35      // Handle the case when no anwser is received.
36      [] replyTimer.timeout {
37          setverdict( fail );
38      }
39    }
40    unmap(mtc:pt_mtc, system:pt_system);
41    stop;
42  }
```

Listing 6.9: Test behavior implementation of test case `tc_TAG`

which can thus be changed after the TTCN-3 compilation process. It specifies the node, on which the application has to be executed. Since the expected response could be an MD5 hash sum or a hexadecimal representation of the output, the message includes the type information "hash". After this stimulus has been sent (line 20), the timer is started with a timeout of 200 seconds (line 23), which guards the following `alt` construct (lines 24-39).

An `alt` statement means that several different alternatives of behavior can take place at a given time. Hence, an `alt` construct that contains several alternatives like in

Listing 6.9 (lines 24-39) blocks until any one of its alternatives matches [40]. In this construct, one alternative is that the port `pt_mtc` waits for a message from the system component port `pt_system` (line 26). After receiving a message at the `pt_mtc` port, it has to be evaluated if the expected message has been received or not (lines 26, 31, 36). If the message matches the template `a_GridClientAnswer(v_id, v_hashsum)` with its belonging and above defined and assigned parameters, the timer will be stopped (line 27) and the local verdict will be set to `pass` (line 28). If a message is received that does not match the template `a_GridClientAnswer(v_id, v_hashsum)` (line 31), the timer will be stopped (line 32) and the local verdict will be set to `fail` (line 33). If no message is received by the MTC port `pt_mtc`, the verdict will be set to `fail` (line 37) after the `timeout` of 200 seconds has been reached (line 36). After the `alt` construct has been left, the two mapped ports will be unmapped (line 40) and all components will be stopped (line 41). The global verdict will be set by evaluating the local verdict, as briefly described in Section 4.2.

The first part of the behavior of test case `tc_TAM` is shown in Listing 6.10. This includes variable definitions, the initiation of the PTCs and the start of their behaviors. This test case runs on an instance of component type `MTCType` and on an instance of component type `SystemType` as an abstract interface to the SUT (line 1).

```
1  testcase tc_TAM() runs on MTCType system SystemType {
2
3      // define application working path
4      var charstring v_workingpath := ”/clusterwork/grid-mandelbrot/”;
5
6      // uniform task parameters
7      var charstring v_application := ”/usr/local/mandelbrot/mandelbrot.py”;
8      var integer v_resolutionX := 300;
9      var integer v_resolutionY := 375;
10     var integer v_red := 255;
11     var integer v_green := 125;
12     var integer v_blue := 125;
13
14     // total job number
15     var integer v_jobnumber := HORIZONTAL*VERTICAL;
16
17     // PTC handling
18     var PTCType PTCArray[NUMBER_OF_PTCS];
19
20     // variables for support of the evalustion of the PTC messages
21     var boolean v_check := false;
22     var boolean v_overall_check := true;
23     var ReactionType JobArray[HORIZONTAL*VERTICAL];
24     var boolean checkArray[HORIZONTAL*VERTICAL];
25     var ReactionType v_Parameters := null;
26
27     // for iteration purposes
28     var integer i:=0, j := 0;
29
30     // initiate the array
31     for (i:=0; i<v_jobnumber; i:=i+1) {
32       checkArray[i] := false;
33       JobArray[i] := setup_Jobarray( v_application, v_red, v_green, v_blue,
34                                      v_resolutionX, v_resolutionY, i,
35                                      v_workingpath);
36       }
37
38       // map mtc ports with system component ports
39       map (mtc:pt_http, system:pt_http);
40       map (mtc:pt_ImageAnswer, system:pt_ImageAnswer);
41
42       // initiate and start the PTCs
43       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
44         // create the PTCs
45         PTCArray[i] := PTCType.create;
46       }
47       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
48         // map the PTC ports to the system ports
49         map (PTCArray[i]:pt_SUTParameters, system:pt_SUTParametersArray[i]);
50         map (PTCArray[i]:pt_Image, system:pt_ImageArray[i]);
51         // connect the ptc to the mtc port
52         connect (self:pt_PTCParameters, PTCArray[i]:pt_PTCParameters);
53       }
54       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
55         // create color for each PTC
56         var Color v_color;
57         var integer variants := NUMBER_OF_PTCS;
58         var integer v_switch := i mod variants;
59         var float v_speed;
60         if (v_switch == 0) {
61           v_color := {0, 125, 255};
62           v_speed := SPEED_PTC3;
63         } else if (v_switch == 1) {
64           v_color := {0, 255, 125};
65           v_speed := SPEED_PTC2;
66         } else if (v_switch == 2) {
67           v_color := {125, 0, 125};
68           v_speed := SPEED_PTC1;
69         }
70         // start the PTC's behaviors
71         PTCArray[i].start(PTCBehavior(  v_application, v_red, v_green, v_blue,
72                                         v_resolutionX, v_resolutionY, i,
73                                         v_speed, v_color));
74       }
75
76     ... // Part two is shown in Listing 6.12
77
78   }
```

Listing 6.10: Test behavior implementation of test case `tc_TAM` (Part one)

First of all, the working path of the IUT is defined (line 4). The variables, declared and assigned from lines 7 to 12, are the expected parameters of the tasks. These parameters are determined from the stimulus through the CGI variables assigned in the template `a_httpStimulus(integer, integer, charstring)`. The number of expected tasks is determined in line 15. Afterwards, the array `PTCArray[NUMBER_OF_PTCS]` of component type `PTCType` is defined (line 18). The size of this array corresponds to the number of PTCs required in this test case. The two variables `v_check` and `v_overall_check` (lines 21, 22) guard the later evaluation of the messages received from the PTCs. The arrays `JobArray` and `checkArray` (lines 23, 24) are required for handling and evaluation of all messages from the PTCs. The variable `v_Parameters` (line 25) is used for temporarily saving received messages of this type. The variables i and j are defined for iteration purposes (line 28).

In the for loop in lines 31 to 36, the two arrays are initialized. The checkArray is related to the JobArray and, therefore, in the checkArray received messages are marked as true. Hence, this array is initialized with false values (line 32). The array JobArray includes all expected messages from the PTCs. It is initialized with the parameterized function setup_JobArray, where all expected messages have been determined (lines 33-35).

The pt_http port of the MTC is mapped to the system component port pt_http (line 39). Afterwards, the pt_ImageAnswer port of the MTC is mapped to the system component port pt_ImageAnswer (line 40).

In the for loop from line 43 to 46, the PTCs are initialized using the TTCN-3 operation create in order to assign them to the array PTCArray[NUMBER_OF_PTCS] (line 45). The ports of each PTC are either mapped or connected in the following for loop (lines 47-53) (see also Figure 6.9). The port pt_SUTParameters of a PTC is mapped to the pt_SUTParametersArray[i] port of the system component (line 49). The port pt_Image of a PTC is mapped to the pt_ImageArray[i] port of the system component (line 50). Additionally, the port pt_PTCParameters of the MTC is connected to the port pt_PTCParameters of each created PTC (line 52). The third for loop (lines 54-74) determines different colors of the images created by the PTCs (lines 60-69) and starts the behavior for each PTC using the TTCN-3 operation start (lines 71-73). The PTC behavior is realized as a parameterized TTCN-3 function and further explained in the following paragraph. The test behavior implementation of test case tc_TAM part two includes the evaluation of the received PTC messages and is explained after the description of the PTC behavior.

The behavior of a PTC, realized as a function that runs as an instance of the component type PTCType, is shown in Listing 6.11. The parameters of the function PTCBehavior include the application name, the color represented as red, green and blue of an image, the image's resolution, the identifier of the component, a timeout and the color of the feigned image. The identifier p_componentId is required in order to distinguish the behaviors of different PTCs.

After the declaration of required variables, an infinite loop is started in order to wait for responses from the SUT. The timer PTCTimer, defined as component type PTCType with a timeout of 50 seconds, is started in line 9 in order to guard the following alt construct. In the alt construct, three alternatives are specified. In the first and second alternative, the port pt_SUTParameters is observed. The third alternative waits for a timeout of the PTCTimer (line 47). The first alternative checks if a received message matches the parameterized template a_Reaction(p_application, ?, ?, ?, ?, p_red, p_green, p_blue, ?, p_resolutionX, p_resolutionY), whereas the ?-parameters represent wild cards (lines 11-13). The content of this received message is assigned to the variable v_Reaction for temporary storage (line 14). The PTCTimer is stopped with the TTCN-3 timer operation stop (line 15), because the expected message has been received within the timeout. The

```
 1 function PTCBehavior( charstring p_application,
 2                       integer p_red, integer p_green, integer p_blue,
 3                       integer p_resolutionX, integer p_resolutionY,
 4                       integer p_componentId,
 5                       float p_timeout, Color p_color) runs on PTCType {
 6   var ReactionType v_Reaction := null;
 7   var integer v_jobnumber;
 8   while (true) {
 9     PTCTimer.start;
10     alt {
11       [] pt_SUTParameters.receive(a_Reaction( p_application, ?, ?, ?, ?,
12                                               p_red, p_green, p_blue, ?,
13                                               p_resolutionX, p_resolutionY))
14                                               -> value v_Reaction {
15         PTCTimer.stop;
16
17         // send received message for evaluation to the MTC
18         pt_PTCParameters.send(v_Reaction);
19
20         // wait until giving response to the SUT
21         PTCApplicationTimer.start(p_timeout);
22         PTCApplicationTimer.timeout;
23
24         // send image specifiaction message to the system component
25         if (COLORED) {
26           pt_Image.send( a_Image( v_Reaction.outputfile, p_color.red,
27                                   p_color.green, p_color.blue,
28                                   p_resolutionX, p_resolutionY));
29         } else {
30           v_jobnumber := ex_subString(v_Reaction.outputfile);
31           if ((v_jobnumber mod 3) == 0) { // every third pic should be black
32             pt_Image.send(a_Image(  v_Reaction.outputfile, 0, 0, 0,
33                                     p_resolutionX, p_resolutionY));
34           } else if ((v_jobnumber mod 3) == 1) { // gray
35             pt_Image.send(a_Image(  v_Reaction.outputfile, 125, 125, 125,
36                                     p_resolutionX, p_resolutionY));
37           } else { // white
38             pt_Image.send(a_Image(  v_Reaction.outputfile, 255, 255, 255,
39                                     p_resolutionX, p_resolutionY));
40           }
41         }
42       }
43       [] pt_SUTParameters.receive {
44         PTCTimer.stop;
45         setverdict(fail);
46       }
47       [] PTCTimer.timeout {
48         setverdict(fail);
49       }
50     }
51   }
52 }
```

Listing 6.11: PTC behavior implementation of test case `tc_TAM`

variable `v_Reaction` is sent by the port `pt_PTCParameters` to the MTC in order to forward the received message for evaluation of all messages through the MTC (line 18).

The simulation of the response time of the nodes is realized with the `PTCApplicationTimer` (line 21). The PTC waits until the timeout of this timer occurs (line 22) and sends, afterwards, an image specification to the SA for creating an image (lines 25-41). The image specifications differ in the color. If the module parameter `COLORED` is set to `true`, the `then` branch is taken (lines 25-28). The color is predefined by the parameter `p_color` and is sent with the other entries of the image specification through the port `pt_Image` using the parameterized template `a_Image`. Each image has a predefined color that corresponds to the PTC, which has created the image. If the module parameter `COLORED` is set to `false`, the `else` branch is taken (lines 29-41). The color of the image is then dependent on the job number. Therefore, the job number is extracted from the name of the output file (line 30) that includes the number of the task. This is realized with the external TTCN-3 function `ex_subString`. In the final composed image, every third part is dependent on the job number colored in black, white, or gray. The color is also integrated in the parameterized template `a_Image` and sent via port `pt_Image` to the SUT (lines 32, 35, 38).

If a message is received that does not match the parameterized template `a_Reaction(p_application, ?, ?, ?, ?, p_red, p_green, p_blue, ?, p_resolutionX, p_resolutionY)`, the second alternative is taken (lines 43-46). Since other messages from the SUT are not expected, the `PTCTimer` is stopped (line 44) and the verdict of this component is set to `fail` (line 45).

If no messages are received within the timeout of timer `PTCTimer`, the third alternative of the `alt` construct is taken (lines 47-49) and the verdict of this PTC is set to `fail`.

This PTC behavior polls for new messages from the SUT. After the PTC has received a message, the message will be evaluated and forwarded to the MTC. The PTC behaviors run parallel to the MTC behavior, which is further explained in the following paragraph.

Listing 6.12 shows the second part of the source code for the MTC behavior of test case `tc_TAM`. After the PTC behaviors have been started as described in the previous paragraphs, a variable for default behavior is defined. Additionally, the variable `i` for iteration purposes is reset (line 7).

```
 1 testcase tc_TAM() runs on MTCType system SystemType {
 2
 3   ... // Part one is shown in Listing 6.10
 4
 5   // init
 6   var default v_receiveAny := activate(alt_receiveAny (pt_PTCParameters));
 7   i := 0;
 8
 9   // send the Stimulus
10   pt_http.send(a_httpStimulus(HORIZONTAL, VERTICAL, v_workingpath));
11
12   // check received messages
13   while(i<HORIZONTAL*VERTICAL) {
14     alt {
15       [] pt_PTCParameters.receive(a_Reaction( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?))
16           -> value v_Parameters {
17         v_check := false;
18         for (j:=0; j<HORIZONTAL*VERTICAL; j:=j+1) {
19           // check if the received message is in the array
20          if (v_Parameters == JobArray[j]) {
21            log("matched ReactionType");
22            checkArray[j] := true;
23            v_check := true;
24            i:=i+1;
25          }
26        }
27        // if the message does not match, testcase failed
28        if (v_check == false) {
29          log("mismatch");
30          v_overall_check := false;
31          setverdict(fail);
32          i:=HORIZONTAL*VERTICAL;
33        }
34      }
35    }
36  }
37
38  deactivate(v_receiveAny);
39
40  // check if to many messages were receiced
41  if (v_overall_check != false) {
42    WaitForCompletionTimer.start;
43    alt {
44      [] pt_PTCParameters.receive {
45        log ("incorrect number of messages received");
46        setverdict(fail);
47      }
48      [] WaitForCompletionTimer.timeout {
49        log ("correct number of messages received");
50      }
51    }
52  }
53
54  // check if every message has been received
55  if (v_overall_check != false) {
56    for (i:=0; i<HORIZONTAL*VERTICAL; i:=i+1) {
57      if (checkArray[i] == false) {
58        log(i);
59        log("message is missing");
60        v_overall_check := false;
61        setverdict(fail);
62      }
63    }
64  }
65
66  // check if the correct image has been created
67  checkImage( MTCTimer, pt_ImageAnswer);
68
69  stop;
70 }
```

Listing 6.12: Test behavior implementation of test case `tc_TAM` (Part two)

The stimulus is sent in line 10 by the port `pt_http` from the MTC to the SUT. This stimulus is specified by the parameterized template `a_Stimulus(integer, integer, charstring)`, which uses the module parameters `HORIZONTAL` and `VERTICAL` and the variable `v_workingpath` that have been described previously.

The `while` loop in lines 13 to 36 adapts the concepts of the TTCN-3 `interleave` operation. Since the messages received from the PTCs do not have to be received in a certain order and the number of involved PTCs or ports is not known at compile time, a mechanism for the reception of messages from different PTCs has been realized in this loop. For this purpose, an `alt` construct is used in order to receive messages through the port `PTCParameters`. The first alternative waits for messages of template `a_Reaction(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)` (line 15). The correct number of parameters and the correct corresponding types of the parameters are required by the message in order to be accepted by this `receive` branch. The received message is saved in the variable `v_Parameters` for further handling and evaluation (line 16). After receiving such a message, the guard variable `v_check` is reset to `false` (line 17). The `for` loop in lines 18 to 26 compares the received message with the messages stored in the array `JobArray`. If the received message stored in the variable `v_Parameters` matches with one of these messages (line 20), the message related to the iterator `j` is marked for later evaluations in the array `checkArray` (line 22). Additionally, the guard variable `v_check` is set to `true` (line 23). After the iteration through all expected messages, the guard variable `v_check` has to be evaluated (lines 28-33). If this variable was not set to `true`, the received message does not match any expected message. Since other messages than the messages stored in the array `JobArray` are not expected at this time, the verdict is set to `fail` in this case (line 31). Therefore, the `while` loop is stopped through increasing the iterator `i` to the maximum `HORIZONTAL*VERTICAL` (line 32). The variable `v_overall_check` is also set to `false` (line 30) in order to jump over the final evaluations. Because the altstep `v_receiveAny` is not required anymore, it is deactivated in line 38.

The final evaluations are required when the previous `while` loop has been passed as expected. In lines 41 to 52, the port `pt_PTCParameter` is observed in order to ensure that no further messages are received after passing the `while` loop. If a message is being received within the timeout, the verdict is set to `fail`. If no message has been received, the evaluation can be continued.

Afterwards, in lines 55 to 64, it is checked through the evaluation of the array `checkArray` if every expected message has been received. If each entry of the array is set to `true`, the check is passed and the number of expected messages has been received. If at least one entry with `false` is detected, the verdict is set to `fail`.

At last, it is checked if the final image has been composed (lines 67-81). The evaluation is done by the function `checkImage(MTCTimer, pt_ImageAnswer)`, which compares an image specification, received within the `MTCTimer` timeout, with the expected specification. This function is dependent on the module parameter `COLORED`.

Recapitulatory, test case `tc_TAM` checks if the parameters of the tasks have been determined by the SUT as expected. It observes if these tasks are submitted to the nodes and executed on them as expected. This observation includes the check whether the correct number of tasks have been called and whether the final task delivers an expected output.

6.5 Adaptation layer

To make the TTCN-3 the test cases `tc_TAG` and `tc_TAM` executable, certain entities of the adaptation layer have to be implemented. The adaptation layer includes the entities SA, PA and CD that are further explained in Section 4.3. The interactions between the entities CD, TE, SA, PA and SUT are described related to test case `tc_TAG` in Subsection 6.5.1. Afterwards, the test adapters of the test cases `tc_TAG` and `tc_TAM` are explained. Details about the adjustments of the SA entity and of the CD entity are given in Subsection 6.5.2 and in Subsection 6.5.3, respectively.

6.5.1 Interactions between the entities

In the SA, the interactions between the TE and the SUT are realized as depicted in the sequence diagram in Figure 6.10. This figure shows a simplified view of the operation invocations that are involved when test case `tc_TAG` is executed.

Before executing test case `tc_TAG`, the test system has to be initialized. This initialization phase includes invoking the TRI operations `triResetSA` and `triResetPA` (interactions 1 and 2 in Figure 6.10), which are provided by the SA and PA entity, respectively. After these entities have indicated their successful initialization, the TE starts executing the `control` part of the test suite that includes a call of test case `tc_TAG`. The TE invokes the operation `triExecuteTestcase` (interaction 3) in order to inform the SA that a new test case will be started. This allows the SA to initialize communication facilities for test case `tc_TAG` [40].

The communication channels towards the SUT have to be established. The TE invokes the `triMap` (interaction 4) operation that needs to be implemented in the SA. A correct completion of the `triMap` operation enables a test component to communicate with the SUT. TTCN-3 makes no assumptions about the communication with the SUT. Therefore, it is necessary to develop concrete operations in order to interact with the SUT by abstract TRI operations [40].

After the communication towards the SUT has been prepared, the communication has to be handled. A received message has to be encoded from structured TTCN-3 values into a form that is accepted by the SUT. The counterpart is the decode operation, where messages received from the SUT are decoded into structured TTCN-3 values. TTCN-3 does not make any assumption on how messages are en- or decoded. Therefore, a concrete encoding has to be developed [40].

Encoding and decoding services are provided by the CD entity, which is attached to the TE via TCI. The operations `tciEncode` and `tciDecode` need to be implemented by the CD entity. The `tciEncode` (interaction 5) operation encodes a requested TTCN-3 message value following the encoding rules and returns it to the TE as a binary string. The `tciEncode` invocation also includes information about the sending test component and the information from which Test System Interface (TSI) port the message has been sent. The encoded message is further passed to the SA via the `triSend` (interaction 6) operation. After invoking this operation, the SA has to make sure that the message will be transmitted

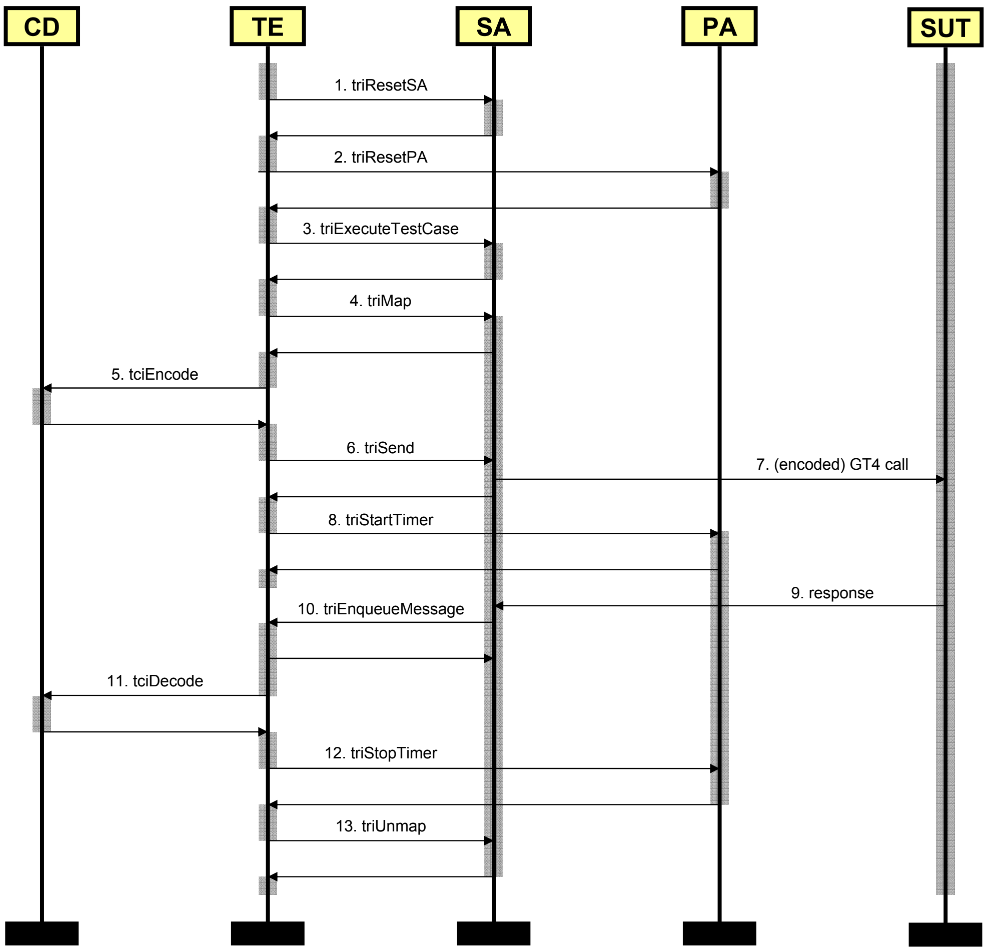


Figure 6.10: Interactions of test system entities while executing test case `tc_TAG`

to the SUT. The encoded message is the call of the GT4 execution service (interaction 7) [40].

After sending this message, a timer is started through the invocation of the `triStartTimer` (interaction 8) operation that is implemented in the PA via the TRI. The `triStartTimer` operation provides timer functionalities, which can be adjusted by the test developer. The PA is generally implemented by the test tool vendor. Therefore, common timers are integrated in the TTworkbench. The call of the operation `triStartTimer` consists of the timer duration and a timer handle that is used to identify the timer in further communications between TE and PA [40].

After the timer has been started, test case `tc_TAG` inside the TE continues with the evaluation of the `alt` construct. This construct includes different alternatives in order to handle different possible reactions of the SUT. The TE checks if a message has been received or a timeout has been reached. If neither any of the alternatives matches, the TE blocks the execution until a condition of a alternative is fulfilled [40].

If the message that has been sent is accepted by the SUT, typically, a response is sent by the SUT and received by the SA (interaction 9). The SA forwards this encoded message via the TRI to the responsible test component inside the TE by invoking the operation `triEnqueueMsg` (interaction 10). The message will be enqueued in the queue of the related port of the responsible test component. Because a message has arrived, the `alt` statement is evaluated inside the TE, whereas the first alternative calls for a matching attempt in order to compare the received message with an expected message. For this comparison, the received message has to be decoded into a structured TTCN-3 value. Therefore, the operation `tciDecode` (interaction 11), which is provided via the TCI by the CD entity, is invoked by the TE. In order to decode the message, the TE has to specify the assumed type of the message. This decoding hypothesis is used within the CD to select a decoding mechanism. After successful decoding, the CD provides the decoded message to the TE. If the message matches with the first alternative, the timer will be stopped by calling the operation `triStopTimer` (interaction 12) implemented in the PA via the TRI. After stopping the timer successfully, the ports are unmapped by the invocation of the operation `triUnmap` (interaction 13), which is the counterpart of the operation `triMap` on the TRI level [40].

6.5.2 System adapters

The SA includes the `triMap` and `triSend` operations. These operations are implemented in Java. For both developed test cases, excerpts of these Java methods are described in the following.

The SA can establish a connection to the SUT for the referenced TSI port in the `triMap` method. This method had to be overwritten for test case `tc_TAM` as it is shown in Listing 6.13. The two parameters of the `triMap` method are the IDs of the ports that have to be mapped (line 1). The parameter `compPortId` is a port reference to a test component port and the parameter `tsiPortId` is a port reference to a system interface port. Each of the two ports has to be registered to the other. This is done by using the default implementation of `triMap` from the interface `TriCommunicationSA` implemented by the tool vendor. This default implementation can be called from the provided base `TestAdapter` by using the handle `CsaDef` (line 3). Whether this registration was successful or not is returned in a status code (line 26).

Before the `triMap` method of the SA for test case `tc_TAM` returns this status, certain activities are realized depending on the ports that have to be mapped. A activity is initialized, respectively, when the port `pt_ImageAnswer` or the port `pt_SUTParameters` is going to be registered (lines 5-24). For both ports, a thread for the possible reception of messages from the SUT is started with different preferences (lines 18-23).

In case that port `pt_ImageAnswer` is going to be mapped, an image path and name is specified (line 9). This is the expected name of the final composed image. The started thread is responsible to check if this image has been created. If this is the case, a received message is enqueued to the port `pt_ImageAnswer` using the method `triEnqueueMsg`, which finally enqueues a message in the message queue of the appropriate component.

If the `triMap` method is called with a `compPortId` related to the port

```
 1 public TriStatus triMap(final TriPortId compPortId, final TriPortId tsiPortId) {
 2     ... // variables definition
 3     final TriStatus mapStatus = CsaDef.triMap(compPortId, tsiPortId);
 4     // start applicable thread when certain ports are mapped
 5     if (    compPortId.getPortName() == "pt_SUTParameters" ||
 6             compPortId.getPortName() == "pt_ImageAnswer") {
 7         if (compPortId.getPortName() == "pt_ImageAnswer") {
 8             // final image
 9             image = "/var/www/grid-mandelbrot/result/mandelbrot.png";
10             socketPort = 0;
11         } else if (compPortId.getPortName() == "pt_SUTParameters"){
12             // port for socket connection to stub
13             image = null;
14             socketPort = 4444;
15         } else {
16             ... // error handling
17         }
18         Runnable runnable = new ResponseListener(   compPortId, tsiPortId, Cte,
19                                                     image, socketPort);
20         // Create the thread supplying it with the runnable object
21         Thread listener = new Thread(runnable);
22         // Start the thread
23         listener.start();
24     }
25     ...
26     return new TriStatusImpl();
27 }
```

Listing 6.13: The `triMap` method of the SA required for `tc_TAM`

`pt_SUTParameters`, a port for a socket connection is defined (line 14). The invoked thread handles a socket connection through this port to the stub that replaces the executable on each PTC. It typically receives a message with the parameters of the tasks that have been called on the nodes. Therefore, this thread is started on each PTC. A received message is enqueued to the port `pt_SUTParameters` using the method `triEnqueueMsg`.

An excerpt of the SA required for test case `tc_TAG`, which includes the implementation of the `triSend` operation, is depicted in Listing 6.14. The method `triSend` overwrites the general `triSend` method that is implemented by the tool vendor and can send messages to the SUT. The parameters of this method include the `componentId` that is the identifier of the sending test component (line 1), the `tsiPortId` that is the identifier of the TSI port via which the message is sent to the SA (line 2), the `sutAddress` that is the destination address within the SUT (line 3) and the `message` that is the encoded message that has to be sent (line 4).

In line 6, a local map is initialized with the hash map that has been set up in the `TriMessage encode(Value)` method in the CD entity. The defined map includes fragments of the TTCN-3 message that has been sent as a stimulus through the ATS. Therefore, these fragments are extracted after line 8. Exemplified in line 9, the identification number is extracted from this map.

```
 1 public TriStatus triSend(   final TriComponentId componentId,
 2                             final TriPortId tsiPortId,
 3                             TriAddress sutAddress,
 4                             final TriMessage message ) {
 5
 6     Map<String, Value> localMap = new HashMap<String, Value>(GridCodec.myMap);
 7
 8     // build command for the global toolkit
 9     final String identification = new String(
10         localMap.get("identification").toString().replace("\"", ""));
11     ...
12     // build whole command
13     String command = "/usr/bin/sudo -u knoppix " +
14         "/usr/local/globus-4.0.3/bin/globusrun-ws -submit " +
15         "-o " + path + host + ".epr " +
16         "-b " +
17         "-F https://" + host + ":8443/wsrf/services/ManagedJobFactoryService " +
18         "-c " + appBuild;
19
20     // call the command
21     try {
22         Runtime.getRuntime().exec(command);
23     } catch (IOException e) {
24         ...
25     }
26     ... // receiverThread definition shown in Listing 6.15
27
28     // start Thread for reveiving messages from the SUT
29     receiverThread.start();
30
31     return new TriStatusImpl();
32 }
```

Listing 6.14: Excerpt of the SA required for `tc_TAG`

The command for calling the GT4 execution service is built from the message extracted by the encoder of the CD entity. The command consists of the service name `globusrun-ws` with its options including the variables that have been determined from the encoded message (lines 13-18). The variables include the host, on which the application has to be executed and the application name with its location and parameters. These parameters are specified in the ATS and encoded through the CD entity. After the command `globusrun-ws` with its given configurations that are specified in the GT4 documentation has been called (line 22), a thread for receiving possible responses from the SUT is defined and started in line 29. A status that this method was completed successfully is returned in line 31.

The thread `receiverThread` mentioned above is shown in Listing 6.15. It checks for messages from the SUT in order to forward them to the TE. In lines 4 and 5 the output image is specified through the variable extractions from the map defined above. The thread polls for the output image of the submitted task in a frequent interval of `sleep_time` (lines 6-8). If the output image has been created, its MD5 checksum is

```
 1 // create polling Thread until output created
 2 Thread receiverThread = new Thread() {
 3     public void run() {
 4         final String fileString = path + identification + ".png";
 5         final File file = new File (fileString);
 6         while (!file.exists()) {
 7             Thread.sleep(sleep_time);
 8         }
 9         // file has been created, means program is terminated
10         if (file.exists()) {
11             ...
12             // generate a message from the SUT output
13             if (execType.equals("hash")) {
14                 // get the MD5 checksum of the image
15                 String Hash = MD5.asHex(MD5.getHash(new File(fileString)));
16                 String Received_messages = identification + " " + Hash;
17                 byte[] Byte_response = Received_messages.getBytes();
18                 // create new message for response to TTCN-3
19                 TriMessage response = new TriMessageImpl(Byte_response);
20                 // method enqueueMsg definition shown in Listing ??
21                 enqueueMsg( tsiPortId, new TriAddressImpl(new byte[]{}),
22                             componentId, response);
23             } else { // byte
24                 ...
25             }
26             ...
27         }
28     }
29 };
```

Listing 6.15: Thread definition for receiving messages from the SUT required for `tc_TAG`

determined (line 15). This hash sum and the identification are integrated in the response message (lines 16-19). The creation of the output image is seen as a message from the SUT and, therefore, decoded by the `decode` method of the CD entity. This decoded message is enqueued via TRI to the responsible port of the related test component by invoking the method `enqueueMsg` (line 21).

The method `triSend` of the SA required for test case `tc_TAM` is partly shown in Listing 6.16. The concept of handling encoded messages with a hash map is adapted from the SA used in test case `tc_TAG` (lines 6, 9, 10) as described previously.

In the `triSend` method, it is distinguished between the types that have been handled by the `GridCodec` class, which corresponds to the CD entity. This differentiation is required, because dependent on the determined type a specific handling has to be started. In case the type `StimulusType` is determined, a thread is defined (lines 13-37) and started (line 38). This thread is responsible for permanently reloading an URL, which is specified in the encoded message in order to send stimuli to the SUT. The thread reloads the URL until the file `finished` has been created by the SUT as described in the specification in Section 5.4. In lines from 20 to 24, the URL is specified with the entries of the encoded message. The protocol, host, file and the CGI variables that are

```
1  public TriStatus triSend(    final TriComponentId componentId,
2                               final TriPortId tsiPortId,
3                               final TriAddress sutAddress,
4                               final TriMessage message ) {
5
6      Map<String, Value> localMap = new HashMap<String, Value>(GridCodec.RequestMap);
7
8      if (GridCodec.Type.equals("DistMandelbrot.StimulusType")) {
9          final String protocol = new String(localMap.get
10             ("protocol").toString().replace("\"", ""));
11         ... // other extractions
12
13         Thread reloadWebsiteThread = new Thread() {
14             public void run() {
15                 final String fileString = workingpath + "finished";
16                 final File finished = new File (fileString);
17                 while (!finished.exists()) {
18                     try {
19                         try {
20                             // Construct data to set the PHP variables
21                             String data = URLEncoder.encode(ResolutionXKey,
22                                 "UTF-8") + "=" +
23                                 URLEncoder.encode(ResolutionXValue, "UTF-8");
24                             ... // add further data, url definition
25                             URLConnection conn = url.openConnection();
26                             ... // init
27                             BufferedReader rd = new BufferedReader(
28                                 new InputStreamReader(conn.getInputStream()));
29                             // read the web page
30                             while (rd.readLine() != null) {}
31                             ...
32                         } ... // error handling
33                         Thread.sleep(reload_interval);
34                     } ... // error handling
35                 }
36             }
37         };
38         reloadWebsiteThread.start();
39     } else if (GridCodec.Type.equals("DistMandelbrot.ImageType")) {
40         ... // handling is shown in Listing 6.17
41     } ... // error handling
42     return new TriStatusImpl();
43 }
```

Listing 6.16: The `triSend` method of the SA required for `tc_TAM` (part one)

specified in the ATS are integrated into the URL. This URL connection is opened with the help of the provided Java class `URLConnection` (lines 25, 26). The object `rd` of type `BufferedReader` reads the stream of the `conn` object in order to load the specified URL as a webpage (lines 27-30). A reload is done in an interval of `time_interval` (line 33) in order to simulate a web browser. The `else` branch in this method is taken, when the type name is equal to `ImageType`. The implementation of the `else` branch is shown in Listing 6.17 and described in the following.

```
 1 public TriStatus triSend( ... ) {
 2
 3     Map<String, Value> localMap = new HashMap<String, Value>(GridCodec.RequestMap);
 4
 5     if (GridCodec.Type.equals("DistMandelbrot.StimulusType")) {
 6         ... // handling is shwon in Listing 6.16
 7     } else if (GridCodec.Type.equals("DistMandelbrot.ImageType")) {
 8         // create the requested image
 9         final String outputfile = new String(localMap.get(
10             "outputfile").toString().replace("\"", ""));
11         ... // further message extractions
12         try {
13             Color color = new Color(red, green, blue);
14             BufferedImage img = new BufferedImage( width, height,
15                                                     BufferedImage.TYPE_INT_RGB);
16             for (int i=0; i<width; i++) {
17                 for (int j=0; j<height; j++) {
18                     img.setRGB(i, j, color.getRGB());
19                 }
20             }
21             ... // other image configurations
22             ImageIO.write(img, "png", new File(outputfile));
23         } ... // error handling
24     } ... // error handling
25     return new TriStatusImpl();
26 }
```

Listing 6.17: The `triSend` method of the SA required for `tc_TAM` (part two)

Inside this `else` branch, the entries of the encoded message are extracted (lines 9-11). These entries represent an image specification. The color, the resolution, the file name and its directory are extracted from the encoded message. An image related to this specification is created in lines 12 to 22 in order to feign an image creation for the IUT.

6.5.3 Coding and Decoding

A message that is supposed to be sent to the SUT has to be encoded. The `encode` method, implemented in Java, of the CD entity required for test case `tc_TAG` is shown in Listing 6.18. A `HashMap` is declared (line 5) in order to store the values of the TTCN-3 message. The values are extracted from the passed parameter `value` and inserted into

```
 1 @Override
 2 public TriMessage encode(final Value value) {
 3
 4     // create a table mapping to save values
 5     myMap = new HashMap<String, Value>();
 6     RecordValue v = (RecordValue) value;
 7
 8     // store values
 9     myMap.put("identification", v.getField("identification"));
10     ...
11 }
```

Listing 6.18: The `encode` method of the CD entity required for `tc_TAG`

the hash map. As an example of the extraction, the content of `identification` that is related to the equally named TTCN-3 variable is stored in the `HashMap myMap` in line 9, whereas the TTCN-3 variable `identification` is the key of its stored content in this `HashMap`. The final encoding is done in the `triSend` method, where the command for the GT4 is built by retrieving the values, which have been stored in the hash map by the `encode` method.

The `encode` method required for test case `tc_TAM` is shown in Listing 6.19. The encoding scheme is the same as in the method `encode` of the CD of test case `tc_TAG`. A hash map is initialized (line 3) in order to store the encoded values of messages dependent on the TTCN-3 types defined in the ATS. This method distinguishes between the types `StimulusType` (line 7), `ImageType` (line 10), `ImageAnswerType` (line 13) and any other types (line 15).

```
 1  public TriMessage encode(final Value value) {
 2      // create a table mapping to store values
 3      RequestMap = new HashMap<String, Value>();
 4      RecordValue v = (RecordValue) value;
 5
 6      Type = value.getType().toString();
 7      if (Type.equals("DistMandelbrot.StimulusType")) {
 8          RequestMap.put("protocol", v.getField("protocol"));
 9          ... // further inputs
10      } else if (Type.equals("DistMandelbrot.ImageType")){
11          RequestMap.put("outputfile", v.getField("outputfile"));
12          ... // further inputs
13      } else if (Type.equals("DistMandelbrot.ImageAnswerType")){
14          RequestMap.put("Checksum", v.getField("Checksum"));
15      } else {
16          RequestMap.put("application", v.getField("application"));
17      }
18      ...
19  }
```

Listing 6.19: The `encode` method of CD entity required for `tc_TAM`

A message that is supposed to be enqueued to a test component has to be decoded. The `decode` method of the CD entity of test case `tc_TAG` is shown in Listing 6.20. In line 3 the `byte_message` of type `byte[]` is defined and assigned to the encoded message, i.e., the received message. If the type delivered by the parameter matches the `GridAnswerType` (line 4), which is part of the decoding hypothesis, the message can be decoded. If the `execType` equals the hash option, the related structured TTCN-3 value will be built, by invoking the method `createGridAnswerType(RecordValue, byte[])` (line 8), which sets the fields of the TTCN-3 value. The returned decoded message will be passed to the TE.

```java
@Override
public Value decode(final TriMessage message, final Type type) {
    byte[] byte_message = message.getEncodedMessage();
    if (type.toString().equals("ClientTester.GridAnswerType")) {
        ...
        if (execType.equals("hash")) {
            RecordValue GridAnswerType = (RecordValue)type.newInstance();
            return createGridAnswerType(GridAnswerType, byte_message);
        } else {
            ...
        }
    } else {
        return null;
    }
}
```

Listing 6.20: The `decode` method of the CD entity required for `tc_TAG`

The method `createGridAnswerType(RecordValue, byte[])` is depicted in Listing 6.21. It builds structured TTCN-3 values from the transmitted `byte_message` through the decoding hypothesis. A part of the decoding hypothesis is initialized with defining the string array `String[] GridAnswerTypeArray = {"identification", "Checksum" };`, which represents the TTCN-3 entry names of the TTCN-3 record type `GridAnswerType`. The other variables defined in lines from 3 to 8 are used for temporary storage or iteration purposes.

```java
1   private RecordValue createGridAnswerType(   RecordValue GridAnswerType,
2                                               byte[] byte_message) {
3       Value value_message = null;
4       int dataLength = byte_message.length;
5       String[] GridAnswerTypeArray = { "identification", "Checksum" };
6       String msg = new String(byte_message, 0, dataLength);
7       String tempMsg = null;
8       Integer tempInteger;
9
10      for (String s: GridAnswerTypeArray) {
11          if (!s.equals("Checksum")) {
12              tempMsg = msg.substring(0, msg.indexOf(" "));
13              tempInteger = Integer.valueOf(tempMsg);
14              value_message =
15                  new de.tu_berlin.cs.uebb.muttcn.runtime.Int(rb, tempInteger);
16          } else { // last entry of the array
17              tempMsg = msg;
18              value_message = new CharString(rb, tempMsg);
19          }
20          setField(GridAnswerType, s, value_message);
21          msg = msg.substring((msg.indexOf(" ")+1), msg.length());
22      }
23      return GridAnswerType;
24  }
```

Listing 6.21: The `createGridAnswerType` method of the CD entity required for `tc_TAG`

In the `for` loop in lines 10 to 22 the string array is iterated in order to build the TTCN-3 record type `GridAnswerType` and cover each of its entries. The received `byte[]` message is converted into `String` (line 6) and divided with help of the whitespace separator (lines 21) into the entries of the TTCN-3 record type `GridAnswerType`. The determined values of the entries are transformed into `Value` type messages (lines 13 and 17). These values stored in the variable `value_message` are set with its belonging value types that are assigned to the iterator `s` and their TTCN-3 record type `GridAnswerType` in line 20 in order to build the TTCN-3 record type structure `GridAnswerType`. This created structure is an understandable message for the TE and is later enqueued to the appropriate component. If the method has been executed correctly, this message includes the expected identification number and the expected determined hash sum of the output image.

The `decode` method of the CD entity of test case `tc_TAM` is shown in Listing 6.22. The decoding has to follow two different decoding hypotheses. It has to be distinguished between the TTCN-3 types `ReactionType` and `ImageAnswerType` that are the expected types of possible responses from the SUT. The responses have to be decoded into TTCN-3 structured types. The methods `createReactionType` and `createImageAnswerType` follow the same concepts as the `createGridAnswerType` method that is explained in the previous paragraph.

```java
public Value decode(final TriMessage message, final Type type) {
    byte[] byte_message = message.getEncodedMessage();
    bitpos = 0;

    if (type.toString().equals("DistMandelbrot.ReactionType")) {
        RecordValue ReactionType = (RecordValue)type.newInstance();
        return createReactionType(ReactionType, byte_message);
    } else if (type.toString().equals("DistMandelbrot.ImageAnswerType")) {
        RecordValue ImageAnswerType = (RecordValue)type.newInstance();
        return createImageAnswerType(ImageAnswerType, byte_message);
    } else {
        return null;
    }
}
```

Listing 6.22: The `decode` method of the CD entity required for `tc_TAM`

6.6 Test execution

TTCN-3 test cases are executed with the help of TTworkbench Enterprise. After specifying the test preferences and compiling the ATS, a Campaign Loader File (CLF) is built by TTworkbench. The CLF is based on the eXtensible and Markup Language (XML) and depicted in Listing 6.23 for test case `tc_TAG`.

The CLF file is required by the TTworkbench execution environment TTman in order to execute test cases. It includes preferences of the test suite, specifies the location of the

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!DOCTYPE campaignloader PUBLIC "-//TESTING TECH//DTD MLF//1.7" "mlf.dtd">
 3  <campaignloader>
 4    <campaign Name="ClientTester" Control="false">
 5      <testadapter Name="GridAdapter" File="lib/adapter.jar" />
 6      <module Name="ClientTester" File="ClientTester.jar" Package="generated_ttcn" />
 7      <parameter Name="HOST" Module="ClientTester">
 8        <description>Host on which the application runs</description>
 9        <type>charstring</type>
10        <value>
11          <![CDATA[
12            <Values:charstring type="charstring" xmlns:Values="Values.xsd">
13              <Values:value>bombay</Values:value>
14            </Values:charstring>
15          ]]>
16        </value>
17        <default>
18          <![CDATA[
19            <Values:charstring type="charstring" xmlns:Values="Values.xsd">
20              <Values:value>bombay</Values:value>
21            </Values:charstring>
22          ]]>
23        </default>
24      </parameter>
25      ... <!-- more module paramters descriptions -->
26      <testcase
27        Name="tc_TAG"
28        Module="ClientTester"
29        Retries="0" Runs="1"
30        ActionOnFail="continue"
31        Selection="true"
32        Verdict="none">
33        <description></description>
34      </testcase>
35      ... <!-- more test case descriptions -->
36    </campaign>
37  </campaignloader>
```

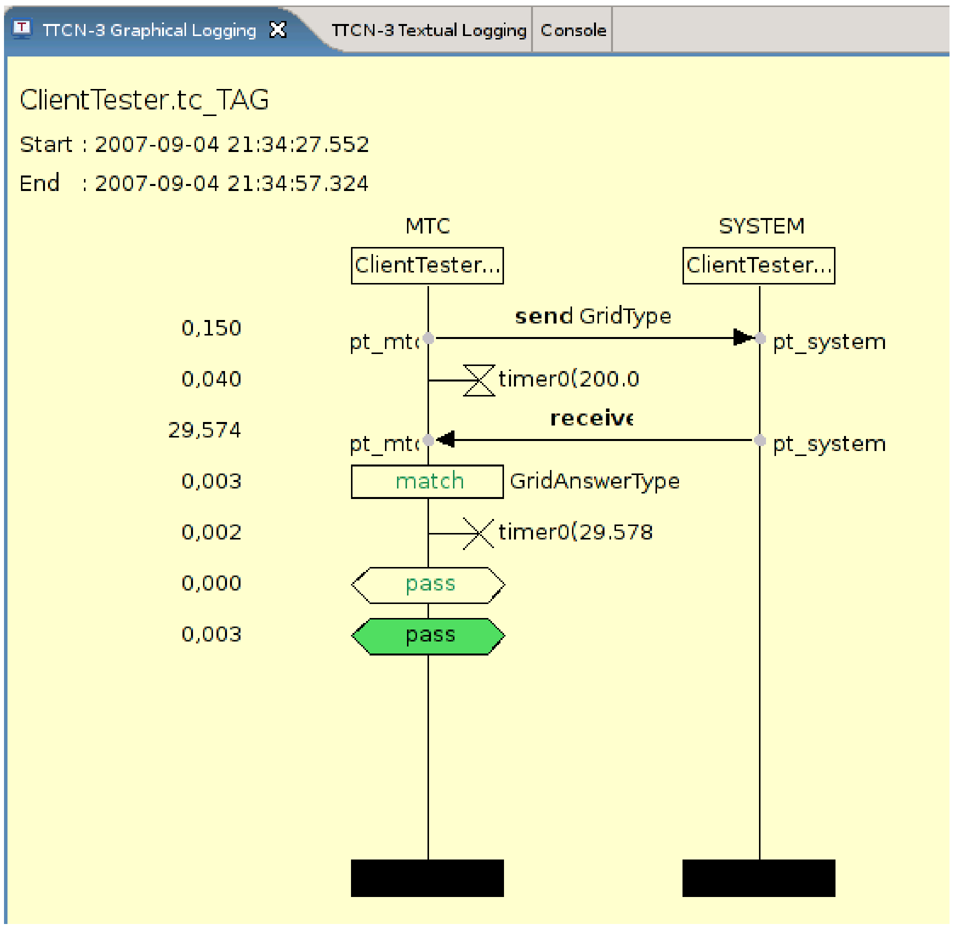
Listing 6.23: An XML based Campaign Loader File required for execution of test case `tc_TAG`

SA and signifies module parameters. A brief explanation of important tags of this XML file is given in the following.

The campaign name is shown in the `campaign` tag in line 4, which is in this case also the main module name. The main module name is also given in the `module` tag, which also includes the name of the TE that is for this test suite the file `ClientTester.jar` as it is shown in line 6. The class name, location and file name of the SA is given by the `testadapter` tag in line 5. Afterwards, the module parameter with its defined type, default value and description as mentioned previously in Listing 6.1 is described between the `parameter` tag in lines 7 to 24. If more module parameters are defined in the test suite, additional `parameter` tags are added by the TTworkbench compilation process. After the module parameter description, the description of the test cases follows. Test case `tc_TAG` is referenced between the `testcase` tag in lines 26 to 34. If more test cases are defined in the test suite, additional `testcase` tags are added by the TTworkbench compilation process.

During and after the execution, a graphical and textual logging for the test case run is provided by the execution environment. The graphical logging consists of a sequence diagram of test case `tc_TAG` and is depicted in Figure 6.11.

As defined in the TTCN-3 specifications explained above, two test components are involved in test case `tc_TAG`. The communication between the MTC and the system component is shown in the graphical logging provided by TTworkbench. A message of type `GridType` is sent from the MTC port `pt_mtc` to the system port `pt_system`. This message is the stimulus of the SUT. Afterwards, the timer with a timeout of 200 seconds is started. Within the timer duration, a message is received by the `pt_mtc` port from the `pt_system` port, whereas the message is matched with an expected message, i.e., in this case with a predefined template. The message matches a template of type `GridAnswerType` and, therefore, the timer is stopped after 29.578 seconds. Since the



ClientTester.tc_TAG

Start : 2007-09-04 21:34:27.552

End : 2007-09-04 21:34:57.324

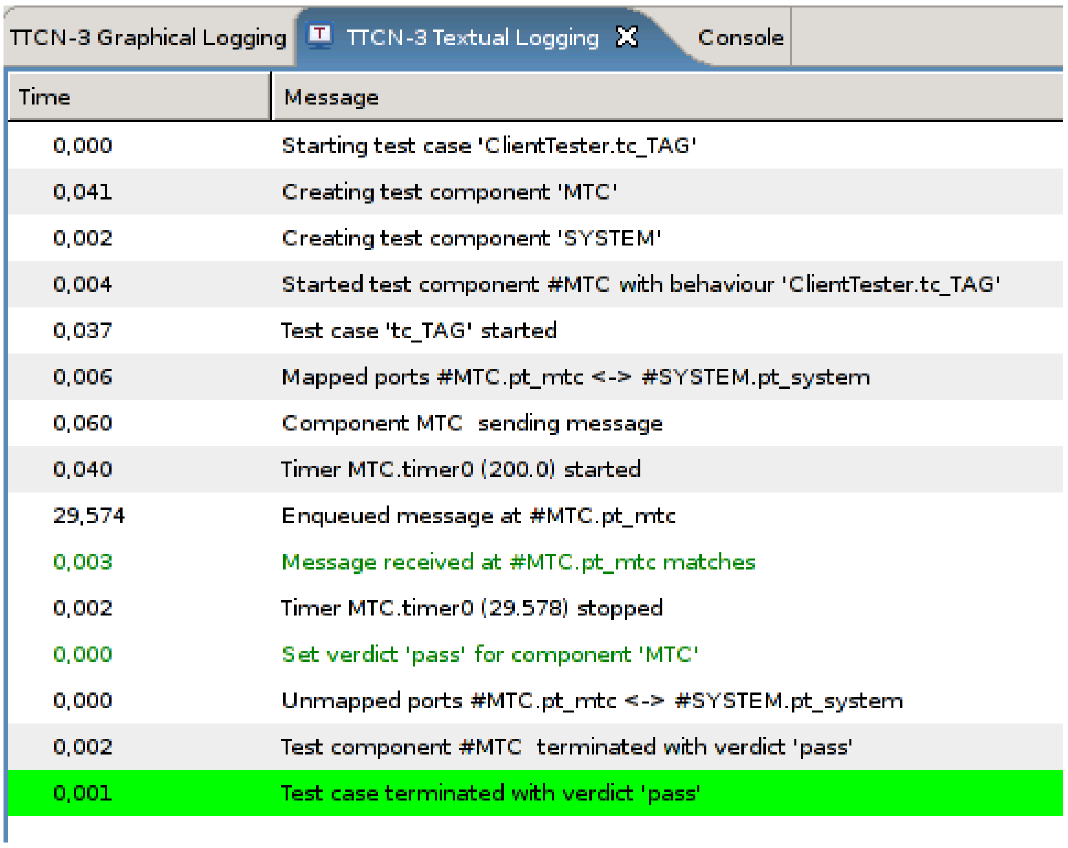
Figure 6.11: Test execution of test case `tc_TAG` - Graphical logging

message matches, the local verdict of this component is set to `pass` and the global verdict of the test case is set to `pass` as well.

The sequence diagram of this test case has similarities to the sequence diagram illustrated in Figure 6.1. Therefore, the behavior of the SUT was expected.

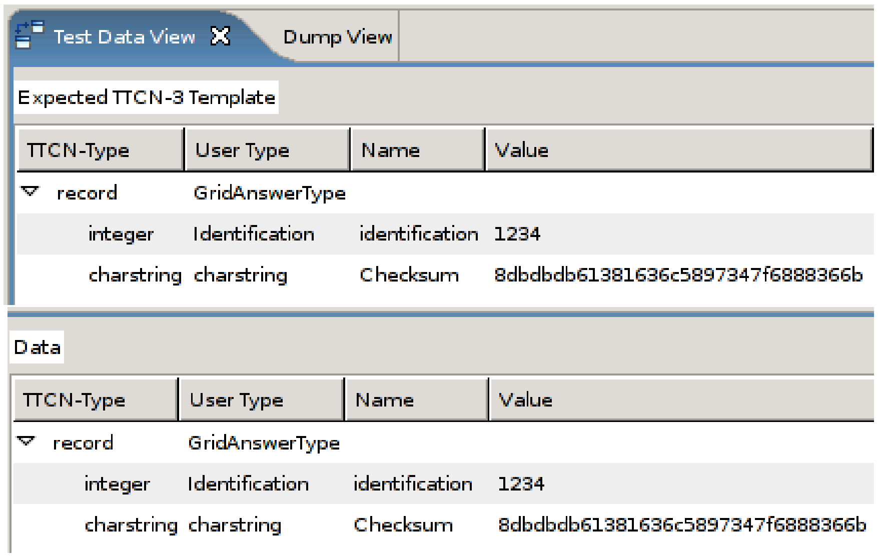
Figure 6.12 shows the textual logging of the execution of test case `tc_TAG`. The content is the same as in the graphical logging. First of all, the MTC and the system component are created. After the test case behavior is started, the port `pt_mtc` of the MTC is mapped with the port `pt_system` of the system. After the MTC sent a message, a timer is started. Within the timeout, a message is received by the MTC. The message matches with the expected result. Therefore, the verdict of this component is set to `pass` after the timer is stopped. The test case terminates and the global verdict is evaluated as `pass` as well.

The expected and actual output data of test case `tc_TAG` can be compared in the test data view that is depicted in Figure 6.13. The actual identification number and the actual MD5 checksum match with the expected data of the TTCN-3 template.



Time	Message
0,000	Starting test case 'ClientTester.tc_TAG'
0,041	Creating test component 'MTC'
0,002	Creating test component 'SYSTEM'
0,004	Started test component #MTC with behaviour 'ClientTester.tc_TAG'
0,037	Test case 'tc_TAG' started
0,006	Mapped ports #MTC.pt_mtc <-> #SYSTEM.pt_system
0,060	Component MTC sending message
0,040	Timer MTC.timer0 (200.0) started
29,574	Enqueued message at #MTC.pt_mtc
0,003	Message received at #MTC.pt_mtc matches
0,002	Timer MTC.timer0 (29.578) stopped
0,000	Set verdict 'pass' for component 'MTC'
0,000	Unmapped ports #MTC.pt_mtc <-> #SYSTEM.pt_system
0,002	Test component #MTC terminated with verdict 'pass'
0,001	Test case terminated with verdict 'pass'

Figure 6.12: Test execution of test case `tc_TAG` - Textual logging



Expected TTCN-3 Template

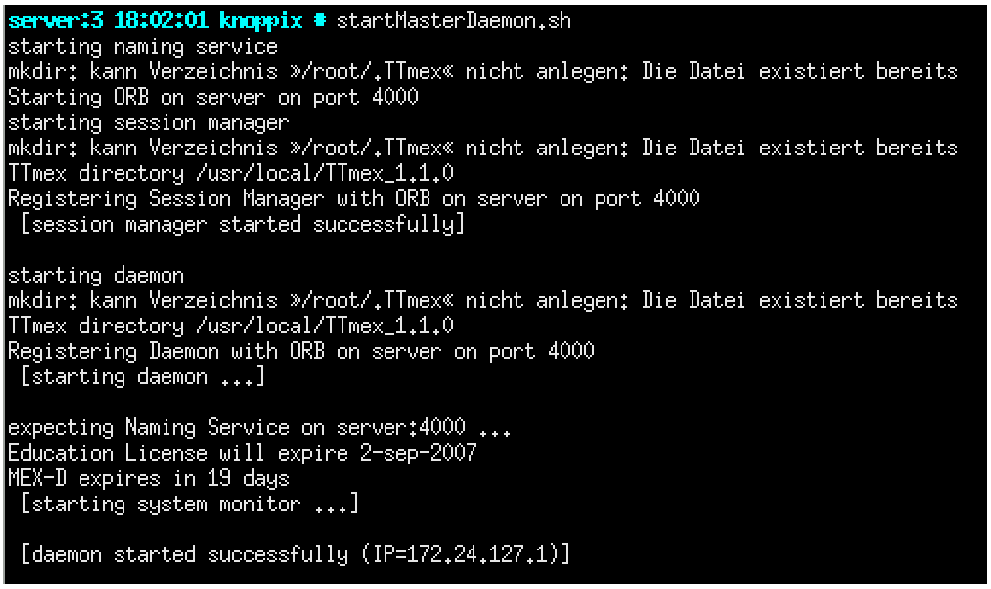
TTCN-Type	User Type	Name	Value
record	GridAnswerType		
integer	Identification	identification	1234
charstring	charstring	Checksum	8dbdbdb61381636c5897347f6888366b

Data

TTCN-Type	User Type	Name	Value
record	GridAnswerType		
integer	Identification	identification	1234
charstring	charstring	Checksum	8dbdbdb61381636c5897347f6888366b

Figure 6.13: Test execution of test case `tc_TAG` *- Test data view*

In the following, the execution of the distributed test case `tc_TAM` is described. Before this test case can be started, the test infrastructure has to be initialized. Therefore, a CORBA based TTmex daemon has to be started on each node that is registered in the Grid. The test engineer has to determine a master node, to which the other nodes are going to connect. In this test environment, the master node is the one on which the IUT runs. On this node, the TTmex master daemon has to be started as shown in Figure 6.14.



```
server:3 18:02:01 knoppix # startMasterDaemon.sh
starting naming service
mkdir: kann Verzeichnis »/root/.TTmex« nicht anlegen: Die Datei existiert bereits
Starting ORB on server on port 4000
starting session manager
mkdir: kann Verzeichnis »/root/.TTmex« nicht anlegen: Die Datei existiert bereits
TTmex directory /usr/local/TTmex_1.1.0
Registering Session Manager with ORB on server on port 4000
 [session manager started successfully]

starting daemon
mkdir: kann Verzeichnis »/root/.TTmex« nicht anlegen: Die Datei existiert bereits
TTmex directory /usr/local/TTmex_1.1.0
Registering Daemon with ORB on server on port 4000
 [starting daemon ...]

expecting Naming Service on server:4000 ...
Education License will expire 2-sep-2007
MEX-D expires in 19 days
 [starting system monitor ...]

 [daemon started successfully (IP=172.24.127.1)]
```

Figure 6.14: Start of the master daemon in order to provide a test infrastructure for test case `tc_TAM`

The master daemon starts the CORBA naming service and the TTmex session manager. Since the test Grid contains three nodes, for the test execution the TTmex master daemon and two TTmex client daemons have to be started. The client daemons connect the master daemon during their starting process. According to adequate configurations as described in the TTmex User's Guide [15], the daemons interact via the CORBA platform.

The distributed test execution in TTmex requires certain configurations. These include three configuration files, which are described in the following. One of them is the Container Configuration File (CCF) that is depicted in Listing 6.24. This file is written in XML format and determines the hosts, on which a container has to be created. In addition, the container names and the sources that have to be deployed to this container are defined.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <containerLoader ...>
 3    <container_list>
 4      <container>
 5        <IP>172.24.127.1</IP>
 6        <name>alpha1</name>
 7        <deploy name="ttcn3/DistMandelbrot.jar" />
 8        <deploy name="lib/TA.jar" />
 9      </container>
10      <container>
11        <IP>172.24.127.252</IP>
12        <name>alpha3</name>
13        <deploy name="ttcn3/DistMandelbrot.jar" />
14        <deploy name="lib/TA.jar" />
15      </container>
16      <container>
17        <IP>172.24.127.253</IP>
18        <name>alpha2</name>
19        <deploy name="ttcn3/DistMandelbrot.jar" />
20        <deploy name="lib/TA.jar" />
21      </container>
22    </container_list>
23  </containerLoader>
```

Listing 6.24: `DistMandelbrot.ccf` required for the distributed execution of test case `tc_TAM`

Since the Grid contains three nodes, three one containers, one for each of them, are described with its attributes between the tags `container` in the CCF. The container specification includes the IP address of the node using the `IP` tag (lines 5, 11 and 17), the name of the node in the test execution using the (`name` tag in lines 6, 12 and 18) and the compiled ETS using the (`deploy` tags). In this example, the same data is deployed in each container.

The Test Component Distribution Language (TCDL) file, which is written in XML, is also required for the configuration of distributed tests. The TCDL allows to specify how the components are distributed over the known containers. This distribution file contains information about the component description, the component assembly, the component mapping rules and the component distribution algorithms. The TCDL file for executing test case `tc_TAM` is shown in Listing 6.25. A TCDL file always starts with the `componentassembly` tag (line 2). Within the `partition` tag, a component of a specified type (between the `component_selector` tag) can be deployed to the containers that are specified inside the `container` tag (lines 10 and 18-20). For this deployment, the distribution algorithm `round-robin`, specified inside the `homes` tag (lines 9 and 17), is used. This means in this case, that a component of type `MTCType` and of `PTCType` is deployed in container `alpha1` and one component of type `PTCType` is deployed, respectively, in container `alpha2` and in container `alpha3`. The collector is a special type of partition and is called when no selector from the previous partitions matches. The collector contains the container `alpha1` (lines 23-25).

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <componentassembly ... >
 3   <description>Grid computing test</description>
 4   <special container="alpha1" />
 5   <partition>
 6     <component_selectors>
 7       <componenttype>MTCType</componenttype>
 8     </component_selectors>
 9     <homes distribution="round-robin">
10       <container id="alpha1" />
11     </homes>
12   </partition>
13   <partition>
14     <component_selectors>
15       <componenttype>PTCType</componenttype>
16     </component_selectors>
17     <homes distribution="round-robin">
18       <container id="alpha1" />
19       <container id="alpha2" />
20       <container id="alpha3" />
21     </homes>
22   </partition>
23   <collector>
24     <container id="alpha1" />
25   </collector>
26 </componentassembly>
```

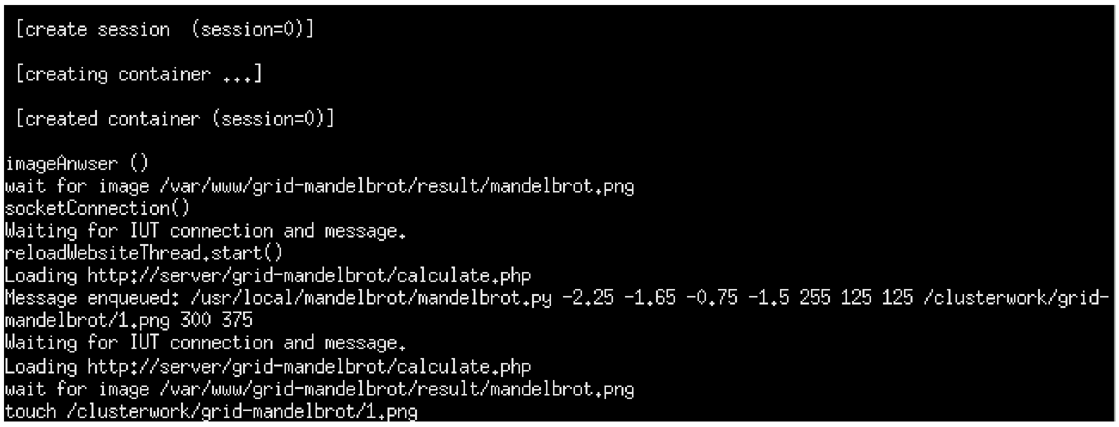
Listing 6.25: `DistMandelbrot.tcdl` required for the distributed execution of test case `tc_TAM`

The third configuration file for distributed testing with TTmex is the mex script that contains commands supported by the TTmex Console. This script creates and configures the TTmex session for a test case that runs in a distributed manner. The mex script used for test case `tc_TAM` is depicted in Listing 6.26. First, a variable for storing the session ID is declared (line 2). Afterwards, a session and all participating hosts are created with the command `make` (line 5), which uses the description given in the file `DistMandelbrot.ccf` as explained above. The configuration of the distribution rules is done by the command `config` with the descriptions that are given in the file `DistMandelbrot.tcdl`. The ETS that is specified in the CLF file is loaded into the specific containers (lines 11-13). Afterwards, a variable for storing the verdict is declared (line 16) and test case `tc_TAM` is started with the command `starttc` (line 19). The `wait` command waits until the termination of test case `tc_TAM` (line 22) in order to collect the final verdict. This verdict is displayed to the test engineer with the `echo` command (line 25). The test log is retrieved with the command `log` and saved in the file `exec_tc_TAM.tlz` (line 28). This file contains the graphical and the textual log, as explained further in the following. The session variable is killed (line 31) and the defined variables are released (line 34). The script ends with the command `exit`.

```
1  # store the session ID
2  var $ses_id
3
4  # create a new session
5  make -ccf DistMandelbrot.ccf -v $ses_id
6
7  # configure the new session (set the deployment rules)
8  config -s $ses_id -tcdl DistMandelbrot.tcdl
9
10 # load the test modules and module parameters as specified in clf file
11 load -s $ses_id -c alpha1 -clf ttcn3/DistMandelbrot.clf
12 load -s $ses_id -c alpha2 -clf ttcn3/DistMandelbrot.clf
13 load -s $ses_id -c alpha3 -clf ttcn3/DistMandelbrot.clf
14
15 # store the verdict
16 var $verdict
17
18 # starts the session
19 starttc -s $ses_id tc_TAM
20
21 # wait until the execution terminates
22 wait -s $ses_id -v $verdict
23
24 # show testcase verdict
25 echo testcase: tc_TAM -v $verdict
26
27 # retrieve log
28 log -s $ses_id -o exec_tc_TAM.tlz
29
30 # kill session
31 kill -s $ses_id
32
33 # release used variables
34 release $ses_id $verdict
35
36 # exit script
37 exit
```

Listing 6.26: The mex script of the TTmex functionality used for the distributed execution of test case `tc_TAM`

The mex script is executed by the TTmex environment. Before executing, this environment has to connect to the previously started master daemon. The master daemon monitoring during the execution of test case `tc_TAM` is depicted in Figure 6.15. As explained before, first a session and then a container is created. The same is done by the other daemons. Afterwards, the `ImageAnswer` thread is started inside the TRI. Then, the main thread that waits for the final result polls for the output file `Mandelbrot.png`. Afterwards, the thread for the socket connection is started in order to wait for a connection and messages from the stub. Then, the `reloadWebsiteThread` is started in order to send the stimulus to the SUT. After the website has been loaded, a message is received by the socket connection thread that enqueues the received message to the appropriate test component. The socket connection thread continues waiting for a possible recurrent socket connection by the stub. The website is loaded again and the test system still waits for the final output. The feigned image for the IUT has been created as it is shown in the last line of Figure 6.15.



```
 [create session  (session=0)]

 [creating container ...]

 [created container (session=0)]

imageAnwser ()
wait for image /var/www/grid-mandelbrot/result/mandelbrot.png
socketConnection()
Waiting for IUT connection and message.
reloadWebsiteThread.start()
Loading http://server/grid-mandelbrot/calculate.php
Message enqueued: /usr/local/mandelbrot/mandelbrot.py -2.25 -1.65 -0.75 -1.5 255 125 125 /clusterwork/grid-
mandelbrot/1.png 300 375
Waiting for IUT connection and message.
Loading http://server/grid-mandelbrot/calculate.php
wait for image /var/www/grid-mandelbrot/result/mandelbrot.png
touch /clusterwork/grid-mandelbrot/1.png
```

Figure 6.15: TTmex master daemon monitoring during the execution of test case `tc_TAM`

After the execution is terminated, the log file can be evaluated by the test engineer. The log file includes the graphical and the textual logging. An excerpt of the graphical logging provided by the TTworkbench is shown in Figures 6.16 and 6.17. Seven test component instances are depicted in this sequence diagram. The MTC instance that is included in container `alpha1` is represented by the first lifeline of the sequence diagram. On each node, a system component is instantiated. The second lifeline represents the instance of the system component of container `alpha3`, the third represents `alpha2` and the fourth `alpha1`. Additionally, on each node, a PTC instance is started and shown in the lifelines five to seven. The names of the related containers are shown in the first line. Each PTC interacts with its instance of the system component that is instantiated by its belonging container.

The depicted behavior is as follows: First of all, the `PTCTimer` is started with a timeout of 50 seconds on each PTC. After the timers have been started, the stimulus of type `StimulusType` is sent by the `pt_http` port from the MTC to the related system component instance of container `alpha1`. After the IUT is stimulated, the PTC instance of container `alpha1` receives a message from its system component instance by its port `pt_SUTParameters`. The received message matches the type `ReactionType`. Therefore, its `PTCTimer` is stopped and the message is forwarded for further evaluation via the port `pt_PTCParameters` to the MTC. After the `PTCApplicationTimer` is started, the MTC confirms the type `ReactionType` of the received message and compares it with the expected messages. After the `PTCApplicationTimer` is timed out as intended, the image specification is sent as type `ImageType` to the belonging system component by the PTC in order to create a feigned image. The same interactions take place by the PTC instances of container `alpha2` and container `alpha3`. These interactions occur repeatedly and are indicated by the dashed lifelines.

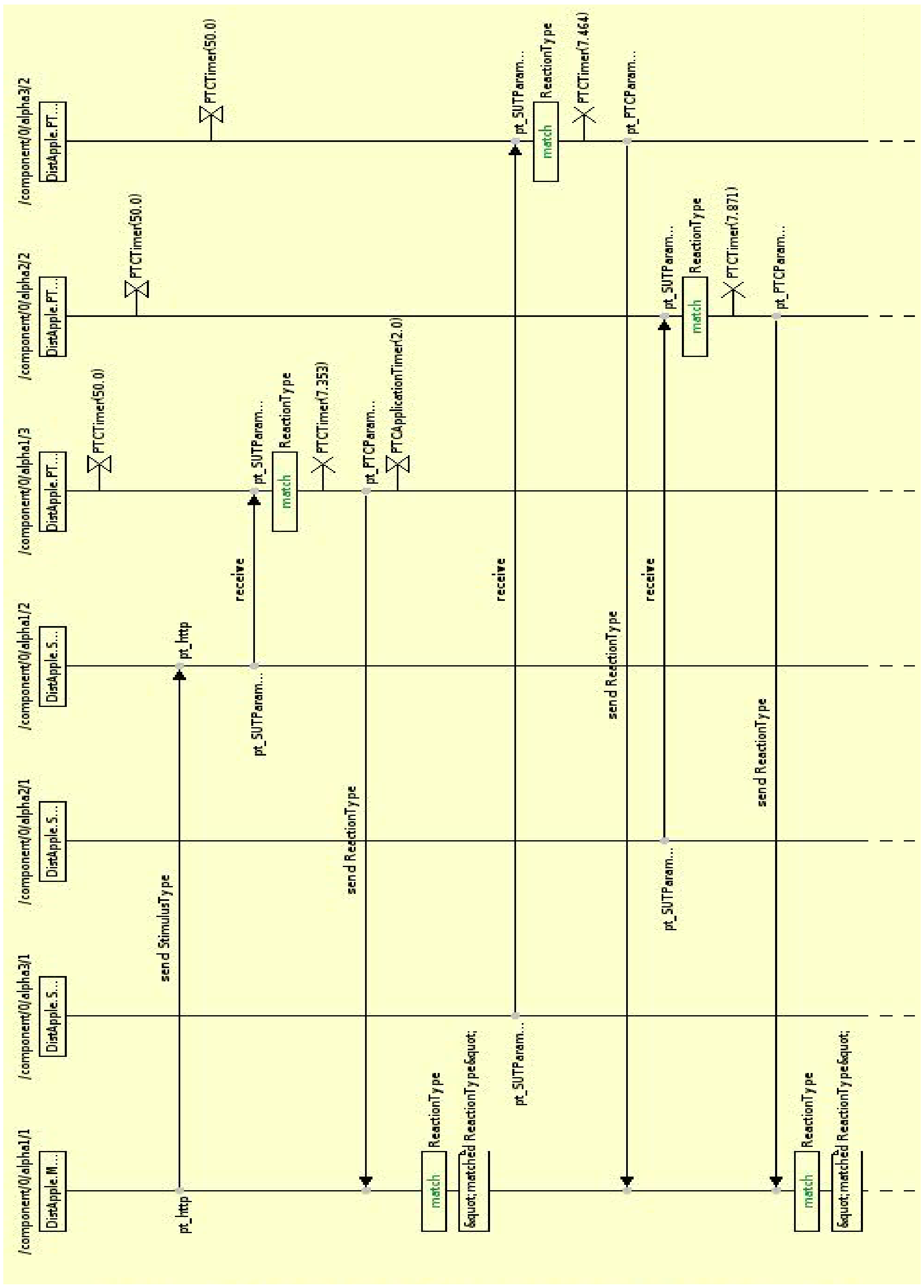


Figure 6.16: Test execution of test case `tc_TAM` *- Graphical logging (part 1)*

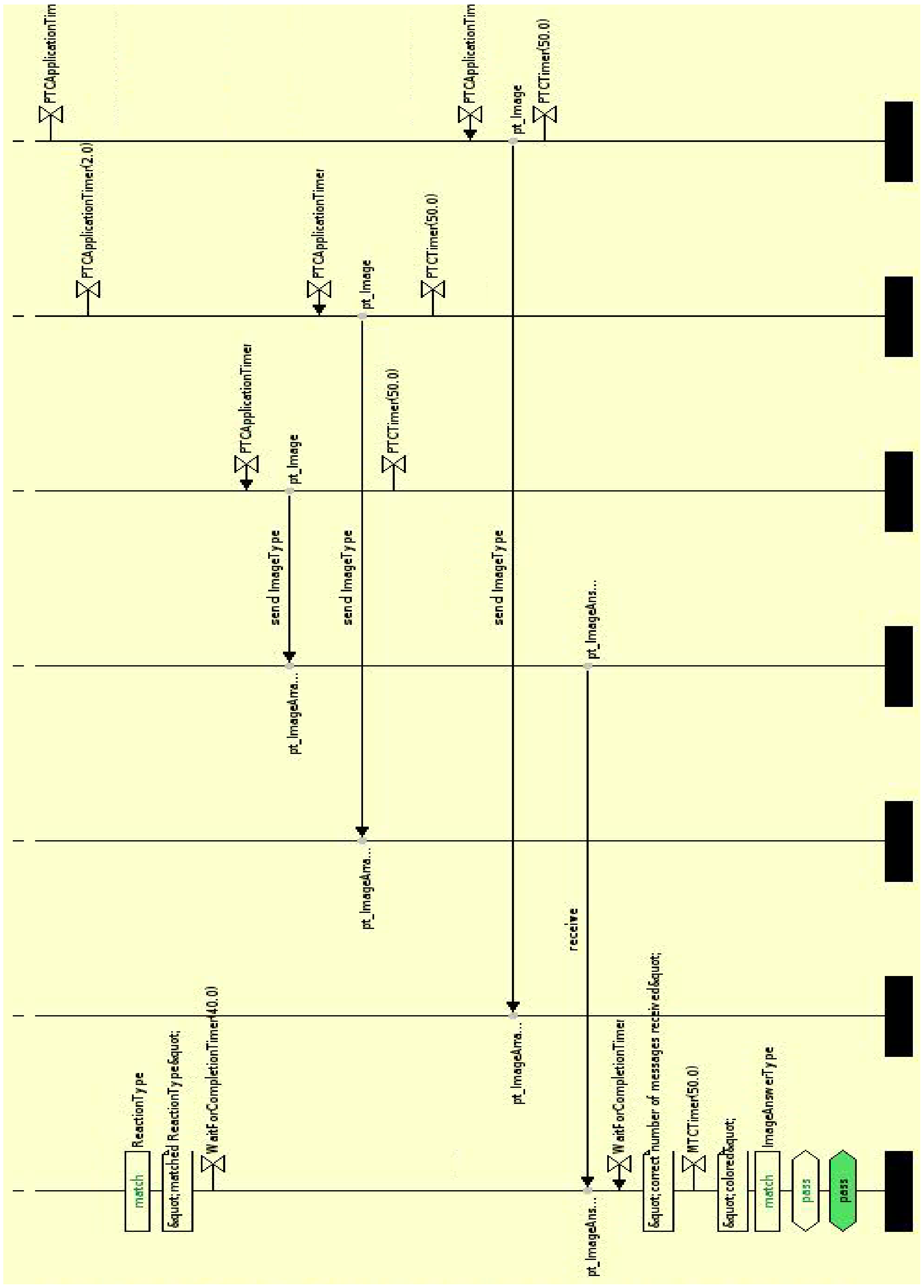


Figure 6.17: Test execution of test case `tc_TAM` *- Graphical logging (part 2)*

After the expected number of messages has been received by the MTC, the timer `WaitForCompletionTimer` is started with a timeout of 40 seconds in order to wait for unexpected messages (Figure 6.17). Afterwards, it is checked by the MTC if the correct number of messages has been received. Within the timeout of `WaitForCompletionTimer`, a message with an image specification is received by the MTC port `pt_ImageAnswer` from the `pt_ImageAnswer` port of the system component. This message matches the type `ImageAnwserType`. Therefore, the verdict is set to `pass`. Because no other verdict than this has been set, the verdict of this executed test case is `pass`.

The TTworkbench also provides a textual logging. Since the logged information includes the same events that are shown in the graphical logging, the textual logging of this test case execution is not further discussed.

7 Conclusion

This chapter summarizes the results, gives an overview of work related to this thesis and provides an outlook. The results are reviewed and discussed in Section 7.1. An overview of work related to this thesis is briefly depicted in Section 7.2. Furthermore, possible prospects are discussed in Section 7.3.

7.1 Summary and discussion

A realization of test cases for Grid computing is essential in order to assure the quality of a Grid computing environment. Testing can raise the user satisfaction with a Grid computing environment or an used Grid application. In this thesis, possibilities of testing an application that runs in a Grid environment using TTCN-3 have been shown, investigated and realized. A Grid example application and its management have been specified and implemented in order to create test items in a Grid computing environment. This application and its management have been analyzed in order to derive test purposes. These test purposes have been realized with test cases in order to evaluate if tests that are using TTCN-3 are applicable to test an application that runs in a Grid environment.

The test case specifications related to these test purposes have been implemented in TTCN-3, while the test harnesses have been implemented in Java. The execution of both test cases demonstrated that TTCN-3 is applicable to test Grid applications. However, the success has to be discussed in relationship to the efforts of specifying and implementing the test cases, the usefulness of the results from the test executions and the reusability of the test cases and harnesses themselves.

In the following, the efforts for implementing and reusing the test environment for a Grid application using TTCN-3 are discussed. For this case study, the efforts of implementing the test harnesses were in the same order of magnitude as for the implementation of the ATS. The implementation of the tests implemented in the case study of this thesis are a base for future developments of tests in a Grid environment. The test cases of this case study have been implemented generically and abstract in order to allow a reuse for similar test environments. The TTCN-3 modules and the test adapters can be reused dependent on the used Grid environment. Therefore, the efforts of adjusting the TTCN-3 test cases and their test adapters differ:

1. when the used Grid middleware changes and
2. when new test cases have to be implemented.

If another Grid middleware than Globus Toolkit 4 is used, the test cases remain unchanged. The efforts have to be invested in the adjustments of the test adapters.

The test adapters have to be changed in order to communicate with the new Grid middleware. That means, for different Grid infrastructures and different Grid application management interfaces the test environment has to be adjusted completely. The TTCN-3 test environment as implemented in the case study of this thesis is preferable applicable for managements, which are realized as web interfaces that run in a Grid computing based on Globus Toolkit 4.

In case a Grid computing environment is used that is based on GT4 but new test cases have to be added, the test adapters can be reused. The implementation of new TTCN-3 test cases are built on the realized test adapter. The realization of these TTCN-3 test cases take the main efforts compared with a minimal effort of adjusting the test adapters.

Much effort was taken to built the Grid environment for the Grid application that had been tested. Grid middlewares, like the Globus Toolkit, normally require a time intensive configuration. Instant-Grid that was used for the case study of this thesis is a Grid environment based on a Linux Live CD that provides an automatic configuration of the Globus Toolkit. Therefore, the effort for these Grid middleware configurations were not so high in the case study. But as the Instant-Grid is integrated in a Live CD image, it is hard to change the distributed test environment or the Grid application. After required configurations, the Live CD image has to be remastered and re-installed in order to be effective. These steps are very time consuming, but may be reduced when using a permanent Grid installation.

7.2 Related work

Testing a Grid environment includes testing of each layer in the Grid architecture. Several Grid initiatives test their computer Grid middleware parallel to their computer Grid middleware developments. The “National Science Foundation (NSF) Middleware Initiative (NMI) Build and Test Lab” provides a distributed, multi-platform framework with automated software building and testing capabilities for a variety of Grid computing projects. This framework is called Metronome [47].

Interesting test environments are provided by the Centre for Development of Advanced Computing (C-DAC). Several test software products are included in the C-DAC Grid Computing Test Suites and Grid Probes that provide for example the “Grid Software-Enabling Applications for Grid Computing Using Globus and C-Language” software. This tool is command-line based and has the objective to check basic Grid capabilities, e.g. remote job submission, validation of proxy or mutual authentication in a Grid environment that is based on GT [52].

The multi-platform and open source software “eInfrastructure for Testing, Integration and Configuration of Software” (ETICS) provides a service to manage complexity and improve the quality of software. This service allows an automation of testing software through using cutting edge Grid software and best practices [48].

Testing Grid computing software is still a challenge in the Grid computing enhancement. Suggestions of strategies for the realization of Grid computing test environments are given in [28]. This include discussions about fundamental test strategies and their realization.

7.3 Outlook

This thesis discussed testing strategies with the scope of testing a Grid computing application using TTCN-3. Additional scopes to cover include testing the Grid middleware itself, realizing performance, stress, end-to-end (response time, auditing, accounting) and usability tests using TTCN-3. The tests have to be performed for all Grid provided services and their combinations, on as many platforms as possible, with full security in place and with the use of meaningful test configurations and topologies. These functionalities can additionally be integrated into the testbed infrastructure created in the case study of this thesis.

Different approaches of realizations of Grid application managements should also be covered by TTCN-3 test solutions. For example, Grid application managements are increasingly realized with the usage of Unified Modeling Language (UML) based workflows, which are required for the determination of the tasks and their synchronized distribution. A realization of interactions between different Grid environments is currently in development, in order to allow global Grid computing. Testing the communication between these Grid environments seems to be worthwhile as well.

Acknowledgments

My special thanks are addressed to Prof. Dr. Jens Grabowski for offering me this Master's thesis in his group and for motivating and advising me during this year. Many thanks to Dr. Helmut Neukirchen who supervised and stimulated my work. Their expertise, understanding and patience, added considerably to my graduate experience. I am grateful for suggestions and ideas from Benjamin Zeiß and Gunnar Krull. I would like to thank Fernanda Dyba, Katrin Rings, Marko Einecke, Michael Cohrs, Adil Faizee, Heiko Voigt and Laura Zechel for proof reading. Finally I would like to thank again Fernanda, Katrin and Marko for their encouragement, and my parents for their unrestricted support throughout all years of study.

Acronyms

API	Application Programming Interface
ATM	Asynchronous Transfer Mode
ATS	Abstract Test Suite
ATSI	Abstract Test System Interface
BMFT	Bundesministerium für Forschung und Technologie
C-DAC	Centre for Development of Advanced Computing
CA	Certificate Authority
CAS	Community Authorization Service
CCF	Container Configuration File
CD	Codec
CD-ROM	Compact Disc Read-Only Memory
CGI	Common Gateway Interface
CH	Component Handler
CLF	Campaign Loader File
CORBA	Common Object Request Broker Architecture
DHCP	Dynamic Host Configuration Protocol
DRS	Data Replication Service
EGEE	Enabling Grids for E-sciencE
ETICS	eInfrastructure for Testing, Integration and Configuration of Software
ETS	Executable Test Suite
FIFO	First In First Out
FIRST	Fraunhofer Institut Rechnerarchitektur und Softwaretechnik
FIZ	Fachinformationszentrum
FTP	File Transfer Protocol
GNU	GNU's Not Unix
GRAM	Grid Resource Allocation and Management
GRIP	Grid Interoperability Project
GSI	Grid Security Infrastructure
GSM	Global System for Mobile communications
GT	Globus Toolkit
GT4	Globus Toolkit version 4
GWDG	Gesellschaft für Wissenschaftliche Datenverarbeitung mbH Göttingen
GWES	Grid Workflow Execution Service
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
IO	Input/Output System
IP	Internet Protocol
IPv6	Internet Protocol version 6
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
IUT	Implementation Under Test
MD5	Message-Digest Algorithm 5
MDS4	Monitoring- and Discovery System
MTC	Main Test Component
NFS	Network File System
NMI	NSF Middleware Initiative
NPACI	National Partnership for Advanced Computational Infrastructure
NSF	National Science Foundation

NTCP	NEESgrid Teleoperations Control Protocol
OGSA	Open Grid Service Architecture
OGSA-DAI	Open Grid Service Architecture Data Access and Integration
OS	Operating System
PA	Platform Adapter
PHP4	Hypertext Preprocessor version 4, previously: Personal Home Page
PoC	Point of Control
PoO	Point of Observation
PTC	Parallel Test Component
PXE	Preboot Execution Environment
RFT	Reliable File Transfer
RGB	Red Green Blue
RLS	Replica Location Service
RTSI	Real Test System Interface
SA	System Adapter
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
SQA	Software Quality Assurance
SQL	Structured Query Language
SUT	System Under Test
TAG	Testing the Application executed on a Grid node
TAM	Testing the Application Management
TC	Test Component
TCDL	Test Component Distribution Language
TCI	TTCN-3 Control Interface
TCP	Transmission Control Protocol
TE	TTCN-3 Executable
TFTP	Trivial File Transfer Protocol
TL	Test Logging
TM	Test Management
TMC	Test Management and Control
TRI	TTCN-3 Runtime Interface
TSI	Test System Interface
TTCN-3	Testing and Test Control Notation Version 3
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URL	Uniform Resource Locator
VO	Virtual Organization
WS	Web Service
WS-I	Web Service Interoperability
XIO	eXtensible and Input/Output System
XML	eXtensible and Markup Language

Bibliography

[1] Parvin Asadzadeh, Rajkumar Buyya, Chun Ling Kei, Deepa Nayar, and Srikumar Venugopal. *Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies. CoRR*, cs.DC/0407001, 2004.

[2] Brett Beeson, Steve Melniko, Srikumar Venugopal, and David G. Barnes. A portal for grid-enabled physics. In Rajkumar Buyya, Paul D. Coddington, Paul Montague, Reihaneh Safavi-Naini, Nicholas Paul Sheppard, and Andrew L. Wendelborn, editors, *ACSW Frontiers*, volume 44 of *CRPIT*, pages 13–20. Australian Computer Society, 2005.

[3] Viktors Berstis. *Fundamentals of Grid Computing.* Redbooks Paper, IBM Corp., 2002.

[4] Manfred Broy, Bengt Jonsson, and Joost-Pieter Katoen. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer, August 2005.

[5] George Din, Sorin Tolea, and Ina Schieferdecker. *Distributed Load Tests with TTCN-3.* In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2006.

[6] Michael Ebner. *UML-based Test Specification for Communication Systems.* PhD thesis, University of Göttingen, 2004.

[7] Ian Foster. *The Grid: A New Infrastructure for 21st Century Science. Physics Today*, 55(2):42–47, 2002.

[8] Ian Foster. *What is the Grid? A Three Point Checklist. Grid Today*, 1(6):22, 2002.

[9] Ian Foster. *A globus primer, An Early and Incomplete Draft.* Technical report, Globus Alliance, 2005.

[10] Ian Foster. *Globus Toolkit Version 4: Software for Service-Oriented Systems.* In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS, pages 2–13. Springer-Verlag, 2005.

[11] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1998.

[12] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, June 28 2002.

[13] Ian Foster, Carl Kesselman, and Steven Tuecke. *The Anatomy of the Grid: Enabling Scalable Virtual Organization. The International Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[14] Ian Foster, Hiro Kishimoto, Andreas Savva, Dave Berry, Abdeslem Djaoui, Andrew Grimshaw, Bill Horn, Fred Maciel, Frank Siebenlist, Ravi Subramaniam, Jem Treadwell, and Jeffrin J. Von Reich. *The Open Grid Services Architecture, Version 1.0.* Technical Report 1.0, The Open Grid Services Architecture Working Group (OGSA-WG), 2005.

[15] Testing Technologies IST GmbH. *Testing Tech TTmex User's Guide.* Testing Technologies IST GmbH, 2007.

[16] Testing Technologies IST GmbH. *Testing Tech TTworkbench Developer's Guide.* Testing Technologies IST GmbH, 2007.

[17] Jens Grabowski. *TTCN-3 – A new Test Specification Language for Black-Box Testing of Distributed Systems.* In *Proceedings of the 17th International Conference and Exposition on Testing Computer Software (TCS 2000), Theme: Testing Technology vs.Testers' Requirements, Washington D.C., June 2000*, June 2000.

[18] Jens Grabowski. TTCN-3 - Testing and Test Control Notation. Presentation, IBM T. J. Watson Research Center, Hawthorne, NY 10532 (USA), November, 13th 2002, November 2002.

[19] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. *An introduction to the testing and test control notation (TTCN-3). Computer Networks*, 42(3):375–403, 2003.

[20] Jens Grabowski and Andreas Ulrich. An Introduction to TTCN-3. Tutorial, Proceedings of 'The TTCN-3 User Conference' at the European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), May 3-5, 2004, May 2004.

[21] Marty Humphrey and Mary R. Thompson. *Security Implications of Typical Grid Computing Usage Scenarios.* In *HPDC*, pages 95–103. IEEE Computer Society, 2001.

[22] Antti Hyrkkänen. *General Purpose SUT Adapter for TTCN-3*. Master's thesis, Tampere university of technology, 2005.

[23] Per Johansson and Hendrik Wallinder. *A Test Tool Framework for an Integrated Test Environment in the Telecom Domain.* D-level thesis, Karlstad University. 2005.

[24] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. *A taxonomy and survey of grid resource management systems for distributed computing. Softw., Pract. Exper.*, 32(2):135–164, 2002.

[25] Yaohang Li and Michael Mascagni. *Grid-Based Monte Carlo Application.* In Manish Parashar, editor, *GRID*, volume 2536 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2002.

[26] Jochen Ludewig and Horst Lichter. *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2007.

[27] Benoit B. Mandelbrot. *Fractals and Chaos: The Mandelbrot Set and Beyond.* Springer New York, 2004.

[28] Alberto Di Meglio. *Grid Software Engineering Challenges.* CERN, The International Collaboration to Extend and Advance Grid Education, 2007.

[29] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, Inc., 2004.

[30] Nataraj Nagaratnam, Philippe Janson, John Dayka, Anthony Nadalin, Frank Siebenlist, Von Welch, Ian Foster, and Steve Tuecke. *The Security Architecture for Open Grid Services.* Open Grid Service Architecture Security Working Group (OGSA-SEC-WG), 2002.

[31] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests.* PhD thesis, University of Göttingen, 2004.

[32] OMG. *The Common Object Request Broker: Architecture and Specification*. Technical Report 2.0, Object Management Group, 1995.

[33] Ina Schieferdecker and Theofanis Vassiliou-Gioles. *Realizing Distributed TTCN-3 Test Systems with TCI.* In Dieter Hogrefe and Anthony Wiles, editors, *TestCom*, volume 2644 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2003.

[34] Marion Schünemann, Ina Schieferdecker, Axel Rennoch, Mang Li, and Claude Desroches. *Improving Test Software using TTCN-3.* GMD Report 153. Forschungszentrum Informationstechnik GmbH. 2001.

[35] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations.* dpunkt., Heidelberg, 2005.

[36] Michael Di Stefano. *Distributed data management for grid computing.* Wiley, Hoboken, NJ, 2005.

[37] Franco Travostino, Joe Mambretti, and Gigi Karmous-Edwards. *Grid Networks - Enabling Grids with Advanced Communication Technology.* John Wiley and Sons, Inc., 2006.

[38] Alain Vouffo-Feudjio and Ina Schieferdecker. *Test Patterns with TTCN-3.* In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 170–179. Springer, 2004.

[39] Ernest Wallmüller. *Software-Qualitätssicherung in der Praxis.* Hanser, 1990.

[40] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiley and Sons, Inc., 2005.

[41] Alexander Willner. *Entwurf und Implementierung einer Ressourcen-Datenbank für das Instant-Grid-Projekt der GWDG*. Master's thesis, University of Göttingen, 2006.

Internet references

[42] Globus Alliance. [Online; http://www.globus.org/ fetched on 05/26/07].

[43] Globus Alliance. *About the Globus Toolkit?* [Online; http://www.globus.org/toolkit/about.html fetched on 05/26/07].

[44] Globus Alliance. *Docs 4.0: globusrun-ws — Official job submission client for WS GRAM.* [Online; http://www.globus.org/toolkit/docs/4.0/execution/wsgram/rn01re01.html fetched on 06/23/07].

[45] Globus Alliance. *GT4 Admin Guide*, 2005. [Online; http://www.globus.org/toolkit/docs/4.0/admin/docbook/ fetched on 05/26/07].

[46] Bob Devaney. *The Mandelbrot Set Explorer - Mathematical Glossary.* [Online; http://math.bu.edu/DYSYS/explorer/def.html fetched on 07/31/07].

[47] National Science Foundation Middleware Initiative Build and Test Lab. *What is Metronome?* [Online; http://nmi.cs.wisc.edu/node/90 fetched on 08/19/07].

[48] CERN. *What is ETICS?* [Online; http://www.eu-etics.org/ fetched on 08/19/07].

[49] Elvior. [Online; http://messagemagic.elvior.ee/index.html fetched on 06/14/07].

[50] EUROGRID. [Online; http://www.eurogrid.org fetched on 05/26/07].

[51] Fraunhofer FIRST and Metarga GmbH. [Online; http://www.ttxp.org/ fetched on 06/14/07].

[52] Centre for Development of Advanced Computing. *C-DAC Grid Computing Test Suites and Grid Probes.* [Online; http://www.cdac.in/HTmL/npsf/gridcomputing/npsfgrid.asp fetched on 08/19/07].

[53] Enabling Grids for E-sciencE. [Online; http://www.eu-egee.org/ fetched on 05/26/07].

[54] CERN European Organization for Nuclear research. *Gridcafe - What is 'The Grid'?* [Online; http://gridcafe.web.cern.ch/gridcafe/whatisgrid/dream/powergrid.html fetched on 04/18/07].

[55] ETSI CTI (Centre for Testing and Interoperability). *What is TTCN-3?* [Online; http://www.ttcn-3.org/WhatisT3.htm fetched on 06/09/07].

[56] Ganglia. [Online; http://ganglia.sourceforge.net/ fetched on 05/26/07].

[57] Testing Technologies IST GmbH. [Online; http://www.testingtech.de/products/ttwb_enterprise.php fetched on 06/14/07].

[58] GO4IT project. *Technical Documentation: General Design Document From Go4IT project.* [Online; http://www.go4-it.eu/modules/mediawiki/index.php/Technical_Documentation:_General_Design_Document fetched on 09/08/07].

[59] Belle Analysis Data Grid. [Online; http://epp.ph.unimelb.edu.au/epp/grid/badg/ fetched on 05/26/07].

[60] Instant Grid. [Online; http://instant-grid.de fetched on 05/26/07].

[61] Open Science Grid. [Online; http://www.opensciencegrid.org/ fetched on 05/26/07].

[62] Gridsphere. [Online; http://www.gridsphere.org fetched on 05/26/07].

[63] Danet Group. [Online; http://www.danet.com/index.php?id=ttcn-3-toolbox&L=6 fetched on 06/14/07].

[64] European Telecommunications Standards Institute. [Online; http://www.etsi.org/ fetched on 06/11/07].

[65] Legion. [Online; http://legion.virginia.edu/ fetched on 05/26/07].

[66] Vyom Technosoft Pvt. Ltd. *One Stop Testing - Your One Stop Guide to Testing?* [Online; http://www.onestoptesting.com/ fetched on 06/09/07].

[67] Hamza Mehammed. *Installationsanleitung für Globus Toolkit 4.0.1*, 2006. [Online; http://www.grid.lrz.de/de/mware/globus/GT4InstallGuide1-0.html fetched on 05/26/07], Leibniz-Rechenzentrum der Wissenschaften.

[68] Sun Microsystems. [Online; http://www.network.com/ fetched on 05/26/07].

[69] Sun Microsystems. *Sun Utility Computing.* [Online; http://www.sun.com/service/sungrid/index.jsp fetched on 05/26/07].

[70] NorduGrid. [Online; http://www.nordugrid.org/ fetched on 05/26/07].

[71] NPACI. [Online; http://npacigrid.npaci.edu/ fetched on 05/26/07].

[72] Australian Virtual Observatory. [Online; http://aus-vo.org/ fetched on 05/26/07].

[73] OpenMolGrid. [Online; http://www.openmolgrid.org fetched on 05/26/07].

[74] OurGrid. [Online; http://www.ourgrid.org/ fetched on 05/26/07].

[75] TOpenTTCN Oy. [Online; http://www.openttcn.com/Sections/Products/OpenTTCN3 fetched on 06/14/07].

[76] POV-Ray. [Online; http://www.povray.org/ fetched on 05/26/07].

[77] Grid Interoperability Project. [Online; http://www.grid-interoperability.org/ fetched on 05/26/07].

[78] The DataGrid Project. [Online; http://eu-datagrid.web.cern.ch fetched on 05/26/07].

[79] SETI@home. [Online; http://seti.alien.de fetched on 05/26/07].

[80] Telelogic. [Online; http://www.telelogic.com/products/tau/tester/index.cfm fetched on 06/14/07].

[81] Gridbus toolkit. [Online; http://www.gridbus.org/ fetched on 05/26/07].

[82] UNICORE. [Online; http://www.unicore.eu/ fetched on 05/26/07].

[83] Wikimedia Foundation, Inc. *Wikipedia: Electric power transmission.* [Online; http://en.wikipedia.org/wiki/Electric_power_transmission fetched on 05/26/07].

[84] Métodos y Tecnología. [Online; http://www.mtp.es/productos.php?id=1 fetched on 06/14/07].

Standards

[85] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. 2007.

[86] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-2 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT)*. 2007.

[87] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-3 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)*. 2007.

[88] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-4 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics*. 2007.

[89] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-5 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)*. 2007.

[90] European Telecommunications Standards Institute (ETSI). *European Standard (ES) 201 873-6 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)*. 2007.

[91] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. 1990.