MuTomVo: Mutation Testing framework for omnet-based simulated enVironments

PABLO CERRO CAÑIZARES

MÁSTER EN INGENIERÍA EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

Madrid, 9 de Febrero de 2015

Calificación obtenida: 10

Director:
Alberto Núñez Covarrubias

Autorización de difusión

Autor
Pablo Cerro Cañizares

Fecha
Madrid, 9 de Febrero de 2015.

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "MuTomVo: Mutation Testing framework for omnet-based simulated enVironments", realizado durante el curso académico 2014-2015 bajo la dirección de Alberto Núñez Covarrubias en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Actualmente, el *testing* es una de las técnicas más extendidas para comprobar la validez de sistemas complejos. Su aplicación en software es una parte fundamental del desarrollo de sistemas, sin embargo, existen varias dificultades a la hora de aplicar estas técnicas, tales como el alto coste económico y computacional. Uno de los aspectos más relevantes en *testing* es la selección de un conjunto de tests adecuado para aplicarlo sobre el sistema que se desea probar. Esto resulta especialmente complicado cuando el sistema sometido a test es de grandes dimensiones, como es el caso de *clusters HPC* (High Performance Computing) o sistemas *cloud*.

El objetivo principal de este trabajo es proporcionar mecanismos que permitan evaluar la idoneidad de los conjuntos de tests, utilizados para chequear sistemas distribuidos, de forma escalable, económica y eficiente. Para ello se propone MuTomVo, un *framework* de mutación de código que integra técnicas de *mutation testing* con técnicas de simulación. Para realizar el modelado y la simulación de sistemas distribuidos se ha utilizado la plataforma SIMCAN.

MuTomVo se ha construido utilizando una arquitectura modular, a través de la cual, se pueden introducir nuevas técnicas de *mutation testing* de manera sencilla. Esto permite realizar una comparación entre dichas técnicas para evaluar la adecuación de cada una de ellas sobre el entorno proporcionado. De esta forma, se pretende reunir en un único *framework* las funcionalidades de diferentes herramientas, tales como simuladores, *frameworks* de mutación y herramientas de generación de tests.

Además, se ha realizado una fase de experimentación para analizar la idoneidad de distintos conjuntos de tests ejecutados en diferentes aplicaciones distribuidas. Cada una de estas aplicaciones, junto con las arquitecturas donde se han ejecutado, han sido modeladas con SIMCAN. Asimismo, los experimentos se han llevado a cabo aplicando técnicas de *mutation testing* sobre estos modelos.

Palabras clave

Modelado, simulación, sistemas distribuidos, herramientas de testing y mutation testing.

Abstract

Currently, testing is the most widely used technique to check the validity of complex systems. Its application in software is a critical part in the development of systems. However, there are several difficulties for applying these techniques, such as high economic and computational cost. One of the main difficulties when applying testing techniques is to obtain an appropriate test suite. This is especially difficult when the size of the system under test is large, like HPC clusters or cloud systems.

The main goal of this project is to provide mechanisms that allow to evaluate the suitability of test suites to check distributed systems, in an inexpensive and efficient way. In this work, we propose MuTomVo, a framework that integrates mutation testing techniques with simulation techniques. For modelling and simulating distributed systems, the SIMCAN simulation platform has been used.

MuTomVo is built on a modular and flexible architecture, where new mutation testing techniques can be easily included. Thus, it is intended to unify, in one framework, functionalities of different tools, such as simulators, mutation frameworks and tools for generating tests.

Moreover, this work presents some experiments for analysing the suitability of different test suites over different distributed applications. Each one of these applications has been modelled with SIMCAN, and therefore, these experiments have been carried out by applying mutation testing techniques on these models.

Keywords

Modelling, simulation, distributed systems, testing tools and mutation testing.

Índice general

Agradecimientos

Quiero agradecer la excelente labor realizada por el director de este trabajo, Alberto Núñez. Tanto sus tareas orientativas como de motivación, han sido cruciales en la consecución de este proyecto. De igual forma, me gustaría dar las gracias a $\mathcal{M}$. G. Merayo por el apoyo y ejercer de oráculo de testing. Y por supuesto, a Sonia Pérez por el soporte gráfico (y su paciencia infinita).

Dedicatoria

A mis padres, por darme todo lo que tengo e inculcarme todo lo que soy.

'Algunas personas quieren que algo ocurra, otras sueñan con que pasará, otras hacen que suceda.'

Michael J. Jordan

Capítulo 1

Introducción

En este capítulo se proporciona una visión global del marco en el cual se ha desarrollado el proyecto. Además de los retos a los que se ha hecho frente durante el desarrollo de este proyecto para alcanzar las contribuciones descritas, se exponen cada uno de los objetivos perseguidos en el mismo.

1.1. Definición y alcance del proyecto

En la sociedad actual, existe un alto porcentaje de empresas tecnológicas que invierten gran parte de su capital en el proceso de evaluación de proyectos software. Parte de su imagen corporativa depende de la calidad de sus productos. El funcionamiento erróneo de alguno de los componentes de un sistema puede causar pérdidas millonarias. Ejemplo de ello es el hecho que ocurrió en 1999 con el satélite MCO (Mars Climate Orbiter), donde un error en el sistema métrico de los ordenadores de tierra provocó un fallo de cálculo que costó la destrucción del satélite valorado en 327 millones de dólares[60]. El uso de técnicas de *testing* reducen el porcentaje de catástrofes ocurridas en sistemas donde el software es un factor crítico.

En las últimas décadas y teniendo como detonante el auge de Internet de alta velocidad, se ha producido un cambio en el modelo de negocio en la sociedad que conlleva un crecimiento de las infraestructuras y servicios informáticos, de forma que se ha intensificado, como consecuencia, el aumento tanto en la generación de datos como en su procesamiento[82]. Esta

evolución requiere el desarrollo y despliegue de nuevas aplicaciones que puedan proporcionar un alto rendimiento para procesar grandes volúmenes de datos, donde intervienen factores determinantes como la escalabilidad, disponibilidad y fiabilidad de los sistemas. Sin embargo, procesar grandes volúmenes de datos incrementa la complejidad de las aplicaciones que los procesan, lo cual también complica el proceso de *testing* de estas aplicaciones.

Actualmente, el *testing* es la técnica más utilizada para chequear la validez de sistemas complejos. Las técnicas de *testing*, usualmente, requieren la generación y aplicación de un conjunto de tests que permita determinar si el comportamiento observado en el sistema se corresponde con el esperado. Una de las dificultades para aplicar técnicas de *testing* es poder contar con un conjunto de tests adecuado. Esto resulta especialmente complicado en el caso de clusters HPC o sistemas *cloud*. Afortunadamente, existen técnicas para poder afrontar estos retos. Una de ellas es conocida como *mutation testing*. Una de las características más relevantes de *mutation testing* es la simplicidad de su funcionamiento, el cual consiste en reproducir los errores cometidos por programadores, que se asume, son competentes, introduciendo cambios sintácticos a lo largo del código fuente original. Los tests previamente proporcionados son evaluados a través de la ejecución del nuevo código modificado, para comprobar su eficacia a la hora de detectar errores. Sin embargo, aplicar técnicas de *mutation testing* en entornos distribuidos puede resultar muy costoso, tanto a nivel económico como en tiempo de ejecución. Además, no siempre se dispone de acceso al sistema físico donde se deben realizar los tests. En aquellos casos en los que ejecutar los tests presenta estos retos, la comunidad científica se ha decantado, cada vez en mayor medida, por el uso de herramientas de simulación.

En este proyecto se van a integrar herramientas de modelado y simulación de sistemas altamente distribuidos, con el objetivo de determinar la idoneidad de los conjuntos de tests para este tipo de sistemas de forma escalable, eficiente y accesible. La aplicación de técnicas de *mutation testing* en entornos simulados, permite desplegar y configurar infraestructuras de una forma más rápida, sencilla y económica que si se hiciera en un entorno físico.

1.2. Motivación

Actualmente, las arquitecturas altamente distribuidas, tales como clusters de alto rendimiento[63] o los sistemas de *cloud computing*[35], son las soluciones más efectivas para los usuarios finales: empresas y científicos. Con el objetivo de utilizar eficientemente los recursos, los sistemas altamente distribuidos necesitan métodos escalables y flexibles para desplegar aplicaciones científicas de alto rendimiento. En muchos de estos sistemas, el esquema más utilizado es *Map-Reduce*, un modelo de programación orientado al procesado de cantidades masivas de datos en sistemas distribuidos.

En un sistema distribuido, uno de los problemas más comunes consiste en garantizar que el comportamiento del mismo se corresponde con la especificación utilizada para implementarlo. Generalmente, cuando se despliega una aplicación basada en el modelo *Map-Reduce*, un alto número de procesos se distribuyen en los nodos del sistema, ejecutándose de forma concurrente, accediendo a los recursos compartidos e intercambiando mensajes por la red de comunicaciones. En estos casos, resulta complejo controlar las acciones de cada uno de estos procesos y, por tanto, garantizar que la aplicación cumple con su especificación. Por este motivo, la comunidad investigadora ha utilizado durante las dos últimas décadas técnicas de *testing* para validar el correcto funcionamiento de sistemas distribuidos. En este trabajo se van a utilizar técnicas de *testing* para poder cumplir los objetivos propuestos. En particular, se van a integrar técnicas de *mutation testing* para inyectar errores en el código de las aplicaciones distribuidas, con el objetivo de evaluar si un conjunto de tests es adecuado para validar dicha aplicación. Sin embargo, la aplicación de estas técnicas en entornos altamente distribuidos requiere enfrentarse a ciertos retos:

- Accesibilidad del sistema sobre el que realizar el test. Debido a las características de las aplicaciones ejecutadas sobre este tipo de sistemas, es necesaria una gran capacidad de cómputo para abordar su resolución, ya que éstas requieren un elevado tiempo de ejecución. Por ello, se necesitan plataformas provistas de un número elevado de nodos de cómputo donde son ejecutados los tests. El acceso a la experimentación sobre este

tipo de plataformas puede resultar complicado debido a su elevado coste económico.

- Modificación y configuración de las infraestructuras. Debido a la gran cantidad y variedad de componentes que forman este tipo de sistemas, como por ejemplo el *cluster* Marenostrum de Barcelona[15], formado por 48.896 núcleos de cómputo, 103 *terabytes* de memoria RAM y 3024 discos duros que alcanzan los 1,9 *petabytes* de almacenamiento, realizar modificaciones sobre su arquitectura puede resultar muy costoso. Además, existen ocasiones donde al usuario final le es imposible realizar estas modificaciones al no tener acceso a la infraestructura.

- Complejidad para reproducir tests. En entornos distribuidos, resulta muy complicado reproducir los resultados de los tests ejecutados. Esto se debe a la variabilidad de factores como la alta concurrencia, latencia de red y especialmente en sistemas *cloud*, la virtualización[85]. En este tipo de sistemas, para maximizar el uso de recursos, se tiende a virtualizar la plataforma. Esto permite la ejecución de tareas por parte de varios usuarios simultáneamente, lo que puede afectar al rendimiento de las aplicaciones.

Estos retos impiden en muchas ocasiones realizar tests sobre los sistemas que se desea probar debido, principalmente, a la falta de accesibilidad a los recursos requeridos. En consecuencia, el uso de herramientas de simulación se convierte en una opción plausible para poder superar las dificultades anteriormente descritas. Actualmente existen numerosas herramientas para simular, desde componentes individuales como discos[7] o CPUs[95], hasta centros de datos completos[13,79], que permiten modelar y representar el comportamiento tanto de la arquitectura como de la ejecución de aplicaciones en entornos altamente distribuidos.

No obstante, el uso de herramientas de simulación presenta varios inconvenientes. Una de las principales desventajas de la simulación consiste en la depuración de los modelos. Cuando se diseña un nuevo modelo, ya sea de una arquitectura completa o una aplicación, el usuario debe realizar una serie de pruebas para asegurarse de que dicho modelo tiene el comportamiento esperado, es decir, que cumple con las expectativas para las que fue

diseñado. Sin embargo, habitualmente, no se suelen aplicar técnicas de *testing*, sino que a través de un conjunto reducido de pruebas, generalmente configurado de forma manual por el usuario, se verifica el modelo. De esta forma, queda sin explorar un dominio extenso de casos potenciales donde el modelo no ha sido evaluado. Esta insuficiencia de casos de prueba eleva la posibilidad de fallo.

En este trabajo se propone la utilización de técnicas de *mutation testing* en entornos de simulación. Con ello se pretende aprovechar las ventajas proporcionadas por los simuladores para mitigar, en la medida de lo posible, las dificultades de aplicar técnicas de *testing* en sistemas distribuidos. Para cumplir este objetivo, se ha desarrollado el *framework* MuTomVo, diseñado para aplicarse en herramientas de simulación basadas en OMNeT++ [93].

1.3. Objetivos

El objetivo principal de este trabajo es **evaluar la idoneidad de conjuntos de tests para chequear sistemas altamente distribuidos de forma escalable, económica y eficiente**. Para poder alcanzar este objetivo, se propone la integración de técnicas de simulación, para modelar y simular sistemas distribuidos, con técnicas de *mutation testing*, utilizadas para la inducción de fallos y comprobación de la calidad de los tests utilizados. Todo ello se ha integrado en un *framework*, llamado MuTomVo, que utiliza la plataforma de simulación SIMCAN [67].

Asimismo, será necesaria la consecución de los siguientes objetivos secundarios:

1. **Diseño de un *framework* flexible y extensible**. MuTomVo se ha construido utilizando una arquitectura modular, a través de la cual, se pueden introducir nuevas técnicas de manera sencilla. Esto permite realizar comparaciones entre dichas técnicas para evaluar la adecuación de cada una de ellas sobre el entorno proporcionado. Es decir, se parte de un modelo inicial, donde se implementan las técnicas necesarias para alcanzar los objetivos de este proyecto, con la particularidad de contar con una arquitectura abierta y flexible que permite su extensión en el futuro. De esta forma, se

pretende reunir en un único *framework* las funcionalidades de diferentes herramientas, tales como simuladores, *frameworks* de mutación y herramientas para generar tests.

2. **Aplicación de técnicas de *mutation testing***. Con el fin de determinar la capacidad de detección de errores de un conjunto de tests previamente definido, es necesaria la generación de *mutantes* del modelo a través de cambios sintácticos en el código fuente original. Una vez generados los mutantes, se mide la eficacia del conjunto de tests para identificar los errores introducidos.

3. **Chequeo automático de los tests ejecutados sobre el modelo**. Una de las características de MuTomVo es la capacidad de determinar si el resultado de la ejecución de un test, aplicado a un modelo, es correcto. La flexibilidad que ofrece el sistema permite la inclusión de técnicas para cumplir con este cometido de manera sencilla.

4. **Optimización en la generación de mutantes para entornos de simulación**. Debido a que MuTomVo está orientado a plataformas que utilicen OMNeT++, se van a utilizar técnicas de generación de mutantes optimizadas para aprovechar al máximo las características de este tipo de sistemas. Por ello, además de implementar las técnicas convencionales de generación de mutantes, van a ser diseñadas técnicas ad-hoc enfocadas en mutaciones para el motor de simulación OMNeT++. De esta forma se reducirán los tiempos de ejecución, tanto en generación de mutantes, ya que se reducirá el número de los mismos, como en su compilación y ejecución.

1.4. Estructura del documento

El resto de este documento está organizado en los siguientes capítulos:

- Capítulo 2, Estado del arte, presenta los trabajos más relevantes en el campo del modelado y simulación de sistemas distribuidos, así como de *mutation testing*. Además, se realiza una comparación con los trabajos existentes que utilizan ambas técnicas.

- Capítulo 3, Descripción de OMNeT++ y SIMCAN, describe las características más relevantes del *framework* de simulación OMNeT++ y de la plataforma de modelado y simulación de sistemas distribuidos SIMCAN.

- Capítulo 4, MuTomVo, describe en detalle cómo se ha realizado la integración del modelado y simulación de sistemas distribuidos con técnicas de *mutation testing*. Inicialmente se presenta la estructura general de MuTomVo. Seguidamente, se describen los operadores de mutación implementados en dicho *framework*, así como el ciclo del proceso de *testing* llevado a cabo para estudiar la idoneidad de conjuntos de tests. Finalmente, se presenta la estructura utilizada para su implementación.

- Capítulo 5, Experimentos, presenta los experimentos realizados utilizando el *framework* MuTomVo. Estos experimentos consisten en evaluar la idoneidad de varios conjuntos de tests, generados automáticamente, para chequear varias aplicaciones ejecutadas en entornos distribuidos.

- Capítulo 6, Conclusiones y trabajo futuro.

Chapter 1

Introduction

The main objective of this chapter is to provide an overview of the MuTomVo framework. In addition to the challenges faced during the development of this project, the objectives and the achieved results are also described.

1.1. Definition and scope

In today's society, there is a high percentage of technology corporations that invest a relevant percentage of their capital in testing software projects. An error during the execution of a system could result in considerable economic loss. As an example, in 1999 an error in the Earth's computer's metric system of the Mars Climate Orbiter caused the destruction of the satellite valued in $327 million[60]. The use of testing techniques can reduce the percentage of catastrophes occurring in these systems.

Over the last decades, there has been a change in business models as a consequence of the high-speed Internet era. This new model carries a growth of the infrastructure and services, increasing both data generation and processing[82]. This revolution requires the development and deployment of new applications that provide high performance to process large volumes of data. However, processing large volumes of data increases the complexity of the applications that process it, which also complicates the process of testing these applications.

Currently, testing is the most widely used technique to check the validity of complex systems. Testing techniques usually require the generation and application of a test suite

that can determine wether the observed behaviour in the system corresponds to the expected one. One of the main difficulties in applying testing techniques is to have an appropriate test suite.

This is especially difficult when the size of the system under test is large, such as distributed clusters or cloud systems. Fortunately, there exist several techniques to deal with these challenges. One of them is known as *mutation testing*. One of the most relevant features of mutation testing is the simplicity of its operation. Basically, mutation testing tries to replicate the mistakes often made by competent programmers, introducing syntactic changes in the original source. The sets of tests are evaluated, through the execution of the modified code, to determine its effectiveness in finding errors. However, the application of *mutation testing* techniques in distributed environments can be very expensive. Moreover, in many cases, there is not access to the physical system in which the tests must be executed. In these cases, the scientific community has opted for the use of simulation tools.

This project integrates modelling tools and simulation of highly distributed systems with the aim of evaluating the effectiveness of test suites for this type of systems in a scalable, efficient and accessible way. The application of mutation testing techniques on applications running in a simulated environment, will allow to deploy and configure different infrastructures faster simpler and cheaper than performing the same process in a physical environment.

1.2. Motivation

Currently, highly distributed architectures, such as high-performance clusters[63] or *cloud*[35] systems, are the most effective solutions for end users: companies and scientists. In order to use efficiently resources, distributed systems need scalable and flexible methods to deploy high-performance applications. In many of these systems, the most widely used scheme is *Map-Reduce*, a programming model for processing large amounts of data in distributed systems. In these systems, one of the most common problems is to ensure that the system

behaviour corresponds to the expected one. Usually, when an application based on the Map-Reduce model is deployed, a large number of processes are distributed among the nodes of the system, running concurrently, accessing to shared resources and exchanging messages through the communications network. In these cases, it is difficult and complex to control the actions of each process, and therefore, ensuring that the application achieve its specification is not possible

Over the past two decades, the research community has used testing techniques to validate the correct operation of distributed systems. In this project we use testing techniques to reach the proposed objectives. In particular, mutation testing techniques are integrated to inject errors in distributed applications, in order to evaluate if a test suite is suitable for checking the system. However, the application of these techniques in distributed environments requires to deal with several challenges:

- Accessibility of the system under test. Due to the characteristics of the applications running on these systems, a high computing power is needed to execute them by requiring a high runtime. As a consequence, many platforms that include a large number of compute nodes are needed to execute the test suites. The experimentation on these platforms can be difficult due to its high cost.

- Customize the infrastructure. Due to the large number and variety of components that conform these systems, like, the cluster Marenostrum Barcelona[15] (48896 computing cores, 103 terabytes of RAM and 3024 hard drives that reach 1,9 petabytes of storage), the modification of its architecture can be very expensive. In addition, in some cases the end user has no access to the infrastructure, which hampers the customization of the environment.

- Complexity to reproduce tests. In distributed environments, it is very difficult to reproduce the results of the executed tests. This is due to the variability of factors such as high concurrency, network latency and, especially in cloud systems, virtualisation[85].

These challenges often hamper executing tests on the system, mainly due to the lack of accessibility to the required resources. Consequently, the use of simulation tools become a plausible option to overcome the previously described difficulties. Currently, there exist several tools for simulating, from individual components such as disks[7] or CPUs[95] to complete cloud systems[13,79], which allow to model and represent the behaviour of both architecture and running applications in highly distributed environments.

However, the use of simulation tools has several disadvantages. One of them is the process of debugging simulation models. Once a new model is designed, the user must execute a test suite to ensure that the model fulfils the expected behaviour. Unfortunately, in most cases testing techniques are not applied. Instead, the model is checked using a small test suites, generally manually configured by the user. Thus, a vast unexplored potential cases are not evaluated. This lack of test cases raises the possibility of failure.

In this project the use of *mutation testing* techniques in simulation environments is proposed. It is intended to get the benefits provided by the simulators to mitigate, as far as possible, the difficulties of applying testing techniques in distributed systems. In order to reach this objective, we have developed a framework called MuTomVo, designed to be used in simulation tools based on OMNeT++[93].

1.3. Goals

The main goal of this project is to **evaluate the suitability of test suites to check highly scalable distributed systems, in an inexpensive and efficient way**. In order to reach this goal, the integration of simulation techniques and mutation testing techniques is proposed. It will allow to model and simulate distributed systems for checking the quality of test suites to be applied. Thus, a framework called MuTomVo, based on the simulation platform SIMCAN[67], has been developed.

In order to achieve this goal, the achievement of the following secondary objectives are required:

1. **Design a flexible and extensible framework**. MuTomVo has been built by using a modular architecture, through which new techniques can be easily introduced. This allows to compare the efficiency of each technique on the provided environment. That is, we start from an initial model, where the required techniques are implemented to achieve the objectives of this project, with the particularity of having an open and flexible architecture that allows the extension of its functionalities. Thus, it is intended the integration of different tools such as simulators, mutation frameworks and tools, to generate and evaluate test suites in a single framework.

2. **Application of mutation testing techniques**. In order to determine the ability of error detection in previously defined test suites, it is required to mutate the models through syntactic changes in the original source code. In this project, only first order mutants are considered.

3. **Automatic evaluation of the test suites executed on the model**. One feature of the proposed framework is to determine whether the results obtained from the application of a test to a given model is correct. The provided flexibility by the system allows the inclusion of techniques to accomplish this task.

4. **Optimisation of the generation of mutants for simulation environments**. Due to the fact that the proposed framework is being oriented to OMNeT-based platforms, optimised techniques for generating mutants are used to maximise the features of these systems. Therefore, in addition to implementing the conventional techniques for generating mutants, we provide ad-hoc techniques focused on mutations for the OMNeT++ simulation engine. In this way, the time required to generate the mutants, the compilation of these mutants and the execution time will be reduced.

1.4. Document Structure

The remainder of this paper is organised into the chapters described below:

- Chapter 2, State of the Art, presents the most relevant work in the field of modelling and simulation of distributed systems, as well as mutation testing. Furthermore, a comparison between current proposals using both techniques, and the one proposed in this project, is presented.

- Chapter 3, Description of SIMCAN and OMNeT++, describes the major features of OMNeT++ simulation framework and the platform for modelling and simulating distributed systems SIMCAN.

- Chapter 4, MuTomVo, describes in detail how the integration of simulating distributed systems and mutation testing techniques has been accomplished. First, the general structure of MuTomVo framework is described. Then, mutation operators implemented in this framework and the testing process cycle are described.

- Chapter 5, Experiments, presents the experiments performed by using the framework MuTomVo. These experiments consist in generating and evaluating the suitability of a test suite to evaluate several applications running on distributed environments.

- Chapter 6, Conclusions and future work.

Capítulo 2

Estado del arte

En este capítulo se presenta un análisis de las técnicas y estudios realizados en los campos de simulación de sistemas *distribuidos*, simulación de sistemas *cloud* y *mutation testing*. Así mismo, se realiza una revisión de los trabajos actuales que emplean técnicas de mutación en entornos simulados y una comparación entre dichos trabajos.

2.1. Modelado y simulación de sistemas distribuidos

Durante la última década, la simulación se está convirtiendo en una de las opciones más demandadas para realizar experimentos científicos[97]. Entre los motivos que hacen de la simulación una tendencia cada vez más utilizada para modelar todo tipo de sistemas, destacan el ahorro de costes, accesibilidad y flexibilidad. Las posibilidades de la simulación pueden aplicarse a un amplio rango de campos, desde la simulación de sistemas sencillos basados en un único nodo, hasta sistemas distribuidos complejos, tales como los sistemas *cloud*.

En el ámbito de la simulación de redes de comunicaciones se encuentran NS-2[1], DaSSF[43], OMNeT++[93], y OPNET[17]. Estos simuladores de redes están enfocados a reproducir con fidelidad detalles importantes como protocolos de red, latencia o fragmentación. Sin embargo, carecen de métodos para simular sistemas de cómputo, sistemas complejos de E/S o virtualización.

Para cubrir esta necesidad, surgen las herramientas para modelar y simular sistemas

distribuidos. Una de las herramientas más influyentes en este campo es COTSon[5], un *framework* de simulación desarrollado conjuntamente por HP Labs y AMD. El objetivo de COTSon es proporcionar una evaluación rápida y precisa de la computación actual y futura. COTSon está orientado a la simulación de sistemas *cluster* formados por cientos de nodos *multi-core* asociados a dispositivos conectados a través de redes de comunicaciones estándar.

SimFlex[39] es un *framework* basado en componentes, orientado a la creación de modelos de tiempos de procesadores para ejecutar aplicaciones comerciales.

Destacada por el alto grado de definición de sus modelos, SIMCAN[66] es una plataforma de simulación cuyo objetivo es modelar arquitecturas de alto rendimiento (HPC). Esta plataforma permite el diseño y modelado de herramientas y arquitecturas distribuidas utilizando un amplio rango de configuraciones. SIMCAN tiene un diseño modular que facilita la integración de diferentes modelos en una única plataforma.

Además de los simuladores descritos anteriormente, existen herramientas enfocadas al diseño y simulación de centros de almacenamiento de datos. Una de estas herramientas es SimSANs[104], *Simulating Storage Area Networks*, especialmente útil en el diseño de infraestructuras y análisis del rendimiento de centros de almacenamiento con redes *Fiber Channel*.

Tras las herramientas descritas para simular sistemas distribuidos, aparecen los simuladores de sistemas *Grid*. Durante la pasada década, los sistemas *Grid*[12] han ofrecido servicios de computación masiva a la comunidad científica. Para realizar investigaciones en este área, se desarrollaron un conjunto de simuladores, entre los que se encuentran GridSim [80], OptorSim[96], SimGrid[46], MicroGrid[101] y GangSim[18].

Finalmente, como evolución de los sistemas *Grid* aparecen los simuladores de sistemas *cloud*. Algunos de ellos usan como base simuladores de sistemas *Grid*, tales como CloudSim o GridSim, mientras que otros fueron escritos completamente para modelar y simular sistemas *cloud*. Los *frameworks* de simulación que mejor se adaptan a los requisitos de modelado y simulación de sistemas *cloud computing* son CloudSim[79], MDCSim[56], GreenCloud[20],

SimGrid[38] e iCanCloud[68].

En el caso de CloudSim, hay numerosos artículos de investigación que muestran los resultados obtenidos con este simulador[11,47,81]. Esta herramienta, basada inicialmente en simulación de sistemas *Grid*[79] (anteriormente conocido como GridSim[2]), incluye una capa nueva implementada para añadir la posibilidad de simular sistemas *cloud*.

MDCSim es un simulador basado en eventos, similar a CloudSim. MDCSim tiene la capacidad de realizar medidas de consumo de potencia en cada uno de los servidores que forman parte de un centro de datos. En las últimas versiones, MDCSim forma parte de una solución comercial llamada CSIM[19].

GreenCloud es una extensión del simulador de redes NS2[1], enfocado a simular las comunicaciones entre procesos ejecutándose en un sistema *cloud* a nivel de paquete. Al igual que NS2, está escrito en C++ y OTcl, lo que es una desventaja para esta herramienta, ya que el uso de dos lenguajes diferentes dificulta la implementación de los experimentos.

SimGrid[38] es un simulador que proporciona funcionalidades básicas para simular algoritmos y aplicaciones distribuidas, tales como *workstations* y entornos *grid*. Los recursos están modelados teniendo en cuenta aspectos como latencia o tasa de servicio. Sin embargo, en las simulaciones no se incluyen detalles hardware tales como acceso a memoria principal o memoria caché, que pueden afectar al consumo de energía. El simulador se encuentra actualmente en desarrollo.

Construida utilizando la plataforma de simulación OMNeT++, iCanCloud es una plataforma que permite el modelado y simulación de sistemas *cloud*. iCanCloud proporciona un conjunto de módulos diseñados para reproducir con fidelidad detalles de comportamiento de los distintos componentes del sistema. A su vez, iCanCloud puede utilizarse con el *framework* $E - mc^2$[14], que incluye mecanismos que permiten realizar mediciones de consumo de energía.

2.2. Mutation testing

Mutation testing tiene como objetivo determinar la eficacia de un conjunto de tests para detectar errores. Para ello, se inyectan fallos en el código, mediante *operadores de mutación*, que corresponden a reglas de transformación de la sintaxis del lenguaje. La idea es crear copias del programa original, cada una de las cuales presenta alguna modificación sintáctica.

El objetivo es determinar cuántas de estas variaciones del programa, denominadas *mutantes*, se comportan de forma diferente al programa original, respecto a un conjunto de tests. Cuando un mutante se comporta de forma diferente al programa original, para algún test `t`, se dice que `t` *mata* al mutante, en otro caso se dice que el mutante está *vivo*. Consideremos el ejemplo del Cuadro 2.1 en el que se muta el programa original de modo que la condición $a > 0 \wedge b > 0$ es sustituida por $a > 0 \vee b > 0$. Esta modificación genera un mutante. Para matar este mutante se requiere un test que provoque un resultado diferente en la ejecución del mutante y del programa original. Si aplicásemos el test $a = -3, b = 5$, el test habría matado el mutante, sin embargo el test $a = 3, b = 5$ no permitirá detectar el fallo. De este modo, los mutantes permiten evaluar lo *buenos* que son los tests seleccionados. Cuantos más mutantes sean matados por el conjunto de tests seleccionados, mayor será la calidad de los mismos.

Algunos mutantes, denominados mutantes equivalentes, presentan el mismo comportamiento que el programa original para cualquier entrada y no pueden ser matados por ningún test. La detección de mutantes *equivalentes* es un problema indecidible, por lo que deben ser detectados manualmente, lo que supone un alto coste en la aplicación de esta técnica[8]. Para tratar de mitigar este problema, existen técnicas, como las que se proponen en el trabajo de Offutt y Craft[71], donde se detallan algoritmos para determinar distintas clases de mutantes equivalentes. Estos algoritmos están basados en análisis de flujo de datos y técnicas de optimización de compiladores. En otro de los trabajos relevantes en este campo[40], Robert Hierons y Mark Harman presentan el uso de técnicas de *slicing programming* aplicadas a la detección de mutantes equivalentes, lo que permite reducir la generación de los mismos.

Programa *p*	Programa *p'*
...	...
if(a>0 **and** b>0)	if(a>0 **or** b>0)
return 1	return 1
else	else
return 2	return 2
...	...

Cuadro 2.1: *Ejemplo de mutación de código*

Mutation testing se basa en dos hipótesis relacionadas con la naturaleza de los errores en los programas. Por una parte, se considera que los programadores tienden a desarrollar programas que son correctos. Esta hipótesis, conocida como *competent programmer hypothesis*[64], establece que los fallos en los programas son pequeños errores sintácticos que podrían corregirse fácilmente. Por lo tanto, los mutantes *simulan* posibles efectos de fallos reales. De este modo, si un conjunto de tests es *bueno* detectando errores en los mutantes, también lo será con los fallos en el programa original. La segunda hipótesis, conocida como *coupling effect hypothesis*[26], establece que los errores complejos están asociados a errores simples, de forma que un conjunto de tests de entrada que detecte todos los fallos simples en un programa, detectará un alto porcentaje de fallos complejos. Existen numerosos estudios que consideran la validez de esta hipótesis, tanto de índole teórica[26,70], como práctica[57].

La Figura 2.1 presenta el esquema de funcionamiento general de *mutation testing*. Dado un conjunto de mutantes, generado a partir de la aplicación de un conjunto de operadores de mutación sobre un programa, y un conjunto de tests, se analizará la adecuación del mismo para detectar los fallos. En primer lugar, el conjunto de tests debe ser aplicado al programa original para asegurar su correcto comportamiento respecto al mismo ①. En el caso que los resultados sean incorrectos, el programa original debe ser corregido ②ⓐ. En caso contrario, los operadores de mutación considerados son aplicados al programa original para la generación de todos los posibles mutantes ②ⓑ. Cada mutante será ejecutado utilizando el conjunto de tests ③. Finalizado este proceso, se calculará el *mutation score*, que indica el porcentaje de

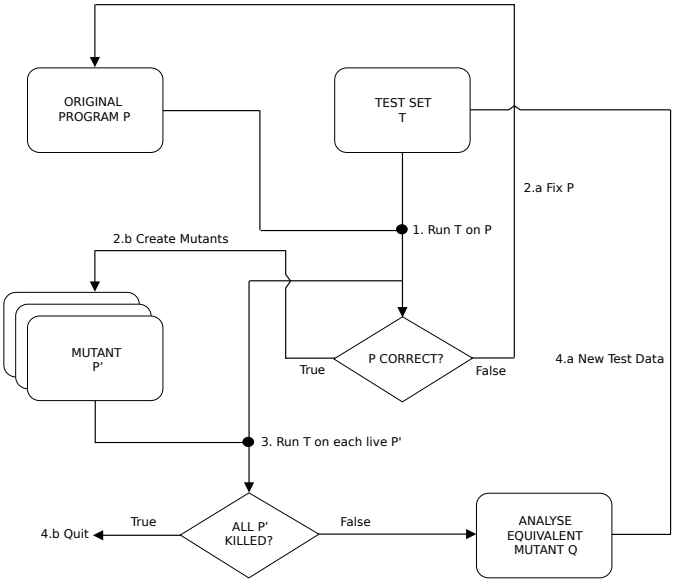

Figura 2.1: *Esquema general del proceso de Mutation Testing*

mutantes no equivalentes que han sido matados por el conjunto de tests. El objetivo es que este porcentaje sea del 100 %, es decir, que todos los mutantes no equivalentes hayan sido matados. Los mutantes vivos indican falta de adecuación del conjunto de tests para detectar fallos potenciales en el programa. Por tanto, teniendo en cuenta el ejemplo del Cuadro 2.1, para un test donde $a = 8, b = 7$ tanto p como p' tendrían el mismo resultado. En este caso p' sería considerado como un mutante vivo. El usuario debe añadir tests adicionales al conjunto inicial y comprobar que el programa original los satisface, para tratar de matar los mutantes que permanecen vivos ④a. Este proceso deberá repetirse hasta que el usuario esté satisfecho con el *mutation score* obtenido ④b.

Aunque cumplen de forma efectiva con el objetivo de analizar la adecuación de los tests, las técnicas de *mutation testing* conllevan varias desventajas que dificultan su empleo en sistemas complejos. Una de estas desventajas es el tiempo invertido para detectar mutantes equivalentes, así como el elevado coste computacional que requiere la ejecución del ciclo completo de *mutation testing* sobre un programa, debido a la elevada cantidad de mutantes generados que deben ser ejecutados utilizando el conjunto de tests. Al tiempo de generación de cada mutante, hay que añadir el tiempo de compilación, ejecución y comprobación

de resultados. Teniendo en cuenta estos factores, las aportaciones científicas orientadas a introducir mejoras a las técnicas de *mutation testing* en las últimas décadas han sido innumerables. Estas técnicas pueden dividirse en técnicas de reducción del conjunto de mutantes y técnicas de reducción del tiempo de ejecución.

2.2.1. Optimizaciones de *mutation testing*

Una de las formas de reducir el tiempo total de la ejecución del esquema de *mutation testing*, consiste en reducir el número total de mutantes generados tras aplicar los operadores de mutación. Alguna de las técnicas más relevantes son: *mutant sampling, mutant clustering y high order mutation.*

En el caso de *mutant sampling*, la mejora se obtiene seleccionando un subconjunto aleatorio de los mutantes generados. Según Wong y Mathur's[62,98] la selección aleatoria del 10 % de los mutantes es sólo un 16 % menos efectiva que el conjunto completo de los mutantes generados en términos de *mutation score.*

Un método más sofisticado de selección de los mutantes generados, se basa en el uso de algoritmos de clustering[42]. *Mutation clustering* trata de reducir el coste computacional disminuyendo el número de mutantes utilizando el algoritmo de *k-means* y el algoritmo de agrupamiento jerárquico. El principio básico de este método es la selección de un único mutante de cada *cluster*, llegando a obtener un *mutation score* de 100 % reduciendo el número de mutantes hasta en un 91 %.

La tercera técnica utilizada para reducir el conjunto de mutantes es conocida como *selective mutation.* El principio básico de *selective mutation* es la reducción del número de mutantes disminuyendo el número de operadores de mutación aplicados. El estudio realizado por Offut et al.[75], donde se extiende el trabajo de Wong y Mathur's[62,98], sugiere que omitiendo dos operadores se mantiene el *mutation score* en 99, 99 %, produciéndose a su vez una disminución de mutantes en un 24 %. De igual forma, omitiendo cuatro operadores, el *mutation score* obtenido es de un 99, 84 %, con una reducción de mutantes de un 41 %. Final-

mente, como último experimento, la omisión seis operadores permite obtener un *mutation score* de un $88,71\,\%$ con una reducción de un $60\,\%$ de los mutantes.

El objetivo principal de las tres técnicas descritas anteriormente, *mutant sampling, mutation clustering* y *selective mutation*, consiste en disminuir el tiempo de proceso realizando una selección de mutantes, una vez han sido generados. De esta forma, al reducir el número de mutantes se disminuye el tiempo requerido para la ejecución de los tests. Entre las técnicas basadas en la reducción del tiempo de ejecución del programa mutado, se puede distinguir entre las enfocadas en la ejecución de los mutantes, las basadas en la optimización del entorno de ejecución y aquellas que usan plataformas avanzadas.

En *strong mutation*[27] se considera que un mutante es matado si la aplicación de un test produce una salida diferente a la que genera el programa original al finalizar la ejecución de ambos. Esto requiere la ejecución completa del mutante para comprobar el resultado de la aplicación del test. En *weak mutation*[41], tan sólo se requiere que el estado del mutante en el punto en el que se realizó la mutación sea diferente al correspondiente en el programa original. La ventaja de *weak mutation* reside en que no es necesario realizar un ciclo completo de ejecución para cada mutante. Existe una tercera técnica conocida como *firm mutation*[100], donde la idea es paliar las dificultades de las dos técnicas anteriores proporcionando soluciones intermedias.

Otra de las tendencias consiste en la introducción de técnicas de optimización en el entorno de ejecución. Con ello se persigue reducir los tiempos de interpretación, compilación y ejecución del código a lo largo del proceso de *mutation testing*. Una de las técnicas más reseñables en este campo es conocida como *compiler-based*, propuesta por DeMillo et al.[25]. Esta técnica pretende reducir el exceso de tiempo invertido en la compilación de los mutantes, siendo esto posible debido a que los cambios sintácticos entre el código fuente original y los mutantes son mínimos, por lo que gran parte del tiempo de compilación es redundante. Para conseguir su propósito el compilador genera dos archivos de salida, el ejecutable del programa original y un archivo que contiene las instrucciones necesarias para convertir el

ejecutable del código fuente original en el ejecutable de cada mutante.

Una técnica que mejora el rendimiento hasta en un 300 % es *mutant schemata*[91,92]. Esta técnica se basa en la generación de un supermutante, al que se le introducen todas las mutaciones posibles de modo que solo se requiere compilar una vez, frente a las múltiples compilaciones realizadas en el esquema tradicional. Una de las aportaciones más recientes en la reducción de coste de compilación es conocida como *bytecode translation*. En esta técnica, se generan los mutantes directamente desde el código ejecutable original, en lugar de construirlos desde el código fuente. Con esto, los mutantes pueden ser generados sin necesidad de realizar la fase de compilación.

Por último y con un uso poco extendido, existen plataformas avanzadas para la ejecución de *mutation testing*. Se trata de técnicas utilizadas para ejecutar mutantes en sistemas con procesadores vectoriales *Single Instruction Multiple Data*[51,52] y máquinas distribuidas *Multiple Instruction Multiple Data*[103].

2.2.2. Aplicaciones de mutation testing

Desde la primera aportación de DeMillo et al. en 1970[27], *mutation testing* ha sido ampliamente estudiado por la comunidad científica. Entre los numerosos estudios realizados en sus diversos campos de aplicación se pueden distinguir dos grandes grupos. Por una parte, la aplicación de mutaciones sobre programas, conocida en la literatura como *program mutation*[24], en la que los cambios son introducidos directamente en el código fuente del programa, lo que la convierte en un esquema de *testing* de caja blanca. Por otra parte, *specification mutation*[36], caracterizada por introducir mutaciones en la especificación del programa, no accede en ningún caso al código fuente durante la fase de *testing*, y por lo tanto corresponde a un esquema de *testing* de caja negra.

En la década de nacimiento de *mutation testing*, *Fortran* era considerado uno de los lenguajes de programación más extendido del momento. Por esto, los experimentos iniciales, tales como la primera definición de operadores de mutación o la primera herramienta

de *mutation testing*, fueron realizados sobre *Fortran*. Con esta herramienta, conocida como *PIMPS*[9,10], se realizaron las pruebas pioneras de *mutation testing* aplicadas a varios programas en Fortran IV[9,64]. La primera especificación formal de operadores no sería realizada hasta 1987, cuando Offut et al.[50,72] propusieron 22 operadores de mutación sobre Fortran. Entre las herramientas más estudiadas se encuentra *MOTHRA*[44]. Los primeros operadores de mutación sobre Ada fueron propuestos por Bowser[6] en 1988. Posteriormente, estos operadores fueron rediseñados por Offut[76], produciendo un conjunto de 65 operadores de mutación.

Las aportaciones iniciales de *mutation testing* en el lenguaje de programación C fueron realizadas por Agrawal et al. En el trabajo[3] se describen 77 operadores de mutación diseñados para el lenguaje ANSI C. Dichos operadores, están categorizados bajo criterios sintácticos.

Existen varias herramientas que aplican técnicas de *mutation testing* en C. Una de ellas fue PROTEUM 1.4, que implementa 75 de los 77 operadores propuestos por Agrawal e incluye la técnica de optimización basada en compilador[23]. Otra de las herramientas más destacadas, diseñada por Yia et al. es MILU[45]. Al contrario que todas las herramientas anteriores que aplicaban todos los operadores al programa testeado, permite seleccionar al usuario un conjunto específico de operadores de mutación. MILU utiliza técnicas de reducción de tiempo de ejecución.

El lenguaje donde se encuentra el mayor porcentaje de las aportaciones de *mutation testing* es Java. La primera aportación fue realizada por S.Kim et al[48]. Se trata de una definición de nuevos operadores de mutación específicos para Java, debido a que los operadores de mutación tradicionales no son suficientes para el paradigma de programación orientado a objetos[49,58]. En el estudio de selección de dichos operadores, se hizo uso de una técnica conocida como HAZOP, que realiza un análisis y guarda los resultados de las desviaciones de los sistemas. HAZOP fue aplicada en la definición de la sintaxis de Java para identificar los posibles fallos que pudiesen ser cometidos por un usuario. A partir de estos fallos, fueron

definidos los operadores de mutación.

Entre las herramientas más destacadas para mutar programas escritos en Java, se encuentra MuJava *(Mutation Java)*[59]. Desarrollado por Offut, MuJava es un *framework* de mutación para el lenguaje de programación Java, que presenta un método para reducir el coste de ejecución en programas orientados a objetos. Para ello utiliza dos técnicas clave, *Mutant Schemata Generation (MSG)* y *bytecode translation*.

Al tratarse de un lenguaje que surgió posteriormente, las aportaciones de *mutation testing* en C# están basadas en el trabajo previo realizado en Java. El conjunto de operadores de mutación de Java se extendió con operadores específicos para C#. La herramienta más destacada de *mutation testing* en este lenguaje de programación es CREAM, cuya particularidad reside en incluir la implementación de técnicas de reducción de costes como *mutation sampling*, *selective mutation* y *mutation clustering*.

Mutation testing tambien es utilizado en el campo de las bases de datos para detectar errores. En 2005, Chan et al.[16] propusieron 7 operadores de mutación para el lenguaje SQL. Posteriormente Tuya et al.[90] presentaron otro conjunto de operadores para consultas SQL. Dentro de las aportaciones de Tuya et al. se encuentra una de las herramientas más utilizada, SQLMutation[89]. SQLMutation genera automáticamente mutantes de consultas SQL. SQLMutation consta de dos interfaces, uno de ellos es una aplicación web para generar interactivamente mutantes y el otro es un *web service* que puede integrarse con otras aplicacions desarrolladas en distintas plataformas.

Otra de las herramientas más reseñables es MUSIC (*MUtation-based SQL Injection vulnerabilities Checking tool*). MUSIC es una herramienta de inyección de vulnerabilidades SQL, que genera automáticamente mutantes para aplicaciones escritas en JSP *(Java Server Pages)* y realiza un análisis del rendimiento del proceso de mutación.

Aspect-Oriented Programming es un paradigma de programación que trata de buscar una modularización adecuada de las aplicaciones, eliminando las preocupaciones transversales propias de aspectos más técnicos. Ferrari et al.[32] proponen 26 operadores de mutación ba-

sados en la generalización de fallos cometidos en estos programas. Recientemente, Delamare et al. introdujeron una técnica de detección de mutantes equivalentes utilizando análisis estático[22].

Las técnicas de *mutation testing* para especificaciones formales engloba un amplio rango de campos de aplicación. Una de las primeras aportaciones en *specification mutation* fue en la especificación de expresiones lógicas, donde Gopal et al. se centraron en las operaciones de especificación de predicados de cálculo. Dentro de este mismo campo, realizando una tarea de refinamiento en las expresiones de cálculo, Aichernig[4] y Woodwarg[99] aportaron la definición de operadores de mutación en especificaciones algebraicas y llevaron a cabo experimentos para proporcionar resultados empíricos en base a código ejecutable.

Más recientemente, se han propuesto numerosas técnicas formales para especificaciones dinámicas de sistemas. Una de ellas es la especificación de máquinas de estado finitas, donde Fabbri et al.[33] proponen 9 operadores de mutación que representan fallos relacionados con los estados, eventos y salidas de una máquina de estados. Pudiéndose considerar una extensión de las máquinas de estado finitas, los diagramas de estados, conocidos en la literatura como *statecharts*, son ampliamente utilizados en la especificación formal de sistemas. Fabbri et al.[34] propusieron el primer conjunto de operadores de mutación para estas especificaciones. Posteriormente, utilizando estos operadores, Yoon et al. introdujeron el concepto *State-based Mutation Test Criterion (SMTC)*[102].

Además de las aplicaciones en los campos descritos, *specification mutation* ha sido aplicado a otros lenguajes de especificación[73,86,87].

En 2001, con motivo del auge de los *web services*, Offut et al.[55] introdujeron un modelo de especificación para formalizar las interacciones entre componentes web. Basándose en este modelo de especificación, se propuso un conjunto de operadores de mutación orientados al modelo de datos XML. Posteriormente, este trabajo fue extendido y orientado a la mutación sobre el lenguaje XML[77] y lenguajes basados en XML[31,54].

Uno de los aspectos más críticos en las comunicaciones, son los protocolos. Para asegurar

su robustez, Probert et al. propusieron un conjunto de operadores de mutación orientados al *testing* de protocolos de red[78]. Posteriormente, Vigna et al. aplicaron el uso de técnicas de *mutation testing* en la detección de intrusiones en red[94].

Para tratar de evitar intrusiones y amenazas que puedan poner en riesgo los sistemas informáticos, se procede a la creación de políticas de seguridad. Por ello, resulta de gran importancia minimizar los defectos en este tipo de políticas, de forma que *mutation testing* es una técnica ampliamente aplicada en este campo, a través de la definición de operadores de mutación orientados a distintos tipos de políticas de seguridad[61,88].

Además de la determinación del nivel de adecuación de los tests, *mutation testing* tiene otras aplicaciones relacionadas con el *testing*. Una de ellas es la generación automática de tests. El objetivo es facilitar la labor del usuario en la creación de conjuntos de tests capaces de *matar* el máximo número de mutantes y alcanzar con ello un *mutation score* lo más alto posible. Para ello, existen técnicas como *Constraint-Based Test data generation (CBT)* propuesta por Offut[69], la cual crea un conjunto de tests automáticamente, analizando los mutantes generados y teniendo en cuenta su accesibilidad, necesidad y suficiencia.

Una de las herramientas de generación de tests automática más influyente es Godzilla[29]. Desarrollada por Offut et al. y haciendo uso de técnicas de CBT, Godzilla consigue *matar* el 90 % de los mutantes en la mayor parte de programas[28].

Otra de las aplicaciones de las técnicas de *mutation testing* en la asistencia en el proceso de *regression testing*. *Regression testing* tiene como objetivo detectar errores, tanto funcionales como no funcionales, dentro de un sistema después de realizar modificaciones. Estas modificaciones se pueden corresponder con los cambios realizados en la configuración, actualizaciones o mejoras. Do et al. utilizaron *mutation testing* para establecer una priorización de tests[30].

Mutation testing ha sido utilizado a su vez en el proceso de minimización de un conjunto de tests, manteniendo, en la medida de lo posible, su efectividad. Para ello, Offut et al. propusieron PingPong[74], una método a partir del cual se realiza una reducción en el conjunto

de tests de hasta un 33 % sin perder efectividad.

2.3. Mutación de código basada en simulación

La validación de aplicaciones en sistemas distribuidos es un proceso complejo. Por ello, el uso combinado de técnicas de *mutation testing* y simulación simplifica la depuración de las aplicaciones. En este campo, el número de aportaciones existentes es muy limitado.

Una de estas aportaciones fue presentada por Rutherford et al.[83], donde se propone una aproximación sencilla para comprobar la adecuación de los tests aplicados en entornos de simulación de sistemas distribuidos. Los sistemas distribuidos propuestos están basados en modelos básicos, donde el conjunto total de nodos utilizado para realizar el estudio es reducido. En la fase de experimentación se utiliza la herramienta SimJava, empleada para simular el comportamiento de la red y MuJava como *framework* de mutación de código. El principio fundamental se basa en medir la eficacia de un test individual, contabilizando el número de mutantes que se han *matado*. Para llevar a cabo los experimentos se proponen dos sistemas distribuidos. Uno de ellos basado en el algoritmo de GBN[53], que se corresponde con el proceso de intercambio de datos entre dos nodos basado en el modelo cliente/servidor. El segundo se basa en el esquema *link-state routing*, donde se analizan distintas topologías.

Como extensión a su propuesta anterior, Rutherford et al.[84] llevan a cabo tres contribuciones al uso y desarrollo de la adecuación de tests para sistemas distribuidos. Se propone un método de *testing* para simulaciones basadas en eventos. Además, se describe una técnica de inserción de errores para evaluar la adecuación de los tests. Por último, se presenta un conjunto de casos de estudio para validar el método propuesto. En estos casos de estudio se experimenta en un escenario formado por dos nodos, donde uno de ellos representa un servidor DNS, que proporciona servicio al otro nodo, que representa un cliente.

2.4. Comparación de herramientas y trabajos actuales

En los trabajos analizados en la literatura, no se ha encontrado ningún *framework* de simulación de sistemas distribuidos que integre técnicas de *mutation testing*. Tal y como se indica en la sección 2.1, existen numerosas herramientas de simulación de sistemas distribuidos, pero en ninguno de los casos expuestos se contempla la aplicación de técnicas de *testing*. Para validar una aplicación, generalmente se realiza un conjunto de pruebas de forma manual y de manera poco exhaustiva, por lo que existe una alta probabilidad de no cubrir todos los casos necesarios con las pruebas realizadas.

Aunque existe una gran variedad de *frameworks* de mutación, algunos de ellos descritos en la sección 2.2, ninguna de estas herramientas es adecuada para la simulación de entornos altamente distribuidos. Esto se debe a que no cuentan con mecanismos específicos para motores de simulación. De esta forma, se aplican un número excesivo de operadores de mutación convencionales que generan a su vez, un elevado número de mutantes, requiriendo un alto coste computacional.

Los estudios descritos en la sección 2.3 proponen trabajos que consideran ambos aspectos[83,84]. Sin embargo, estas propuestas están diseñadas utilizando herramientas que no resultan adecuadas para sistemas de gran escala. En el caso del *framework* de simulación SimJava, no proporciona modelos realistas de recursos de computación básicos, como discos, procesadores o distintas tecnologías de interconexión de equipos. Estos modelos están representados por programas de 400 líneas de código, modelando el sistema con un nivel de abstracción muy alto, lo cual dista de ser una simulación realista de sistemas distribuidos. A su vez, la cantidad reducida de nodos que forman los escenarios, limitan en gran medida la escalabilidad de los sistemas simulados.

Capítulo 3

Descripción de OMNeT++ y SIMCAN

Este capítulo presenta un breve resumen del *framework* de simulación OMNeT++, así como de la plataforma de modelado y simulación de sistemas distribuidos SIMCAN. El objetivo de este capítulo es mostrar las características principales de los entornos de simulación utilizados en este trabajo. De esta forma, se pretende facilitar la lectura de los capítulos posteriores que describen las técnicas de mutación empleadas sobre estos entornos.

3.1. Introducción a OMNeT++

OMNeT++ es un *framework* de simulación para construir simuladores orientados a modelos de red. La estructura de OMNeT++ está compuesta por un conjunto de módulos que envían y reciben mensajes a través de un canal de comunicaciones previamente configurado. De esta forma, la definición de los módulos, junto con los canales de comunicaciones, forman una red que simula la arquitectura deseada. Actualmente, el uso de este *framework* está muy extendido en la comunidad científica, existiendo numerosas herramientas de simulación basadas en OMNeT++, tales como INET, Castalia, MiXiM, etc. Además, durante el año 2014 se publicaron 368 artículos que utilizan este *framework*.

Los módulos de OMNeT++ están implementados en C++ y su comportamiento puede ser programado utilizando dos modelos de programación diferentes:

- Modelo orientado a eventos. Se recibe un mensaje en cada unidad de tiempo, de forma

que una función del módulo es ejecutada utilizando el mensaje como parámetro de la función.

- Modelo basado en co-rutinas. El módulo ejecuta una función principal. Los mensajes son procesados cuando la función principal ejecuta una sentencia *receive*.

El modelo basado en co-rutinas es más intuitivo y fácil de desarrollar. Sin embargo, requiere una cantidad elevada de memoria, la cual debe asignarse a cada módulo. Por esto, este modelo no es viable para simulaciones a gran escala.

Los módulos tienen una organización jerárquica que facilita la construcción de arquitecturas de redes de computación. Existen dos tipos de módulos:

- Módulos simples, que únicamente incluyen el módulo en sí.
- Módulos compuestos, que incluyen otros módulos como componentes.

Al igual que los módulos, los mensajes están implementados en C++. Los mensajes son esencialmente una colección de datos y funciones. Estas funciones son utilizadas para realizar el tratamiento de datos. Hay distintas maneras de implementar un objeto:

- Escribiendo una especificación del mensaje en lenguaje NED. Esta especificación es precompilada para generar el objeto C++. Este método únicamente funciona con mensajes compuestos de valores simples.
- Escribiendo objetos C++ que hereden de la clase general que representa el mensaje. El nuevo objeto puede modificar el comportamiento definido en la clase general.

Los objetos mensaje pueden obtener a su vez otros mensajes utilizando la característica de encapsulación ofrecida por OMNeT++. La encapsulación está limitada a un único elemento por mensaje.

En OMNeT++, una simulación se configura mediante ficheros de texto plano utilizando el lenguaje NED. En estos ficheros se especifican tanto el número de instancias como los parámetros de los módulos que van a ser simulados. Esto ofrece la ventaja de poder configurar varios modelos y lanzar las simulaciones sin necesidad de recompilar el código fuente.

Una de las principales desventajas de OMNeT++ es su estilo de programación orientado a eventos. Este estilo de programación dificulta el desarrollo de nuevos módulos, ya que el programador tiene que contar con la complejidad añadida de gestionar y sincronizar los eventos. A diferencia del estilo de programación imperativo, en este caso las acciones no se llevan a cabo cuando se invoca un método de un objeto, sino cuando el evento generado a partir de esa invocación llega a su módulo correspondiente. En modelos que representen arquitecturas de gran escala, tales como sistemas cloud o clusters de alto rendimiento, esta tarea puede llegar a ser extremadamente compleja, sobre todo en la fase de depuración, debido a la enorme cantidad de mensajes y eventos presentes en el sistema.

Algunos de los métodos más utilizados para desarrollar nuevos módulos en OMNeT++ se detallan en el Listado 3.1. Los métodos *initialize* y *finish* inicializan un módulo y finalizan su ejecución, respectivamente. Es muy común, sobre todo en programadores noveles, no incluir o colocar en un lugar incorrecto las llamadas a estos métodos.

El envío de mensajes se realiza con los métodos *send* y *scheduleAt*. Mientras que el primero puede enviar un mensaje *msg* a través del canal *gateid* a otros módulos, el segundo permite el envío retardado del mensaje *msg*, después de t segundos, al mismo módulo que invocó el método. Esta función es especialmente útil para la implementación de temporizadores.

Algunos métodos, como *cancelEvent*, tienen la funcionalidad de cancelar un determinado evento, lanzado mediante el método *scheduleAt*. El uso de este método resulta especialmente delicado cuando el programador debe controlar varios eventos lanzados en el mismo módulo, ya que de no realizar correctamente la sincronización entre ellos, es muy probable que se genere un error durante la simulación. Por otro lado, el método *endSimulation* finaliza la

simulación completa del modelo.

Para poder gestionar mensajes enviados entre módulos existe el método *handleMessage*. Puesto que este método debe implementarse en cada módulo con un comportamiento específico, es uno de los métodos más relevantes en el desarrollo de nuevos módulos.

```
1  void initialize();
2  void finish();
3
4  int send (cMessage *msg, int gateid);
5  int scheduleAt (simtime_t t, cMessage *msg);
6
7
8  cMessage* cancelEvent (cMessage *msg);
9  void endSimulation ();
10
11 void handleMessage(cMessage *msg);
```

Listado 3.1: *Métodos proporcionados por OMNeT++*

3.2. Introducción a SIMCAN

SIMCAN es una plataforma de simulación orientada al análisis y estudio de aplicaciones paralelas en entornos distribuidos. El objetivo de SIMCAN es mitigar, en la medida de lo posible, las carencias que ofrecen otras herramientas de simulación, tales como flexibilidad, precisión, rendimiento y escalabilidad. Estas características convierten a SIMCAN en una plataforma potente para modelar y simular sistemas distribuidos.

El proyecto SIMCAN fue iniciado en 2008[65]. Las principales contribuciones de esta plataforma de simulación se detallan a continuación:

- Balance entre escalabilidad, rendimiento y precisión para conseguir una simulación rápida en entornos a gran escala, con un nivel de detalle variable. De esta forma, se ha integrado en una única plataforma de simulación el modelado de CPU, almacenamiento, red de comunicaciones y memoria.

- Modelado del sistema de almacenamiento muy detallado, incluyendo en su repositorio distintos modelos de sistemas de ficheros, tales como Ext2, PVFS y NFS.

- Proporciona un API basado en POSIX y una librería adaptada de MPI[37] para facilitar el modelado y simulación de aplicaciones paralelas. Además, pueden ejecutarse trazas de aplicaciones reales y utilizar diagramas de estados.

- Debido a su arquitectura flexible, los componentes pueden ser intercambiados fácilmente.

Con el objetivo de mantener el equilibrio entre rendimiento, precisión, flexibilidad y escalabilidad, el núcleo principal de esta plataforma se basa en un diseño flexible que permite integrar cada sistema básico en un único modelo de simulación. Estos sistemas son el sistema de almacenamiento, el sistema de memoria, el sistema de procesamiento y el sistema de red de comunicaciones.

La Figura 3.1 muestra la arquitectura de SIMCAN. En este esquema, se puede apreciar cómo las aplicaciones acceden a los recursos de cada sistema básico a través de un API común. De esta forma, dependiendo de los requisitos del usuario, se pueden modelar distintos sistemas básicos, cada uno con su correspondiente nivel de detalle.

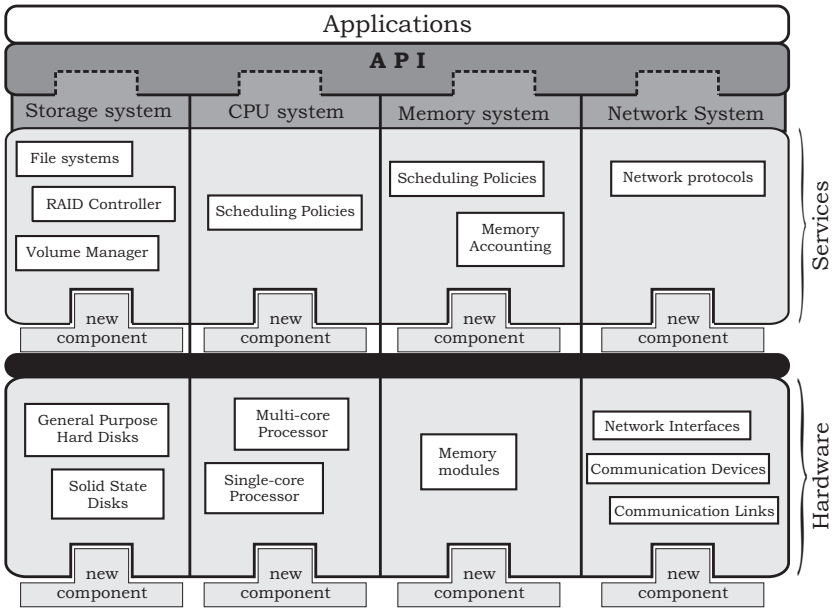

Figura 3.1: *Arquitectura de SIMCAN*

SIMCAN proporciona un conjunto de componentes existentes utilizados para modelar distintas arquitecturas. Estos componentes están organizados jerárquicamente en un repositorio de módulos. Actualmente, en este repositorio se ofrecen modelos completos de sistemas distribuidos con distintos niveles de detalle y escalabilidad, pudiéndose añadir y diseñar nuevos componentes al repositorio. Para proporcionar mayor flexibilidad, se permite incluir simuladores existentes al repositorio, como por ejemplo, diskSim [7].

En un sistema de computación, el nodo es el componente más relevante. De igual forma, en SIMCAN un nodo es un bloque de construcción para crear sistemas distribuidos. La simulación de un único nodo está basada en unir los módulos que simulan aplicaciones con los componentes que simulan cada uno de los cuatro sistemas básicos (ver Figura 3.2).

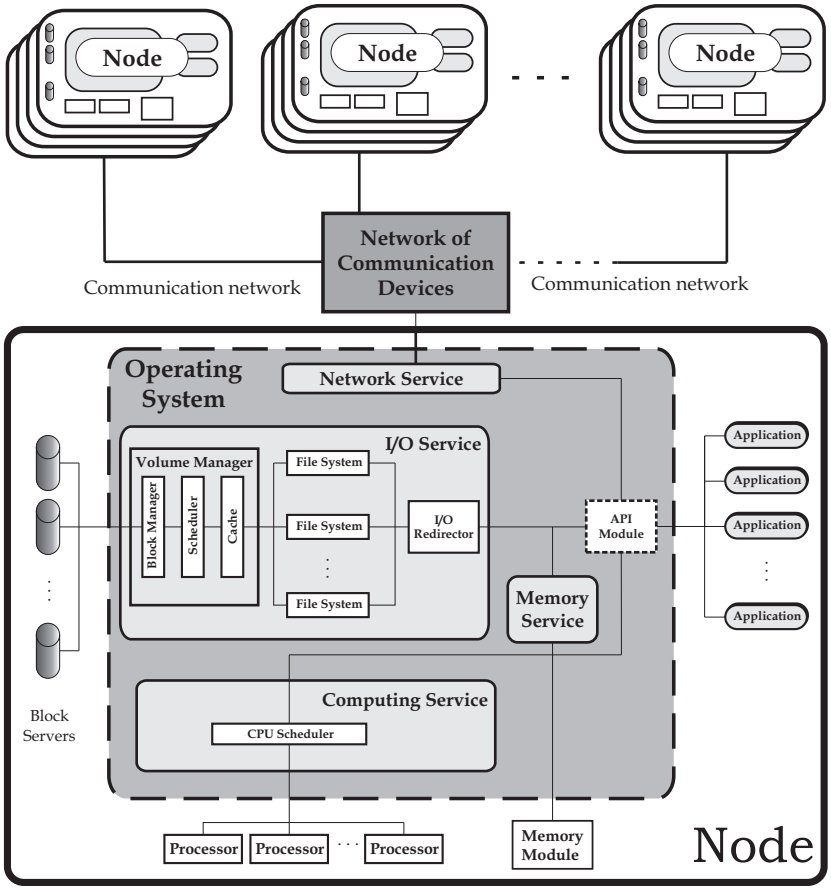

Figura 3.2: *Estructura de un nodo en SIMCAN*

Con el propósito de facilitar la tarea de construir y configurar entornos distribuidos altamente escalables, la plataforma SIMCAN proporciona una clasificación flexible de nodos de agregación basado en bloques, la cual imita la agregación utilizada en sistemas reales (ver Figura 3.3). A continuación se describe una colección de nodos de agregación proporcionado por SIMCAN, que permite crear las arquitecturas distribuidas más comunes:

- **Nodo de computación**: Este módulo simula el comportamiento de un nodo. Contiene los módulos necesarios para simular los sistemas que incluye un nodo real. Los componentes de cada nodo pueden ser modificados y configurados para variar su comportamiento, pudiendo actuar como nodo de cómputo, nodo de almacenamiento o una mezcla entre ambos.

- **Placa de nodos**: Este módulo es un conjunto de nodos localmente desplegados. Estos nodos incluyen un *switch* local para poder comunicarse mediante una red local, actuando como único elemento de comunicación con el exterior.

- **Rack**: Este módulo es un conjunto de placas de nodos. Cada placa de nodos en el rack tiene su propio canal de comunicaciones.

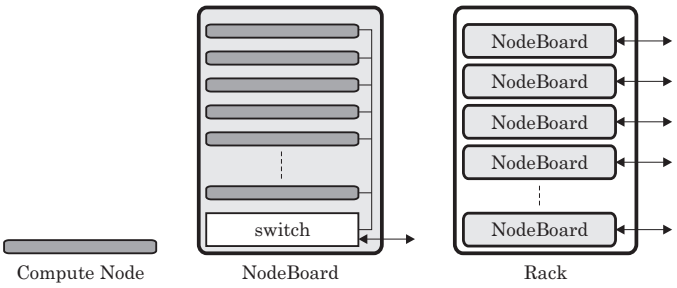

Figura 3.3: *Agregación de nodos en SIMCAN*

3.2.1. Modelado y configuración de sistemas distribuidos

En SIMCAN, la definición y configuración de un entorno está dividido en tres niveles distintos, donde cada uno de ellos se corresponde con un nivel de abstracción. Estos niveles de describen a continuación:

- El nivel 1 consiste en la definición del sistema correspondiente al entorno que va a ser simulado. Básicamente, este nivel contiene una lista con todos los componentes y sus conexiones correspondientes. Esta descripción está especificada en texto plano utilizando el lenguaje NED.

- El nivel 2 consiste en la configuración de cada componente definido en el nivel anterior. Esta configuración consiste en asignar valores al conjunto de parámetros de cada componente, con el objetivo de personalizar el comportamiento del entorno simulado. Tal como ocurre en el nivel 1, esta configuración se realiza en ficheros de texto plano.

- El nivel 3 consiste en un conjunto de archivos de configuración adicionales para personalizar los componentes específicos que lo requieran. En la mayor parte de los casos, estos archivos son usados en componentes que modelan sistemas con un nivel alto de detalle.

3.2.2. Módulo API de SIMCAN

El módulo API lleva a cabo un papel fundamental en el componente que simula un sistema operativo. Básicamente, este módulo contiene un conjunto de llamadas agrupadas como un API (*Application Programming Interface*) para ejecutar aplicaciones en SIMCAN. Una llamada al sistema es el mecanismo utilizado por las aplicaciones para solicitar recursos del sistema operativo. Estas llamadas proporcionan un interfaz entre las aplicaciones y el sistema operativo (ver Figura 3.2).

Para facilitar la portabilidad de las aplicaciones, las funciones ofrecidas por este API están basadas en POSIX. POSIX es el nombre de la familia de estándares especificada en IEEE, el cual define un API que permite un amplio rango de funciones de computación comunes, compatibles en distintos sistemas operativos. Las funciones del API que proporciona SIMCAN se muestran en los listados 3.2, 3.3, 3.4, y 3.5.

El Listado 3.2 ofrece un interfaz para utilizar el servicio de cómputo. Básicamente, las aplicaciones que hacen peticiones a la CPU deben invocar la función *simcan_request_cpu*

y especificar el número de instrucciones que deben ser ejecutadas, medidas en MIs (Millones de Instrucciones).

```
1 void simcan_request_cpu (long int numInstructions);
```

Listado 3.2: *Funciones ofrecidas por el API de cómputo del sistema*

Con el objetivo de interactuar con el sistema de memoria, las aplicaciones deben implementar un interfaz especificado en el Listado 3.3. Principalmente, este interfaz consiste en dos funciones. La primera de ellas, llamada *simcan_request_allocMemory*, es utilizada para reservar memoria. La segunda, llamada *simcan_request_freeMemory*, es utilizada para liberar memoria reservada anteriormente. El parámetro *memorySize* indica la cantidad de memoria requerida, especificada en bytes, para llevar a cabo la operación correspondiente. El parámetro *region* indica la región de memoria involucrada en esta operación.

```
1 void simcan_request_allocMemory (int memorySize, int region);
2 void simcan_request_freeMemory (int memorySize, int region);
```

Listado 3.3: *Funciones ofrecidas por el API de memoria del sistema*

El sistema de almacenamiento se encarga de todas las gestiones relacionadas con el acceso a los datos. El conjunto de funciones mostrado en el Listado 3.4 ofrece un interfaz para interactuar con el sistema de almacenamiento, que consiste básicamente en un conjunto de funciones para gestionar archivos.

```
1 void simcan_request_open (char* fileName);
2 void simcan_request_close (char* fileName);
3 void simcan_request_create (char* fileName);
4 void simcan_request_delete (char* fileName);
5 void* simcan_request_read (char* fileName, unsigned int offset, unsigned int size);
6 void* simcan_request_write (char* fileName, unsigned int offset, unsigned int size);
```

Listado 3.4: *Funciones ofrecidas por el API de almacenamiento del sistema*

Las funciones *simcan_request_open* y *simcan_request_close* se utilizan para abrir y cerrar los archivos indicados en el nombre proporcionado como parámetro a las funciones. De igual forma, las funciones *simcan_request_create* y *simcan_request_delete* se encargan de crear y borrar un archivo.

Finalmente, las funciones *simcan_request_read* y *simcan_request_write* se encargan de leer y escribir datos, respectivamente, en el archivo especificado en el parámetro *fileName*.

La cantidad de datos que deben leer y escribir se indica en el parámetro *size*. Finalmente, el parámetro *offset* indica el punto de inicio del archivo donde se realiza el procesado de datos.

El sistema de redes de comunicación se encarga de la gestión de conexiones con otras aplicaciones ejecutadas en nodos remotos. El API del sistema de comunicaciones se muestra en el Listado 3.5. Utilizando este interfaz, las aplicaciones pueden intercambiar datos con otras aplicaciones.

```
1 void simcan_request_createListenConnection (int localPort, string type);
2 void simcan_request_createConnection (string dAddress, int dPort, int id, string type);
3 void simcan_request_sendDataToNetwork (SIMCAN_Message *sm, int id);
4 void simcan_request_receiveDataFromNetwork (SIMCAN_Message *sm, int id);
```

Listado 3.5: *Funciones ofrecidas por el API de comunicaciones del sistema*

La función *simcan_request_createListenConnection* crea una conexión entrante. Por otro lado, la función *simcan_request_createConnection* establece una conexión con una aplicación remota.

Las función *simcan_request_sendDataToNetwork* se utiliza para enviar datos desde la aplicación que la invoca, a una aplicación remota especificada en el parámetro *sm*. Por otro lado, *simcan_request_receiveDataFromNetwork* se encarga de recibir los datos desde la aplicación remota especificada en *sm*.

3.2.3. Funciones del interfaz MPI para la simulación de aplicaciones distribuidas

Además de las funciones del API anteriormente descritas, SIMCAN proporciona un subconjunto de las funciones proporcionadas por MPI para programar aplicaciones paralelas en arquitecturas distribuidas. Esta sección describe las funciones MPI implementadas en SIMCAN para reproducir el comportamiento de las funciones MPI ejecutadas por una aplicación real.

Las llamadas MPI implementadas en SIMCAN pueden clasificarse en dos clases. La primera clase, conocida como operaciones de comunicación punto a punto, consiste en diferentes operaciones para realizar envío y recepción de datos (ver Listado 3.6).

```
1 void mpi_send (unsigned int rankReceiver, int bufferSize);
2 SIMCAN_MPI_Message * mpi_recv (unsigned int rankSender, int bufferSize);
```

Listado 3.6: *Funciones de MPI orientadas a la comunicación punto a punto*

Las llamadas encargadas de realizar operaciones colectivas pertenecen a la segunda clase. Este tipo de llamadas, que proporcionan sincronización y operaciones entre grupos de procesos, se muestra en el Listado 3.7.

```
1 void mpi_barrier ();
2 void mpi_bcast (unsigned int root, int bufferSize);
3 void mpi_scatter (unsigned int root, int bufferSize);
4 void mpi_gather (unsigned int root, int bufferSize);
```

Listado 3.7: *Funciones de MPI colectivas implementadas en SIMCAN*

Generalmente, para establecer un punto de sincronización entre los procesos que forman parte de la aplicación distribuida, se utiliza la llamada *mpi_barrier*. Cuando un proceso invoca esta función, se queda bloqueado hasta que todos los procesos hayan realizado la misma llamada.

El resto de llamadas colectivas requieren de un proceso encargado para controlar la operación. Este proceso, denominado proceso *root*, se debe indicar como parámetro en la llamada de las funciones. Por ejemplo, la llamada *mpi_bcast* invocada por el proceso *root*, envía un conjunto de datos, de tamaño *bufferSize*, al resto de procesos de la aplicación. Las llamadas *mpi_scatter* y *mpi_gather* se utilizan para repartir y recolectar datos entre los procesos.

Para realizar la sincronización entre procesos en llamadas colectivas, cada proceso tiene una cola local asociada. En esta cola, se reciben todos los mensajes y son almacenados. Cuando un proceso espera un mensaje concreto, cada vez que un mensaje es recibido, se almacena en la cola local hasta que reciba el mensaje esperado.

Capítulo 4

MuTomVo

Uno de los objetivos principales de este trabajo es utilizar las ventajas proporcionadas por las herramientas de simulación y las técnicas de *mutation testing* para construir el *framework* MuTomVo. En este capítulo se presenta una propuesta para afrontar las dificultades de esta integración, donde se exponen los esquemas detallados de la arquitectura del sistema, el proceso completo de *testing* utilizado y el cálculo de la idoneidad de los conjuntos de tests utilizados en este proceso.

4.1. Diseño general de la arquitectura de MuTomVo

En esta sección se muestra una descripción detallada de los pasos necesarios para llevar a cabo el proceso de *testing*, cuyo objetivo es calcular la idoneidad de un conjunto de tests para testear una aplicación ejecutada en un entorno distribuido. La Figura 4.1 presenta el diseño de la arquitectura propuesta para implementar MuTomVo.

En este esquema, se puede observar cómo se integran las herramientas de modelado y simulación de sistemas distribuidos con las técnicas de *mutation testing*, así como los pasos necesarios para poder realizar el proceso completo de *testing*.

Inicialmente, el usuario debe construir un modelo utilizando la *GUI* (*Graphical User Interface*) proporcionada por el simulador SIMCAN (ver Apéndice 2). Este modelo consiste básicamente en la configuración del sistema distribuido que se desea modelar, incluyendo tanto las máquinas físicas como las conexiones de la red de comunicaciones y la aplicación

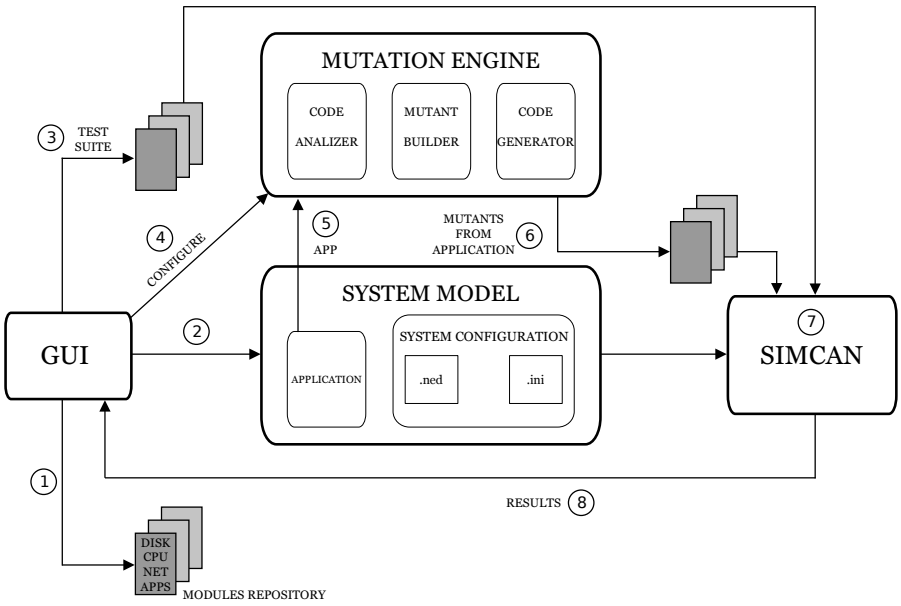

Figura 4.1: *Arquitectura básica del framework MuTomVo*

que se desea testear. Para realizar este proceso, el usuario cuenta con una serie de módulos predefinidos en el repositorio de SIMCAN ①. Estos módulos, a los que se puede acceder a través de la *GUI*, simulan el comportamiento de los dispositivos que forman el sistema distribuido, tales como CPUs, discos, memorias y redes de comunicaciones.

Una vez finalizado el diseño del modelo, MuTomVo genera los ficheros de configuración oportunos para que SIMCAN pueda simular el entorno diseñado por el usuario ②.

Con el modelo del sistema generado se debe proporcionar un conjunto de tests ③. Estos tests pueden ser, o bien generados manualmente por el usuario, o bien generados automáticamente por el propio *framework*.

El siguiente paso consiste en realizar la configuración del módulo *mutation engine* ④. Para ello, el usuario, a través del módulo *GUI*, debe introducir los parámetros correspondientes para configurar el proceso de mutación, tales como el número máximo de mutantes generados y los operadores de mutación que se utilizarán para generar estos mutantes.

Una vez realizada esta configuración, el módulo *mutation engine* toma como entrada la aplicación seleccionada por el usuario en el paso ② para comenzar el proceso de la generación de mutantes ⑤. Una vez finalizado este proceso, se generan los mutantes oportunos ⑥. Este proceso se describe en detalle en la sección 4.2.

En este punto, el *framework* ya contiene el modelo del sistema donde se van a ejecutar los tests, la aplicación y los mutantes generados a partir de esta aplicación. El modelo del sistema lo recibe como entrada el simulador SIMCAN ⑦ para construir, tanto la arquitectura del mismo, como la topología de red previamente configurada por el usuario en el paso ①.

Seguidamente, el conjunto de tests se ejecuta, tanto sobre la aplicación original, como sobre los mutantes generados. Una vez ejecutados todos los tests, se reportan los resultados obtenidos al módulo *GUI* para ser presentados al usuario ⑧.

4.2. Descripción de los operadores de mutación

Durante la fase de desarrollo de una aplicación, los errores pueden cometerse de diversas formas. Teniendo en cuenta la *competent programming hypothesis*, los errores cometidos por el programador van a consistir en un conjunto de pequeños errores, que alteran la semántica del programa provocando un comportamiento no esperado. Estos errores pueden clasificarse dentro de distintas categorías, de modo que los operadores se agrupan de acuerdo a las mismas, tal como se puede observar en el Cuadro 4.1.

Debido a que la plataforma de simulación OMNeT++ está diseñada para el desarrollo de aplicaciones en el lenguaje C++, una de las categorías propuestas es el uso de operadores generales de dicho lenguaje de programación, tales como operadores aritméticos, relacionales, condicionales, etc. El segundo grupo se centra en operadores relacionados con las llamadas proporcionadas por OMNeT++. Finalmente, los grupos SIMCAN y MPI son operadores destinados a las llamadas del sistema proporcionadas por SIMCAN y el interfaz de intercambio de mensajes utilizado para el desarrollo de aplicaciones paralelas.

La inclusión de este tipo de operadores en MuTomVo no pretende aportar una inno-

Operadores de mutación			Descripción
Generales	Aritméticos	AOR	Arithmetic Operator Replacement
		AOD	Arithmetic Operator Deletion
	Relacionales	ROR	Relational Operator Replacement
	Condicionales	COR	Conditional Operator Replacement
		COD	Conditional Operator Deletion
	Lógicos	LOR	Logical Operator Replacement
		LOD	Logical Operator Deletion
	Asignación	ASR	Assignment Operator Replacement
OMNeT++		OOD	OMNET Operator Deletion
		OOMU	OMNET Operator Movement Up
		OOMD	OMNET Operator Movement Down
		OOR	OMNET Operator Replacement
		OOSP	OMNET Operator Shuffle Parameters
SIMCAN		SOD	SIMCAN Operator Deletion
		SOMU	SIMCAN Operator Movement Up
		SOMD	SIMCAN Operator Movement Down
		SOR	SIMCAN Operator Replacement
		SOSP	SIMCAN Operator Shuffle Parameters
MPI		MOD	MPI Operator Deletion
		MOMU	MPI Operator Movement Up
		MOMD	MPI Operator Movement Down
		MOR	MPI Operator Replacement
		MOSP	MPI Operator Shuffle Parameters

Cuadro 4.1: *Tabla con los operadores de mutación de MuTomVo*

vación en el campo de *mutation testing*, desde el punto de vista de su implementación en una plataforma de simulación. Actualmente existen numerosos trabajos y herramientas de mutación de código que implementan estos operadores. El motivo por el cual se han incluido e implementado estos operadores en MuTomVo, es evitar el uso de varias herramientas, pudiendo así realizar completamente el proceso de *testing* utilizando únicamente el *framework* propuesto. Además, esto permitirá comparar, desde el punto de vista de rendimiento y eficacia, los resultados obtenidos al utilizar tanto los operadores generales como los nuevos operadores propuestos, de forma cómoda y sencilla. De esta forma se evita el uso de herramientas adicionales.

4.2.1. Operadores de mutación generales

La categoría general engloba la introducción de errores propios del uso lenguaje de programación C++. Estos errores son introducidos por el programador al escribir sentencias donde se realizan operaciones básicas. Una de ellas es el cálculo de operaciones aritméticas. Para ello se define un conjunto de operadores, clasificados en dos categorías, binarios *op + op, op - op, op * op, op / op y op % op*, que representan operaciones aritméticas básicas, y unarios *-op* y *+op*, que representan el signo de los operadores. A su vez, se definen cuatro operadores aritméticos característicos de este lenguaje *op++, ++op, op−−* y *−−op*, denominados *short-cut*.

Con estos operadores, pueden realizarse distintas mutaciones que representan las diferentes categorías de fallos que pueden cometer los programadores a la hora de realizar operaciones aritméticas. En los ejemplos que se muestran a continuación, se ha utilizado la siguiente notación: OP representa un operador de mutación, OP_b indica operador de mutación binario, OP_s indica operador de mutación *short-cut* y OP_u operador de mutación unario.

- AOR_b : Arithmetic Operator Replacement. Con este operador se produce un reemplazo de un operador aritmético binario (ver Listado 4.1), por un operador aritmético binario diferente (ver Listado 4.2).

```
1 startDelay = par ("startDelay");
2 inSize  = (int) par ("inSize") * MB;
```

Listado 4.1: *Código original*

```
1 startDelay = par ("startDelay");
2 inSize  = (int) par ("inSize") / MB; ◁
```

Listado 4.2: *Mutante AOR_b*

- AOR_u : Arithmetic Operator Replacement. A través de este operador se produce el reemplazo, en el código fuente, de un operador aritmético unario (ver Listado 4.3) por otro operador unario diferente (ver Listado 4.4).

```
1 writeOffset = -(outputDataSize);
```

Listado 4.3: *Código original*

```
1 writeOffset = +(outputDataSize); ◁
```

Listado 4.4: *Mutante AOR_u*

- AOR_s : Arithmetic Operator Replacement. Con este operador se produce el reemplazo

en el código de un operador aritmético *short-cut* (ver Listado 4.5), por un operador aritmético *short-cut* diferente (ver Listado 4.6).

```
1      if (sm->getResult() == SIMCAN_OK){
2          executeCPU = false;
3          executeWrite = false;
4          executeRead = true;
5          currentIteration++;
6          delete (sm);
7      }
```

Listado 4.5: *Código original*

```
1      if (sm->getResult() == SIMCAN_OK){
2          executeCPU = false;
3          executeWrite = false;
4          executeRead = true;
5          currentIteration--; ◁
6          delete (sm);
7      }
```

Listado 4.6: *Mutante* AOR_s

- AOD_u : Arithmetic Operator Deletion. Mediante este operador se produce la eliminación del código fuente de un operador aritmético unario básico (ver Listado 4.8).

```
1
2  writeOffset = -(outputDataSize);
```

Listado 4.7: *Código original*

```
1
2  writeOffset = (outputDataSize); ◁
```

Listado 4.8: *Mutante* AOD_u

- AOD_s : Arithmetic Operator Deletion. Por medio de este operador se produce la eliminación del código fuente de un operador aritmético *short-cut* (ver Listado 4.10).

```
1  if (sm->getResult() == SIMCAN_OK){
2      executeCPU = false;
3      executeWrite = false;
4      executeRead = true;
5      currentIteration++;
6      delete (sm);
7  }
```

Listado 4.9: *Código original*

```
1  if (sm->getResult() == SIMCAN_OK){
2      executeCPU = false;
3      executeWrite = false;
4      executeRead = true;
5      currentIteration;    ◁
6      delete (sm);
7  }
```

Listado 4.10: *Mutante* AOD_s

La categoría de operadores de mutación relacional, agrupa los fallos cometidos por el programador a la hora de realizar operaciones de comparación entre operandos. Entre ellas, las proporcionadas por el lenguaje de programación C: *op >op, op >= op, op <op, op <= op, op == op y op != op.*

- ROR : Relational Operator Replacement. A través de este operador se sustituye un operador relacional (ver Listado 4.11) por otro operador relacional diferente (ver Listado 4.12).

```
if (sm_io->getResult() == SIMCAN_OK){
    executeCPU = false;
    executeWrite = false;
    executeRead = true;
    currentIteration++;
    delete (sm);
}
```

Listado 4.11: *Código original*

```
if (sm_io->getResult() != SIMCAN_OK){ ◁
    executeCPU = false;
    executeWrite = false;
    executeRead = true;
    currentIteration++;
    delete (sm);
}
```

Listado 4.12: *Mutante ROR*

Con el objetivo de proporcionar mecanismos para realizar operaciones a nivel de bit entre dos operandos, se encuentra la categoría de operadores lógicos. Estos operadores pueden clasificarse tanto en binarios, tales como *op* & *op, op* | *op y op* ˆ *op*, que representan operaciones relaciones, como en unarios ∼*op*.

- LOR_b: Logical Operator Replacement. Realiza un reemplazo de operadores del tipo binario (ver Listado 4.13) por otro operador del mismo tipo (ver Listado 4.14).

```
if ((res & opMask) == SIMCAN_OK){
    executeCPU = true;
}
```

Listado 4.13: *Código original*

```
if ((res | opMask) == SIMCAN_OK){    ◁
    executeCPU = true;
}
```

Listado 4.14: *Mutante* LOR_b

- LOD_u: Logical Operator Delete. Mediante este operador se produce la eliminación del código fuente de un operador condicional unario (ver Listado 4.16).

```
result = ~runningThreads;
if(result == SIMCAN_OK){
    finishedExecution = true;
}
```

Listado 4.15: *Código original*

```
result = runningThreads;          ◁
if(result == SIMCAN_OK){
    finishedExecution = true;
}
```

Listado 4.16: *Mutante* LOR_u

La categoría condicional reúne aquellos operadores utilizados para realizar operaciones condicionales entre operandos. El lenguaje C proporciona cinco operadores binarios *op* && *op, op* || *op, op* & *op* | *op y op* ˆ *op*, y uno unario, !.

- COR_b : Conditional Operator Replacement. Mediante este operador se reemplaza un operador condicional binario (ver Listado 4.17) por otro operador condicional (ver Listado 4.18).

```
if (sm->getRes() || !sm->getFail()){
    executeCPU = false;
    executeWrite = false;
    executeRead = true;
    currentIteration++;
    delete (sm);
}
```

Listado 4.17: *Código original*

```
if (sm->getRes() && !sm->getFail()){ ◁
    executeCPU = false;
    executeWrite = false;
    executeRead = true;
    currentIteration++;
    delete (sm);
}
```

Listado 4.18: *Mutante* COR_b

- COD_u: Conditional Operator Deletion. Por medio de este operador se produce la eliminación del código fuente de un operador condicional unario (ver Listado 4.20).

```
if (sm->getRes() || !sm->getFail()){
    executeCPU = false;
}
```

Listado 4.19: *Código original*

```
if (sm->getRes() || sm->getFail()){◁
    executeCPU = false;
}
```

Listado 4.20: *Mutante* COD_u

Los operadores de asignación establecen un valor a una variable. Además de los operadores básicos de asignación, en C se proporcionan once operadores *short-cut*. Estos operadores realizan una operación sobre la parte derecha de la expresión y asignan el resultado a la parte izquierda. La operación realizada viene determinada por el uso de alguno de los siguientes operadores, *op += op, op -= op, op *= op, op /= op, op %= op, op & = op, op |= op, op ˆ op, op <= op, op>>= op* y *op >>>= op*.

- ASR_s: Short-Cut Assignment Operator Replacement. Realiza el reemplazo de un operador del tipo *short-cut* (ver Listado 4.21) por otro operador del mismo tipo (ver Listado 4.22).

```
if (executeCPU == true){
    currentIteration+=1;
}
```

Listado 4.21: *Código original*

```
if (executeCPU == true){
    currentIteration-=1;    ◁
}
```

Listado 4.22: *Mutante* ASR_s

4.2.2. Operadores OMNET

Con el propósito de representar los errores cometidos por el programador a través del uso de la plataforma de simulación OMNeT++, se propone un conjunto de operadores de mutación que representan los fallos más comunes cometidos en este entorno. La diferencia

más notable reside en que en lugar de introducir cambios en el uso del lenguaje de programación C, el conjunto de operadores propuestos está orientado a la modificación de las funciones proporcionadas por OMNeT++, las cuales se detallan en la sección 3.1. Para ello, se proponen los siguientes operadores de mutación:

- OOR: OMNET Operator Replacement. Con este operador se produce el reemplazo en el código de una llamada del API de OMNET (ver Listado 4.23), por otra diferente (ver Listado 4.24).

```
1 sm->removeLastModuleFromTrace ();
2 send (sm, gateId);  ◁
3 modulesSent++;
```

Listado 4.23: *Código original*

```
sm->removeLastModuleFromTrace ();
scheduleAt (gateId,sm); ◁
modulesSent++;
```

Listado 4.24: *Mutante OOR*

- OOD: OMNET Operator Deletion. Mediante este operador se produce la eliminación del código fuente de una llamada del API de OMNET (ver Listado 4.26).

```
1 sm->removeLastModuleFromTrace ();
2 send (sm, gateId);
3 if(updateSentMessages()>=MAX_MSG)
4 {
5     endSimulation();
6 }
```

Listado 4.25: *Código original*

```
sm->removeLastModuleFromTrace ();
if(updateSentMessages()>=MAX_MSG) ◁
{
    endSimulation();
}
```

Listado 4.26: *Mutante OOD*

- OOMU: OMNET Operator Movement Up. Se realiza un cambio de posición de una sentencia, a la inmediatamente anterior a la actual (ver Listado 4.28).

```
1 sm->removeLastModuleFromTrace ();
2 send (sm, gateId);
3 if(updateSentMessages()>=MAX_MSG)
4 {
5     endSimulation();
6 }
```

Listado 4.27: *Código original*

```
send (sm, gateId); ◁
sm->removeLastModuleFromTrace ();
if(updateSentMessages()>=MAX_MSG)
{
    endSimulation();
}
```

Listado 4.28: *Mutante OOMU*

- OOMD: OMNET Operator Movement Down. Se realiza un cambio de posición de una sentencia, a la inmediatamente posterior a la actual (ver Listado 4.30).

```
sm->removeLastModuleFromTrace ();
send (sm, gateId);
if(updateSentMessages()>=MAX_MSG)
{
    endSimulation();
}
```

Listado 4.29: *Código original*

```
sm->removeLastModuleFromTrace ();
if(updateSentMessages()>=MAX_MSG)
{
    send (sm, gateId); ◁
    endSimulation();
}
```

Listado 4.30: *Mutante OOMD*

4.2.3. Operadores SIMCAN y MPI

Estos operadores tienen el objetivo de representar los errores cometidos por el programador a través del uso de la plataforma SIMCAN. Para ello, al igual que en el caso de los operadores OMNET, el conjunto de operadores propuestos está orientado a la modificación de las llamadas proporcionadas por el API de SIMCAN, descritas en la sección 3.2.2. De igual forma, SIMCAN proporciona un subconjunto de llamadas MPI para desarrollar aplicaciones paralelas en arquitecturas distribuidas, detalladas en la sección 3.2.3. Por ello, con el objetivo de proporcionar una descripción más clara, ambos operadores son agrupados en esta sección, los cuales están representados a través de modificaciones de las llamadas a sus respectivos APIs. Por ello, tanto los grupos SIMCAN como MPI cuentan con los siguientes operadores:

- OR: Operator Replacement. Con este operador se produce el reemplazo en el código de una llamada del API (ver Listado 4.31) por otra diferente (ver Listado 4.32).

```
// Reading data for each worker process
SIMCAN_request_write (outName, offWrite,
    slice_KB*KB);
```

Listado 4.31: *Código original*

```
// Reading data for each worker process
SIMCAN_request_read (outName, offWrite,
    slice_KB*KB); ◁
```

Listado 4.32: *Mutante SOR*

- OD: Operator Deletion. Mediante este operador se produce la eliminación del código fuente de una llamada del API (ver Listado 4.34).

```
// Send data to each process
dataSize = 512000;
mpi_send (getMyMaster(myRank), dataSize);
```

Listado 4.33: *Código original*

```
// Send data to each process
dataSize = 512000;◁
```

Listado 4.34: *Mutante MOD*

- OMU: Operator Movement Up. Se realiza un cambio de posición de una sentencia, a la inmediatamente anterior a la actual (ver Listado 4.36).

```
1 result=SIMCAN_request_create(strPath);
2 if(result=SIMCAN_OK)
3     SIMCAN_request_read(strPath);
4 SIMCAN_request_close(strPath);
```

Listado 4.35: *Código original*

```
result=SIMCAN_request_create(strPath);
SIMCAN_request_read(strPath);    ◁
if(result=SIMCAN_OK)
    SIMCAN_request_close(strPath);
```

Listado 4.36: *Mutante SOMU*

- OMD: Operator Movement Down. Se realiza un cambio de posición de una sentencia, a la inmediatamente posterior a la actual (ver Listado 4.38).

```
1 result=SIMCAN_request_create(strPath);
2 if(result=SIMCAN_OK)
3     SIMCAN_request_read(strPath);
4 SIMCAN_request_close(strPath);
```

Listado 4.37: *Código original*

```
result=SIMCAN_request_create(strPath);
if(result=SIMCAN_OK)
    SIMCAN_request_close(strPath);
SIMCAN_request_read(strPath);    ◁
```

Listado 4.38: *Mutante OOMD*

- OSP: Operator Shuffle Parameters. A través de este operador, se produce un intercambio entre los parámetros de una llamada del API (ver Listado 4.40).

```
1 mpi_send (i, dataSize);
```

Listado 4.39: *Código original*

```
mpi_send (dataSize, i); ◁
```

Listado 4.40: *Mutante MOSP*

4.3. Proceso de mutación

En esta sección se describe el proceso completo de mutación. Para ello, se ha realizado una descripción tanto de los componentes de la arquitectura de MuTomVo como de los pasos realizados desde el momento en el cual el usuario proporciona un modelo del sistema, hasta que obtiene los resultados del proceso.

4.3.1. Proceso de generación de mutantes

El módulo encargado de orquestar la generación de mutantes, descrito en la Figura 4.2, es conocido como *mutation engine*. Inicialmente, este módulo recibe como entrada el código fuente de una aplicación, que es enviada al módulo *code analyzer*.

A través de un análisis sintáctico del código fuente, el módulo *code analyzer* se encarga de localizar los puntos del código a los que se puede aplicar alguno de los operadores de

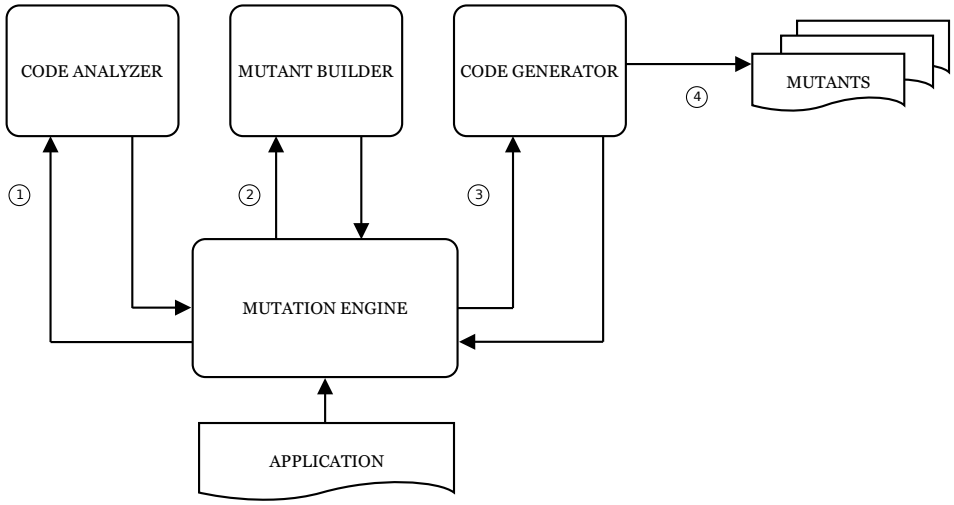

Figura 4.2: *Arquitectura de mutation engine*

mutación que aparecen en el Cuadro 4.1. Una vez han sido localizados, se almacena su posición y tipo de operador en una estructura de datos, que es enviada al módulo *mutant builder*.

Haciendo uso de la información proporcionada por el módulo de análisis, el módulo *mutant builder* se encarga de generar una colección de objetos de mutación, que consiste en un conjunto de objetos que representan todos los mutantes resultantes de aplicar operadores de mutación sobre el código fuente. Estos objetos contienen un conjunto de parámetros que son necesarios en fases posteriores para generar cada uno de los mutantes. Los parámetros asociados a cada objeto representan el tipo de operador, identificador, posición y el operador de mutación a aplicar.

Por último, el módulo *code generator* se encarga de insertar en el código fuente original las modificaciones representadas en cada uno de los objetos proporcionados por el módulo *mutant builder*. Cada mutante es generado introduciendo un único cambio en el código fuente original, es decir, sólo se generan mutantes de primer orden. Una vez concluida la generación de mutantes, éstos son almacenados y asociados a identificadores únicos. Estos identificadores son utilizados durante las siguientes fases del proceso.

4.3.2. Ciclo completo del proceso de *testing*

El funcionamiento del proceso de generación y ejecución de mutantes está ilustrado en la Figura 4.3. Inicialmente, el proceso comienza cuando el usuario proporciona el modelo, la aplicación original y un conjunto de tests para comprobar su correcto funcionamiento. Para ello, debido a que la plataforma de simulación OMNeT++ está escrita en lenguaje C++, la aplicación es compilada utilizando el compilador GCC.

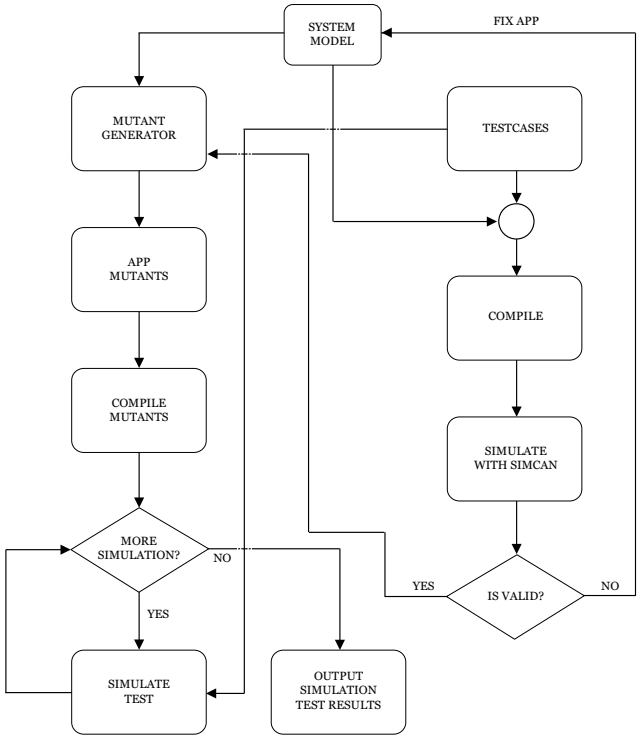

Figura 4.3: *Ciclo completo del proceso de testing*

Al terminar la fase de compilación, cada test es ejecutado sobre el programa original a través de su simulación en SIMCAN, comprobando que las salidas generadas son correctas. En el caso de que la salida sea incorrecta, se ha detectado un error en el programa original y debe ser corregido antes de continuar con el proceso. Si la salida es correcta se procede a la fase de generación de mutantes, detallado en la sección 4.3.1.

Una vez generado el conjunto de mutantes de la aplicación, se procede a la compilación

de cada uno de ellos. En el siguiente paso, cada test es ejecutado sobre cada mutante de la aplicación a través de SIMCAN. Seguidamente, se comprueba que cada una de las salidas obtenidas es correcta, aplicando los criterios detallados en la Sección 4.4. En el momento de detectar una salida incorrecta, no se ejecutan más tests sobre el mutante que se está procesando. En caso contrario, se continúan ejecutando tests sobre dicho mutante.

Una vez se hayan realizado todas las simulaciones, los resultados del proceso de mutación son mostrados al usuario a través de la *GUI* del sistema.

4.3.3. Diseño de la estructura de MuTomVo

Debido a la continua aparición de nuevas contribuciones en el campo de *mutation testing*, es necesario contar con una arquitectura flexible que permita incluir nuevos operadores y técnicas de optimización.

Por ello, uno de los objetivos del trabajo es construir MuTomVo utilizando una arquitectura modular y flexible, de forma que permita introducir nuevas técnicas de forma sencilla. Esto reduce los tiempos de integración, aumentando la viabilidad de contar con un número elevado de técnicas en el sistema.

La Figura 4.2 muestra los cuatro módulos para generar mutantes, *mutation engine, code analyzer, mutant builder* y *code generator*.

En primer lugar, *mutation engine* proporciona un alto grado de flexibilidad, ya que es el encargado de la comunicación entre el resto de módulos. A su vez, cuenta con mecanismos que permiten el intercambio de cualquiera de los módulos sin necesidad de realizar cambios en la estructura general.

El módulo *code analyzer* está formado por el conjunto de clases representado en la Figura 4.4. Este módulo analiza sintácticamente el código para determinar las localizaciones donde es posible aplicar los operadores de mutación. A través de la clase *StateMachineIndexer*, se añade la funcionalidad de realizar búsquedas a partir de una cadena de texto. En el caso de que se necesite introducir un nuevo analizador sintáctico, únicamente se debe imple-

mentar la nueva funcionalidad en base al interfaz *IMutatorIndexer*. Este interfaz encapsula el módulo *code analyzer* del resto del proceso de mutación, permitiendo intercambiar distintos analizadores sin afectar al resto del programa.

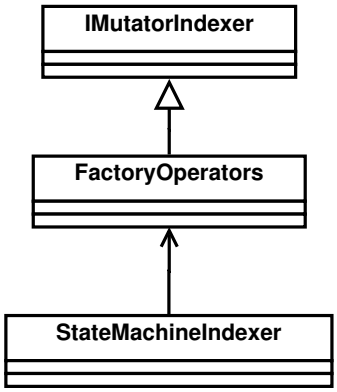

Figura 4.4: *Módulo de análisis de código*

La generación de mutantes es realizada por el módulo *mutant builder*, formado por el conjunto de clases ilustrado en la Figura 4.5. La clase principal es *Mutator*, que permite la generación de mutantes a través del uso de distintas técnicas. En este caso *MutantBuilder* implementa el esquema tradicional de *mutation testing*. Para poder proporcionar técnicas nuevas, éstas deberían ser introducidas en extensión a la clase *Mutator*, tal como se ilustra en la clase *HOMBuilder*, la cual sirve como ejemplo de nueva integración. A su vez, *MutantBuilder* proporciona un conjunto de funcionalidades utilizada por los constructores de operadores de mutación propuestos en el Cuadro 4.1. Cada uno de ellos proporciona un conjunto de operaciones determinadas por cada uno de sus interfaces. Entre ellos, cabe destacar la existencia del elemento *TextBasedBuilder*, con el que se realiza la generación de mutantes basados en operadores OMNET, SIMCAN y MPI.

Por último, el encargado de generar y almacenar el código fuente resultante del proceso de mutación es el módulo *code generator*. Su estructura se presenta con un diagrama de clases en la Figura 4.6. *Code generator* cuenta con una clase base en la que están implementadas las operaciones básicas, tales como la exportación del código fuente a disco o numeración de mutantes, extendida con generadores de código específicos. En este caso se cuenta con dos

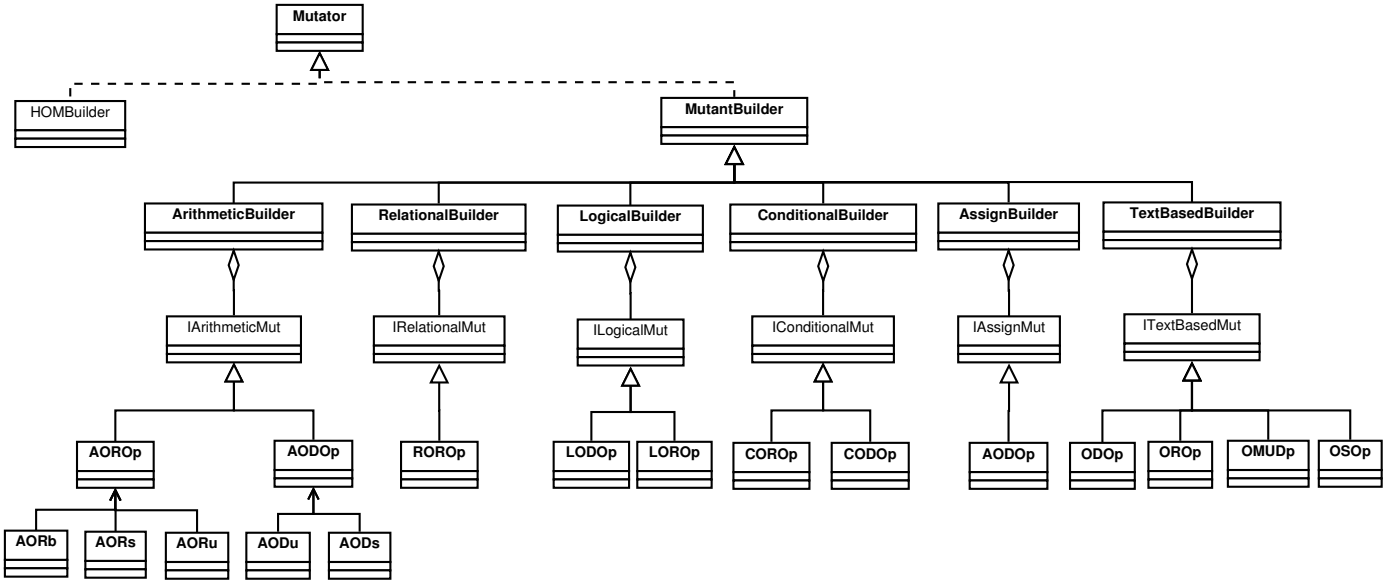

Figura 4.5: *Módulo de construccion de mutantes*

generadores de código, en primer lugar *StdCodeGenerator*, encargado de generar el código fuente correspondiente a los operadores generales y *TextBasedGenerator* responsable de la generación de código de los mutantes cuyas definiciones están basadas en cadenas de texto.

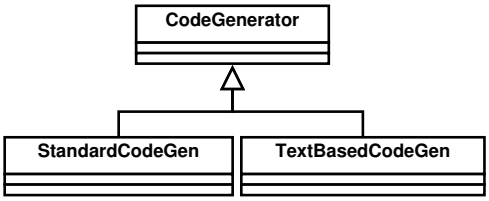

Figura 4.6: *Módulo de generación de código*

4.4. Comprobación de la validez de los mutantes

Una etapa fundamental en el ciclo de *mutation testing* es la comprobación de la validez de los resultados obtenidos mediante la ejecución de los mutantes. Por ello, es necesario proporcionar algún tipo de mecanismo que permita comprobar cuando un mutante es matado o no por un test.

Para ello, tal como se describe en la sección 2.2, se debe tener en cuenta si el resultado

de la ejecución de un test, sobre un mutante, difiere del resultado de la ejecución del mismo test sobre la aplicación original. En ese caso se dice que *se mata* a este mutante. En cambio, si todos los resultados de los tests, aplicados tanto a la aplicación original y al mutante coinciden, se dice que el mutante está vivo.

Cuando termina la ejecución del conjunto de tests, se debe comprobar si se ha *matado* a todos los mutantes generados. En caso contrario, el usuario debe elegir entre proporcionar un número adicional de tests con el objetivo de matar dichos mutantes, o terminar con el proceso. Existen ocasiones en las que incluir iterativamente nuevos tests no alcanza el objetivo de *matar* a todos los mutantes. Esta situación sucede cuando la aplicación original y alguno de sus mutantes son equivalentes.

Una de las técnicas existentes para la comparación de resultados es conocida como *strong mutation*. En ella, se realiza la ejecución tanto del programa original como de los mutantes y se produce la comparación entre los resultados finales de cada ejecución. Este comportamiento se ve reflejado en el Listado 4.41, donde se aplica un operador de mutación aritmético que modifica la semántica del programa original. Este cambio provoca que el resultado generado por el mutante (ver Listado 4.42) sea distinto al resultado obtenido por la aplicación original.

```
1
2  void ApplicationHPC::sendResults()
3  {
4  unsigned int dataSize;
5  // Master Process WRITE!
6  if (!workersWrite)
7      dataSize = sliceT_KB*KB;
8  else
9      dataSize = METADATA_MSG_SIZE;
10
11 // Send data to each process
12 mpi_send (getMyMaster(myRank), dataSize);
       calculateNextState ();
13 }
14 void ApplicationHPC::finish()
15 {
16 showResultMessage ("App [%s] - Rank:%d --
       IO:%f NET:%f CPU:%f",
17 moduleIdName.c_str(), myRank, totalIO.dbl
       (), totalNET.dbl(), totalCPU.dbl());
18
19 // Finish the super-class
20 AppMPI_Base::finish();
21 }
```

Listado 4.41: *Código original*

```
1
2  void ApplicationHPC::sendResults()
3  {
4  unsigned int dataSize;
5  // Master Process WRITE!
6  if (!workersWrite)
7      dataSize = slice_KB+KB; ◁
8  else
9      dataSize = METADATA_MSG_SIZE;
10
11 // Send data to each process
12 mpi_send (getMyMaster(myRank), dataSize);
13 calculateNextState ();
14 }
15 void ApplicationHPC::finish()
16 {
17 showResultMessage ("App [%s] - Rank:%d --
       IO:%f NET:%f CPU:%f",
18 moduleIdName.c_str(), myRank, totalIO.dbl
       (), totalNET.dbl(), totalCPU.dbl());
19
20 // Finish the super-class
21 AppMPI_Base::finish();
22 }
```

Listado 4.42: *Mutante* AOR_b

Al concluir el proceso se obtiene el *mutation score*, que se corresponde con la proporción de mutantes de la aplicación matados sobre el número de mutantes generados.

$$MS = \frac{mutantesMatados}{totalMutantes}$$

Capítulo 5

Experimentos

En este capítulo se describen los experimentos llevados a cabo con el framework MuTomVo. Estos experimentos consisten en analizar la idoneidad de diferentes conjuntos de tests ejecutados sobre diferentes aplicaciones distribuidas. Cada una de estas aplicaciones, junto con las arquitecturas donde se han ejecutado, han sido modeladas con SIMCAN. De esta forma, los experimentos se han llevado a cabo utilizando las técnicas de *mutation testing* anteriormente descritas sobre estos modelos.

5.1. Generación de los conjuntos de tests

El objetivo principal de los experimentos presentados en este capítulo consiste en calcular la idoneidad de varios conjuntos de tests ejecutados sobre aplicaciones distribuidas. Los tests generados se ejecutarán sobre una aplicación modelada en la plataforma de simulación SIMCAN. En SIMCAN, cada aplicación contiene un conjunto de parámetros que deben ser configurados antes de poder ejecutarse. Estos parámetros se definen mediante un par <nombre, valor>. De esta forma, definimos un test t como un conjunto de parámetros tal que:

$$t = \{< param_1, value >, < param_2, value >, \ldots, < param_n, value >\}$$

Por ejemplo, supongamos una aplicación con 2 parámetros de entrada, llamados *iterations* y *processingTime*, ambos de tipo entero, donde *iterations* indica el número de itera-

ciones que debe realizar la aplicación y *processingTime* el tiempo de procesamiento de cada iteración, medido en segundos. Un posible test para esta aplicación que ejecute 2 iteraciones, donde cada una de ellas simula el procesamiento de 1000 segundos de CPU, se representa de la siguiente forma:

$$t = \{< iterations, 2 >, < processingTime, 1000 >\}$$

Puesto que el usuario hace uso de la GUI para poder configurar la ejecución de cada test, este proceso se realiza de forma sencilla. Sin embargo, proporcionar manualmente un conjunto de tests que pruebe de forma exhaustiva una aplicación es un proceso muy costoso y propenso a generar valores erróneos. Para aliviar estos inconvenientes, se han desarrollado dos métodos para generar tests automáticamente a partir de unos parámetros de configuración.

El primer método utiliza el Algoritmo 1 para generar los tests (ver Apéndice I). Este algoritmo es configurable, de forma que el usuario puede definir el número de tests que serán generados y los rangos utilizados para asignar los valores a cada parámetro de estos tests. Esta configuración debe ser proporcionada por el usuario a través de la GUI de MuTomVo, la cual genera una lista de objetos de tipo *ParamValues* (ver Listado 5.1), que recibe como entrada el Algoritmo 1. Básicamente, esta configuración consiste en asignar, para cada parámetro, el valor máximo, el valor mínimo y el intervalo entre cada par de valores. El segundo método consiste en generar los tests de forma aleatoria.

```
1 public class ParamValues{
2     private String name;              // Parameter name
3     private String type;              // Parameter type (int, float, boolean, ...)
4     private String minValue;          // Minimum value assigned to this parameter
5     private String maxValue;          // Maximum value assigned to this parameter
6     private String interval;          // Interval between two values of this parameter
7     private List values;              // List of values
8     . . .
9 }
```

Listado 5.1: *Clase que representa los valores asignados a cada parámetro en los tests*

5.2. Análisis de la idoneidad de los conjuntos de tests

Los experimentos realizados en este trabajo se han llevado a cabo utilizando dos aplicaciones diferentes. La primera aplicación, llamada *appCPU*, realiza operaciones de cómputo intensivo sobre un conjunto de datos. Inicialmente, el conjunto de datos está almacenado en el disco de un servidor remoto. La aplicación carga este conjunto de datos en memoria para posteriormente realizar el cálculo correspondiente. Una vez procesado el conjunto de datos, el resultado se escribe en el disco del servidor.

Los parámetros que se utilizan para configurar esta aplicación se detallan en el Listado 5.2. El parámetro *inputDataSize* indica el tamaño del conjunto de datos a procesar, *outputDataSize* indica el tamaño del fichero obtenido como resultado de procesar el conjunto de datos, *MIs* indica el cómputo realizado medido en millones de instrucciones e *iterations* indica el número de conjuntos de datos que se procesan durante la ejecución de la aplicación. La Figura 5.1 muestra el sistema donde se ejecuta esta aplicación.

```
1 int inputDataSize;          // Size of data-set
2 int outputDataSize;         // Size of results
3 int MIs;                    // Computing for each iteration
4 int iterations;             // Number of iterations
```

Listado 5.2: *Parámetros de configuración de la aplicación appCPU*

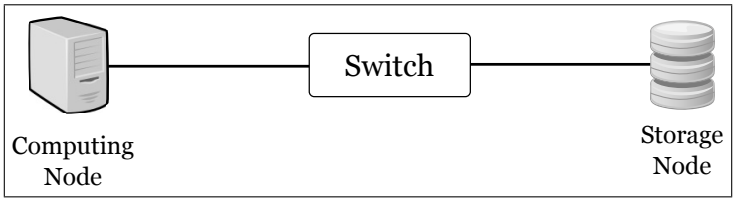

Figura 5.1: *Esquema de ejecución de la aplicación appCPU en SIMCAN*

La segunda aplicación, llamada *appMR*, consiste en una versión simplificada del modelo Map-Reduce[21] propuesto por Google. Esta aplicación tiene como objetivo el procesamiento paralelo de un conjunto de datos cuyo tamaño se configura previamente a su ejecución. En esta aplicación intervienen dos tipos de procesos: coordinadores (*coordinator*) y trabajadores (*workers*). Estos procesos están agrupados en *frames*, de forma que cada *frame* contiene

un proceso coordinador y un conjunto de procesos trabajadores. La Figura 5.2 muestra el esquema de ejecución de esta aplicación.

La ejecución de la aplicación empieza leyendo el conjunto de datos de los nodos de almacenamiento (*storage nodes*) ①. Los procesos coordinadores llevan a cabo esta tarea. Seguidamente, cada proceso coordinador envía una porción del conjunto de datos, denominado dominio, a cada proceso trabajador ②. Cuando el dominio es recibido por un proceso trabajador, éste empieza a procesarlo ③. Una vez finalizado el procesamiento del dominio, los resultados obtenidos son enviados al proceso coordinador correspondiente ④. Finalmente, cuando el proceso coordinador recibe los resultados de todos los procesos trabajadores que tiene asociados, los escribe en disco ⑤, de forma que éste procesa un nuevo dominio comenzando en el paso ①. Así, los procesos coordinadores reparten dominios a los procesos trabajadores hasta que el conjunto de datos se ha procesado completamente.

El entorno simulado donde se ha ejecutado la aplicación *appMR* está formado por 64 nodos de cómputo y 8 nodos de almacenamiento. Cada nodo tiene un procesador Dual-Core Intel(R) Xeon(R) CPU, 4 GB de RAM, un interfaz de red Ethernet Gigabit y una unidad de disco de 500GB. La topología del entorno es TOR (*Top Of Rack*).

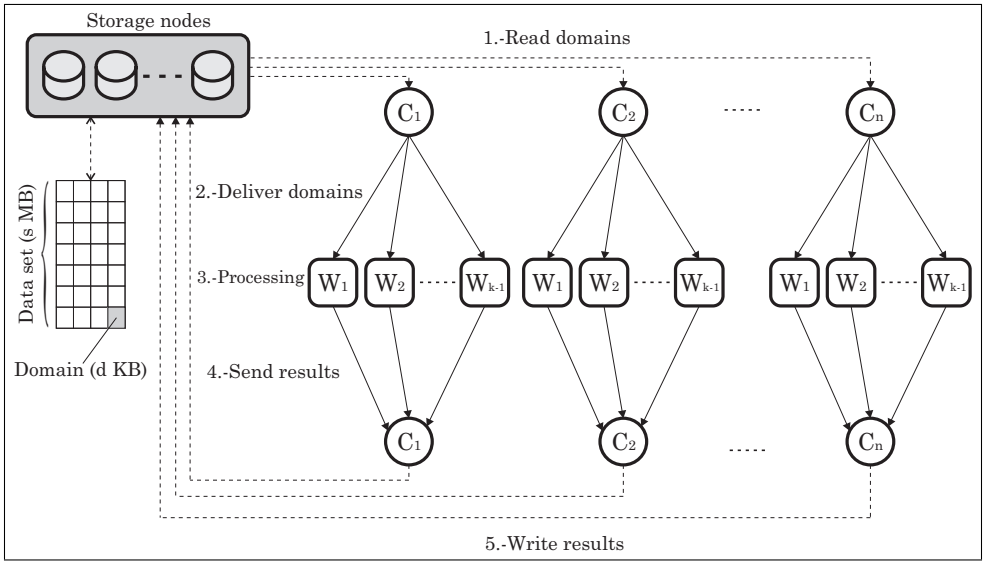

Figura 5.2: *Esquema de ejecución de la aplicación appMR en SIMCAN*

Los parámetros utilizados para configurar esta aplicación se muestran en el Listado 5.3, donde *workersSet* indica el número de procesos de un *frame*, *sliceToWorkers* es el tamaño de cada dominio medido en Kbytes, *sliceToMaster* es el tamaño de los resultados obtenidos por los procesos trabajadores medido en Kbytes, *sliceCPU* es el cómputo realizado por los procesos trabajadores medido en MIs, *workersWrite* indica si los procesos trabajadores escriben los resultados directamente en disco en lugar de enviarlos a su proceso coordinador y *numIterations* representa el número de iteraciones realizadas.

```
1 int workersSet;          // Number of worker processes per frame
2 double sliceToWorkers;   // Slice of data (in Kbytes) sent to each worker process
3 double sliceToMaster;    // Slice of data (in Kbytes) received for master process
4 int sliceCPU;            // CPU processing (in MIPS) for each worker process
5 bool workersWrite;       // Worker processes write results on disk
6 int numIterations;       // Number of iterations
```

Listado 5.3: *Parámetros de configuración de la aplicación appMR*

Para llevar a cabo los experimentos descritos en esta sección, se han generado varios conjuntos de tests utilizando dos métodos diferentes, generación de tests con el Algoritmo 1 y generación de tests de forma aleatoria. El Cuadro 5.1 muestra la configuración utilizada para generar los conjuntos de tests con el Algoritmo 1.

Para comparar los distintos conjuntos de tests se ha utilizado la siguiente notación: $tsAppCPU$ denota un conjunto de tests empleado para probar la aplicación *appCPU*, mientras que $tsAppMR$ denota un conjunto de tests empleado para probar la aplicación *appMR*. El método con el cual se han generado los conjuntos de tests se indica mediante los sufijos *algo1* y *random*, donde *algo1* indica que los tests han sido generados con el Algoritmo 1 y *random* indica que los tests han sido generados aleatoriamente.

Para la realización de los experimentos descritos en esta sección se han generado 4 conjuntos de tests, $tsAppCPU_{algo1}$ y $tsAppCPU_{random}$, formados por 72 tests y $tsAppMR_{algo1}$ y $tsAppMR_{random}$, formados por 144 tests. A su vez, los mutantes utilizados han sido seleccionados a través de la técnica *mutant sampling*.

Una vez generados los conjuntos de tests, se ha procedido a la evaluación de los mismos siguiendo el proceso de testing descrito en la Figura 4.3. Para ello, el equipo utilizado ha

Tests	Aplicación	Parámetro	Tipo	minValue	maxValue	Interval
$tsAppCPU_{algo1}$	appCPU	inputDataSize	int	10MiB	20MiB	5MiB
		outputDataSize	int	10MiB	20MiB	5MiB
		MIs	int	$10 \cdot 10^6$	$25 \cdot 10^6$	$5 \cdot 10^6$
		iterations	int	1	3	2
$tsAppMR_{algo1}$	*appMR*	workersSet	int	4	8	4
		sliceToWorkers	double	64KiB	32KiB	128KiB
		sliceToMaster	double	16KiB	32KiB	16KiB
		sliceCPU	int	$10 \cdot 10^6$	$20 \cdot 10^6$	$5 \cdot 10^6$
		workersWrite	boolean	true	false	n/a
		numIterations	int	1	3	2

Cuadro 5.1: *Configuración para la creación de tests utilizando el Algoritmo 1*

sido un MacBook Pro Intel Core i5 2.5 Ghz con 16 GB RAM y 500 GB de disco.

Los Cuadros 5.2 y 5.3 muestran el coste computacional de cada una de las fases del proceso, utilizando los conjuntos de tests generados mediante el Algoritmo 1 y de forma aleatoria, respectivamente. La columna *Tests* indica el conjunto de tests aplicado, *Aplicación* indica la aplicación sobre la cual se han ejecutado los tests, *Operadores* representa los operadores utilizados para realizar el proceso de mutación, *#Mut.* indica el número de mutantes generados, finalmente, *Mutación*, *Compilación* y *Ejecución* muestran el tiempo para generar los mutantes, el tiempo para compilarlos y el tiempo para ejecutarlos, respectivamente. Existen dos casos para los que no se han generado mutantes, ya que *appCPU* no contiene llamadas de *MPI* y *appMR* no contiene llamadas de OMNeT++.

Los resultados obtenidos tras la ejecución del proceso de testing para los conjuntos de tests generados con los dos métodos anteriormente descritos, son muy similares. Esto indica que, en términos de rendimiento sobre las aplicaciones propuestas, los resultados son independientes del método utilizado para generar los conjuntos de tests.

Los Cuadros 5.4 y 5.5 muestran los resultados obtenidos al finalizar el proceso de testing. Como puede apreciarse en los cuadros, un mutante puede morir por dos causas. La columna

Tests	Aplicación	Operadores	# Mut.	Mutación	Compilación	Ejecución
$tsAppCPU_{algo1}$	appCPU	Generales	87	520 ms	2m 33s	2h 21m
		OMNeT++	2	267 ms	1m 1s	3m 24s
		SIMCAN	12	103 ms	1m 11s	5m 22s
		MPI	n/a	n/a	n/a	n/a
$tsAppMR_{algo1}$	*appMR*	Generales	121	870ms	3m 21s	26h 31m
		OMNeT++	n/a	n/a	n/a	n/a
		SIMCAN	24	396ms	1m 33s	4h 25m
		MPI	23	488ms	1m 35s	4h 15m

Cuadro 5.2: *Costes del proceso de testing utilizando tests creados con el Algoritmo 1*

Tests	Aplicación	Operadores	# Mut.	Mutación	Compilación	Ejecución
$tsAppCPU_{random}$	appCPU	Generales	87	475 ms	2m 33s	2h 41m
		OMNeT++	2	254 ms	1m 12s	4m 45s
		SIMCAN	12	123 ms	1m 15s	7m 12s
		MPI	n/a	n/a	n/a	n/a
$tsAppMR_{random}$	*appMR*	Generales	121	793ms	2m 43s	26h 11m
		OMNeT++	n/a	n/a	n/a	n/a
		SIMCAN	24	390ms	1m 20s	4h 08m
		MPI	23	438ms	1m 12s	5h 35m

Cuadro 5.3: *Costes del proceso de testing utilizando tests creados aleatorioamente*

Falla test indica los mutantes que, habiendo finalizado su ejecución, obtienen unos resultados que no coinciden con los resultados obtenidos por la aplicación sin mutar, mientras que la columna *Error* indica los mutantes que no han finalizado su ejecución. Esta situación puede ocurrir por varios motivos, como por ejemplo llegar a un punto en el que la aplicación se queda bloqueada o realizar una operación no permitida con un puntero (*segmentation fault*).

Los mutantes vivos representan aquéllos que han finalizado su ejecución y han obtenido los mismos resultados que la aplicación sin mutar. Para cada tipo de operador, la idoneidad del conjunto de tests se mide mediante el *Mutation Score* con la fórmula descrita en la

Conjunto de tests	Operadores	# Mutantes	Matados		Vivos	Mutation Score
			Falla test	Error		
$tsAppCPU_{algo1}$	Generales	87	13	19	55	0.36
	OMNeT++	2	0	2	0	1.0
	SIMCAN	12	0	9	3	0.75
	MPI	n/a	n/a	n/a	n/a	n/a
$tsAppMR_{algo1}$	Generales	121	8	43	70	0.42
	OMNeT++	n/a	n/a	n/a	n/a	n/a
	SIMCAN	24	2	20	2	0.92
	MPI	23	2	19	2	0.91

Cuadro 5.4: *Evaluación de los conjuntos de tests creados con el Algoritmo 1*

Conjunto de tests	Operadores	# Mutantes	Matados		Vivos	Mutation Score
			Falla test	Error		
$tsAppCPU_{random}$	Generales	87	13	19	55	0.36
	OMNeT++	2	0	2	0	1.0
	SIMCAN	12	0	9	3	0.75
	MPI	n/a	n/a	n/a	n/a	n/a
$tsAppMR_{random}$	Generales	121	8	43	70	0.42
	OMNeT++	n/a	n/a	n/a	n/a	n/a
	SIMCAN	24	2	20	2	0.92
	MPI	23	2	19	2	0.91

Cuadro 5.5: *Evaluación de los conjuntos de tests creados aleatoriamente*

sección 4.4. En este caso, puesto que el objetivo es automatizar el proceso de evaluación de cada conjunto de tests empleando un número elevado de mutantes, no se han tenido en cuenta los mutantes equivalentes.

Detectar mutantes equivalentes es un problema no decidible[8] para un conjunto de test, por lo que este proceso requiere la intervención manual por parte del usuario. Si tenemos en cuenta el número de mutantes vivos obtenidos, su análisis para poder concluir si éstos son o no equivalentes requeriría un tiempo y esfuerzo muy alto por parte del usuario.

Los resultados obtenidos muestran unos valores de MS idénticos para los conjuntos de tests generados con ambos métodos, tanto utilizando el Algoritmo 1, como generando los tests de forma aleatoria. Por ello, en lo que respecta al MS obtenido, estos resultados son independientes del tipo de conjunto de test utilizado para realizar el proceso de testing.

5.3. Evaluación de los resultados obtenidos

Esta sección presenta una discusión de los resultados obtenidos en los experimentos. Para poder realizar una comparación con mayor claridad, se han incluido varias gráficas que representan los resultados expuestos en la sección anterior.

Las Figuras 5.3 y 5.4 muestran los resultados obtenidos utilizando conjuntos de tests generados con el Algoritmo 1 y de forma aleatoria, respectivamente.

Ambas gráficas utilizan una escala logarítmica con el coste computacional de cada fase del proceso de *testing*, medido en minutos de cómputo. Los resultados están agrupados por tipos de operador y conjunto de tests, de forma que los sufijos *Mut*, *Comp* y *Ejec* hacen referencia a los tiempos de mutación, compilación y ejecución respectivamente.

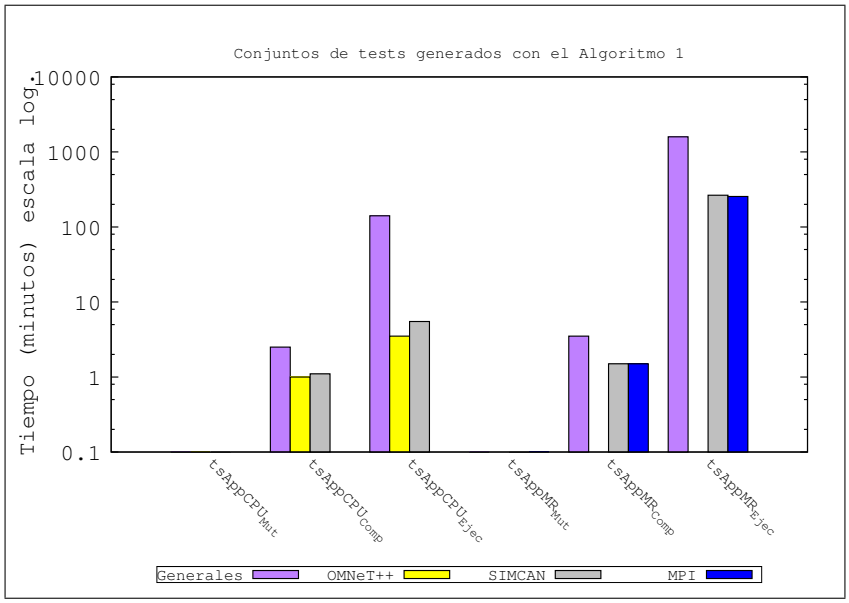

Figura 5.3: *Rendimiento del proceso de testing utilizando tests creados con Algoritmo 1*

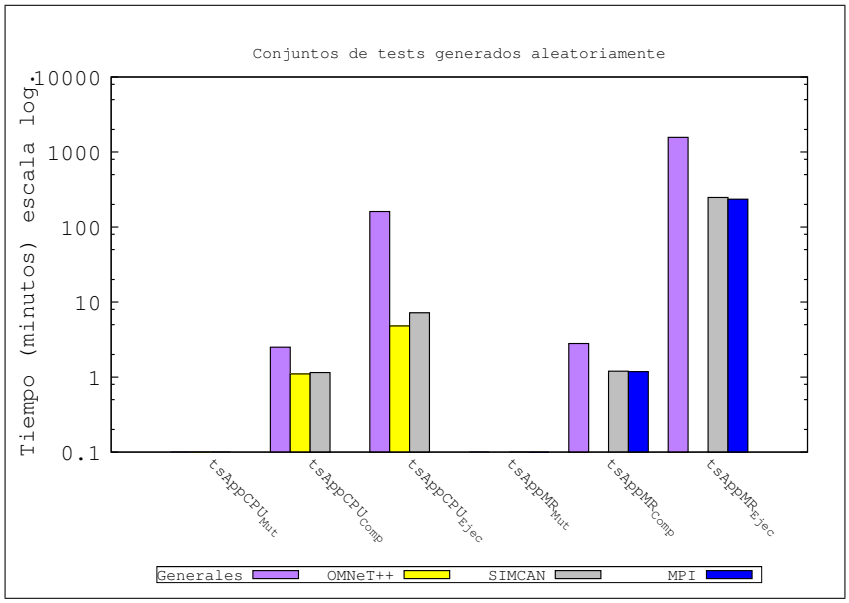

Figura 5.4: *Rendimiento del proceso de testing utilizando tests creados aleatoriamente*

Como puede apreciarse en ambas gráficas, los tiempos de generación de mutantes son muy pequeños en comparación con el coste de las otras fases. En todos los casos esta fase requiere menos de un segundo para llevarse a cabo.

La compilación de mutantes requiere un tiempo sustancialmente mayor al que pueda requerir una aplicación convencional. Esto sucede principalmente por la estructura de SIMCAN, ya que al incluir nuevas aplicaciones en su estructura de archivos, no se compilan únicamente los ficheros con el código fuente de los mutantes, sino una parte del propio simulador. De esta forma, cada vez que se incluye un nuevo mutante en el simulador, esto conlleva un coste adicional además de la compilación del propio mutante. Como puede apreciarse en las gráficas, el tiempo requerido para compilar es directamente proporcional al número de mutantes involucrados en el proceso. El caso más costoso, utilizando 121 mutantes, requiere 3 minutos y 21 segundos, mientras que el caso más rápido en compilar, utilizando únicamente 2 mutantes, requiere 1 minuto y 1 segundo. Ambos casos se corresponden con conjuntos de tests generados con el Algoritmo 1.

El proceso de ejecución de los mutantes es claramente el que más tiempo requiere. Es importante enfatizar que hay una diferencia notable entre el tiempo de ejecución de los

mutantes generados utilizando los operadores generales y el tiempo de ejecución de los mutantes generados a partir de los operadores propuestos en este trabajo. Esto se debe, fundamentalmente, a que el número de mutantes generados utilizando los operadores de mutación específicos para la simulación, es significativamente inferior al número de mutantes obtenido utilizando operadores generales.

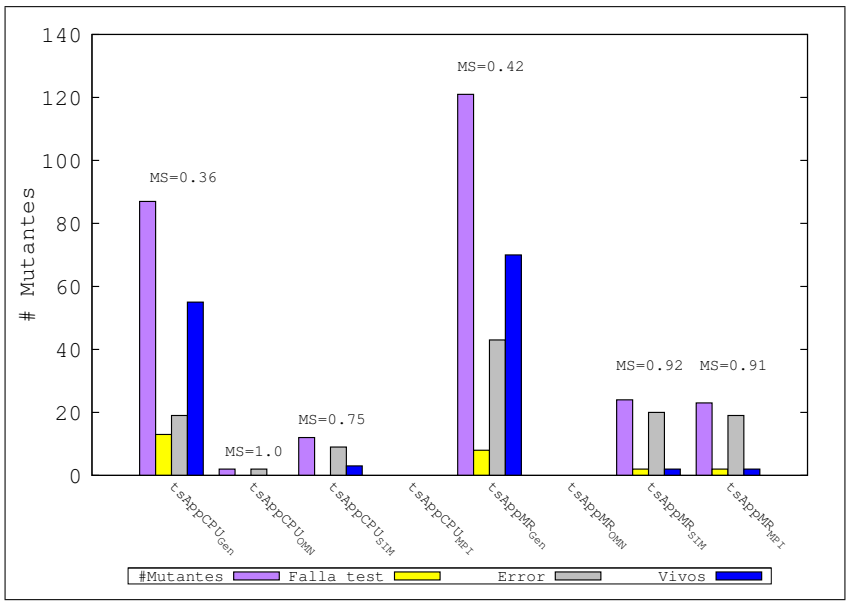

Figura 5.5: *Gráfica con los resultados obtenidos de evaluar los conjuntos de tests*

En cuestión de rendimiento para testear *appCPU*, el conjunto de tests $tsAppCPU_{algo1}$ es mejor que $tsAppCPU_{random}$. Sin embargo, para testear *appMR*, el conjunto de tests $tsAppMR_{random}$ es mejor que $tsAppMR_{algo1}$ en casi todos los casos, excepto cuando se utilizan operadores de mutación MPI. Esto se debe a que, para la mayoría de los casos, el valor del parámetro que define el tamaño de los datos, generado aleatoriamente, es menor que el valor generado por el Algoritmo 1. Este hecho se refleja en un menor tiempo de ejecución de la aplicación y, en consecuencia, de los tests.

La Figura 5.5 muestra los resultados de la evaluación de los conjuntos de tests. Puesto que los resultados mostrados en los Cuadros 5.4 y 5.5 son iguales, únicamente ha sido utilizada una representación gráfica de los mismos. En esta gráfica los resultados se han agrupado por el tipo de operador utilizado para generar los mutantes, donde el sufijo *Gen* representa

los operadores generales, *OMN* los operadores basados en las llamadas de OMNeT++, *SIM* los operadores basados en las llamadas de SIMCAN y *MPI* los operadores basados en las llamadas de *MPI*.

El conjunto de tests *tsAppCPU* obtiene un valor de MS=0.36 cuando utilizamos operadores generales para realizar las mutaciones, tales como operadores aritméticos y operadores lógicos. Este resultado indica que este conjunto de tests ha matado pocos mutantes, dejando vivos a 55 de 87, con lo que podemos concluir que es un resultado muy pobre. Analizando los resultados para el resto de operadores, podemos ver que los valores de MS mejoran sustancialmente. En particular, se obtiene MS=1.0 para los mutantes generados con los operadores de OMNeT++ y MS=0.75 para los mutantes generados a partir de los operadores de SIMCAN. Sin embargo, el número total de mutantes generados para estos dos últimos casos es muy pequeño. Básicamente, esto se debe a que *appCPU* contiene muy pocas llamadas de OMNeT++ y SIMCAN, por lo cual se han generado únicamente 2 y 12 mutantes, respectivamente.

Los resultados para *tsAppMR* mejoran ligeramente cuando utilizamos los operadores generales para generar los mutantes. Así, este conjunto de tests es capaz de matar a 51 mutantes de 121, obteniendo un MS=0.42. Este valor, aunque mejora el ofrecido por el *tsAppCPU*, sigue siendo pobre, ya que mata menos de la mitad de los mutantes generados. Los resultados obtenidos cuando se utilizan los mutantes generados por los operadores *SIMCAN* y *MPI*, son más prometedores, obteniendo valores de MS=0.92 y MS=0.91 respectivamente. En este caso, se han generado 24 mutantes utilizando operadores de *SIMCAN* y 23 mutantes utilizando operadores de *MPI*.

Como conclusión podemos decir que, independientemente del método empleado para generar los conjuntos de tests, los resultados obtenidos son prácticamente idénticos. Además, los conjuntos de tests analizados son más eficaces matando mutantes generados con operadores específicos de simulación. En general, si creamos los mutantes utilizando estos operadores, obtenemos buenos resultados, siendo 0.75 el valor de MS más pequeño. Esto se

debe, fundamentalmente, a que los mutantes generados con operadores convencionales son más difíciles de matar. Además, el propio entorno orientado a eventos donde se ejecutan los tests incrementa la dificultad de matar mutantes generados con operadores convencionales.

Capítulo 6

Conclusiones y trabajo futuro

En este trabajo se ha desarrollado MuTomVo, un *framework* de mutación de código que integra técnicas de *mutation testing* con técnicas de simulación, a través de la plataforma de modelado y simulación de sistemas distribuidos SIMCAN.

Las dificultades abordadas durante su desarrollo han sido mayores que las estimadas inicialmente. Esto se debe a los problemas añadidos de la integración de mecanismos de *testing* en un entorno simulado. En primer lugar, ha sido necesario adquirir conocimientos técnicos, tanto en las plataformas de simulación OMNeT++ y SIMCAN, como en técnicas de *mutation testing*. Además, teniendo en cuenta que el paradigma de programación de estas plataformas está basado en eventos, la aplicación de técnicas de *mutation testing* resulta más complicada que en paradigmas de programación tradicionales. Básicamente, la dificultad radica en que las variables analizadas en el código dependen del orden en que se ejecuten los eventos, los cuales no son procesados cuando éstos se generan, sino cuando llegan al módulo correspondiente. Esto produce una modificación en el flujo de ejecución de una aplicación, complicando la tarea de selección de un conjunto de operadores de mutación adecuados.

Una vez finalizado MuTomVo, se ha llevado a cabo una fase de experimentación. Los experimentos realizados han consistido en analizar la idoneidad de distintos conjuntos de tests ejecutados en diferentes aplicaciones distribuidas. Cada una de estas aplicaciones, junto con las arquitecturas donde se han ejecutado, han sido modeladas con SIMCAN.

Los experimentos realizados muestran que el coste computacional para llevar a cabo el

proceso de *testing* es muy elevado. En este punto queremos matizar que hay una diferencia notable en el tiempo de ejecución del proceso de *testing*, el cual depende directamente de los operadores empleados para generar los mutantes. De esta forma, utilizando los operadores de mutación propuestos en este trabajo, se genera un número más reducido de mutantes que utilizando operadores de mutación convencionales, por lo que, consecuentemente, se obtiene una mejora sustancial en términos de rendimiento.

Cabe destacar, tal como reflejan los resultados obtenidos, la obtención de un MS (*Mutation Score*) superior con el uso de los operadores propuestos, orientados a simulación, con respecto a la aplicación de operadores de mutación generales. Esto es debido a que los operadores de mutación generales, tales como operadores aritméticos o condicionales, generan mutantes que son más difíciles de matar que los generados con los operadores orientados a simulación.

Además, se han comparado conjuntos de tests creados con dos métodos diferentes, utilizando un algoritmo configurable basado en asignar valores por medio de intervalos y generando tests aleatoriamente. Los experimentos realizados muestran que ambos conjuntos de tests obtienen resultados muy parecidos, tanto a nivel de rendimiento como de efectividad. Esto es debido a la naturaleza de las aplicaciones utilizadas en los experimentos. Concretamente, la variación periódica de los valores asociados a los parámetros utilizados en estas aplicaciones, tales como por ejemplo *inputDataSize* o *workersSet*, no afecta al índice de MS resultante del proceso de *mutation testing*. Como conclusión, podemos decir que los operadores de mutación propuestos obtienen mejores resultados para los tests generados de forma automática en los entornos de simulación distribuidos utilizados en este trabajo, tanto en tiempo de ejecución como en MS, que los operadores tradicionales.

La aplicación de las técnicas de *mutation testing* requiere un alto coste computacional. Por ello, como trabajo futuro se propone la utilización de técnicas de paralelismo para aumentar el rendimiento del proceso de ejecución de mutantes y reducir así el tiempo necesario para llevar a cabo el proceso completo de *mutation testing*. A su vez, también se

propone la inclusión de nuevos operadores para matar más mutantes con los conjuntos de tests generados.

Chapter 6

Conclusions and future work

This work presents MuTomVo, a mutation testing framework that integrates mutation testing techniques with simulation techniques.

The difficulties faced during the development of this work have been higher than initially expected. This is due to the additional problems of integrating testing mechanisms in a simulated environment. First, it was necessary to obtain technical expertise in several platforms such as OMNeT++ and SIMCAN, and mutation testing techniques. Moreover, considering the event-based programming paradigm of these platforms, the application of mutation testing techniques is more difficult than traditional programming paradigms. Basically, this difficulty lies in managing different events that modify variables in the code. This produces a change in the execution flow of an application, complicating the task of selecting a suitable set of mutation operators.

Once the development of MuTomVo has been completed, different experiments have been conducted. These experiments consist on analysing the suitability of different test suites over different distributed applications modelled by using SIMCAN.

These experiments show that the computational cost for performing the testing process is very high. At this point, we want to remark that there is a significant difference in the execution time of the testing process, which directly depends on the operators used to generate mutants. Thus, using the mutation operators proposed in this project generate a smaller number of mutants than using conventional mutation operators. Consequently, a

substantial improvement is obtained in terms of performance.

The results show that a better MS (Mutation Score) is obtained when using the proposed operators focused on simulation, with respect to the general mutation operators. This is due to the general mutation operators, such as arithmetic and conditional operators, generate mutants that are harder to kill than those generated with operators focused on simulation.

Also, different test suites have been evaluated, which have been generated by using two different methods, using a customizable algorithm and randomly generating tests. The experiments carried out show that both test suites obtain similar results. This is mainly caused due to the nature of the evaluated applications.

As a final conclusion, we can say that the proposed mutation operators outperform for tests generated automatically in distributed simulation environments used in this work, both at runtime and MS, which traditional operators.

A line of future work is to use techniques for exploiting parallelism to increase the performance of executing mutants and, therefore, reducing the time needed to perform the entire process of mutation testing.

Bibliografía

[1] Network simulator NS-2. http://www.isi.edu/nsnam/ns.

[2] A. Sulistio and U. Cibej and S. Venugopal and B. Robic and R. Buyya. A toolkit for modelling and simulating Data Grids: An extension to GridSim. *Concurrency and Computation: Practice and Experience*, 20(13):1591–1609, 2008.

[3] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the c programming language. techreport SERC-TR-41-P, Purdue University, West Lafayette, Indiana, March 1989.

[4] B. K. Aichernig. Mutation testing in the refinement calculus. *Formal Aspects of Computing*, 15(2-3):280–295, November 2003.

[5] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[6] J. H. Bowser. Reference manual for ada mutant operators. techreport GIT-SERC-88/02, Georiga Institute of Technology, Atlanta, Georgia, 1988.

[7] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. *The DiskSim Simulation Environment Version 4.0 Reference Manual.* Dept. Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008.

[8] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, March 1982.

[9] T. A. Budd, T. A. DeMillo, T. J. Lipton, and F. Gerald Sayward. The design of a prototype mutation system for program testing. In *Proceedings of the AFIPS National Computer Conference*, pages 623–627, Anaheim, New Jersey, 5-8 June 1978.

[10] T. A. Budd and F. G. Sayward. Users guide to the pilot mutation system. techreport 114, Yale University, New Haven, Connecticut, 1977.

[11] R. Buyya, A. Beloglazov, and J. Abawajy. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. In *Proc of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, USA, 2010.

[12] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[13] G. G. Castañé, A. Núñez, and J. Carretero. icancloud: A brief architecture overview. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 853–854, July 2012.

[14] G. G. Castañé, A. Núñez, P. Llopis, and J. Carretero. E-mc2: A formal framework for energy modelling in cloud computing. *Simulation Modelling Practice and Theory*, 39:56–75, January 2013.

[15] Barcelona Supercomputing Center. MareNostrum III User Guide. Technical report, 2014. http://www.bsc.es/support/MareNostrum3-ug.pdf.

[16] W. K. Chan, S. C. Cheung, and T. H. Tse. Fault-based testing of database application programs with conceptual data model. In *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, pages 187–196, Melbourne, Australia, 19 -20 September 2005.

[17] X. Chang. Network simulations with OPNET. In *WSC'99: Proceedings of the 31st conference on Winter simulation*, pages 307–314. ACM, 1999.

[18] C.L. Dumitrescu and I. Foster. GangSim:a simulator for grid scheduling studies. In *Cluster Computing and the Grid*, volume 2, may 2005.

[19] CSIM Development Toolkit for Simulation and Modeling. Web page at http://www.mesquite.com/. Date of last access:27th July, 2011.

[20] D. Kliazovich and P. Bouvry and S. U. Khan. Greencloud: A packet-level simulator of energy-aware cloud computing data centers. *Journal of Supercomputing (in press)*.

[21] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[22] R. Delamare, B. Baudry, and Y. Le Traon. Ajmutator: A tool for the mutation analysis of aspectj pointcut descriptors. In *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, pages 200–204, Denver, Colorado, 1-4 April 2009.

[23] M. E. Delamaro, J. J. Maldonado, and A. P. Mathur. Integration testing using interface mutation. In *Proceedings of the seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, pages 112–121, White Plains, New York, 30 October - 02 November 1996.

[24] R. A. DeMillo. Program mutation: An approach to software testing. techreport, Georgia Institute of Technology, 1983.

[25] R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. An overview of the mothra software testing environment. techreport SERC-TR-3-P, Purdue University, West Lafayette, Indiana, 1987.

[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[27] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[28] R. A. DeMillo and A. J. Offutt. Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol.*, 2(2):109–127, April 1993.

[29] R. A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[30] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 411–420, Budapest, Hungary, 25-30 September 2005.

[31] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for ws-bpel 2.0. In *Proceedings of the 21th International Conference on Software and Systems Engineering and their Applications (ICSSEA'08)*, Paris, France, 9-11 December 2008.

[32] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, pages 52–61, Lillehammer, Norway, 9-11 April 2008.

[33] S. C. P. Ferraz, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 220–229, Monterey, California, 6-9 November 1994.

[34] S. C. P. Ferraz, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, page 210, Boca Raton, Florida, 1-4 November 1999.

[35] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Proc. Grid Computing Environments Workshop*, pages 1–10, Austin, USA, November 2008.

[36] A. S. Gopal and T. A. Budd. Program testing by specification mutation. techreport TR 83-17, University of Arizona, Tucson, Arizona, 1983.

[37] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference*. MTI-Press, 2nd edition edition, 1998.

[38] H. Casanova, A. Legrand and M. Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*, pages 126 –131, april 2008.

[39] N. Hardavellas, S. Somogyi, T. F. Wenisch, E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31:31–35, 2004.

[40] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, December 1999.

[41] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[42] S. Hussain. *Mutation Clustering*. phdthesis, King's College London, UK, 2008.

[43] J. Liu and D. M. Nicol. *DaSSF 3.1 User's Manual*. Dartmouth College, April 2001.

[44] J.A. Offutt. Mothra. Web page at http://cs.gmu.edu/~offutt/rsrch/mut.html. Date of last access:7th February, 2015.

[45] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, 29-31 August 2008.

[46] K. Fujiwara and H. Casanova. Speed and accuracy of network simulation in the simgrid framework. In *Proc. of the 1st Intl. Workshop on Network Simulation Tools (NSTools)*, Nantes, France, 2007.

[47] K.H. Kim, A. Beloglazov, and R. Buyya. Power-aware provisioning of cloud resources for real-time services. In *Proc. of the 7th Intl. Workshop on Middleware for Grids, Clouds and e-Science*, Urbana Champaign, Illinois, USA, 2009.

[48] S. Kim, J. A. Clark, and J. A. McDermid. The rigorous generation of java mutation operators using hazop. In *Proceedings of the 12th International Cofference Software and Systems Engineering and their Applications (ICSSEA 99)*, Paris, France, 29 November-1 December 1999.

[49] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. In *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, pages 207–225, San Jose, California, 6-7 October 2001.

[50] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software:Practice and Experience*, 21(7):685–718, October 1991.

[51] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 171 – 177, Banff Alberta, July 1988.

[52] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, May 1991.

[53] J. F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach.* Pearson, 2012.

[54] S. Lee, X. Bai, and Y. Chen. Automatic mutation testing and simulation on owl-s specified web services. In *Proceedings of the 41st Annual Simulation Symposium (ANSS'08)*, pages 149–156, Ottawa, Canada., 14-16 April 2008.

[55] S. C. Lee and A. J. Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, pages 200–209, Hong Kong, China, November 2001.

[56] S. Lim, B. Sharma, G. Nam, E. Kim, and C.R. Das. Mdcsim: A multi-tier data center simulation platform. In *Intl. Conference on Cluster Computing and Workshops (CLUSTER)*, New Orleans, USA, 2009.

[57] R. J. Lipton and F. G. Sayward. The status of research on program mutation. In *Proceedings of the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[58] Y. Ma, A. J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, June 2005.

[59] Y. Ma, A. J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[60] Mars Climate Orbiter Mishap Investigation Board. Mars Climate Orbiter mishap investigation board Phase I report, November 10, 1999.

[61] E. E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, pages 667–676, Banff, Alberta, Canada, 8-12 May 2007.

[62] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9 – 31, 1994.

[63] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2010. http://www.top500.org.

[64] A. S. Namin and J.H. Andrews. Finding sufficient mutation operators via variable reduction. In *Mutation Analysis, 2006. Second Workshop on*, pages 5–5, Nov 2006.

[65] A. Núñez, J. Fernández, and J. Carretero. Simcan. http://www.arcos.inf.uc3m.es/.

[66] A. Núñez, J. Fernández, R. Filgueira, F. García, and J. Carretero. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. *Simulation Modelling Practice and Theory*, 20(1):12–32, 2012.

[67] A. Núñez, J. Fernández, J. D. Garcia, L. Prada, and J. Carretero. Simcan: A simulator framework for computer architectures and storage networks. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools'08)*, pages 73:1–73:8, 2008.

[68] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente. iCanCloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[69] A. J. Offutt. *Automatic Test Data Generation.* phdthesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988.

[70] A. J. Offutt. The coupling effect: Fact or fiction. *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, December 1989.

[71] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, September 1994.

[72] A. J. Offutt and K. N. King. A fortran 77 interpreter for mutation analysis. *ACM SIGPLAN Notices*, 22(7):177–188, July 1987.

[73] A. J. Offutt, Y. Ma, and Y. R. Kwon. An experimental mutation system for java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, September 2004.

[74] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the 12 International Conference on Testing Computer Software*, pages 111–123, Washington, DC, June 1995.

[75] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pages 100–107, Baltimore, Maryland, May 1993.

[76] A. J. Offutt, J. V., and J. Payn. Mutation operators for ada. techreport ISSE-TR-96-09, George Mason University, Fairfax, Virginia, 1996.

[77] A. J. Offutt and W. Xu. Generating test cases for web services using data perturbation.

In *Proceedings of the Workshop on Testing, Analysis and Verification of Web Services (TAV-WEB)*, pages 1 – 10, Boston, Massachusetts, 11-14 July 2004.

[78] R. L Probert and F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In *Proceedings of the Workshop on Protocol Test Systems*, pages 57–76, Leidschendam, Netherland, 15-17 October 1991.

[79] R. Ranjan R. Buyya and R.N. Calheiros. Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *Proc. of the 7th High Performance Computing and Simulation Conference (HPCS)*, 2009.

[80] R. Buyya and M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency & Computation: Prac. & Exp.*, 14:1175–1220, 2002.

[81] R. N. Calheiros and R. Buyya and C. A. De Rose. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience*, 40(5):405–429, 2010.

[82] S. Ried, H. Kisker, P. Matzke, A. Bartels, and M. Lisserman. Sizing the cloud - a BT futures report. Understanding and quantifying the future of cloud computing. Forrester Research Report, 2011.

[83] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Simulation-based testing of distributed systems. Technical Report CU-CS-1004-06, Department of Computer Science, University of Colorado, January 2006.

[84] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 34:452–470, 2008.

[85] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3:460–471, 2010.

[86] T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a csp security example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, page 340–350, Chiang Mai, Thailand, 10-12 December 2003.

[87] T. Sugeta, J. C. Maldonado, and W. E. Wong. Mutation testing applied to validate sdl specifications. In *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems*, page 2741, Oxford, UK, 17-19 March 2004.

[88] Y. Traon, T. Mouelhi, and Benoit Baudry. Testing security policies: Going beyond functional testing. In *The 18th IEEE International Symposium on Software Reliability*, pages 93–102, Trollhättan, Sweden, 5-9 November 2007.

[89] J. Tuya, Maria J. S. Cabal, and C. de la Riva. Sqlmutation: A tool to generate mutants of sql database queries. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 1, Raleigh, North Carolina, November 2006.

[90] J. Tuya, M. J. Suarez, and C. de la Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, April 2007.

[91] R. H. Untch. Mutation-based software testing using program schemata. In *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)*, pages 285–291, Raleigh, North Carolina, 1992.

[92] R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 139–148, Cambridge, Massachusetts, 1993.

[93] A. Varga. The OMNeT++ discrete event simulation system,. In *Proc. of the European Simulation Multiconference (ESM)*, Prague, Czech Republic, 2001.

[94] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 21–30, Washington DC, USA, 2004.

[95] W. Liu and T. Fan. Live migration of virtual machine based on recovering system and cpu scheduling. In *Information Technology and Artificial Intelligence Conference (ITAIC), 2011 6th IEEE Joint International*, volume 1, pages 303–307, aug 2011.

[96] W.H. Bell, G. David, C. Capozza, L. Capozza, A.P. Millar, K. Stockinger and F. Zini. Simulation of dynamic grid replication strategies in OptorSim. In *Proc. of the 3rd Intl. Workshop on Grid Computing (Grid)*, Baltimore, USA, 2002.

[97] E. Winsberg. *Science in the Age of Computer Simulation.* University of Chicago Press, 2010.

[98] W. E. Wong. *On Mutation and Data Flow.* phdthesis, Purdue University, West Lafayette, Indiana, 1993.

[99] M. R. Woodward. Objtest: an experimental testing tool for algebraic specifications. In *Proceedings of the IEE Colloquium on Automating Formal Methods for Computer Assisted Prototying*, page 2, 14 Jan 1990.

[100] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutationtesting issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 152–158, Banff Albert, Canada, July 1988.

[101] X. Liu. *Scalable Online Simulation for Modeling Grid Dynamics.* PhD thesis, Univ. of California at San Diego, 2004.

[102] H. Yoon, B. Choi, and J. O. Jeon. Mutation-based inter-class testing. In *Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC'98)*, page 174, Taipei, Taiwan, 2-4 December 1998.

[103] C. N. Zapf. *A Distributed Interpreter for the Mothra Mutation Testing System.* phdthesis, Clemson University, Clemson, South Carolina, 1993.

[104] C. L. Zhu. Data center storage networking simulation, 2009. SimSANs version 3 is available at http://www.simsans.org.

Algoritmo 1 Generación automática de tests

Require: $List < ParamValues > list$;
Ensure: Conjunto de tests

```
    // Para cada parámetro de entrada
 1: for (i = 0 to list.size() − 1) do
 2:    // Si el tipo del parámetro actual es boolean
 3:    if (list[i].type == 'boolean') then
 4:       list[i].values.add (list[i].minValue)
 5:       if (list[i].minValue != list[i].maxValue) then
 6:          list[i].values.add (list[i].maxValue)
 7:       end if
 8:    end if
       // Si el tipo del parámetro actual es int
 9:    if (list[i].type == 'int') then
10:       x ← Integer.parseInt(list[i].minValue)
11:       while (x ≤ Integer.parseInt(list[i].maxValue)) do
12:          list[i].values.add (x)
13:          x ← x+Integer.parseInt(list[i].interval)
14:       end while
15:    end if
       // Si el tipo del parámetro actual es double
16:    if (param.type == 'double') then
17:       z ← Double.parseDouble(list[i].minValue)
18:       while (z ≤ Double.parseDouble(list[i].maxValue)) do
19:          list[i].values.add (z)
20:          z ← z+Double.parseDouble(list[i].interval)
21:       end while
22:    end if
23: end for
24: for (p_0 = 0 to list[0].values.size()) do
25:    for (p_1 = 0 to list[1].values.size()) do
26:       ...
27:       for (p_{list.size()−1} = 0 to list[list.size()-1].values.size()) do
28:          test ← (<list[0].name, list[0].values[p_0]>,<list[1].name, list[1].values[p_1]>
29:          , ..., <list[list.size()-1].name, list[list.size()-1].values[p_{list.size()−1}]>)
30:       end for
31:       ...
32:    end for
33: end for
```

Apéndice II

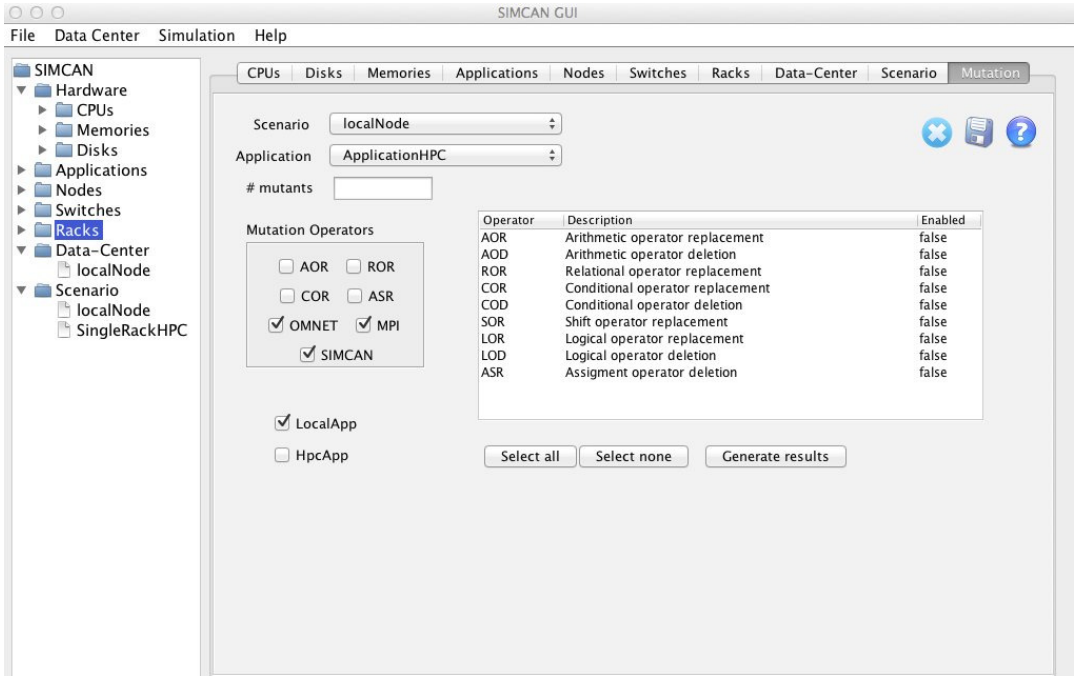

Figura 1: *Interfaz de usuario de MuTomVo*

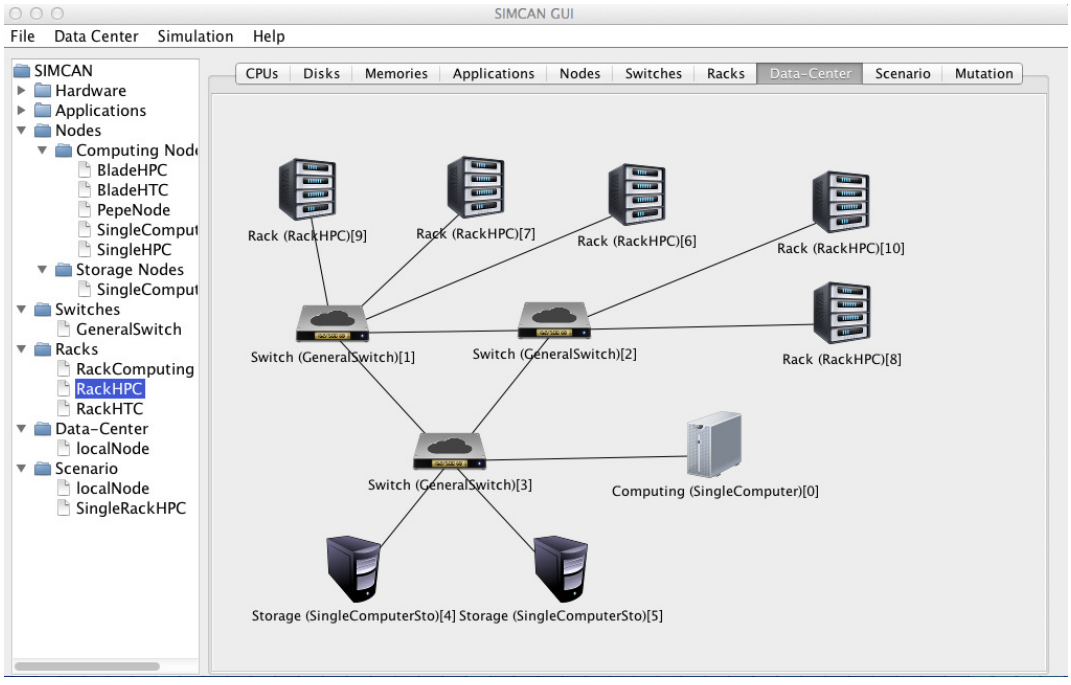

Figura 2: *Interfaz de modelado de sistemas distribuidos*