Aalborg University
Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Øst

TITLE:
A Test Framework for WMPI II

TOPIC:
Distributed Systems and Semantics,
Software Testing

AUTHOR:

Anders Lyhne Christensen

SUPERVISOR:
Josva Kleist

SEMESTER:
DAT7

PROJECT PERIOD:
September 2002 to January 2003

NUMBER OF COPIES:
4

NUMBER OF PAGES:
50

Synopsis:

This report documents the development of a test framework for cost-effective system and integration testing of WMPI II prior to its release. WMPI II is a commercial implementation of the Message Passing Interface (MPI) version 2 developed at Critical Software SA. A short introduction to MPI, software testing concepts, and the architecture of WMPI II is provided.
The requirements for the test framework are developed in three parts. One part for the type of coverage that the framework should provide, one for the functionality it should implement, and finally a part for the user interface requirements. The main idea behind the framework is parameter abstraction, where parameters to functions of the implementation under test are encapsulated in classes. By developing tests based on this concept and by letting instances of abstract parameters act as oracles, the code for testing a single or few WMPI II functions can be reused for input coverage, stress, and volume testing.
The high-level architecture for the test framework is presented and discussed, followed by a presentation of the test plan for WMPI II. The test plan is based on an incremental approach through the features provided by the test framework, such that functions and components are tested in progressively more complex settings.

Contents

Preface

This project is a DAT7 project from the Computer Science Department of Aalborg University Denmark in the area of "Application-oriented research". I have spent six months in Coimbra/Portugal at Critical Software SA working as a software engineer supported by a Marie Curie Fellowship Programme. The mission of the Marie Curie Fellowship Association is to stimulating the training and mobility of researchers in the European Community. The focus of the first part of my fellowship has been to develop and implement a cost-effective platform for testing and debugging WMPI II. WMPI II is a commercial implementation of the Message Passing Interface standard, thus a middleware product for high performance, distributed computing.

Chapter 1 contains a short introduction to the history of MPI, the scope of the project, and definitions of the concepts related to software testing used in the remaining parts of the report. Throughout Chapter 2 the requirements for a test platform are developed and discussed. Chapter 3 contains the high level design of the platform. In Chapter 4 the test plan is presented. Finally, Chapter 5 contains concluding remarks and a discussion on future additions and changes to the test platform.

Readers are expected to be familiar with the basic concepts behind MPI-1 as well as to have some knowledge about additional features in MPI-2. It is advised to keep the MPI-2 specification handy while reading the report, not because any details are thoroughly treated, but because a number of source code examples are provided to illustrate different approaches to testing. The MPI-2 specification is freely available from the MPI Forum's homepage:

`http://www.mpi-forum.org`

Introduction

Chapter 1

Testing is a central ingredient of every software development process, from testing the requirements, analysis, and design, to compiling and running our implementations. Informally, software testing is the process of uncovering evidence of defects in software systems, [Boo01]. Defects can be introduced in any part of the development process as results of bugs - mistakes, misunderstandings, and omission on the part of developers. Complementary methods for testing range from being based solely on human intuition and insight, such as code inspection and code review, to fully automated testing by software, such as automated test generation based on pre- and post-conditions and model checkers. Although complementary, interesting, and often necessary for quality assurance[1], we will not consider nor discuss the whole spectrum of methods in this project. Instead our focus is on *system* and *integration* testing. A definition of these terms and more elaborate scope for this project are given after a concise presentation of MPI 2, the standard implemented by the software, which we want to test.

1.1 The Message Passing Interface

The Message Passing Interface, MPI, standard is a specification of a platform independent API, forming an abstraction layer on which high performance, distributed applications can be built. An MPI-based application consists of a number of independently executing processes, which communicate through the interface provided by MPI. The first version of MPI, released in 1994, specified an interface for point-to-point communication and collective communications. The motivation behind the definition of such a standard arose from the desire of the parallel computing[2] community to have a single library interface for the message passing model for parallel computers, [GHLL$^+$98]. A single library standard would allow scientists and engineers to make practical use of the newest computing technology, [SOO$^+$98]. The effort paid off since MPI currently is the leading standard for message passing libraries for parallel computers, [SOO$^+$98].

In 1997 the MPI Forum published an ambitious version 2 of the MPI standard, which includes new functionalities such as:

- *Process Creation and Management*: A number of processes can spawn new processes at runtime and initiate communication, two independent MPI applications, each consisting of multiple processes, can connect at runtime, exchange information, and disconnect.

[1] Contrary to some definitions of quality assurance we choose to let it include testing.

[2] The term *parallel computing* refers to multiple processes working on a task in parallel with the intent of combining multiple hardware resources in order speed up computation. Parallel computing should not be confused with the use of *concurrency* as a logical mean for structuring applications.

- *Parallel I/O*: Files can be created, written, read, and deleted collectively, meaning that every process has access to all file data, and that data can be read and written using a shared file pointer.
- *One-sided communication*: Remote memory access enables processes to access regions of memory exposed by other processes without their explicit participation.

While a number of implementation of the Message Passing Interface (MPI) standard exist for version 1.x only few, often incomplete, implementations of the MPI-2 standard exist[3]. This is most likely due to the complexity of MPI-2. There are 128 functions in MPI-1 while MPI-2 contains on less that 287 functions and a rich set of additional functionalities. The size of MPI is partly due to the fact that MPI is intended to be used on a variety of platforms, thus the MPI Forum has included multiple functions performing the same operations, but with slightly different semantics, to allow implementers to take advantage of a particular platform. WMPI II is a full MPI-2 implementation for Microsoft Windows currently being developed at Critical Software. The implementation is expected to be among the first full MPI-2 implementations for standard Intel x86 compatible PCs.

1.2 WMPI II

WMPI 1 was a based on MPICH, an open-source implementation of the MPI 1.2 standard. It started as a research project at the Univeridade de Coimbra and was later commercialized. After the release of the MPI-2 standard, it soon became clear that a new core of the implementation was needed, since WMPI 1 relied on a static, shared, global view of the world, meaning that every process known every other process from the very beginning to the bitter end of an execution. With the introduction of dynamic processes, that is process creation and management, the core - and therefore the majority of the existing code - had to be redone since the global view assumption was no longer valid. As opposed to a static, global view of the world, the core of WMPI II now relies on a dynamic and local view in which each process only knows the processes to which it is directly connected at any given point in time.

WMPI II is an implementation of the MPI-2 standard for x86 Intel clusters[4] running Microsoft Windows NT, 2000, or XP. WMPI II is due for release in the beginning of 2003 and one of the remaining tasks is to thoroughly *test* the implementation. This is a difficult task, partly due to the size of MPI-2, but primarily due to its distributed nature. Since WMPI II is a middleware product in can be used in an infinite number of ways by application with unique execution and communication patterns. And since the majority of MPI-2 functions are concerned with transferring data between or synchronizing independent processes, there is no simple input/output relationship for functions. Thus, the challenge lays in finding and developing a cost-effective method for this task.

[3] A list of available MPI implementations and their support for MPI-1 and MPI-2 features can be found on `http://www.lam-mpi.org/mpi/implementations/fulllist.php`.

[4] A cluster is a type of parallel or distributed system that consists of a collection of interconnected whole computers used as a single, unified computing resource, [Pfi98].

1.3 Scope

The scope of this project is as follows: In the context of WMPI II and Critical Software, how do we test the implementation in order to determine whether it has a sufficient level of quality for release.

1.3.1 The Context

The context of this project is Critical Software and the WMPI II team. The team consists of two software engineers and a supervisor on the technical side, a business engineer part time on the project, as well as the business unit manager. Hence, the available resources in terms of developing, running, and responding to test results are limited to the two software engineers. Since Critical Software is a company concerned with staying in business, the release of WMPI II is due sooner rather than later; hence, *time* is an important factor.

1.3.2 Test

The term "test" refers mainly to asserting correctness with respect to the MPI-2 standard. This means that WMPI II should provide reliable message passing with the semantics and via the interface dictated by the specification. The term does *not* refer to testing different algorithms and implementations in order to compare performance; this should and have been done earlier in the development process. The only performance issues considered at this stage, are those resulting from minor code defects, e.g. a thread releasing a critical section later than necessary, thereby blocking other threads and delaying communication.

1.3.3 A Sufficient Level of Quality

An objective measure for quality of an MPI-2 implementation can be defined in a number of ways. At this point in the process the main concern is: *Correctness*. More specifically, the criterion for releasing WMPI II is that it implements the standard as dictated by the specification. Ideally, we want to meet the following criterion:

Given a valid state of the implementation and valid input, every function should provide the functionality according to the standard, terminate, and leave the implementation in a valid state.

Where *state* is defined as the content of all internal data structure of all instances running on all participating nodes, as well as measurable timing of concurrent events. Validity is defined as consistency. *Input* is simply defined as the value of the actual parameters passed to an MPI-2 function.

It is, however, impossible to determine whether the criterion stated above is met by WMPI II based on testing, since tests can never show the absence of defects - only their presence, (Dijkstra). Thus, the only way to determine whether the criterion is really met is to do exhaustive testing, hence run tests for all valid states and all valid inputs. But theoretically there are an infinite number of valid inputs, as well as an infinite number of states so we would need an infinite number of tests.

This issue calls for a less ambitious and more subjective measure of quality in order to determine whether WMPI II is releasable, and at the same time also requires a more selective (creative) approach to the testing process. Ultimately, a "sufficient level of quality" of WMPI II is a business decision and as software engineers it is our job to provide data on which such a decision can be made.

1.4 Concepts, Terminology, and Definitions

In this section we present the concepts, terminology, and definition concerned with software testing used throughout this project.

First we arrange software into a hierarchy of levels and name the types of testing usually applied to the different levels. Then we present different methods of testing, e.g. black-, white-, and grey-box testing, regression testing, stress testing, etc. Finally, we define the terminology that we use in the remaining part of the report.

1.4.1 Software and Testing Levels

Although hard to define rigorously software can be divided into a number of levels for which different types to testing are applicable. At one extreme we have the statement level, and at the other we have application level. In this project we will use the following levels:

- *Statement:* The statement level refers to individual statements such as arithmetic expressions and decision statements leading to branching based on one or more conditions.
- *Function:* Functions are the elements of the specification or the code that define distinct pieces of system functionality. E.g. the elements at the function level in MPI-2 are the individual function specified by the standard such as the `MPI_Send` function.
- *Component:* A component is a part of a system comprising a collection of related functions, such as all point-to-point functions or functions related to process creation and management. Components are also known as *functional units* or *modules*.
- *Application:* The application level refers to the complete system; hence all components described in the system specification. In our case the application level is the entire MPI-2 API.

Depending on our perspective we can sometimes view the same piece of code as being on different levels. E.g. for a client WMPI II can be viewed as a single component, while during early development a single (complex) MPI-2 function might be viewed as a component, since it calls a number of internal functions. Given the current phase of the development process we choose the division presented above.

There is no general term for testing at the statements level[5], however for the remaining three levels the following terms are normally used:

[5] Normally *white-box* testing is associated with the testing performed at the statement level. However, strictly speaking white-box testing refers to a test case design method for which the goal can be some kind of coverage, but not in general to tests performed at the statement level, [Mye79]

- *Unit Testing:* Unit testing is the process of testing individual functions of an application *in isolation*. Unit testing is usually the first type of testing applied during the implementation phase of a piece of software.
- *System Testing:* System testing is the process of testing individual component in isolation.
- *Integration Testing:* Integration testing refers to the process of testing multiple integrated components, which eventually make up the complete implementation.

1.5 Testing Methods

When we perform unit, system, and integration testing, we design our test cases based on different information and with the intent of uncovering different kinds of defects. Two main categories of tests can be referred to as *dynamic analysis* and *static analysis*. In dynamic analysis code is tested by executing it, whereas static analysis bestows on external applications analyzing the source code for possible defects. Some static analysis is performed by modern compilers, like type-checking, detecting unreachable code, and usage of uninitialized variables. In this project, however, we focus only on dynamic analysis.

1.5.1 Black-box, White-box and Grey-box Testing

Black-box testing, also known as *specification-based* testing, is a method where software is tested solely based on its specification. In some cases this can be done formally with a high degree of automation, if the specification is present in a formal, mathematical form - sufficiently precise, concise, and unambiguous, [Sto93]. In specification-based testing the tester sees the implementation under test as a black box and tests are developed without consideration as to how the inside of the box is implemented.

White-box testing is a method where the source code of the implementation under test is used extensively to specify and select test cases. The focus of white-box testing is on testing the code, whereas the focus of black-box testing is to test if an implementation lives up to its specification (or requirements). White-box testing usually involves measuring coverage. Coverage, e.g. what a test case or a test suite covers, can be measured different ways, some of which are the following:

- **Statement coverage:** A measure that reports the number of statements of an implementation under test, which have been encountered during the execution of a test case or a test suite.
- **Decision coverage:** A measure that reports whether boolean expressions tested in control structures have evaluated to both `true` and `false`.
- **Condition coverage:** A measure that reports the `true` or `false` outcome of each boolean sub-expression.
- **Path coverage:** A measure that reports whether each of the possible paths in each function has been followed. A path is a unique sequence of branches from the function entry to the exit.

- **Race coverage:** A measure that reports whether multiple threads execute the same code and/or use the same resources concurrently.

Grey-box testing is a hybrid between black-box and white-box testing. The objective of grey-box testing is like black-box testing to test whether an implementation lives up to its specification; however, the internal structure of the implementation is used to select and design test cases.

1.5.2 Regression Testing

During a software development and maintenance cycle existing code is modified, due to defect correction, change, addition or removal of functionality. In order to verify that new defects have not been introduced or that the code changes have not regressed other parts of an implementation during such modifications, *regression testing* is performed.

1.5.3 Volume Testing

In volume testing the implementation under test is subjected to large amounts of data, [Mye79]. E.g. in WMPI II a volume test could be to start an absurdly large number of processes (>10.000) or sending a message of one gigabyte.

1.5.4 Stress Testing

In stress testing the implementation under test is subjected to heavy loads. This should not be confused with volume testing; a heavy load is a peak volume of data encountered over a short span of time, [Mye79]. E.g. a large number of threads trying to send a message at the same time, one process continuously sending to another process, a group of processes repeatedly spawning new processes.

1.6 Test Terminology

In order to test a piece of code through execution, we feed the code some input and records its output and determine if it is valid. The piece of code can be of a statement, function, component, or application, and *input* and *output* can be stated more or less formally. For a decision statement we might define the input in term of an assignment of the variables, which are evaluated in the decision, and the output can be defined as the value of the instruction pointer after executing the statement. For a function reading n bytes from a file, the input could be a file handle and the valid output could be n bytes from the file. For a component, e.g. a lexer, the input could be a file name and the output a list of tokens. For an application, like a C compiler the input could be the filename of a C file and the valid output an executable consistent with the source code.

1.6.1 Test Case

A test case is a single instance of a test, hence a single pair of input data and a set of valid output data. A test suite is a set of test cases.

1.6.2 Input and Output Domains

The input domain (ID) refers to the types and the values of the input parameters to a test. Since we are concerned with system and integration level testing of WMPI II, the input testing refers to entities such as buffers, communicators, windows, files, etc. The *valid input domain* (VID) is the sub-domain of the ID for which a function is defined according to the system specification. E.g. the ID for `MPI_Bcast` is any values for a buffer (pointer), a count (integer), a dataype (`MPI_Datatype`), a communicator (`MPI_Comm`), and a root (integer). The VID for `MPI_Bcase`, on the other hand, is the sub-domain, for which the buffer is a valid pointer, pointing to a chunk of memory large enough to contain count (non-negative) instances of the datatype, and where the value of root is valid for the communicator specified ($0 \leq$ root $<$ communicator size). The output domain (OD) refers types and values of the output of a test.

1.6.3 Oracle

An oracle is the means by which the output of a test is used to judge the correctness of the operation performed by the implementation under test. This can be done by hand or automated oracles can be implemented. If the oracle function is performed by hand the tester can modify the source code in order to allow for manual inspection of output, e.g. dump output to the console or a file. If the oracle function is performed automatically, already validated data from previous versions, an alternative implementation, or data generated by the oracle code itself, can be compared to the output of the piece of code under test to assert its correctness. In some cases an automated oracle needs only to perform a consistency check, e.g. an oracle for a test of function, which is supposed to output a prime number, need only to verify that the output is indeed a prime number; hence, it does not need to generate the prime numbers itself.

1.6.4 Success/Failure

Intuitively, a successful test from the point of view the developer responsible for the implementation under test, is one that shows that the implementation behaves correctly according to the specification. However, a contrasting view exists, namely that testing is a destructive activity where one attempts to find defects in a program. A test case simply showing that an implementation behaves in accordance with its specification is considered a waste of time since this result cannot be used to improve the quality of the implementation under test. We choose to adopt the latter view[6], namely that a successful test is one that gives evidence of defects in the implementation under test.

1.6.5 Equivalence Partitioning

Since exhaustive input testing, e.g. running test cases for each element in the VID, is impossible for all but the simplest programs, some selection among the possible test cases has to be

[6] For a more elaborate discussion on the subject, please refer to "The Art of Software Testing" by Glenfold J. Myers, and his ideas on the psychology behind software testing.

performed. Equivalence partitioning is used to divide the VID into a finite number of subdomains for which each element has the same error detecting ability as the others. E.g. if we have verified that we are able to send one element of the datatype `MPI_Char` of value 5 using `MPI_Send`, it would be reasonable to expect that we can also send an element with the value 6, etc. Hence, effort is most likely better spent developing different test cases, e.g. for testing `MPI_Send` with integers, doubles, user-defined datatypes, etc. In case we are dealing with the amount of data transferred, we have to be careful though: If we are able to send a small amount of data, it does not necessarily mean that we can send large amounts of data, since buffering on the receiving side is used for small messages, while a rendezvous protocol is used for bigger messages. Hence, some knowledge about the implementation under test can help to refine the equivalence partitions.

1.6.6 Random Testing and Boundary-value Analysis

In random testing, the elements in a domain used in test cases are chosen at random. In boundary-value analysis, on the other hand, functions are tested based on the boundaries of the input and/or output domain. If we for instance want to test a function with the input domain $\{-10, -9, .., 10\}$, instead of selecting any element, we chose -10 and 10. Continuing the discussion above on the size of the data transferred: If we apply boundary-value analysis to message sizes, we choose the largest message for which buffering is used and the smallest message for which the rendezvous protocol is used.

1.7 The State of WMPI II September 2002

Before we go into details on what to test and how, we take a brief look at the state of WMPI II, its overall architecture, what has been implemented, what has already been tested, and how it was done.

1.7.1 The Architecture

WMPI II is a layered architecture. The architecture is shown in Figure 1-1 on the facing page. At the bottom resides a number of *devices*, or device drivers, which exposes a common interface to the WMPI Management Layer (WML). The devices are responsible for connecting and communicating with other processes (local and remote), and creating new processes. Currently, there are two devices implemented, namely *tcp* and *shmem*. The tcp device is a *threaded device* uses the TCP/IP stack as the means of communication. The term *threaded device* means that part of the device runs in its own thread in order to respond to message arrivals as they occur. The other device, shmem, uses shared memory for inter-process communication for processes residing on the same machine. The shmem device is non-threaded, thus the code of the device essentially runs in the same thread as the client application. This means that new messages are only handled (passed to an operation waiting for a particular message or put on a queue for unexpected messages) when the device is instructed to do so.

The middle layer of WMPI II consists of the WMPI Management Layer (WML). This is where the core functionality is implemented. The chosen shape, wide at the top and more narrow

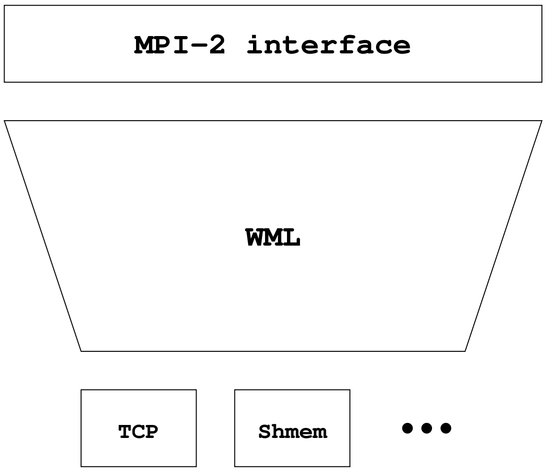

Figure 1-1: The high-level architecture of WMPI II. The bottom layer consists of a number of devices. In the middle layer is the WMPI Management Layer where the core functionality is implemented. The MPI-2 interface at the top is responsible for checking parameter consistency and converting parameters from MPI types to WML types (e.g. mapping handles to pointers).

at the bottom, is meant to illustrate that all the WML functionality, this being collective communication, parallel IO, one-sided communication, and so on, relies on simple point-to-point primitives, which are implemented at the bottom of the WML.

The top of Figure 1-1 is occupied by the MPI-2 interface. Almost all MPI-2 functions, as described by the standard, have their counterpart WML function. The MPI-2 interface is responsible for checking parameter consistency and converting parameters from MPI types to WML types, such as mapping handles to pointers and calling the corresponding WML function.

1.7.2 The Development

The development WMPI II has been performed in a number of stages by a number of different developers. Besides from being a commercial product, the components of WMPI II have largely been developed as research projects in collaboration with the Universidade de Coimbra. Thus, many different developers have worked on separate parts of the implementation from time to time.

1.7.3 The Tests

Due to the initial research nature of WMPI II, where the focus has primarily been on performance and not on testing, quality assurance has not been top priority. However, during the final stages of the development phase some attention has been devoted to a more structured approach to testing. Three types of testing have been applied:

- **Unit testing:** Some of the part of the WMPI II implementation has undergone unit testing using the freely available package "CPP Unit", [SFL+02]. The user-defined datatypes for instance, has been unit tested by testing the each of the functions involved in creating, packing, and unpacking the datatypes, *in isolation*.

- **System testing:** Some system testing has been performed through in-house developed, standalone applications, each testing a single or few functionalities within a component. These kinds of tests have been named "Completeness tests" and their sole purpose has been to verify that all necessary parts of a component have been implemented, hence that *a module is complete but not necessarily correct*.

- **Stress/Volume:** Tests combining stress and volume testing for a few selected areas have been written and run.

From the list presented above it is obvious that no integration testing has yet been performed. The completeness tests have proved difficult to use for regression testing, first of all because they consist of numerous standalone applications, in few cases one for each MPI function but often multiple applications for each function, and secondly because they are written for a particular setup, that is a fixed number of processes. This makes them ill-suited for regression testing since they are not easy to maintain and run. Their coverage in terms of the state of WMPI II is also limited since each of the tests has the same simple structure: Initialize MPI, call one or more functions to test a particular functionality, and finalize MPI. In that sense the "completeness tests" have proven not to be complete.

The stress/volume tests suffers from the same problem as many of the completeness tests, namely that they are tied to a specific hardware setup. Moreover, they only stress few, selected parts of WMPI II.

1.7.4 Conclusion

Based on the architecture of WMPI II and the testing that has been done, we permit ourselves to draw the following conclusion: Based on the layered architecture, where much of the functionality in the WML is built on top of the same core, which again is built on top of one thread and one non-threaded device, we need some tests, which stresses the state of the implementation using multiple hardware configurations. Moreover, the completeness tests are insufficient for uncovering defects, thus a more thorough, structured method of testing is necessary. The focus should be on developing a way of implementing and running tests, such that tests provide better coverage, and so that they are easier to maintain. Another part of specification with has not yet been submitted to much testing is the thread-safety of WMPI II. It has been designed to be thread-safe, but tests are needed to verify the implementation is indeed thread-safe.

Requirements

CHAPTER 2

In this chapter we develop requirements for a test framework in three stages. First we discuss and describe what we want to test, hence what situations we need to catch in our test cases (model layer). Secondly, we discuss what functionalities the test framework should provide (function layer). Finally, we discuss two user interfaces, one for tester executing tests and debugging - the external user interface, and one for the test case developer - the internal user interface (interface layer).

2.1 The Task

The MPI-2 standard defines 287 functions. The brute-force way of testing these functions is to write separate tests for each one. However, if we consider this strategy it has a number of drawbacks. To illustrate why we use an example. Say we want to test `MPI_Send`, we could write the following program[1]:

```
#include<mpi2.h>
#include<stdio.h>
/* Run with two processes only! */
int main(int argv, char** argv) {
  int        nRank;
  int        nPayload = 0;
  MPI_Status msStatus

  MPI_Init(argv, argc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    nPayload = 42;
    MPI_Send(&nPayload, 1, MPI_INT, 1, 100, MPI_COMM_WORLD);
  } else {
    MPI_Recv(&nPayload, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, &msStatus);
  }

  MPI_Finalize();
```

[1] We use the Hungarian notation in our examples. In this notation identifiers are given names in which information about their type is embedded. In this way the type of a variable or a complex type can be determined based on the prefix of its name, e.g. classes are prefixed by `C`, objects by `c`, integers by `n`, pointers by `p`, etc. For instance `pvBuffer` a void pointer to a buffer.

```
  if (nPayload == 42) {
    printf(``Test failed'');
    return -1;
  } else {
    printf(``Test successful'');
    return 0;
  }
}
```

Let first consider what we have tested: If this test fails, hence we actually do the transfer, we know that MPI_Send and MPI_Recv can transfer an integer of the value of 42. If this is the case it would be reasonable to assume that we could transfer an integer of any value[2]. Unfortunately, not much else can be concluded based on a fail test run. Can we transfer two integers? Three integers? A million? Can we transfer a char? Can we use other communicators than MPI_COMM_WORLD? Can we send an integer in the other direction (from rank 1 to rank 0)? These questions cover only the dimension of the input to the two functions. Another set of questions concerning the state of the implementation can be raised: Can we do another MPI_Send, MPI_Recv pair before calling MPI_Finalize? Are sends and receives done correctly when we use multiple communicators simultaneously? Can we do a collective operation before an MPI_Send or a MPI_Recv? Can we do one after[3]?

Another important issue here is that of *execution paths*: Running the test case shown above *once* will test *one* execution path. Either rank 0 will send a message to rank 1 before rank 1 is expecting it (before MPI_Recv was called). Another possibility is that rank 0 enters MPI_Send while the matching MPI_Recv is waiting for the message. Finally, the scenario where the rank 0's message arrives while rank 1 has entered MPI_Recv but is not yet expecting the message.

In every one of the three cases different execution paths are tested, in the first example (**A**) in Figure 2-1 on the next page rank 0's message is put on the *unexpected* queue, and when rank 1 enters MPI_Recv it searches the queue, find the message and performs the receive. In the second case (**B**) the message is expected, and is handed directly to MPI_Recv. In the third case (**C**), when the message from rank 0 arrives at rank 1 while rank 1 is *in the process* of setting up the receive, we should have an execution path of either of the two cases described above. Hence, strictly speaking we do not have a third execution path, however it is important to test this case to determine if sufficient and correct mutual exclusion is done in the code. E.g. imagine that the message from rank 0 arrives while the thread of MPI_Recv is searching through the unexpected queue, no matching message is found, so MPI_Recv registers that it is expecting a message and starts waiting. However, just before the registration is done,

[2] In a heterogeneous environment this assumption might not be valid, e.g. in an environment containing both little-endian and big-endian architectures we would need to boundary-value analysis to test data conversions routines, however since WMPI II is presently only implemented to run on Intel x86 architectures the assumption can be made.

[3] The same questions could be raised about I/O, dynamic process creation, and one-sided communication: Does any other communication, valid according to the standard, have an influence on whether we can do sends and receives? And vice versa?

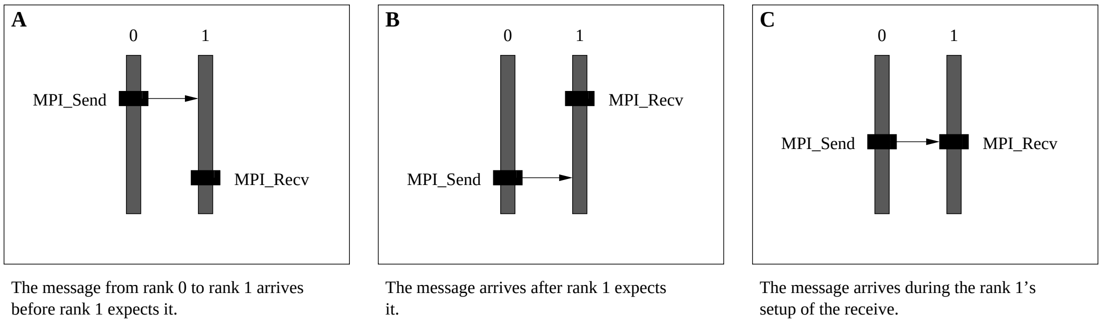

Figure 2-1: Three possible execution paths to consider in the simple send/receive test.

the thread of the device that have received rank 0's message finds no operation waiting for the message and puts it on the unexpected queue. We have a deadlock since the code of `MPI_Recv` will not search through the unexpected queue again but expects to get notified, however, this will not happen since the device has already finished processing the message.

Thus, another question can be added to the list: How do we test the different execution paths? Since we are running multiple processes, the only way that we can be absolutely sure that every one of the execution paths is executed is by implementing some external synchronization mechanism. However, such an external mechanism is 1) time-consuming and might require code changes in WMPI II, and 2) might have side effects which "silence" defects that could otherwise have been detected. By running the send/receive test a large number of times we might assume that we have tested the different execution paths.

The questions in the paragraphs above are summarized as the coverage requirements for the set of test cases that we want to run.

2.1.1 Coverage Requirements

Below is the list of abstract coverage requirements for the system and integration testing of WMPI II:

1. **Input coverage:** The test cases have to provide boundary-value analysis.
2. **State coverage:** The test cases have to provide coverage of different states of WMPI II. Hence, some test cases have to include a sequence of MPI-2 calls, where some are executed serially and some in parallel (in multiple threads).
3. **Path coverage:** The test cases have to provide some path coverage for different states of WMPI II; hence some (not necessarily perfect) mechanism for controlling the order of concurrent events is needed.
4. **Race coverage:** The test cases have to provide some race coverage in order to check the thread-safety of WMPI II.

Notice, that we have chosen to exclude "path coverage" from "state coverage". Some would rightfully argue that execution path could be considered part of the state, however, since the state determined by the functions executed *before* a test case, while the execution path is determined *during* a test case, the two requirements are likely to be meet by different mechanisms, hence the separation.

2.2 A Test Framework

From the discussion of the coverage requirements and the simple send/receive example presented in the previous section, it seems clear that these requirements can hardly be meet by a set of test cases each implemented as individual executables. The number of executables would simply be astronomical, and even if this was not the case, the tests would contain a significant amount of identical code, e.g. setting up MPI, creating and querying communicators, and finalizing MPI (known as *scaffolding functions*). Hence, it seems reasonable to abandon the individual executables in favor of a *single* application comprising all test cases. We call this system the "Test Framework" (TFW). The fundamental idea behind the TFW is based on *parameter abstraction*.

2.2.1 Parameter Abstraction

Ideally, we what to develop the TFW so that we only write each function of the MPI-2 specification as few times as possible, while still fulfilling the coverage requirements. This means that the code for testing one or more function should be reusable for completeness, regression, stress, and volume testing. One way to achieve this to some degree is through parameter abstraction.

To illustrate this concept, assume that we wish to test the `MPI_Send` function:

```
MPI_Send(void *buffer,
         int count,
         MPI_Datatype datatype,
         int *rank,
         int *tag,
         MPI_Comm comm);
```

Obviously, the function takes one void pointer, three integers, a value of the type `MPI_Datatype`, and a value of the type `MPI_Comm`. However, if we consider the semantics of the parameters we know that the first three parameters, `buffer`, `count`, and `datatype` are all interdependent. The `buffer` parameter needs to point to a valid memory region, large enough to hold `count` elements of `datatype`. Hence, each of the three parameters are essentially attributes of the same concept: The data that we wish to send. Likewise, `comm` and `rank`, are attributes of the same concept: The destination of the send.

Now if we encapsulate these two concepts in two classes, `CMessage` and `CP2PComm` (for point-to-point communication), we can write the part of a test for `MPI_Send` where the function is actually called in the following way:

```
MPI_Send(cMsg.GetBuffer(),
         cMsg.GetCount(),
         cMsg.GetDatatype,
         cP2PComm.GetRecvRank(),
         nTag,
         cP2PComm.GetComm());
```

where cMsg and cP2PComm are instances of CMessage and CP2PComm, respectively. At this point it might seem that we have just added more code to the implementation by doing the encapsulation, however, the advantage of this approach becomes clear if we consider the similarities between tests of similar MPI-2 functions. Given that the code for the CMessage and CP2PComm classes has already been written, testing a function like MPI_Isend can be implemented to use instances of those classes, since its prototype:

```
MPI_Isend(void *buffer,
          int count,
          MPI_Datatype datatype,
          int *rank,
          int *tag,
          MPI_Comm comm,
          MPI_Request *request)
```

only differs from that of MPI_Send by the additional request parameter of the type MPI_Request*. Hence, to write the part of a MPI_Isend test where the function is called in the following way:

```
MPI_Isend(cMsg.GetBuffer(),
          cMsg.GetCount(),
          cMsg.GetDatatype,
          cP2PComm.GetRecvRank(),
          nTag,
          cP2PComm.GetComm(),
          &request);
```

Furthermore, the CMsg class can be reused for tests for collective operations like MPI_Bcast. We can even promote reusage further by extending the class methods to capture the concept of *operators* used in MPI functions such as MPI_Reduce, as well as partial messages for tests of functions such as MPI_Gather and MPI_Scatter.

2.2.2 Granularity

To meet the coverage requirements, we need fine granularity, in the sense that if we write a test where multiple MPI calls are made, we want to be able to interleave those calls and other MPI calls. E.g. if we have a test containing a call to MPI_Isend and a call to MPI_Wait, we want to be able to call another test, e.g. one testing `MPI_Bcast`, in between the two, to test if WMPI II implements the specification on this point. Of course we could implement a new test comprising all three functions, however then we are half-way back to starting point where special tests needed to be developed for every sequence of calls. If we instead divide each test into a number of *steps*, where some MPI function is called in each step, then the TFW could interleave steps from different tests. E.g. the `MPI_Isend` test could be implemented in the following way:

```
void ExecuteStep(int n_step) {
  if (n_step == 0) {
    MPI_Isend(..)
    ...
  } else if (n_step == 1) {
    MPI_Wait(..)
    ...
  }
}

unsigned int GetNumberOfSteps() const {
  return 2;
}
```

The TFW can then call the `MPI_Isend` first with `n_step = 0`, then call the `MPI_Bcast` test, and finally call the `MPI_Isend` test with `n_step = 1`.

2.2.3 Oracles

Until this point we have not discussed how oracles should be implemented in the TFW. First we will consider what is required of oracles in the TFW and then how these requirements fit into the idea of parameter abstraction.

After a call to e.g. `MPI_Send` we need to determine:

1. If the correct operation was performed, hence whether the correct data is received at the destination.
2. If `MPI_Send` terminates.
3. If `MPI_Send` leaves WMPI II in a valid state, thus we should be able to perform other MPI function after calling `MPI_Send`.

Anyone of the oracle functions could be performed by hand, however, this is both time-consuming and error-prone. Therefore, we strive for at much automation as possible. The first oracle function, checking whether the correct data is received at the destination, is most conveniently checked at the destination. If we assume that objects `cMsg` and `cPT2PTComm`, *identical* to those at the sending side, are present at the receiver, we write the following code:

```
MPI_Recv(pvTempBuffer,
         cMsg.GetCount(),
         cMsg.GetDatatype(),
         cPT2PTComm.GetSendRank(),
         nTag,
         cPT2PTComm.GetComm(),
         &msStatus);
```

Now that received message is stored in `pvTempBuffer` of the type void pointer, while the *correct* message should be known by the `cMsg` object. Hence, we let the object `cMsg` do the job of verifying that the correct message has been received:

```
if (!cMsg.CheckBuffer(pvTempBuffer))
    [...ERROR!...]
```

In this way we can write a single test for `MPI_Send` and `MPI_Recv`, which can be used in test cases for messages of any datatype and any size.

The second oracle function, to check that `MPI_Send` terminates cannot - for obvious reasons - be verified by the test itself. Hence, we will have to rely on some external mechanism, another thread or another process, to act as the oracle in this case. We will refer to this mechanism as the *watch dog*.

The third oracle function, to determine whether WMPI II is left in a valid state after a call to `MPI_Send` can be tested in different ways. One could define a validity criterion based the values of all variables and data structure within the running implementation. This strategy, although quite thorough, is hardly practical, since it, first of all, is difficult to take a "snapshot" of a concurrent system, secondly the implementation (80.000+ lines) is prohibitively large, and finally the oracle would not be very maintainable. If only subsets of the implementation is considered such a semi-white-box strategy might be useful, not only for testing but also for debugging.

A different approach, less thorough but far more practical, is simply by testing. After `MPI_Send` has been tested, we re-run the test possibly with different `cMsg` and `cPT2PTComm`, or we could run a different test. The only major consequence of this strategy is that if a test succeeds, hence an incorrect behavior is detected, we cannot conclude that the incorrectness is related to the function(s) tested immediately before it is detected. A successful test case might be successful because the state before the test case was not a valid one, thus a symptom of a defect introduced by a previous test case.

2.2.4 Functional Requirements

The following is a list of functional requirements:

1. **Regression testing:** The TFW should be usable for regression testing. This means that is should be relatively easy to run a battery of test cases giving broad coverage. Test selection should also be flexible so that a more thorough set of test cases concentrating on the area of modification can be run. Regression testing is not allowed to take a significant amount of time in order to be used.

2. **Volume and stress testing:** The TFW should be useful for volume and stress testing. It should not be necessary to write special tests for this purpose.

3. **Multi-threaded:** The test framework should allow and support concurrent test cases executing in different threads.

4. **Compatibility:** A minimum of effort should be necessary to convert existing test suits to the TFW.

5. **Logging and statistics:** The TFW should be able to log and output progress and coverage information. To achieve good software quality it is necessary to keep track of what have been tested, when, and the results of the tests.

6. **Performance:** The TFW should spend a minimum amount of time on house-keeping and the amount of logging should be configurable. Experience has shown that generating and outputting log information can force specific execution paths, thereby degrading path coverage and thus hiding defects.

7. **Reusable in other projects:** The TFW should be generic and flexible enough to eventually be used in other projects. Since a considerable amount of resources has been dedicated to development of the framework, management would like to see it reused in other projects. Thus, the design has to be clean and that the framework logic has to be separated from the MPI specific tests.

8. **Boot mechanism:** A part of the TFW should be a boot mechanism, which can successively start executions of different sets of test cases. Such a boot mechanism is necessary since a number of defects, often related to the state of the implementation, can be detected only during the clean-up phase in `MPI_Finalize`. Unfortunately, `MPI_Init` and `MPI_Finalize` can only be done once in a process, hence each time we want to call `MPI_Finalize` we have to shut down the current process, start a new one, and continue testing. Since we do not want to do this manually, part of the TFW should be a TestBooter.

9. **Debugging:** The TFW should provide some debugging functionalities. Even though debugging is normally not considered part of the testing process, there are two major reasons for including some functionalities to assist the debugging process in the TFW: Obviously, we want to correct the defects found, and secondly while developing tests the tests themselves might contain defects. Debugging functionalities will speed up test development and strengthen the confidence in the test results.

10. **Maintenance phase:** The TFW should be usable in the maintenance phase. In this phase clients often report defects. The symptoms of these defects are usually discovered

either due to specific hardware configurations or due to a specific pattern of usage. In the latter case, the TFW should be usable to implement a test of the sequence of MPI calls leading to the client discovered defect. The purpose is two-fold: Since the TFW has some debugging functionalities, these might help in finding the defect and a sequence of events leading to it, and secondly by implementing a test specific for a previously discovered defect will improve the quality of the regression testing.

2.3 Test Development and Execution

TFW has two faces with respect to the user interface. One is the interface to the compiled framework, which could either be by command-line or some graphical interface, hence what we traditionally mean when we *user interface*. The second interface is that provided by the TFW to the test developer.

2.3.1 External User Interface Requirements

In the first version the test framework is intended for developers, who know what they are doing. This means that the interface, configuration, and output should sacrifice overview for flexibility and immediate access to functionality. Hence, a command-line and configuration files are preferred over a graphical user interface. A later version might be shipped to clients in order to collect detailed information in case of site-specific problems. In this case a graphical interface might become relevant.

The following is a list of requirements for the external user interface for the first version:

1. **Configurable through XML**: The TFW should be configurable through Extensible Markup Language (XML) files. We require the configuration to be sufficiently flexible, so that no recompilation and redistribution of executables on a cluster of machines is necessary, unless one or more new tests have been implemented. We want developers to be able to create one configuration file, run the TFW, change the configuration file, re-run the TFW, without recompiling the framework. XML has been chosen, since it is a well-defined standard sufficient expressive for our purposes. Given the wide use of XML a number of tools are available for authoring and validating XML files.

2. **Automatic distribution of configuration**: The TFW should automatically distribute the configuration from one central place to all nodes. Since human precision scales badly with cluster size it is beneficial to rely on an automatic mechanism to keep all nodes synchronous with respect to configuration. Much time has been wasted on tracking "defects" arising from incompatible versions of software and configurations.

3. **Progress output**: The TFW should output progress information (progressively outputting what test cases are being executed). The progress output should be in an XML format compatible to the input configuration format. Thus, the progress output of a run leading to a successful test case, should be usable as input for a new run. This can be practical for debugging purposes.

4. **Test report**: The framework should output test reports (what has been tested, when, and results) to a user-defined source, e.g. a file or database.

5. **Debug output**: The framework should output debugging information (test specific) to a user-defined source, e.g. a file or socket.

2.3.2 Internal User Interface Requirements

The internal user interface consists of architecture of the test framework visible to test developer, including the functionalities it offers and the restrictions it imposes. The idea is that each specific test inherits, either directly or indirectly, from an abstract test class. There are no rigorous requirements for the internal user interface, only the following design guideline:

The interface should be as simple as possible in terms of the amount of code needed for each test and the amount of knowledge about the internal structure of the TFW required on behalf of the test developers.

2.3.3 Test Requirements

The test requirements, are requirements for the tests implemented in the TFW themselves as well as for the TFW. The TFW should operate correctly for tests meeting these requirements. Tests are required to meet the following requirements provided that the system under tests lives up to its specification:

1. **Terminate:** A test is required to eventually terminate.

2. **Thread-safe:** Each test is a class, and the TFW is required to execute each step of each instance serially, in ascending order. However, when multiple threads are running the same test might be instantiated in more than one thread, hence any access to global, non-thread-safe resources are up to the test to synchronize.

3. **Independent of hardware:** Tests are required to operate correctly independent of the hardware configuration, e.g. a test cannot require a cluster with an even number of machines to operate correctly. Tests are allowed to inquire about the current hardware configuration and skip the actual test code if the current configuration is not compatible with the test. However, this behavior is only allowed for compatibility reasons, and is not recommended. The user should be notified in case a test is “skipped” due to configuration incompatibilities.

4. **Complete:** A test must complete all operations started in the test. This means that a test issuing an `MPI_Isend` must also issue a matching `MPI_Wait`. From this requirement and requirement 1 it follows that the test must also ensure that a matching receive is issued by any destination rank. Moreover, each step must be complete, meaning that if a test issues an `MPI_Send` on step x in one process, the matching receive operation done by the destination process also has to occur in step x. This requirement is necessary to prevent deadlock when steps of test cases are interleaved.

5. **Idempotent:** Given the same input a test is required to perform the exact same operation, in order know exactly what has been tested. This is especially useful for completeness and regression testing, as well as for debugging. Naturally, we do not require a test to provoke a deterministic order of events in a concurrent system under test, we merely require that given some specific input a test performs a deterministic set of interactions (function calls and the values of their actual parameters) with the system under test, in a deterministic order.

High-level Design

CHAPTER 3

In this chapter we present the high-level design of the TFW. We will not go into detail about specific interfaces between different components but focus on the ideas and motivation behind the overall design, how responsibilities are divided, and provide some examples.

In order to make the TFW reusable in other projects we have designed an abstract framework, which captures the concept of parameter abstraction, test scheduling, configuration, test reports, and debugging functionalities. We have then *instantiated* the framework for WMPI II; hence implemented MPI specific tests, abstract parameters classes, and other functionalities in order to test WMPI II. The design presented in the following section is the design of the abstract framework, while the examples are taken from the WMPI II specific framework - WMPI II Test Framework (W2TFW).

Both frameworks are implemented in C++.

3.1 The Class Diagram

The architecture of the TFW is shown in Figure 3-1 on the following page. At the top resides the test manager(`CTestManager`), which is aggregates a number of SyncGroups (`CSyncGroup`) and is associated to a configuration source (`CConfigurationSource`), a watch dog (`CWatchDog`), and a results manager (`CResultManager`).

Each SyncGroup aggregates a number of resource combiners (`CResourceCombiner`), resource pools (`CResourcePool`), and a number of tests (`CAbstractTest`). The resource pools each again aggregates a number of resources (`CResource`). These classes and the structure will be explained in further detail in the following sections.

In order to make Figure 3-1 more readable a few classes have been omitted: Every class shown in the figure inherits from a common class named `CFrameworkObject`, which has a number of convenient methods like the `toString` method similar to that of Java. Each framework object also has a name (a string) in order to ease configuration and debugging. Every configurable object inherits from a class called `CConfigurableObject`. Moreover, a number of helper classes such specialized STL containers have been omitted.

3.2 Tests

Tests in the TFW should be general, meaning that the code written to test one or a few functions should be usable for testing those functions in completeness, stress, and volume test cases. When tests contain only few MPI functions they can serve as building blocks for larger more complex test cases without any code modifications.

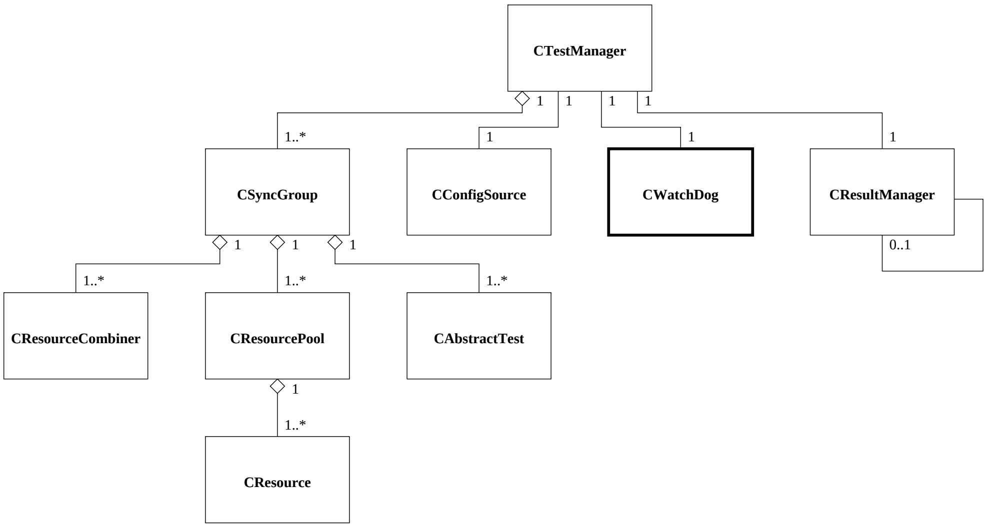

Figure 3-1: The class diagram of the Test Framework.

A test is divided into one or more steps, e.g. in a test for `MPI_Isend` and `MPI_Irecv` there is one step for setting up the receives, one for the sends, and one for waiting (`MPI_Waitall`) for completing the communication.

The class `CAbstractTest` serves as the super class for all tests implemented in the TFW. It implements a number of auxiliary functions for outputting debug information and obtaining the abstract parameters. Furthermore, `CAbstractTest` defines two methods that a subclass needs to overwrite in order to implement a test which can be instantiated:

```
class CAbstractTest : CConfigurableObject {
   ...
   virtual int GetNumberOfSteps() const = NULL;
   virtual int Execute(CTestInfo* pcTestInfo) = NULL;
   ...
}
```

The method `GetNumberOfSteps` returns the number of steps into which a test is divided, while `Execute` contains the actual code for a test. An instance the class `CTestInfo` contains information on the current step to execute.

By keeping `CAbstractTest` simple we mainly address two of the requirements. Namely that the internal user interface should be simple, and that the TFW should be compatible with existing test cases not written for the TFW. An existing test cases can relatively easy be converted to the TFW by implementing a class inheriting from `CAbstractTest`, specifying one step, and copying the code for the test case into `Execute`.

Error reporting and handling are naturally important issues in the TFW. As can be seen on the partial interface above `Execute` returns an `int`. There are two ways that a test can inform the TFW that an error has occurred (a successful test), namely by returning a special value or by throwing an exception (operating system exceptions such as access violations are also caught). We allow for both ways to ease test development. Sometimes it might be convenient for a test developer to return a function result directly, e.g.:

```
class CBarrierTest : CAbstractTest {
    virtual int GetNumberOfSteps() const { return 1; }
    virtual int Execute(CTestInfo* pcTestInfo) {
        return MPI_Barrier(MPI_COMM_WORLD);
    }
}
```

While in other situations it might be more convenient to use exceptions, for instance when a test relies on another method, possibly from a different object such as an instance of an abstract parameter, to check consistency of some received data, it does not have to check for and propagate a possible error code.

Allowing two ways for an implementation of the `Execute` method to report errors adds only little complexity to the TFW, since `Execute` is only called during the inner loop of the test execution (see page 29); hence, it is only in a single place that the two types of error reporting need to be handled.

3.3 Resource and Resource Pools

The TFW is based on parameter abstraction, meaning that parameters to MPI functions are not hardwired in tests, but instead represented by objects. Moreover, meta parameters are parameters passed to tests which are not directly used in MPI function calls, but describe some other flexible/modifiable behavior of a test. For example a test for `MPI_File_open` (See page 211 of the MPI-2 specification, [LILS+97]):

```
MPI_File_open(MPI_Comm comm,
              char* filename,
              int amode,
              MPI_Info info,
              MPI_File* file);
```

The abstract parameters would be the communicator, filename, and the access mode (`amode`), while meta parameters could denote whether or not the *info* object should be used to specify the file server, if a file should be created before calling `MPI_File_open`, etc.

From the TFW point of view abstract parameters and meta parameters have the exact same purpose: A combination specify exactly one test case. When we refer to abstract and meta parameters we simply refer to them as *resources*. Resources can be everything from entities specific to the implementation under test, such as windows, datatypes, files, etc. in MPI-2, to more general types such as integers, integer pairs, strings, booleans, and so on. A class implementing a resource should inherit from `CResource`.

Resource pools are resource containers, which must contain one or more resources of the same type, e.g. a resource pool of windows, a resource pool of datatypes, a resource pool of integers, etc.

3.4 Resource Combiners

Resource combiners are responsible for combining resources for a test. Usually a resource pool contains more than one resource, and the resource combiner will create a number of combinations, $|C|$, where each combination should be valid for the test. The test is then run $|C|$ times, where different combinations are used in each *iteration*.

A simple resource combiner creates all possible combinations: Assume that a test uses the resource pools $RCP_1, RCP_2, \ldots, RCP_n$. All possible combinations would then be: $C = RCP_1 \times RCP_2 \times \ldots \times RCP_n$. In some cases the number of combinations might become prohibitively large. Assume, for instance, that we run the send/receive test with using integer resource pool for the sending rank, one integer resource pool for the receiving rank, and one resource pool for the datatype. If we have a communicator with 40 processes and 10 different datatypes, the number of possible combinations will amount to $|C| = 40 \cdot 40 \cdot 10 = 16.000$. Time is probably better spent testing some other functions, thus a different resource combiner can be developed, which combines resources such that each every resource in every resource pool is included in a combination at least once (hence the number of combinations are determined by the resource pool containing the largest number of resources).

A few tests might require special resource combiners, namely tests which have convex input domains. For instance, a test which writes to file using `MPI_File_write` can only use datatypes that are compatible with the elementary type of the process' current view of the file, (See page 226 of the MPI-2 specification, [LILS$^+$97]). This means that not all combinations of files and datatypes are valid, thus a special resource combiner can be developed to sort out compatible combinations. Alternatively, the code for determining resource compatibility for a test could be placed in the test itself. This alternative, however, has two drawback: If a test does not actually do any testing due to incompatible resources, CPU cycles are wasted and the quality of stress tests is deteriorated, secondly there are no less than 12 functions(!) for writing to files, hence the resource combiner can be reused for tests for all of those functions. Due to these drawbacks we recommend developing special resource combiners for tests with convex input domains, however, the framework allows for both approaches.

Resource combiners should inherit from `CResourceCombiner`.

3.5 SyncGroups

SyncGroups are containers for tests and resource pools. The term "SyncGroup" has been chosen since tests running in different, communicating processes need to be synchronous in the sense that all members of a communicator need to agree on what to do and in which order.

Different processes might be members of different communicators and SyncGroups allow us to do the synchronization in a clean way, since SyncGroups in the W2TFW correspond to a single intra-communicator. Processes that are members of a SyncGroup (intra-communicator group) execute the tests contained in the SyncGroup, while non-member processes do not.

3.6 The Test Manager

The `CTestManager` is a singleton object residing at the top of the class diagram (See Figure 3-1 on page 26). The test manager is responsible for scheduling and running tests, communicating with the watch dog, and passing the results of tests to the result manager. The test manager is a container for a number of SyncGroups.

The test manager's main function is to schedule tests. Tests can be scheduled in four ways:

1. **Test Interleaving:** SyncGroups are selected in a round-robin fashion and tests within the SyncGroups are selected in a round-robin fashion. Each test is run for the number of iterations dictated by the resource combiner before a new test is selected from the next SyncGroup.

2. **Iteration Interleaving:** A test is selected from each SyncGroup and one iteration (all the steps of a test) is run before an iteration of the test from the next SyncGroup is run. When a test has been run for the number of iterations dictated by its resource combiner, a new test from the same SyncGroup replaces it.

3. **Step Interleaving:** A test is selected from each SyncGroup and one step is run for the selected tests in a round-robin fashion. When all iterations of a test have been run it is replaced by a new test from its SyncGroup.

4. **Multi-threaded:** The test manager spawns a number of threads so tests in different SyncGroups are executed concurrently. Tests from SyncGroups assigned to the same thread can be scheduled using any of three the scheduling schemes described above.

One might expect the multi-threaded scheduling is the type of scheduling that puts most stain (tests best) on the implementation under test. However, this is not always true. Stress testing, for instance, often requires the same test to be run over and over again with as little overhead as possible. Thus in some situations test interleaving is the better choice. Moreover, debugging is likely to be easier using one of the more coarse scheduling schemes if using such a scheme can reproduce the symptom of a defect.

3.6.1 Test Execution

Before any tests are run a method called `PrepareRun` is called for all objects such that output files can be created, resources initialized and combined, threads created, and so on. When the clockwork has been synchronized the following method is called:

```
void CTestManager::Run(CThreadConfig* pcThreadConfig) {
  // Get the first test info:
  CTestInfo* pcTestInfo = GetNextTestInfo(pcThreadConfig);

  // While no one has requested termination and there are more
  // test steps to run, run one step:
  while (!GetTerminate() && pcTestInfo != NULL) {
    int nResult;
    CAbstractTest* pcTest = pc_info->GetTest ();

    // Inform the result manager what step we are about
    // to execute:
    m_pcResultManager->BeforeStep (pc_info);

    // Try to run the step:
    try {

      nResult = pcTest->Execute(pc_info);
      // Store the result:
      m_pcResultManager->Store(nResult, pc_info);
    }

    // Catch exceptions thrown by a framework object:
    catch (CTestException cException) {
      m_pcResultManager->Exception(cException, pc_info);
    }
    // Catch OS exceptions:
    catch (unsigned int nOSException) {
      m_pcResultManager->SystemException(nOSException, pc_info);
    }

    // If we do not have to terminate, get the next
    // test step to execute:
    if (!GetTerminate())
      pcTestInfo = GetNextTestInfo(pcThreadConfig);
  }
}
```

Thus, we keep on executing test steps until all test steps have been executed or until some object has requested termination. Usually, the result manager will request termination after an incorrect return code or an exception has been encountered, however, other objects such as the watch dog and even tests are allowed to request termination as well. The result manager implemented for the W2TFW treats all return codes except from `MPI_SUCCESS` as fatal. The result manager is called before and after execution of each step to allow for progress and debug output. If the result manager were only to be called after executing a test step it would be tedious to determine the step causing a hang or a crash, especially for a scheduling scheme such as step interleaving.

The method `GetNextTestInfo` is responsible for selecting the next test to run based on the selected scheduling scheme. Each time called it returns an instance of `CTestInfo`. One instance of the class `CTestInfo` exists for each instance of a test, it contains a reference to the test itself, the current iteration, the current step for the test, and the resources to use for the current iteration.

3.7 Configuration

One of the main motivations behind developing the TFW was to reduce the time spent on implementing tests by making the code for a test reusable in multiple test cases. Thus, for this scheme to work, a convenient way of setting up and configuring test cases is necessary - without the need for recompilation or code changes for each new setup. But what does it mean to *configure* a set of test cases? We want to be able to specify which test cases to run, this means that we have to be able to specify which SyncGroups should be created, how tests should be scheduled, which resources tests should use, and finally which tests should be run. We also need to be able to configure the type and amount of debug and process output that the TFW should generate.

We have chosen XML as the format of configuration files. Alternative, we could have introduced a proprietary format or a more powerful language like Python, however XML was chosen for since it has a sufficient level of expressiveness. Moreover, XML is a widely used standard, which means that a number of authoring tools exists. Its wide usage also means that developers from other projects already know or relatively easy can acquire knowledge to read and modify configuration files. We use a freely available, multi-platform XML engine called *Xerces*, [Pro02].

3.7.1 Configuration File Example

An example of a simple configuration file is given below:

```
<testmanager class="CWMPITestManager" name="testmanager">
  <outputter class="CFileOutputter" name="outputter" level="10">
    <filename>project_example.tfw</filename>
  </outputter>

  <syncgroup class="CWMPISyncGroup" name="Collective">
    <resourcepool class="CIntegerPool" name="root" create"true">
      <range comm="*"/>
    </resourcepool>

    <resourcepool class="CDatatypePool" name="datatype">
      <resource class="CIntDatatype" name="int"/>
      <resource class="CVectorDatatype" name="vector"/>
      <resource class="CDoubleDatatype" name="double"/>
    </resourcepool>

    <resourcepool class="CIntegerPool" name="buffersize">
```

```
      <resource class="CIntegerResource" name="small">
        <value>1</value>
      </resource>
      <resource class="CIntegerResource" name="big">
        <value>100000</value>
      </resource>
    </resourcepool>

    <resourcecombiner class="CInterpolationCombiner" name="combiner">
       <maxcombinations>10</maxcombinations>
    </resourcecombiner>

    <test class="CBcastTest" name="bcast"/>
    <test class="CScatterTest" name="scatter"/>
    <test class="CGatherTest" name="gather"/>
  </syncgroup>
  <run/>
</testmanager>
```

The configuration shown above creates one SyncGroup containing three collective communication tests. One for `MPI_Bcast`, one for `MPI_Scatter`, and one for `MPI_Gather`. There are three resource pools used by the tests: A resource pool of integers containing one integer for each rank in the communicator (`<range comm="*">`), one containing the datatypes to use, and one integer resource pool of buffer sizes. Moreover, the SyncGroup uses a resource combiner, which creates at most 10 combinations.

Objects have both a `class` attribute and a `name` attribute. We need the class name to in order to instance an object of the correct subclass, while the `name` attribute (the name of an object) is used to identify its purpose. Notice that both the buffer size and the root resource pools contain integers, they are both of the class `CIntegerPool`, but they are used for different purposes. The names of the resource pools (`root` and `buffersize`) identify their purpose in the example shown above.

3.7.2 Configuration and Inheritance

If the example configuration file from the previous section is compared to the class diagram in Figure 3-1 on page 26 it is obvious that the XML tree corresponds to this architecture. The `<testmanager>` tag contains the `<syncgroup>` tag, which again contains the `<resourcepool>`, `<test>`, and the `<resourcecombiner>` tags, and so forth. In order to make the TFW flexible the configuration file is parsed recursively: The test manager is handed the subtree of the `<testmanager ...>` tag. When the `<syncgroup ...>` tag is reached an object of the class `CWMPISyncGroup` is instantiated and passed the subtree of the tag, etc.

If a different design had been chosen where configuration files were read and interpreted by either the test manager or some external entity, the framework would have been monolithic, since this entity would have to know every possible option of every SyncGroup, resource pool, test, etc. By letting individual objects parse their parts of the configuration files modifications and additions are kept local.

In order to make the parsing of configuration files easy for test developers, all configurable objects inherits from a common super class, CConfigurableObject. In case a test developer needs to respond to a custom tag all that is required is to register a tag handler. A tag handler is a function of the following type:

```
typedef void (CConfigurableObject::*TagHandler)(DOMElement* pc_dom_element);
```

where the DOMElement is a node in the XML tree.

For example, in the integer resource pool named "root" in the configuration example of Section 3.7.1 on page 31, resources are not created explicitly:

```
...
<resourcepool class="CIntegerPool" name="root" create"true">
  <range comm="*"/>
</resourcepool>
...
```

instead <range comm="*"/> is used to specify that one integer resource should be created for each process in the communicator of the SyncGroup. This is implemented by in the class CIntegerPool, which registers a tag handler for the "range" tag.

3.8 Example: A MPI_Bcast Test

Below is the source code for a MPI_Bcast test taken from the W2TFW:

```
/*
    Constructor - Specify the used resource pools:
*/
CMPIBcastTest::CMPIBcastTest(const string& str_obj_name,
                             CSyncGroup *pc_parent) :
                             CWMPITest(str_obj_name, pc_parent)
{
  UseResourcePool (RESOURCEPOOL_NAME_ROOT);
  UseResourcePool (RESOURCEPOOL_NAME_INTRACOMM);
  UseResourcePool (RESOURCEPOOL_NAME_DATATYPE);
  UseResourcePool (RESOURCEPOOL_NAME_BUFFERSIZE);
}

/*
    GetNumberOfSteps: One step:
*/
int CMPIBcastTest::GetNumberOfSteps() const {
  return 1;
}
```

```
/*
    Execute:
    - Get the resources used.
    - Allocate memory for the broadcast data.
    - The root initializes the data.
    - Do the broadcast.
    - Check the data.
    - Free memory used for the broadcast data.
*/
int CMPIBcastTest::Execute (CTestInfo* pc_info)  {
  // Get the resource for this combination:
  CCommResource*     pcCommResource       = (CCommResource*)
                       GetResource(RESOURCEPOOL_NAME_INTRACOMM);
  CIntegerResource*  pcBufferSizeResource = (CIntegerResource*)
                       GetResource(RESOURCEPOOL_NAME_BUFFERSIZE);
  CIntegerResource*  pcRootResource       = (CIntegerResource*)
                       GetResource(RESOURCEPOOL_NAME_ROOT);
  CDatatypeResource* pcDatatypeResource   = (CDatatypeResource*)
                       GetResource(RESOURCEPOOL_NAME_DATATYPE);
  unsigned int nBufferSize = pcBufferSizeResource->GetValue();
  unsigned int nRoot       = pcRootResource->GetValue() %
                                   pcCommResource->GetSize ();

  // Allocate memory for the broadcast data:
  void* pvBuffer = malloc(nBufferSize * pcDatatypeResource->GetBufferSize());

  // The root rank fills the buffer:
  if (nRoot == pcCommResource->GetRank ()) {
    pcDatatypeResource->InitBuffer (pvBuffer, nBufferSize, nRoot);
  }

  // Do the broadcast - in case MPI_Bcast returns anything but MPI_SUCCESS
  // return the error code:
  MPI_CALL (MPI_Bcast, (pvBuffer,
                        nBufferSize,
                        pcDatatypeResource->GetDatatype (),
                        nRoot,
                        pcCommResource->GetComm ()));

  // Check the data received:
  pcDatatypeResource->CheckBuffer (pvBuffer, nBufferSize, nRoot);

  // Free memory and return:
  free (pvBuffer);
  return MPI_SUCCESS;
}
```

In the constructor the resource pools used by the this test are specified. Every resource pool is identified by its name (`RESOURCEPOOL_NAME_...` are string constants). In the `Execute` method the resources for this test case are located through a member method `GetResource`, which obtains the resources from the resource combiner through the `pc_info` object; thus test developers need not be concerned with interfacing to the resource combiner. Memory is allocated by all processes based on the buffer size and the datatype, and the root process fills this memory. The broadcast is made, every process checks the data for consistency, and memory is freed.

In the example, memory is allocated and freed in each iteration. Since memory allocation may take time, thereby degrading the quality of stress tests, an alternative approach is to move the memory allocation and freeing into a method called `BeforeFirstInteration` and a method called `AfterLastInteration`, respectively. These methods are called either before any (after all) tests are run or right before the first (after the last) iteration of a test is run, depending on the configuration of the TFW.

Side note: Moving the freeing of memory also allows for some clean-up in case `MPI_Bcast` fails and throws an exception or returns an error. However, we are not too concerned about memory leaks in this case since WMPI II is not required to be fault tolerant. Hence, in case an error occurs, the state of WMPI II is undefined and following test results are considered useless.

3.9 Lessons Learned

After having implemented a large number of tests, resources and resource pool, and used the W2TFW for testing and debugging, we have found that *documentation* is an important but difficult issue. When test cases are being designed it is essential that the tester is able to 1) determine what is tested, and 2) what configuration options are available for tests and resources, without consulting the source code for each individual test. This is not a new issues resulting from using the TFW, similar problems exist with previously developed test suites for WMPI II. Our solution so far has been to clearly state what functions are tested and to give an example configuration file in the header files for tests. This solution, however, is far from optimal, since it still requires a significant amount of effort on behalf of the tester designer, as well as discipline on behalf of the test developer when changes are made tests.

A possible solution could be to develop an interactive configuration module on top of the TFW. Currently all tests, resources, resource pools, and resource combiners explicitly register tag handlers, hence which tags they recognize. Moreover, tests register which resource pools they use. Thus, the framework could be extended with a module that enables users to design test cases interactively.

Developing tests for the W2TFW has proved more challenging than developing standalone executables (the previous approach to testing in the WMPI II project), since the tests written in the framework should be as generic as possible. Designing and developing a test in such a way that it is as flexible and reusable requires thinking. However, we view this consequence of using the TFW as an advantage, since the majority of software developers perform best when being challenged, [Hum97].

Test Planning

Chapter 4

In this chapter we present and discuss the test plan for WMPI II. This includes addressing two fundamental questions, namely how do we select is a good set of test cases and when do we stop testing?

First we discuss the strategy for deriving a test plan. Secondly, we give an overview of what has been tested previous to the introduction of the TFW in the WMPI II project. Finally, we present the test plan for WMPI II and discuss how the features provided by the TFW have been used in the design of the test plan.

4.1 Strategy

A test plan for testing WMPI II should include:

- What should be tested.
- What methods should be applied.
- How test cases are prioritized.
- Which kind of defects should be corrected and when.

We want to focus our effort on the most important functionality and the most severe defects. Unfortunately no objective metrics exist for neither test effectiveness, functionality importance, nor the severity of defects, [Hut01]. If some metrics are applied, they are usually defined for a specific project, department, or organization, based on implementation under test, system specification, time, cost, and to some extend experience, [Hut01, KPM01]. In the following paragraphs we will discuss these aspects in general and how they apply to the WMPI II project.

Test Effectiveness

A measure of the effectiveness of a set of test cases can be based on the number of defects it finds. However, if the effectiveness of test cases is only measured by their success rate, there is no guarantee that the selected test cases sufficiently cover the functionality of the implementation under test.

A more useful measure of test effectiveness is (from [Hut01]):

$$E = \frac{\text{Defects found while testing}}{\text{Total number of defects}},$$

where the "Total number of defects" is the sum of the number of defects found while testing and the number of defects found by users after the product release. Hence, an effective set of test cases should test the functionality that clients are likely to use. In case of WMPI II "expected usage" is difficult to classify precisely, since it is a middleware product, and different clients are likely to use different areas of MPI-2 depending on their application. However, we define expected usage in the following broad terms:

A typical client uses multiple components of WMPI II for high performance applications.

Thus, we do not expect clients to be focused on a single component, such as collective communication or process creation and management, but expect that client applications use functionality from multiple components of WMPI II. Some clients might use one set of components while others use a different set depending on the clients' needs. Moreover, we expect clients to use WMPI II for high performance applications, where CPU and network loads are high for longer periods of time (hours or even days). Hence, in order to maximize test effectiveness we have to prioritize integration testing (multiple areas) as well as stress tests (high CPU and network load), and the test cases should be run continuously for longer periods of time.

Severity

In "Software Test - Methods and Metrics", [Hut01] the defects are classified by severity according to the following criteria:

- **Severity 1:** Program ceases meaningful operation.
- **Severity 2:** Severe function error but application can continue.
- **Severity 3:** Unexpected result or inconsistent operation.
- **Severity 4:** Design or suggestion.

These severity classes might work well for GUI applications such as word processors, spreadsheets, browsers, web sites, etc. but they are hardly applicable to WMPI II. An MPI-2 implementation allows users to build distributed applications in order to utilize multiple hardware resources to perform complex tasks. Thus, in many situations an application built on MPI-2 may run for a long period of time with little or no user intervention. This means that a defect in WMPI II of Severity 2 (Severe function error but application can continue) or Severity 3 (unexpected result or inconsistent operation) is likely to result in a crash or a hang in a client application; hence a defect of Severity 1 (Program ceases meaningful operation). Even if a defect of Severity 3 does not cause a crash or a hang, we would still render it catastrophic, since an unexpected result or an inconsistent operation could produce an incorrect final result.

We categorize defects into three, less optimistic, classes of severity:

- **Non-deterministic:** Defects which result in any incorrect operation (hang, crash, or incorrect result) non-deterministically, e.g. "*Sometimes* an application hangs or crashes after one hour". These defects are hard to reproduce and find, and usually they are the hardest to fix, since they are often related to a specific timing of events.

- **Deterministic:** Defects which result in any incorrect operation (hang, crash, or incorrect result) deterministically, e.g. "New processes cannot be spawned unless rank 0 of the communicator is the root". These defects are usually not as hard to locate as non-deterministic defects, and they are often caused by missing or incorrect implementation of a one or a few functions in the upper layers of the WML. Deterministic defects may be dependent on sequence of MPI function calls performed in the same thread, but they are *not* related to timing of events such as the order message arrivals, locking of critical sections, etc.
- **Cosmetic:** Defects which cause inconvenience but has no influence on the stability or performance of the implementation, e.g. "If `MPI_Send` is given an uncommitted datatype, it does not return the correct error code".

Both the non-deterministic and deterministic can be fatal to a client application; however, we consider the non-deterministic defects as the more severe of the two classes. Due to their nature, they can cause great troubles to clients: Parallel applications are not easy to debug and if we throw in a couple of non-deterministic defects in WMPI II, it may become virtually impossible. Clients are likely to spend a significant amount of time diagnosing defects of this class. Deterministic defects, on the other hand, are usually easier to diagnose, report, and work around.

Before releasing WMPI II we will not allow any defects classified as non-deterministic, whereas deterministic defects are accepted to some degree. In case a deterministic defect is found a choice is made to either fix the defect or remove and document the absence of the functionality. If cosmetic defects are found, they are corrected if this can be done with little effort; otherwise they are registered and corrected after the initial release.

Prioritizing Functionality

One of the central new features in MPI-2 is Process Creation and Management, this includes functionality for creating new processes and joining independent computations at runtime. For WMPI II process creation and management is considered the most important new component. The main reason is that the ability to create processes at runtime is the functionality is expected to make some Parallel Virtual Machine (PVM) users move to MPI.

The two other new areas, namely parallel I/O and one-sided communication, are naturally important too, but since WMPI II is currently for Microsoft Windows only, these features have less impact compared to dynamic processes. The Windows file system, NTFS, does not natively support parallel I/O like IBM's GPFS, [SH02]. Moreover, Windows clusters tend to be loosely connected, which means that the performance of one-sided communication does not differ significantly from that of point-to-point operations (although some algorithms might be expressed more elegantly using remote memory access than point-to-point communication).

Historical Data

Historical data on recorded defects in earlier versions of a software product, similar projects, and even performance data on individual developers can be used to improve the testing process.

This data can form the basis for an empirical model, which to some extend can be used to predict the number of defects injected in each phase of the software development process, [Car02]. The predictions made by such a model can be used to estimate the remaining number of defect as well as to estimate the reliability of a software system, [KM97].

However, a quantitative approach to software quality is part of the Capability Maturity Model's level 4, [PCA+91], which Critical Software has not yet reached. Given the history of the WMPI II project, which started as a research project within distributed systems and not quality assurance, introducing an extensive quantitative approach at this phase of the development process is likely to be neither possible nor beneficial.

We will use some historical data to derive a test plan. We will not use any empirical or statistical models, but instead rely on the following simple phenomenon:

"The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.", [Mye79, page 15]

This might seem counterintuitive, but if we consider the defect-finding-rate, hence the number of defects found during some unit of time, a component with a comparatively high rate would indicate many defects. Naturally, this is only true if we make the (bold) assumption that defects in different components are equally hard to find. Another explanation for the phenomenon is that defects are more often found in the complex parts of the code where the defect-density tend to be higher.

We take a brief look at what has previous been done in terms of testing WMPI II; what methods have been applied, how much time has been spent, and the number of defects found.

- **System testing:** Some system testing has been performed by developing standalone applications, each testing a single or few functionalities within a component (the completeness tests).
- **Stress/Volume:** Tests combining stress and volume testing for a few selected areas have been written and run.
- **MPI-1:** Testing has been performed (partly by clients) since the first release of WMPI 1. As changes in the core have been implemented, in order to allow a local, dynamic world view, these changes have been gradually integrated in recent WMPI 1.x versions. Thus, most MPI-1 functionality of WMPI II is considered of sufficient quality for the release, since it has proved stable for industrial use.

From the list shown above it is obvious that no integration testing of MPI-2 functionality has yet been performed, and only selected areas have undergone stress and volume tests. Moreover, the completeness tests used for system testing have proved to be insufficient do to the limited amout of VID coverage they provide. Another major issue is thread-safety; WMPI II has been designed to be thread-safe, but the MPI-2 functionality has not been tested thoroughly to verify that the implementation is in fact thread-safe.

Table 4-1 on the facing page contains the total number of defects found in the components of MPI-2 and the amount of time since February 2002 devoted to developing tests, testing, debugging, and correcting the defects found. The table lists the number of defects found *after* some initial unit testing had been perform and the component was considered of a sufficient quality for completeness and stress testing.

	Defects	Time spent
MPI-1	8	-
Devices	8	-
One-sided communication	13	12 weeks
Process creation and management	9	4 weeks
Parallel I/O	5	8 weeks
Other	16	-

Table 4-1: The table lists the total number of defects found in each component, the amount of time spent on developing, testing, debugging, and correcting defects.

The devices have not been tested in isolation, hence the defects found in their implementations where found while testing other areas. Moreover, some of defects listed under MPI-1 are genuine defects related to MPI-1, while others are related to MPI-1 functions operating with MPI-2 specific entities such as new user-defined datatypes. The category "**Other**" in Table 4-1 denotes defects found in other areas MPI-2 functionality, such as functions using inter-communicators (extended collective operations), and house-keeping in the WML, such as memory-leaks and threads not being correctly terminated during `MPI_Finalize`.

The following factors should be considered when interpreting the numbers shown in 4-1: The defect tracking discipline might have changed gradually since February 2002, and since times shown include debugging and defect correction, they may not truly reflect the relative amounts of time spent on *testing* each module. Moreover, the quality of the tests written may vary in terms of their defect finding ability, although the majority are completeness tests based on the same simple structure[1].

We are reluctant to draw any definite conclusion from the numbers listed in Table 4-1, due to the limited amount and the uncertainty of the data. If any conclusion should be drawn, based on the assumption that the amount of defects already found in a component is proportional to the number of remaining defects, it would be that especially process creation and management and one-sided communication should be prioritized in the test plan. One definite conclusion that we *can* draw, is that a more rigorous approach to defect tracking procedure is necessary to improve future test planning.

4.2 The Test Plan

In this section we present the assumptions made for the testing process, followed be a presentation and a discussion of the test plan for WMPI II.

[1] One way to compensate for differences the defect finding ability of test cases is to assign each test case a score, although no objective way of doing so exists, [KPM01].

We make the following assumptions:

- Initial testing of WMPI II and the tests themselves will be performed at Critical Software using a cluster of five machines: 1) Intel Pentium III 450 MHz, 192 MB ram, 2) Intel Pentium III 733 MHz 256 MB ram, 3) Dual Pentium III Xeon 550 MHz 512 MB ram, 4) Dual Intel Pentium III 800 MHz 512 MB ram, and 5) Pentium IV 1.7 GHz 256 MB ram. All machines are inter-connected through Fast Ethernet.

- After the initial testing has been performed and the defects found have been corrected, acceptance testing will be performed on the Universidade de Coimbra/DEI cluster, which consists of 22 nodes: 21 Celeron 900 MHz 256 MB ram nodes and one Quad Pentium III Xeon 2.0 GHz, 2 GB RAM, all inter-connected through Fast Ethernet.

- The machines in both clusters are assumed be installed with Windows NT, 2000, or XP, member of a domain, and correctly configured to run WMPI II.

- All tests will be written in the TFW. A small number of tests for central functions of all components have already been implemented[2] and run using completeness, volume, and stress test settings.

- When non-deterministic defects are discovered they will be corrected before any further testing is performed. If a deterministic defect is found testing can proceed in parallel with the debugging process.

The overall test plan for testing WMPI II before the release is the following:

1. **Integration tests**: Stress and volume tests of selected functions of all modules of MPI-1 and MPI-2.

2. **Process creation and management:** Completeness, volume and, stress tests.

3. **Thread-safety level 1:** Tests running in multiple threads simultaneously using different resources.

4. **One-sided communication:** Completeness, volume, and stress tests.

5. **Parallel I/O:** Completeness, volume, and tests.

6. **Thread-safety level 2:** Tests running in different threads using shared resources.

7. **Other:** Extended collective operations, error-handlers, external interfaces, MPI-1 functionality.

[2] To prove that the concept behind the TFW was applicable in WMPI II, a small number of tests was implemented for central functionality of all components of WMPI II during the development of the TFW. The first component for which we developed tests was process creation and management, since that component is considered the most important in WMPI II. Furthermore, it is the most challenging functionality for which to develop generic tests. Thus, if the framework could be used to test process creation and management, it would most like be possible to implement tests for the other components as well.

1. Integration Tests

The list shown above is a prioritized list. It is unlikely that time allows for the implementation and execution of all the tests necessary to complete the list. This is partly why integration testing has received top priority. As mentioned above, no integration testing has yet been performed, and it is the type of testing that most closely simulates how clients are expected to use WMPI II. Moreover, a sufficient number of tests have already been implemented in the TFW to perform some integration testing. Thus, by starting out by integration testing of WMPI II, we do a type of testing that has not yet been performed at little cost. This will provide some indication of the general stability of WMPI II, and the test results of can therefore be used to refine the test plan.

2. Process Creation and Management

The second item on the test plan is "Process Creation and Management". Compared to the other major areas of MPI-2, the process creation and management component includes relatively few functions (12). Given the size and the fact that some tests have already been developed, testing process creation and management should not be very time consuming. When the component has shown to be stable in system testing the idea is to expand the integration testing of item 1 to include the additional test implemented for process creation and management.

3. Thread-safety level 1

Testing the thread-safety of WMPI II has been divided into two phases. The first phase (item 3) focuses on the thread-safety in applications using different resources in each thread. This means that each thread uses a unique set of communicators, windows, and files. No special tests need to be implemented in order to test for this level of thread-safety. The amount of effort needed is simply changing the way that the test manager schedules the tests included in the integration testing.

4. One-sided communication and 5: Parallel I/O

One-sided communication has been placed over parallel I/O based on knowledge of the implementation. One-sided communication allows for three different kinds of synchronization. In all types of synchronization an epoch for a window is opened, a number of transfers are done (get, put, and/or accumulate), and the epoch is closed. The standard allows for an MPI-2 implementation to block until each transfer has been completed, but only requires that the data in windows and get-buffers is consistent after the epoch closure. The latter opens for a number of optimizations, some of which have been implemented in WMPI II at the cost of some complexity. Thus, compared to parallel I/O, which relies on a central file server, the implementation of one-side communication is somewhat more complex and therefore is expected to contain more defects.

As with the tests for process creation and management, the tests implemented for parallel I/O and one-sided communication are added to the configuration files of the integration test suite and the thread-safety test suite, and both are rerun.

6. Thread-safety level 2

For the second round of thread-safety tests the focus is on testing WMPI II in situations were resources are shared between multiple threads, e.g. multiple functions writing to the same file, using the same window (within the same epoch), or the same communicator, concurrently. The idea is to focus on race coverage. For this level of thread-safety a new set of tests and oracles are necessary, since some synchronization within these objects is necessary. Only a small set of functions will be tested using shared resources due to the complexity of implementing such tests. Furthermore, only few clients are expected to require this level of thread-safety.

7. Other

The last item, 7, consists of all other functions. We do not have the ambition of implementing tests for all 287 MPI-2 functions before the release. After the release, however, tests for all the functions will assist reproduction and debugging of client discovered defects, and the tests will be necessary in order to test the planned version of WMPI II for Linux and other Unix flavors as well as interoperability between these implementations and the Windows version.

4.2.1 Conclusion

The test plan presented has been designed such that when a test has been implemented the function(s) tested are subjected tests in the following order: Completeness, volume, stress, integration, thread-safety level 1, and thread-safety level 2. This design has been chosen since we expect to find defects that need to be corrected. By going from a simple method of testing to progressively more complex, defects are discovered in an order which eases the debugging. Except for the second level of thread-safety, the TFW allows us to do stepwise integration of tests of functions and components in progressively more complex settings at little cost.

A similar approach is to be applied for the hardware used to run tests: First tests are run on a single CPU machine (during test development), then a dual, then the Critical Software cluster, and finally on the Universidade de Coimbra cluster. When the haystack is small the needle is easier to find.

Conclusion

Chapter 5

5.1 Conclusion

Previous to the introduction of the test framework in the WMPI II project each test case was written as a standalone executable. Given the size, 287 functions, and the complexity of the MPI-2 API this task proved cumbersome and the approach insufficient. Moreover, tests were narrow terms of the their valid input domain coverage and path coverage, since parameter values and communication patterns were hardwired into the tests. We have addressed these issues by developing a test framework based on parameter abstraction. The framework itself has been designed to be independent from WMPI II and can be used in other projects, not necessarily related to distributed systems.

The main idea behind the test framework is that tests should be written to use the abstract parameters in order to make them generic and thus reusable. The abstract parameters can be objects encapsulating entities specific to the implementation under test as well as more general types such as booleans, integers, floats, etc. The philosophy is that each test implemented should only test a single or a few functions of the specification to allow for fine granularity and flexibility. When the tests are kept simple, they can serve as building blocks in large test cases, thereby reducing the amount of code necessary to perform the different types of testing. Moreover, the test framework is designed such that part of the oracle function is performed by the abstract parameters to allow tests to be generic and simple to develop. Since all tests have a common interface and since they are part of the same application, some effort has justifiably been spent on result reporting and debugging functionalities, since these features can be used globally.

With the introduction of the test framework in the WMPI II project, we have attained the capability to use the same test code for valid input domain coverage (completeness), stress, volume, and thread-safety test cases for both system and integration testing. This means we are now able to take an incremental approach to testing at no cost implementation-wise. We have designed the test plan so that MPI-2 functions and components are tested in progressively complex test cases. Moreover, since all tests are part of a single application regression testing, which especially becomes important during the maintenance phase, is likely to be applied with less effort and of higher quality compared to the former testing approach.

The main focus of the test plan presented in Chapter 4 is integration testing; hence test cases where multiple components of WMPI II are tested in conjunction. Alternatively, integration testing could have been performed through a relatively small set of applications where multiple MPI-2 functions are tested in each application. Depending on the complexity of these applications, they might have taken less time to develop, while still providing "reasonable" state and input coverage. However, such an approach would have made debugging harder, due to the complexity of such applications. Furthermore, two or more versions of such applications would be needed to test thread-safety of WMPI II, unless the initial versions were to be multi-threaded, making the debugging process even more complex. An ad-hoc approach would most likely not be very flexible and it would be hard to track what exactly has tested under which conditions.

5.2 Future Work

Besides from the task of implementing and converting the tests required to complete the test plan, the following areas are planned for additions to the test framework, going from near to distant future:

- **Interactive Test Design:** As mentioned, designing test cases can be tedious. Comments and sometimes even code has to be consulted when test cases are designed. Information about which resource pools tests use and which tags they recognize is already explicitly represented in the framework. Hence, it should be relatively straightforward to build an extension to the test framework, which allows configuration files to be built interactively. One way in which this approach can be assisted is by extending the methods for registering resource pools and recognized tags to take a (non-empty) help string as a parameter. This would keep documentation and code close, hence reducing the need for external documentation, which sometimes tend to be both incomplete and out of date.
- **Platform Independence:** There are two major reasons for making the framework platform independent, namely that the porting of WMPI II to Linux and other Unix flavors will start in 2003, and that the framework should be usable in other projects at Critical Software of which the majority are implemented for other operating systems than Microsoft Windows. Platform independence refers to encapsulating OS dependent code in an abstraction layer that has to be implemented for each OS on which the framework should run. Currently, the only platform dependent code present in the test framework is for managing threads and synchronization, thus the effort necessary to isolate this code is expected to be quite limited.
- **Profiling:** The test framework can be used for profiling different algorithms and implementation approaches. Especially the latter will be necessary when porting WMPI II to Linux. The devices are some of the components to be ported first, where parameters such as the message size for changing from a buffering scheme to a rendezvous protocol can be optimized. Such parameters as well as synchronization schemes can be profiled using the framework. This can be done by implementing a special result manager, which keeps track of the time spent in each test.
- **Graphical User Interface:** A graphical front-end and a couple of additional features would enable the support team to send the test framework to clients. Through the interface, clients should be able to retrieve configuration files through HTTP or FTP and to run test suites. In this way, the framework can collect site-specific test data in order to assist debugging in situations where the symptoms of a defect are irreproducible

at Critical Software. The front-end should also handle sending the results to Critical Software, e.g. through FTP or HTTP. When test cases can be retrieved via the Internet, a configuration file can be easily modified and deployed. This opens for improved support, since developers can modify configuration files and review results locally, while the effort necessary on behalf of the clients is reduced to a few mouse clicks.

- **Robustness:** In the Ballista Project is aimed at testing the robustness and "hardening" commercial of-the-shelf software components, [Koo97]. Robustness is measured by calling functions with different parameter values in the component and determining whether each function correctly identifies both valid and exceptional input. Hardening is achieved through software generated code layer checking for and filtering any exceptional input, not correctly recognized by a component. WMPI II can be tested for robustness in a similar way and consequently hardened. Robustness can be tested by adding a new set of oracles and resource pools of invalid inputs. Although robustness is not a requirement by MPI-2 standard, it would ease the development of WMPI II-based applications due to correct and meaningful error codes, as well as graceful abortion of all processes in case of erroneous input.

Another area, which would be interesting from an academic point of view but less likely to be pursued in context of the WMPI II project, is automatic optimization of paths leading to an inconsistent state. During development and debugging a significant percentage of defects only show their presence sometimes (non-deterministic defects). The test framework could be extended with a component that attempts to optimize the path - a sequence of test cases - leading to a hang, a crash, or an incorrect operation. This could be done through simple manipulation of configuration files based on heuristics. Paths could be optimized based on measures such as execution time, length, and success rate. However, it is (unfortunately) not likely that automatic path optimization will be implemented due to the effort necessary - not to mention the risk of it not being useful in practice, since the optimization might prove too time-consuming and the results insignificant for debugging.

Bibliography

[Boo01] Rumbaugh Booch, Jacobson. *A Practical Guide to Testing Object-Orientated Software*. Assison-Wesley, 2001.

[Car02] David N. Card. *Managing Software Quality with Defects*. IEEE Computer Society, 2002.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nizberg, William Saphir, and Marc Snir. *MPI - The complete Reference, Volume 2, The MPI Extensions*. The MIT Press, 2 edition, 1998.

[Hum97] Watts S. Humphrey. *Managing Technical People*. Addison Wesley, 1997.

[Hut01] Marnie Hutcheson. *Software Testing - methods and metrics*. MIT Press, 2001.

[KM97] Saileshwar Krishnamurthy and Aditya P. Martur. On the estimation of reliability of a software system using the reliability of its components, 1997.

[Koo97] Philip Koopman. Ballista design and methodology, 1997.

[KPM01] S. H. Kan, J. Parrish, and D. Manlove. In-process metrics for software testing, 2001.

[LILS+97] Ewing Lusk, Steve Iluss-Lederman, Bill Saphir, Marc Snir, William Gropp, Anthony Skjellum, and Bill Nitzberg. Mpi-2: Extensions to the message-passing interface, 1997.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. Wiley Interscience, 1979.

[PCA+91] M. C. Paulk, B. Curtis, E. Averill, J. Bamberger, T. Kasse, M. Konrad, J. Perdue, C. Weber, and J. Withey. Capability maturity model for software. Technical Report CMU/SEI-91-TR-24 ADA240603, 1991.

[Pfi98] Gregory F. Pfister. *In Search of Clusters*. Prentice Hall, 2 edition, 1998.

[Pro02] The Apache XML Project. Xerces-c++ 2.1.0, 2002.

[SFL+02] Eric Sommerlade, Michael Feathers, Jerome Lacoste, J.E. Hoffmann, Baptiste Lepilleur, Bastiaan Bakker, and Steve Robbins. Cppunit, 2002.

[SH02] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters, 2002.

[SOO+98] Marc Snir, Steve Otto, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The complete Reference, Volume 1, The MPI Core*. The MIT Press, 2 edition, 1998.

[Sto93] Philip Alan Stocks. Applying formal methods to software testing, 1993.