UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Integrated system architecture for internet of things model-driven design with applications in medicina

Arquitectura de un sistema integrado para diseño dirigido por modelos en el contexto de internet de las cosas con aplicaciones en medicina

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Kevin Henares Vilaboa

Directores

José Luis Risco Martín
José Luis Ayala Rodrigo
Román Hermida Correa

Madrid

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Integrated system architecture for Internet of Things model-driven design with applications in medicine.

Arquitectura de un sistema integrado para diseño dirigido por modelos en el contexto de Internet de las Cosas con aplicaciones en medicina.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR:

Kevin Henares Vilaboa

DIRECTORES:

José Luis Risco Martín
José Luis Ayala Rodrigo
Román Hermida Correa

Madrid, 2021

FACULTAD DE INFORMÁTICA

DOCTORADO EN INGENIERÍA INFORMÁTICA

Integrated system architecture for Internet of Things model-driven design with applications in medicine.

Arquitectura de un sistema integrado para diseño dirigido por modelos en el contexto de Internet de las Cosas con aplicaciones en medicina.

Memoria para optar al grado de doctor
presentada por:

Kevin Henares Vilaboa

Directores:

José Luis Risco Martín
José Luis Ayala Rodrigo
Román Hermida Correa

Madrid, 2021

Acknowledgement

I would like to express my gratitude to the people that helped in the development of this thesis. First, thanks to my supervisors, José Luis Risco Martín, José Luis Ayala, and Román Hermida, for their guidance and dedication. I also want to thank my advisors during my research stays in Switzerland and Canada, Marina Zapater and Gabriel Wainer, for their time and implication, and for facilitating the collaboration between our research groups. Thanks to my office partners at the computer science faculty of the UCM for accompanying me in this project and making research times easier. Finally, I want to thank my family for a lifetime of unconditional support.

Table of Contents

List of Figures

List of Tables

Resumen

A lo largo de los últimos años hemos visto cómo las arquitecturas de procesamiento y almacenamiento se vuelven más baratas y eficientes, las infraestructuras de comunicación se hacen más rápidas y escalables, y se desarrollan multitud de nuevas formas de interactuar con el mundo que nos rodea. Cada día más dispositivos se conectan a la red, y la generación de datos a nivel mundial está creciendo exponencialmente. En este contexto, el Internet de las cosas promete ser la nueva revolución tecnológica, como en su día lo fue la introducción de la red de redes o la accesibilidad móvil universal. Este paradigma promete difuminar la tradicional barrera entre el mundo físico y el digital, embebiendo sensores y actuadores en los objetos físicos que nos rodean y creando nuevos sistemas de información. Poco a poco, estamos viendo cómo este concepto está transformando multitud de ámbitos y permitiendo una gran variedad de nuevas aplicaciones y servicios. Las ciudades inteligentes emergen poco a poco, creando nuevas formas de transporte, optimizando sus infraestructuras y monitorizando y regulando aspectos como el tráfico de vehículos, la contaminación o los residuos generados por la población. En la industria se están implementando numerosas formas de monitorización y automatización, que están ayudando a minimizar los fallos, reducir el desperdicio y optimizar los procesos. En el ámbito médico vemos cómo el desarrollo de sistemas de monitorización y plataformas de apoyo a la toma de decisiones están provocando una transición desde el clásico sistema de diagnóstico y tratamiento post-facto hacia un marco proactivo de prevención y pronóstico más centrado en la potenciación y personalización de la salud que en el tratamiento de la enfermedad. Este contexto está favoreciendo el desarrollo de sistemas cada vez más complejos, con un gran número de componentes heterogéneos interactuando entre sí. Para abordar esta complejidad la industria ha ido incorporando diferentes técnicas de modelado y simulación que facilitan su diseño, verificación y validación. Entre estas técnicas se encuentran los formalismos de modelado y simulación, que proporcionan un marco de trabajo con un fuerte respaldo teórico y favorecen el desarrollo de sistemas robustos y escalables.

Esta tesis se orienta al desarrollo y estudio de modelos y escenarios de monitorización y predicción dentro del ámbito de la salud, haciendo uso del formalismo de simulación DEVS. Durante el desarrollo de la tesis se han realizado numerosas contribuciones en esta línea, desde tres niveles distintos de abstracción. En el de más bajo nivel, se ha contribuido al desarrollo de simuladores y herramientas de modelado con eventos discretos. Se ha añadido un simulador Python al entorno de modelado y simulación xDEVS, se han propuesto implementaciones alternativas reduciendo notablemente la carga introducida por este tipo de simuladores, se ha contribuido a la definición de métricas para el análisis y comparación de simuladores DEVS, y se han presentado varias herramientas que permiten la verificación de este tipo de sistemas. Desde el punto de vista intermedio del modelado de sistemas de salud, hemos desarrollado varias propuestas orientadas a la predicción y estimación de distintos eventos o sucesos clave para distintas enfermedades. En las primeras etapas, hemos estudiado cómo un sistema DEVS puede ser fácilmente implementado en una FPGA mediante su traducción a un lenguaje de especificación hardware. Para ello, se desarrolló un prototipo de sistema de monitorización sanitaria basado en un modelo de predicción de migraña, recogiendo y formateando información desde sensores médicos reales y alertando de la proximidad de nuevas fases de dolor mediante el uso de varios conjuntos de modelos predictivos. Para la generación de este tipo de modelos, también presentamos una metodología modular que permite automatizar la creación de bases de conocimiento y simplificar la producción de modelos predictivos mediante especificaciones XML. Además, heos desarrollado modelos epidemiológicos que nos permiten analizar y comprender cómo afectan distintos tipos de medidas y escenarios en la propagación de distintas epidemias. Finalmente, desde el mayor nivel de abstracción, nos planteamos cómo este tipo de modelos podrían ser usados de forma segura y escalable en entornos del Internet de las cosas. En este sentido, hemos realizado dos contribuciones principales. Una de ellas consiste en la mejora y extensión de SFIDE, un simulador para el estudio de estrategias de distribución de trabajos computacionales en centros de datos. Rediseñamos la plataforma, permitimos la comunicación con el conocido gestor de cargas SLURM, usado en multitud de centros de datos en todo el mundo, y posibilitamos el modelado de escenarios que conecten esos centros de datos con dispositivos finales a través del modelado de una capa de red intermedia. Por otra parte, realizamos un estudio de optimización que pretende analizar cómo influye a nivel de consumo energético la localización de distintos micro centros de datos en un contexto de monitorización médica continua en un ámbito urbano. Este escenario contempla la existencia de miles de pacientes de migraña monitorizados para la detección temprana de eventos críticos en su enfermedad, y cómo las cargas de trabajo generadas por los dispositivos de monitorización se reparten en los micro centros de datos y afectan al consumo energético total del sistema.

Con el desarrollo de estas temáticas, y abarcando numerosas herramientas, metodologías y casos de uso, esta tesis proporciona una amplia visión de cómo el modelado y la simulación supone una herramienta fundamental a la hora de desarrollar sistemas complejos, y cómo se pueden aprovechar estas técncias para la elaboración y despliegue de sistemas de modelado predictivo en el ámbito de la salud.

Abstract

Over the past few years, we have seen how processing and storage architectures become cheaper and more efficient, communication infrastructures become faster and more scalable, and many new ways of interacting with the world around us are being developed. Every day more devices are connected to the network, and the generation of data worldwide is growing exponentially. In this context, the Internet of Things promises to be the new technological revolution, as was the introduction of the network of networks or universal mobile accessibility in its day. This paradigm promises to blur the traditional barrier between the physical and digital world, embedding sensors and actuators in the physical objects that surround us and creating new information systems. Little by little, we see how this concept is transforming many fields and allowing a great variety of new applications and services. Smart cities are continually evolving, creating new forms of transport, optimizing their infrastructures, and monitoring and regulating aspects such as vehicle traffic, pollution, or waste generated by the population. Numerous forms of monitoring and automation are being implemented in the industry, helping to minimize failures, reduce waste, and optimize processes. In the medical field, we see how the development of monitoring systems and decision-making support platforms are causing a transition from the classic post-facto diagnosis and treatment system towards a proactive prevention and prognosis framework more focused on empowerment and personalization of health than in the treatment of diseases. This context favors the development of increasingly complex systems, with a large number of heterogeneous components interacting with each other. To address this complexity, the industry has been incorporating different modeling and simulation techniques that facilitate its design, verification, and validation. These techniques include modeling and simulation formalisms, which provide a framework with strong theoretical support and favor the development of robust and scalable systems.

This thesis is oriented to the development and study of monitoring and prediction models and scenarios within the health field, using the DEVS simulation formalism. During its development, numerous contributions have been made in this regard from three different levels of abstraction. At the lowest level, it has contributed to the development of discrete-event simulators and modeling tools. A Python simulator has been added to the xDEVS modeling and simulation environment. Alternative simulator implementations have been proposed that notably reduce the load introduced by this type of simulators. Also, the thesis has contributed to the definition of metrics for the analysis and comparison of DEVS simulators, and present several tools allowing the verification of this type of systems. From the intermediate point of view of health systems modeling, we have developed several proposals to predicting and estimating different key events for specific diseases. In the early stages, we have studied how a DEVS system can be easily implemented in an FPGA by translating it into a hardware specification language. For this, a health monitoring system prototype has been developed based on a migraine prediction model, collecting and formatting information from actual medical sensors and alerting the proximity of new pain phases through various sets of predictive models. For the generation of this type of models, we also present a modular methodology that allows automating the creation of knowledge bases and simplifying predictive models' production using XML specifications. Also, we have developed epidemiological models that enable us to analyze and understand how different types of measures and scenarios affect the spread of epidemics. Finally, from the highest level of abstraction, we have considered how this type of model could be used safely and scalably in the Internet of Things environments. In this regard, we have made two main contributions. One of them consists of the improvement and extension of SFIDE, a simulator for studying strategies for the distribution of computational work in data centers. We have redesigned the platform, allowed communication with the well-known SLURM workload manager used in many data centers around the world, and made possible the modeling of scenarios that connect those data centers models with end devices by modeling of an intermediate network layer. On the other hand, we have carried out an optimization study that aims to analyze how the micro data centers' location influences energy consumption in an urban healthcare scenario. This scenario considers the existence of thousands of migraine patients monitored for the early detection of critical events in their disease, and how the workloads generated by the monitoring devices are distributed in several micro data centers affecting the system's overall energy consumption.

With the development of these topics, and covering numerous tools, methodologies, and use cases, this thesis provides a broad vision of how modeling and simulation is a fundamental tool when developing complex systems, and how these techniques can be used for the development and deployment of predictive modeling systems in the healthcare field.

Chapter 1

Introduction

The data generation ratio worldwide is increasing exponentially over the years. In every instance, the domain-specific information is progressively taken into account to create valuable knowledge that helps understand our reality and make our procedures and technologies more effective and efficient. According to the International Data Corporation (IDC), worldwide created, captured, and replicated data will grow to 175 zettabytes by 2025. Since 2018, in which 33 zettabytes were registered, this represents an increment of 530% and a compound annual growth rate of 61%[174].

This scenario of the so-called data era opens many doors to provide new business values and technologies for society. Data is changing everything around us, and we are beginning to see the first glimpses of the opportunities that it can offer us. Some examples that are already a reality are the automatic detection of production defects by image, tools to predict and prevent breakdowns in industrial machinery, the principles of the autonomous car, real-time translation between languages, or models that diagnose all kinds of diseases and predict their events and outcomes. These examples are only a minimal part of the value that data is offering us, as we are in a transitional phase. These technologies are constantly growing for adapting to our needs, and soon they will be a fundamental part of our society.

However, the systems managing these data are becoming more complex every year. They present a large number of components, usually behaving dynamically. They are steadily evolving, incorporating new features and interacting with external systems. Over time, they tend to include knowledge of several disciplines and incorporate solutions with different levels of detail. For instance, a data center simulation model could include a highly-detailed representation of the processors functioning, and also describe high-level allocation strategies. Moreover, a system can combine models with several temporal scales (i.e., discrete and continuous models), or allow the generation of outcomes with different levels of precision. For studying such complex systems, we usually first try to develop analytical solutions. However, often the resulting mathematical models are not flexible or powerful enough for representing all their particularities and interactions. A common alternative to overcome these limitations consists in the development of simulation models, which allow describing the behavior of a system through a combination of mathematical descriptions and algorithms reproducing its life-cycle. This approach requires the creation of a conceptual model developed based on our study of reality, which is later transformed into a computational model using a programming language. This resulting executable model is then used to study and generate knowledge, adapting its inputs and configuration parameters based on our needs, and comparing its outcomes with real-world data. This process can be performed iteratively, improving the conceptual model relying on the outcomes of the simulation. Through this approach, we can reproduce complex situations that would be highly expensive or even infeasible to study in reality.

Due to these advantages, a multitude of Modeling and Simulation (M&S) techniques have been widely used in recent decades, both in academia and in industry. Although these modeling formalisms were born and matured in academia, companies around the world have already adopted them for the development of their software. This transition came first with the introduction of Model-Based Systems Engineering (MBSE)[96], which focused on generating different types of domain models as the primary documentation source. It favored better coordination between the development teams and increased workflow productivity. Over the years, and especially as of 2014, this evolved in Modeling and Simulation-Based Systems Engineering (M&SBSE)[68,140], a new perspective that seeks to reduce the gap existing between the domain models and the resulting software products. The domain models are replaced by executable models, able to directly reproduce behaviors present in the system of interest and simulate large amounts of configurations in manageable times. The main goal of this dissertation is to make M&SBSE the main driving force in the design of complex systems with a focus on the Internet of Things (IoT) and healthcare applications as the main use case. In the following sections, we describe some contributions oriented to this goal, giving a brief overview of the related state of the art.

1.1 M&S formalisms: benefits and challenges

Despite the inherent potential in the introduction of M&S mechanisms to the development workflow, it is important to select methodologies with a solid theoretical foundation and able to manage high complexity levels. According to Castro et al.[229], models are frequently perceived as "islands of knowledge", implemented for specific simulation engines. This makes it difficult to generalize or reuse models and becomes an obstacle when addressing the interdisciplinary challenges present in large-scale projects. However, there exist different modeling formalisms that help to overcome such setbacks, providing clear, reusable, and unambiguous specifications with which we can represent all kinds of heterogeneous complex systems. Among all these M&S formalisms, Discrete Event Systems (DESs)-based environments are widely used due to their intuitive and powerful nature[34]. They describe the state of the model as a discrete set that evolves as different events occur. Amid all the available DES approaches (e.g., Markov chains or Petri nets), the Discrete Event System Specification (DEVS) formalism[230] showed great success in integrating other discrete-event and continuous-time systems[207]. It also allows combining submodels of different nature to compose a hybrid system, usually using Quantized State Systems (QSS) to approximate continuous-time models through discrete events[111].

There exist plenty of DEVS-based frameworks available in different programming languages and containing a great variety of specific tools. These frameworks can be used to implement models based on a variety of DEVS-based formalisms (e.g., CDEVS, Cell-DEVS, DynDEVS, FDDEVS, PDEVS), perform parallel and distributed simulations, graphically design and visualize our models' hierarchy, and export our results in several formats. However, they also present some common drawbacks, such as the lack of interoperability standards or the shortage of powerful verification tools. As each framework contains a particular DEVS simulator implementation, we deal with different APIs and programming languages depending on the tools used to develop our system. Hence, reusing and combining the resulting models is still a tough task. For overcoming this inconvenience, the DEVS community proposed several web-based solutions to interconnect heterogeneous simulators. Touraille et al.[198] proposed an XML-based definition for defining standard distributed simulation linking DEVS and non-DEVS simulations. Risco-Martin et al.[176] implemented a framework to connect heterogeneous DEVS simulation combining Service Oriented Architecture (SOA) with Web Services Description Language (WSDL). As a result, this platform can execute simulations without local access to modeling components. Bae et al.[15] also contributed to this trend, presenting a DEVS-based plug-in framework for the interoperability of simulators. With this approach, it can communicate heterogeneous models implemented in accordance with the PlugSim interface and includes algorithms to support data exchange and time synchronization between DEVS and non-DEVS models.

During the development of the thesis, we have made exhaustive use of different DEVS modeling and simulation frameworks. Moreover, **we have made direct contributions to the core of the xDEVS M&S toolkit, implementing a Python DEVS simulator, developing several tools for facilitating the verification of models, and proposing new implementation methodologies to reduce overhead introduced by the DEVS simulators. Moreover, we have extended the DEVStone benchmark, providing an objective way of analyzing and compare different DEVS simulator implementations.**

1.2 Verification, Validation & Optimization of M&S Systems

The design, development, and implementation of complex systems continue to be a challenging effort at the systems engineering level. The problem is much more accentuated today in the data era, with the dawn of paradigms like the Internet of Things and Machine Learning. Modeling and Simulation are continuously demanding new formal methodologies to manage the design, development, and implementation of such ultra-large systems with a high level of quality and accuracy while fulfilling a wide range of real-time constraints[137]. The way we perform M&S must be adapted, providing new ideas and tools to separate the model from the simulator, and the implementation from the analysis. These M&S techniques often remain difficult to verify and validate. Performing Verification and Validation (V&V) is an exhaustive exercise for any simulation model. Due to the inherent complexity in current simulation models that comprise multi-faceted data-driven methodologies or co-simulation methodologies, V&V is a challenge of its own[139].

In the software industry, these activities are a solid part of the development workflow. They have been developing V&V methodologies for decades and applying them to their projects to improve the development performance and reduce the project development time and the overall costs. However, the simulation field is experiencing a slow adoption of these V&V methodologies, and the most popular M&S frameworks provide a reduced offer of verification tools. The CD++ DEVS-based simulator accepts basic test cases defining the expected component outputs[211]. Saadawi and Wainer[180] also proposed an alternative method to check DEVS models by translating the model components hierarchy to Timed Automata representations and using the UPPAAL model checker over them[21]. It was exemplified with eCD++, an extension of the CD++ simulator oriented to the development of real-time models. Zengin et al.[232] adapted the verification and validation technique proposed by Forrester and Senge[187] to the DEVS-Suite environment, exemplifying them using an Open Shortest Path First (OSPF) simulation model. This approach generates test case sets, defining its contribution to the overall model confidence. These tests cover different modeling aspects, as the objective specification, and the structure and behavior of the model. Members in the DEVS community also developed external V&V solutions, applying them over some publicly available simulators. Li et al.[119] designed a DEVS-based testing framework combining black-box and white-box approaches, aiming to check both the model specification and behavioral correctness of DEVS simulators. For performing this verification, they proposed a standard XML representation for event-and-state-traces and implemented a tool for generating test case suites covering all possible DEVS constructs and their combinations. They applied their methodology over the PythonDEVS and DEVS++ simulators, being able to point some flaws of their implementations. Hwang et al.[97] defined Finite & Deterministic DEVS (FDDEVS), a variant of the DEVS formalism oriented to facilitate the equivalence between the requirements set and the model behavior. Mittal and Douglass[138] developed a DEVS Domain Specific Language (DSL) called DEVS Modeling Language (DEVSML) to write correct DEVS models by construction. This language was integrated by Mittal and Martin[141] in an Eclipse-based DEVSML Studio.

We have contributed to this trend by implementing several verification tools. We have developed a unit testing platform following the principles of the Experimental Frame (EF), allowing the definition of XML-based test cases containing the expected states and outcomes of the different components of the system. We have also developed a verification tool that takes advantage of the clear differentiation of the modeling and simulation layers to embed a transparent and constraint-based algorithm in the execution of models. These constraints are specified in terms of arithmetic and logical expressions and can be used to ensure certain relations among the outputs of the subcomponents of the system in a straightforward manner. Finally, there are some specific cases in which we cannot ensure the outputs produced by our system. This is known in the testing world as the oracle problem. In the testing theory this case can be covered by many methodologies, but one of the most popular is metamorphic testing[236]. This approach establishes a set of relations between the input and outputs of the system. **We have also translated this into the simulation world, with the development of a platform-independent and flexible tool to perform metamorphic testing over DEVS models.** This tool reads the logging files produced by the simulator of our choice and parses them through simulator-specific translation plugins, getting clear information about the time of the events, the inputs, and the outputs of the system. This information is progressively passed to the relations evaluator module, which checks the compliance of the relations for every change produced in the input and output sets. The relations can be defined with external files using a custom notation, or through a set of Python functions following the notation of specific interfaces provided by the verification tool. This leads to our next contribution, detailed below.

1.3 Internet of Things

In tandem with the volume of the data, the number of worldwide connected devices is also increasing rapidly. Its current number already exceeded the 20-billion barrier, and it's growing at a 10% per year pace[125]. This fits into the already established paradigm of the Internet of Things (IoT). Things around us are progressively equipping microcontrollers, transceivers for digital communication, and suitable protocol stacks that will make them able to communicate among them and with the users[226]. In this way, they become an integral part of the Internet and open the door for multiple new use cases. Some examples are the urban IoT, where the traditional public services and infrastructures are optimized and evolved to allow interacting with the citizens, or the healthcare IoT, where the real-time monitoring through networked sensors is showing great potential to evolve different aspects of medicine such as the prediction, diagnose, and treatment of a wide range of physical and mental diseases.

Despite the new opportunities offered by the IoT paradigm, it also introduces some new requirements that need to be addressed adequately. The increase in the number of connected devices creates high storage and computing needs, that often can not be covered without the help of additional infrastructures because of the design and power consumption restrictions of the end devices. Cloud computing is one of the enabling platforms to support these needs, providing on-demand networked access to a shared pool of configurable computing resources. However, despite its flexibility and scalability, it introduces communication delays and can increase the overall system consumption. To alleviate its disadvantages it is usually combined with other technologies as Fog and Edge Computing. Although it has a similar characterization to the Cloud, Fog nodes usually present more heterogeneous storage, computing, and networking resources, and are located near the edge devices. This closeness gives Fog Computing a supporting role to the Edge computations, reducing the latency that would involve sending the data to the Cloud by providing intermediate computational and storing resources. Edge Computing is usually done in the same device that generates the data. Hence, it avoids the latency provoked by communication networks. As there is no transmission of data, they also stay safer and more reliable. However, Edge devices are usually restricted by their battery power constraints and present limited computing and storage resources.

This three-layer IoT framework allows the use of efficient policies capable of distributing the computing needs over the different nodes composing the network. This distribution is performed assuming a trade-off among parameters such as acceptable delay, power consumption, or security and legislation policies. Often, the Edge layer only performs simple calculations, or preprocessing tasks to leverage the data transmitted over the network. The rest of the tasks are distributed between Fog and Cloud layers according to the main goals of the service to be offered. They even can be dynamically adjusted based on parameters as priorities, expected precision of results, or computing load levels. Distributed scenarios also become possible, sharing computing loads between multiple devices to improve latency and improve the overall performance of the system. In this context dynamic on-demand networks can also be configured, in which similar nodes collaborate to provide functionalities that they would lack by themselves. For instance, optimized traffic regulation based on smart vehicle fleets or more precise weather forecasts based on networks of monitoring nodes deployed over a territory.

As the development of these complex systems involves a great effort in terms of design and implementation, it is highly convenient to use Modeling & Simulation (M&S) techniques to simulate and optimize their performance. Over the last few years, a great deal of research has been done in this regard, designing different allocation strategies to efficiently distribute the load processing over the nodes of the network, and implementing multiple frameworks that facilitate the design and implementation of IoT scenarios. Some of them focus on scenarios covering only certain types of computational resources[53,192], while others are oriented to specific types of processing methodologies[104,122].

This thesis contributes to these research lines from two perspectives. First, as a collaboration with the *École Polytechnique Fédérale de Lausanne*, **we have improved and extended the SFIDE data center simulator.** SFIDE allows the study of different allocation strategies, modeling the entire cooling and processing infrastructures, and generating detailed reports of task allocation, power energy consumption, and cooling-related information. **We have redesigned its implementation, adding some additional features like compatibility with SLURM**, a popular workload manager used in data centers over the world. Moreover, originally SFIDE only accepted a local generation of tasks. We have extended this perspective, **allowing the definition of IoT devices and intermediate networks**. In this way, we can now obtain information about the power consumption regarding network communications and end nodes, as well as useful insights of the communication delays. On the other hand, **we have performed an optimization case study, studying the impact that the specific locations of Micro Data Centers have on the infrastructure energy consumption.** In this scenario, the data centers receive and process model training and inference tasks, from a population of monitored migraine patients. We describe in detail the whole workflow, including the extraction of the different building layouts and infrastructures of the selected urban area, the modeling of the population behavior, and the implementation of the IoT scenario itself using a data stream-oriented IoT framework.

1.4 Healthcare Monitoring Systems

Healthcare systems present several features suitable for the application of M&S techniques. They are complex systems affected by high uncertainty and variability. They often require a stochastic approach to solve their inner challenges. M&S has been applied for a multitude of purposes, including health and care systems operation, disease progression modeling, screening-related modeling, or health behavior modeling. Although M&S traditionally has had a considerably lower adoption than in other sectors such as business and commerce, aerospace, and the military, we have seen a rise in the M&S-aided development of healthcare systems in the last decade. A wide range of healthcare applications, services, and infrastructures are being developed, at the same time that the systems to develop became more and more complex. They have also been powered by the progression of technologies like the Internet of Things and Machine Learning. Networked sensors, either worn on the body or embedded in our living environments, make possible the collection of useful data that can be processed later to obtain relevant conclusions by the medical staff. Specifically, the continuous monitoring through the use of Healthcare Monitoring Systems (HMSs) opens the door to a wide range of applications supporting medical and healthcare services. Contrary to the post-facto diagnose-and-treat reactive paradigm, having these data allows the prognosis of diseases in their initial stages[80]. Moreover, some diseases have specific symptoms for each patient. Monitoring the patients in those cases would allow the generation of personalized treatments based on the specific needs of each individual. As a result, the overall performance of the health system can be improved while reducing the involved costs. We have also seen how M&S helped to face the events derived from the recent COVID-19 pandemic[73,144]. Plenty of models analyzing the spread were developed by research groups all over the world, assuming a great role in government decision-making and helping to reduce its impact and consequences.

During the development of the thesis, we have made many contributions to healthcare modeling. First, **we have studied the model-driven design and implementation of an HMS of a migraine prediction system.** Based on a previously developed DEVS model, **we have adapted the migraine prediction system using a hardware description language.** As a result, we have obtained a fully operational implementation of the system, able to get data directly from medical sensors, and to generate alerts when a new pain phase within the migraine cycle is approaching. **We have also designed a modular methodology to simplify the creation of predictive models from data coming from symptomatic disease patients through custom XML specifications.** The methodology has been used to design a fully-operational system able to anticipate stroke types and outcomes in the early stages of new stroke crises. For this development, actual monitorization data coming from the Stroke Care Unit of the *Hospital Universitario de la Princesa* was used, combining that with the diagnoses of the medical staff, and storing the resulting models in a central database in an automated way. The system also covers the specification of end nodes, capable of downloading suitable predictive models from the central database and generate inferences. **Finally**, in collaboration with the Advanced Real-Time Simulation Laboratory (ARS) of Carleton University, **we have been involved in the development of different epidemiology models. These models combine cellular automata and discrete-event simulations to generate a prediction of the evolution of the epidemy over time**, basing its behavior on the discretization of a set of differential equations characterizing the virus spread.

1.5 Publications

1.5.1 Conference papers

This thesis has generated the following articles in international conferences:

- Henares, K., Pagán, J., Ayala, J. L., Risco-Martín, J. L. Advanced migraine prediction hardware system. In *Proceedings of the 50th Computer Simulation Conference* (Bordeaux, France, 2018), SummerSim'18, Society for Computer Simulation International.

- Henares, K., Risco-Martín, J. L., Zapater, M. Definition of a transparent constraint-based modeling and simulation layer for the management of complex systems. In *Proceedings of the Theory of Modeling and Simulation Symposium* (Tucson, Arizona, USA, 2019), SpringSim'19, Society for Computer Simulation International.

- Henares, K., Risco-Martín, J. L., Hermida, R., Roselló, G. R., Cárdenas, R. Modular framework to model critical events in stroke patients. In *Proceedings of the 2019 Summer Simulation Conference* (Berlin, Germany, 2019), SummerSim'19, Society for Computer Simulation International.

- Henares, K., Risco-Martín, J. L., Ayala, J. L., Hermida, R. Unit testing platform to verify DEVS models. In *Proceedings of the 2020 Summer Simulation Conference*, SummerSim'20, Society for Computer Simulation International (pp. 1-11).

- Cárdenas, R., Henares, K., Ruiz-Martín, C., Wainer, G. Cell-DEVS Models for the Spread of COVID-19. *Cellular Automata for Research and Industry* conference. (ACRI 2020).

- Cárdenas, R., Henares, K., Ruiz-Martín, C., Arroba P., Wainer, G., Risco-Martín, J. L A DEVS simulation algorithm based on shared-memory for enhancing performance. In *Proceedings of the 2020 Winter Simulation Conference*, WinterSim'20, Society for Computer Simulation International.

1.5.2 Book chapters

This thesis has generated the following book chapters:

- Henares, K., Pagán, J., Ayala, J. L., Zapater, M., Risco-Martín, J. L. Cyber-Physical Systems Design Methodology for the Prediction of Symptomatic Events in Chronic Diseases, in S. Mittal, A. Tolk (eds.) *Complexity Challenges in Cyber Physical Systems: Using Modeling and Simulation (M&S) to Support Intelligence, Adaptation and Autonomy* (2019), Wiley & Sons.

- Henares, K., Martín, J. L. R., Pagán, J., González, C., Ayala, J. L., Hermida, R. Cyber-Physical Systems Design Flow to Manage Multi-channel Acquisition System for Real-Time Migraine Monitoring and Prediction. In J.L.R Martin, S. Mittal, T. Oren (eds.) *Simulation for Cyber-Physical Systems Engineering* (pp. 283-304). Springer, Cham (2020).

1.5.3 Other publications

The author has also contributed in the following journal article:

- Gago-Veiga, A. B., Pagán, J., Henares, K., Heredia, P., González-García, N., De Orbe, M. I., ... Vivancos, J.. To what extent are patients with migraine able to predict attacks?. *Journal of pain research* (2018).

Chapter 2

Modeling and Simulation of Complex Systems

Complex systems are usually understood as organizations that include a large number of interacting components. They usually have a dynamic nature and increase their complexity over time. Some examples of complex systems are communication infrastructures, the Earth climate system, economies, organisms, or socio-economic organizations as cities. Given their inner complexity, we usually need mathematical models to study them. Furthermore, in many cases, they cannot be studied only with analytical solutions. A common alternative are simulation models, which allow to describe the behaviors of a system through a plethora of model types (e.g. Ordinary Differential Equations (ODE), Machine Learning, fuzzy models), and algorithms reproducing its life-cycle based on these descriptions.

When developing a system it is important to determine the context within it has to be composed and used. This context is usually referred in the literature as the *experimental frame*. The experimental frame is also crucial for performing and validating simulations[49]. The behavior of a simulation model is impacted by this context, and its outcomes can also alter its state. Figure 2.1 shows the relation between these entities. From this perspective, a model is a representation of the system itself. A simulation implies the use of a simulation engine for reproducing the behaviors of the system under study through the model specification.

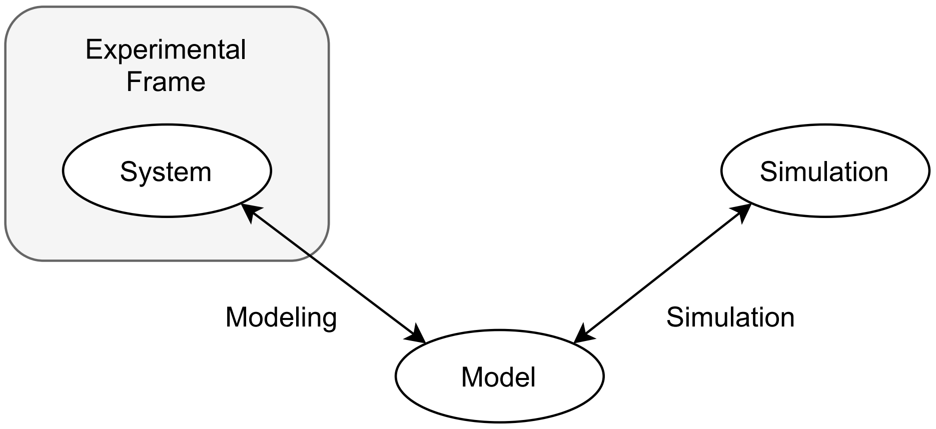


Figure 2.1: System, model and simulation relation

There are several phases and activities involved in the development of a simulation model. Figure 2.2 summarizes this process, showing the main modeling and assessment activities (dashed and solid lines, respectively). First, some analysis is performed over real-life phenomena to extract knowledge, as mathematical definitions, or behavioral rules. The conceptual model is constructed with this knowledge. Then, we can implement a computerized model based on this conceptual model, using the programming language and modeling framework of our preference. By interpreting the simulation outcomes of the computarized model we are able to analyze real-world conditions in a flexible and powerful way. We can perform studies that would be infeasible in the real world, and we can adapt the initial conditions and behavior parameters of the model to perform an in-depth analysis of the systems. However, several relevant inaccuracies or errors can be introduced in these models during the development process. To avoid these situations, we have to introduce several assessment activities.

The qualification activity aims at assuring that the conceptual model correctly extracts information of the system with the desired level of detail. The verification activity checks the relationship between the conceptual and computerized model (i.e. it verifies that the computerized model faithfully represents the conceptual model description, without introducing any unexpected additional behaviors derived from the programming implementation). Fi-

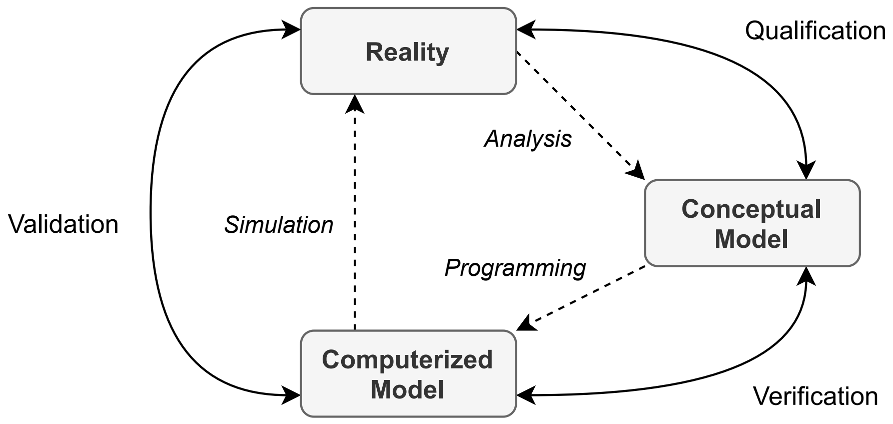


Figure 2.2: The model lifecycle.

nally, the validation activity aims at checking whether the simulation outcomes represent correctly the reality. Section 2.2 provides more detail about the Verification and Validation (V&V) of models, extending these concepts and detailing some inner challenges and methodologies.

Over time, models increase their complexity in different ways. They progressively accumulate knowledge of several disciplines and increase the level of detail of certain phenomena representations as the different development iterations are completed. The same model can combine low-level and high-level descriptions, and analyze a problem with different level of detail. For instance, a data center simulation model could include a highly-detailed representation of the processors included in its processing units, and also describe the allocation strategies of incoming computational tasks. This example illustrates the concept of multi-resolution modeling (MRM). According to Davis and Bigelow[50], MRM involves the development of a single model or family of models including different levels of resolution for a specific problem. Resolution is the detail with which a system (or attribute) is modeled[169]. The different levels of resolutions are usually associated with the level of abstraction desired to describe the situation. Figure 2.3 depicts some of the resolution dimensions present in modeling and simulation:

- System: a model can represent an individual system or a hierarchical composition

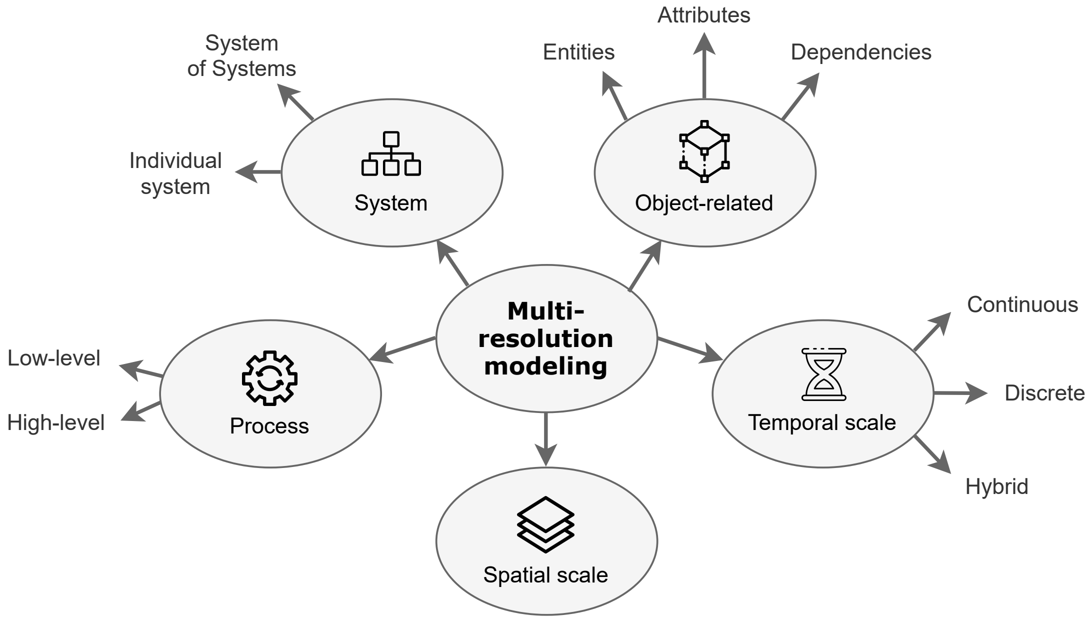


Figure 2.3: Resolution dimensions in modeling and simulation.

of systems. This composition can be performed iteratively, breaking down specific systems, or aggregating external systems.

- Object-related: different resolution can be achieved by altering the entities, attributes, or dependencies of the system, or modifying how they interact among them.
- Process: individual processes can be described with different levels of detail depending on the desired level of abstraction.
- Temporal-scale: a simulation model can be executed with different time bases. Discrete models only change their state variables at certain time instants. In continuous models, the state variables change continuously over time. Hybrid models combine these two approaches. Section 2.3.1 gives more detail about this classification.
- Spatial-scale: different levels of detail can be obtained in the simulation outcomes by changing the operational units of the model.

A complex model usually has a large number of heterogeneous components (usually referred to as agents). The combination of the size of these models and their diversity often leads to large-scale behaviors known as emergent behavior. These complex behaviors, unpredictable from the knowledge of the separate components, play a central role in the theory of complex systems. They usually appear when we extrapolate properties and make predictions of the whole system based on incomplete knowledge of its parts and configurations[12]. As the systems grow, the properties of the entire system become very different from those of its components.

Although the theoretic foundation for modeling emergent behavior has been reviewed by several authors in the literature[? ?], and several tools have been created to facilitate the specification of these kind of phenomena[? ?], reproducing emergent behaviors in artificial environments is still a challenging task. As the models produced in the modeling workflow represent simplified versions of the real systems, the inner omission of some low-level details implies an information loss that may cancel these emerging behaviors. To avoid this, the model has to be supported by strong hypothesis or theoretical constructs and be extensively simulated to check its correctness. However, the computational complexity of the simulator execution may also introduce unintended behaviors that contribute to inaccurate emergent behavior[142]. Therefore, the use of robust verification and validation methodologies and techniques has an critical role when ensuring the reliability of a complex system.

2.1 Advantages of M&S when implementing complex systems

M&S encompasses a great variety of procedures that produce models of phenomenons or infrastructures and simulate their behavior. A model is a simplified representation of the structure and behavior of a particular system of interest. However, as an approximation to the actual system, it should include its main features, be able to process the same inputs, and produce expected outputs. Hence, a good model should be a judicious trade-off between realism and simplicity[158]. Usually, simpler models are created in the early stages of development, evolving and increasing in complexity as the project progresses. Many M&S approaches are not based on a well-defined theoretical framework, and therefore they cannot assume this increasing complexity. According to Castro et al.[229] models tend to be treated as islands of knowledge, that are quickly encapsulated in specialized tools and that are then very difficult to generalize or reuse. This can push M&S in a opposite direction to that required to address the challenges related to interdisciplinary modeling of large projects. In this regard, the use of modeling formalisms represents a solution for this problem, offering clear, reusable, and unambiguous specifications able to deal with the complexity and interaction among heterogeneous subsystems. Some examples are Petri Nets, Markov Chains, or DEVS. The DEVS formalism can be used to represent accurately any discrete event system, and approximate any continuous system with the desired level of detail.

Although these modeling formalisms were born and matured in the academia, companies around the world have already adopted them for the development of their projects. The Model-based Systems Engineering (MBSE) methodology has gained a lot of popularity over the last decade and tends to replace the classical documentation structures with domain models. Moreover, several efforts propose the introduction of executable models in the development workflow[68]. This evolution makes its way from MBSE to the Model & Simulation Based Systems Engineering (M&SBSE), helping to overcome the gap between the system model specification and the respective implementation. By performing simulations with these executable models it is possible to reproduce behaviors present in the system of interest, allowing the study of its correctness and performance based on the outcomes of these executable models. Usually, these models allow simulating large amounts of configurations in manageable times, which would be too expensive or infeasible to execute in the actual system.

In this M&SBSE context, modelers have benefited from classic and new M&S frameworks or toolkits. Some of them implement well-known and robust simulation formalisms consolidated over decades. For instance, we can see M&S tools implementing formalisms as Finite

State Machines (JFLAP[177], jFAST[217], ASSIST[81]), Petri Nets (CPNTools[172], ExSpect[202], GreatSPN[7]), SysML (Astah SysML[209], Modelio[143], Papyrus[55]), or DEVS (CD++[211], DEVSJava[183], PythonPDEVS[25], xDEVS[175]). These tools provide a formal basis while contributing to reduce costs, improve the scalability, and increase the quality of products and systems. Also, several of them provide simple and intuitive Application Programming Interfaces (APIs) for generating executable models using the most popular programming languages.

Nowadays, M&S plays a crucial role in understanding key concepts of complex systems. Among other things, it allows to test hypotheses for feasibility, identify bottlenecks in the flow of entities or information, or introduce several performance metrics for analyzing system configurations from different points of view. Also, simulations allow us to compress time to observe certain phenomena over long periods, or expand time to observe a volatile phenomenon in detail[127]. These advantages have allowed researchers to study with greater detail complex systems or environments, in a great variety of fields. For instance, we can see relevant M&S-related studies in areas like business[150], critical infrastructures[79,157], genetics[51], healthcare[60], or military[41,225], to name just a few.

2.2 Verification and Validation of simulation models

Verification and validation (V&V) techniques are used to guarantee that models are accurate representations of their corresponding systems. Validation is the process of checking if a model meets the needs of the customer and other identified stakeholders. For this, input and output trajectories between the source system (whether real or conceptual) and the model under test must be generated. The validity, whether replicative, predictive, or structural, requires these trajectories to be equal[74]. On the other hand, verification is the attempt to establish that the simulation relation holds between a simulator and a model (i.e. the simulator faithfully implements the model's dynamic behavior). This is generally accomplished through two main approaches: formal proof of correctness and extensive testing[74][182].

Testing consists of the dynamic verification of a program behavior on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior[201]. This procedure usually implies the generation of representative test cases sets. They include the necessary inputs to complete the system under tests and the expected results. Also, they can include prefix and postfix values, to put the system into an appropriate state to receive the input and to prepare the system for the next tests, respectively. Test cases can be defined based on the knowledge of domain experts or can be generated following different coverage criteria. These criteria are usually based on graphs of the behavior of the system, logic relation between its entities, input-space partitioning, or syntax-based methodologies[6]. To contrast these test cases against the actual system, several techniques have been used over the years in the software industry, usually applying different techniques in each activity of the development flow. Some of them imply monitoring the internal states and data flows inside the artifact under testing, and others apply black-box testing. As a result, there exist a great variety of techniques and criteria[103]. However, although some authors have brought these methods closer to the M&S field[92][47], they are not well-established yet. As part of these efforts, we focused on transfer one of these popular techniques, Unit Testing, to an M&S framework. This testing method[95,179] performs a low-level assessment of the units produced in the implementation phase. Hence, it verifies these *units* have the functionality that the developer (or a modeler, in a M&S scope) expects. Usually, once defined, unit tests are executed periodically in an automated way and separated from the actual system. In this way, it allows finding bugs in the early stages, which helps to increase confidence in the software artifact, and facilitates the integration of subsystems. They can be defined both in text or in code, and specify test cases based on pairs of sets of inputs and outputs. These combinations of inputs and their expected outputs should cover the complete behavior of the components. In this way, unit tests easily detect the introduction of changes that affect the expected behavior, alerting the developer about the failed test cases and the affected components. These tests are often defined after the model implementation.

However, they can be developed in parallel or even before the model definition (test-driven development). This growing practice has been widely studied and usually results in defect reduction and quality improvements in the final software artifact[100].

Although this approach can be applied in much of the most common simulation formalisms, it is especially interesting in modular systems like DEVS. The modular and hierarchical nature of DEVS makes the extraction of *units* a straightforward task. The individual behaviors are defined as atomic modules, while the complex functionalities are encapsulated in coupled modules. Both of them have a fixed number of input and output ports, with certain expected data types and formats. Taking advantage of this, we have developed several verification tools during the development of this thesis. Sections 3.3 and 3.4 provide extra details about these contributions.

2.3 M&S techniques to tackle complex systems

2.3.1 Continuous, discrete and hybrid models

Starting from an abstract model, and following the definitions of different modeling formalisms, multiple model artifacts can be generated. These artifacts could differ in terms of the time base, how the time evolves, or the state-space. However, they also usually share a common structure. According to Wymore et al.[220] a casual and deterministic system SYS can be described as:

$$SYS =< T, X, \Omega, Q, \delta, Y, \lambda >$$

where:

- T is the time base.
- X is the input set. It describes the different input values (possibly represented as an product set).

- $\Omega = \omega : T \rightarrow X$ is the input segment set. Each input segment ω represent the values associated with each input $x \in X$ and time $t \in T$.

- Q is the state set. Represent all the unique states in which the system can be.

- $\delta : \Omega \times Q \rightarrow Q$ is the transition function. It determines how the system changes among the different states. For determining a new state, the transition takes into account the previous state and a subset $\omega \in \Omega$.

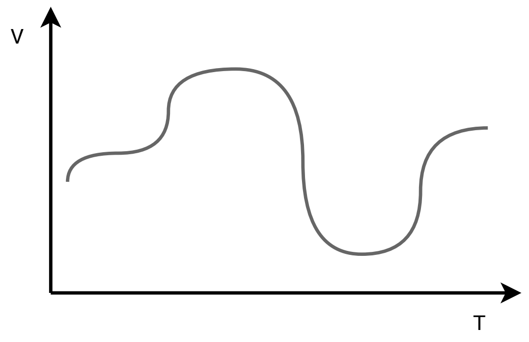
- Y is the output set. It describes the different output values.

- $\lambda : Q \times X \rightarrow Y$ is the output function. It determines when and how the outputs of the systems are generated.

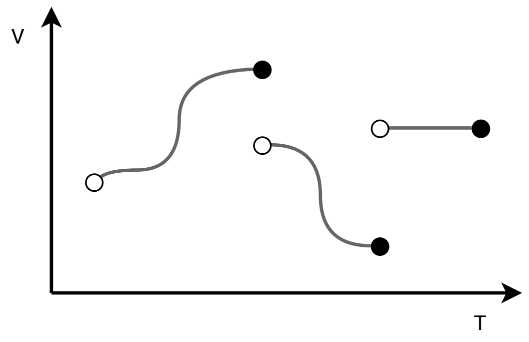
Based on the time basis, we can distingish among different types of models (depicted in Figure 2.4). When $T = \mathbb{R}$ we call them continuous-time models. When $T = \mathbb{N}$, we call them discrete-time models. It is worth noting that some discrete-event models also have $\mathbb{R}$ as the time base, but they only process events in particular time instants. Untimed models as Finite State Automata (FSA) also fall in this category. In these models, the explicit time base is replaced by a notion of progression[206]. These two time categorizations can also be combined, giving rise to the so-called hybrid or discrete/continuous models. In these models, time advances continuously until some specific conditions are met. When it happens, they pause the continuous evolution and go through many discrete states. Only when this process has been finished, the time continues to advance again.

2.3.2 Simulation formalisms

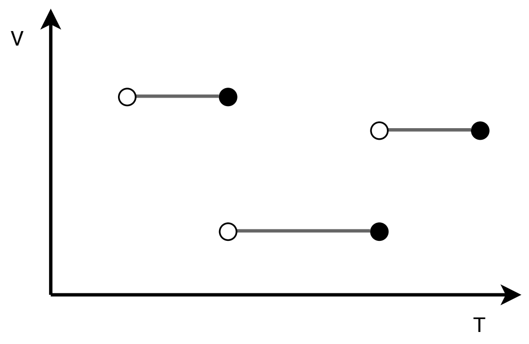
Discrete Event System Specification (DEVS)[230] is a modular and hierarchical formalism presented in 1976 by Bernard P. Zeigler. It is able to describe complex systems based on two types of modules. Atomic modules describe the behavior of the system and respond to a finite set of events, changing their state and producing outputs. Coupled modules reflect



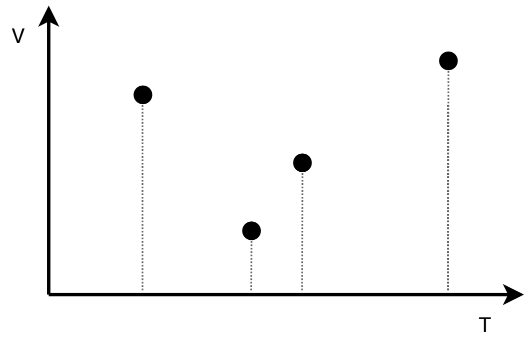
(a) Continuous segment.



(b) Piecewise continuous.



(c) Piecewise constant.

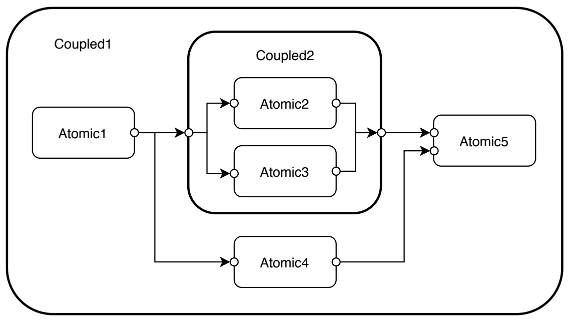


(d) Discrete events.

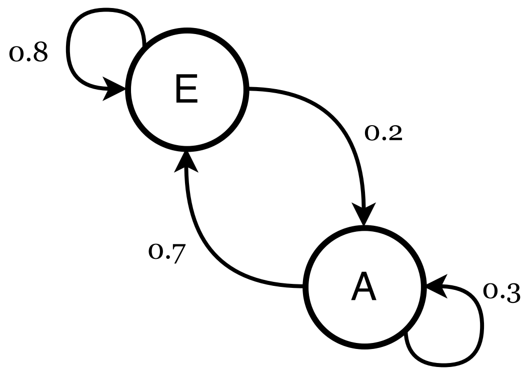
Figure 2.4: Model segment types.

the system structure, grouping other atomic and coupled modules. Both of these module types can have input and output ports, to receive and send information to other modules. These ports are connected by directed couplings, linking pairs of output and input ports. Figure 2.5a depicts a simple representation of this kind of structure. A formal description of this formalism can be found in Chapter 3.

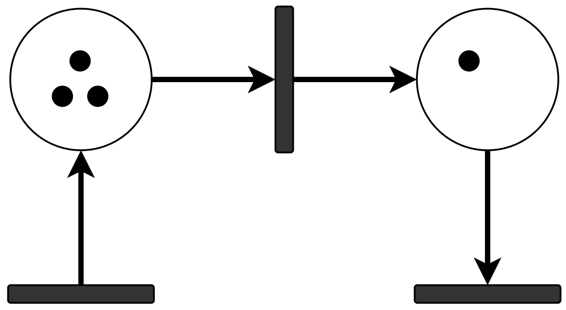
Markov chains, kemeny1976markov were introduced by Andréi Markov in 1906. This kind of model defines discrete stochastic processes where the probability of occurrence of an event only depends on the previous state (what is known as Markovian property). It consists of a finite set of states and a probabilistic definition of transitions. This is depicted in Figure 2.5b. The model starts in a specific state, and in each transition event it evolves in a stochastic manner based on the probability factors specified in the transition arrows. From this idea, multiple variations and extensions have been proposed over time. A popular



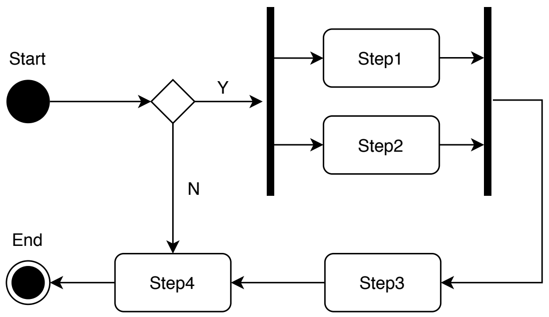
(a) Discrete Event System Specification.



(b) Markov chains.



(c) Petri Nets.



(d) SysML Activity Diagrams.

Figure 2.5: Examples of several modeling specification representations.

one is the Generalized Semi-Markov Process (GSMP), a formalism to describe and analyze discrete-event systems[71].

Petri Nets (PNs) are a discrete-event system model introduced in the early 1960s by Carl Adam Petri[167]. Although it encompasses several types of system models, analysis techniques, and notational conventions, the most used class of PNs is called place/transition nets. These logic models only represent the order in which the events are produced, not the time related to the events. They consist of places, transitions, arcs, and tokens. These tokens are located inside of the places, and the arcs connect pairs of place-transition or transition-place. The places from which an arc runs to a transition are called the input places of the transition, whereas the places to which arcs run from a transition are called the output places of the transition. When transitions are activated, they consume tokens from the input places and generate add tokens to the output places. This activation is produced when there are tokens in all his input places. Figure 2.5c shows a basic example of these nets.

Systems Modeling Language (SysML)[64] is a general-purpose modeling language oriented to systems engineering applications. It was developed extending the Unified Modeling Language (UML), reducing the software-centric restrictions of the UML definitions, and adding new types of diagrams. Through its diagrams, SysML allows capturing both the structure and the behavior of the systems. Although it was originally designed to develop static models, several efforts have been made to create executable SysML definitions and frameworks over the years. For instance, Balestrini-Robinson et al.[16] presented a framework containing a Python programming interface for SysML models and integrate it with OpenMDAO, a multi-disciplinary design, analysis, and optimization framework developed by NASA Glenn Research Center. Kapos et al.[105] propose transforming the SysML metamodel to a DEVS model using the Query/View/Transformation (QVT) language. For this purpose, additional simulation properties and enriched system descriptions are added alongside the SysML basic definition.

2.4 Application fields

As we have seen M&S brings great benefits and is being progressively implemented in a multitude of fields of application. In this section, we focus in two highly coupled fields related to the scenarios developed in this thesis, Internet of Things (IoT) and Healthcare. The impact of these fields is described below, providing some context and stating how they benefit from M&S.

2.4.1 Internet of Things

The Internet of Things (IoT) is a paradigm that groups a variety of physical objects and an infrastructure to connect them and enable their interaction and control. The concept of smart devices was first discussed as early as 1982 at Carnegie Mellon University with the introduction of a versioned Coke machine able to report its inventory and the temperature of the newly introduced drinks[200]. After some years of development of this new idea, Kevin Ashton et al. introduced in 1999 the concept of *Internet of things*[13] to describe a system where the Internet connects to the physical world via ubiquitous sensors. This connectivity empowered computers with mechanisms to gather information from connected devices without human intervention. In this way, it is possible to determine when things need replacing, repairing, or recalling, improving the processes efficiency and, consequently, reducing the inner wastes, losses, and costs.

This idea has been developed over the years[131,214], increasing its presence in our day to day and affecting numerous application fields. Nowadays, IoT comprises a great variety of heterogeneous devices, from simple sensors and actuators to vehicles or buildings. Supporting this growth, multiple IoT protocols have been developed, focusing on optimizing the bandwidth and reducing the latency of the communications between the Internet and the connected devices. It also benefited from the improvements of technologies like Wireless Sensor Networks (WSNs), barcodes, intelligent sensing, RFID, NFCs, or low energy wireless communications[72]. Supported by the large amount of data generated by IoT devices, we have also seen a convergence with multiple technologies like cloud computing, wireless networking, real-time analytics, machine learning, sensors, and embedded systems[58].

IoT is still in continuous development and a common consensus over the IoT definitions and standards is yet to be achieved. However, the IoT is currently going through a phase of rapid growth. According to Cisco Research, there will be 27.1 billion networked devices by the end 2021, up from 17.1 billion in 2016[42]. Globally, there will be 3.5 networked devices per person. IDC estimates that these numbers will continue growing, reaching 55.7 billion connected devices worldwide by 2025, 75% of which will be connected to an IoT platform. They estimate data generated from connected IoT devices to be 73.1 ZB by 2025, growing from 18.3 ZB in 2019[98]. Consistently with this increase, the global IoT market is expected to reach a value of $1.256 trillion by 2025 from $0.690 trillion in 2019 at a compound annual growth rate of 10.53%, during the period 2020-2025[124].

In the coming years, the number of IoT applications and services is expected to grow at a great pace. As a consequence, multiple application domains will be affected by this technology. Some domains where IoT is already helping to boost the efficiency of the inner processes are agriculture, healthcare, smart cities, and nano-scale applications[66]. Smart cities are expected to be great contributors to this trend, paving the way for the introduction of intelligent and unmanned transportation[196] and optimized deliveries[213], and optimizing their infrastructures[17,52,129]. Improved industrial monitoring and automation techniques will also help to minimize failures, reduce waste, and optimize processes. In healthcare, accurate patient monitoring and pharmaceutical management, added to the prediction of risk factors in highly-impact diseases, will result in huge cost savings. These predictions will be facilitated by the implementation of WSNs where patients with specific target diseases wear unintrusive devices that allow to continuously monitoring patients state, registering variables like heart rate, ElectroCardioGram (ECG), ElectroDermal Activity (EDA), ElectroEncephaloGram (EEG), or peripheral oxygen saturation (SpO2). These variables, related to the Autonomic Nervous System (ANS), can be modeled to relate them with specific diseases symptoms or outcomes, generating useful predictions for patient diagnosis and treatment[5,160].

These scenarios come accompanied by a need for storage infrastructures and computing capabilities. This trend has led to the increasing use of data centers to store and process data traditionally located in endpoints. Cloud computing is one of the enabling platforms to support these needs. The National Institute of Standards and Technology (NIST)[136] defines cloud computing as "*...a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*". Armbrust et al.[61] simplify this concept defining *cloud* as the "*datacenter hardware and software that provide services*". However, in addition to raw computing and storage, cloud computing providers usually offer multiple software services, APIs, and development tools that allow developers to build seamlessly scalable applications upon their services[210]. This Infrastructure as a Service (IaaS) model approaches large and powerful cloud infrastructures to individuals, allowing them to develop solutions to simplify and optimize different aspects of everyday life through a new range of innovative services and applications. Through its use, it is possible to easily improve the performance of systems, making use of these storing and processing capabilities, and reduce systems overall cost.

However, there are also some drawbacks when using these services. Some applications may not assume the latency derived from the intermediate communications or being restricted by security or privacy concerns. In these cases, Fog Computing is often used. This paradigm tries to solve these aspects by approaching these infrastructures to end-users and comes with several advantages. In addition to the lower latencies derived from the closer infrastructures, it enables the use of mid-range IoT protocols and helps to reduce problems related to bandwidth bottlenecks. Also, the reliability of the connection is increased, due to the existence of multiple interconnected channels. Some examples of fog nodes include industrial controllers, Micro Datacenters (MDCs), and video surveillance cameras.

These two paradigms can also be combined in more complex scenarios, benefiting from their individual advantages. Figure 2.6 depicts a typical architecture of a Cloud-Fog scenario. The final devices often establish a connection with the nearest Fog node, although they can also connect to alternative ones if the main node is down or overloaded. The application requests requiring a fast response are usually processed in these nodes. They usually have moderate computational power, and can also operate as a bridge to connect different types of devices and applications.

When the request involves less urgent processing or requires higher computational power, Fog nodes can opt for sending them to the upper Cloud layer. In this layer, we usually find more powerful processing units and a higher amount of services. This load distribution is usually a key aspect of Cloud-Fog scenarios[149,156], being able to adapt it according to the

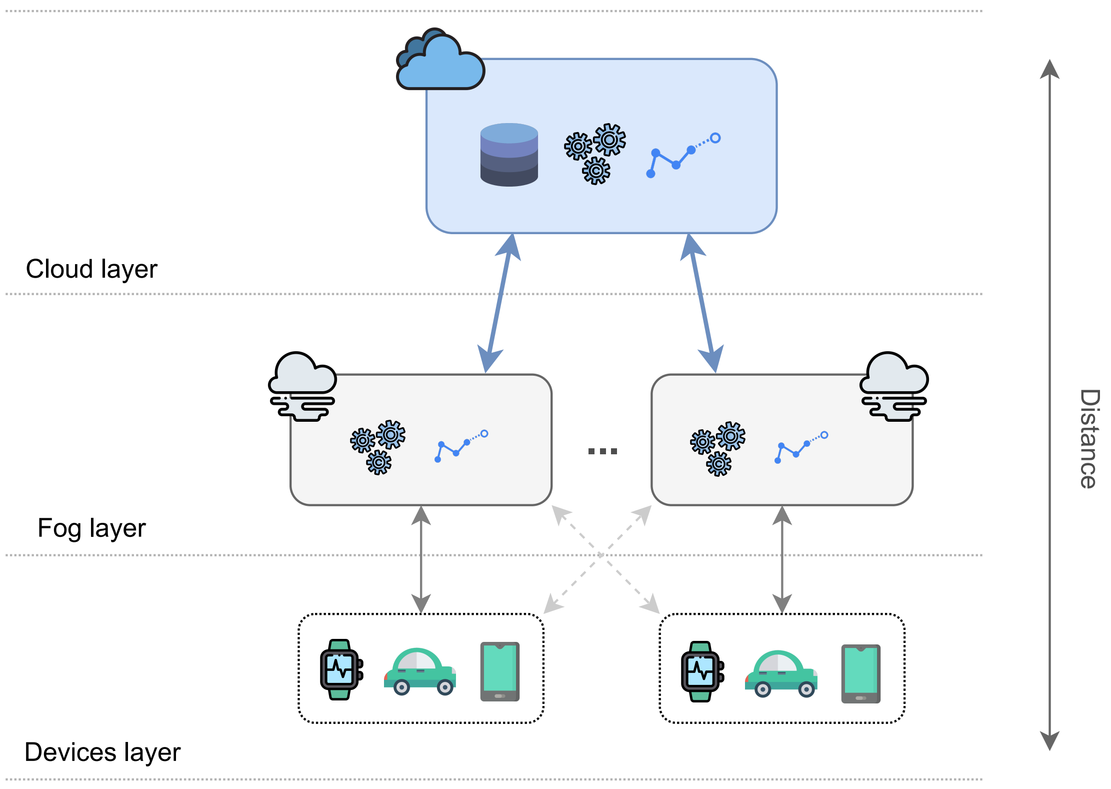


Figure 2.6: Arquitecture of Edge/Fog/Cloud scenarios.

goals of usability, energy consumption, and scalability of the system.

Such complex scenarios make IoT a great M&S application field. In the last years, a good amount of research has been done in this regard. Multiple frameworks have been implemented for defining IoT scenarios from different perspectives. We have simulators focusing on the different computing layers, covering Cloud computing[29,53,151], Fog computing[32,76,116], or even Edge Computing[192]. Some of them were developed focusing on specific types of processing, as Map-Reduce[104,122]. Others are centered in specific types of architectures or applications, as federated networks[112], or data stream-oriented IoT applications[32]. These and other IoT frameworks have been used for analyzing a large variety of scenarios. For instance, Mahmoud et al.[126] used the iFogSim framework for proposing energy-aware allocation strategies for placing application modules on Fog devices, comparing the resulting performance with Cloud-based solutions. Sarkar et al.[184] also used this simulator to present a fog based intelligent surveillance system able to continously process the images recorded by the security cameras to find specific events. Several parameters are used to compare the scenario with an alternative cloud architecture, including network usage, energy consumption, latency, and execution cost. Gudi et al.[75] propose a novel architecture to improve the human-robot communications in industrial environments by moving the processing of data from Cloud to Fog, reducing notably the latency of communications.

2.4.2 Healthcare

Several reasons made healthcare a suitable application field for applying M&S techniques. First, healthcare systems are characterized by uncertainty and variability, often requiring a stochastic approach. Also, they are highly complex, so they benefit from modeling approaches that can deal with this complexity. Thirdly, the key role played by human beings in healthcare systems requires an approach that allows interaction and communication between modeler and user or client[224]. Although M&S has been used for modeling healthcare systems for over forty years, traditionally has had a considerably lower adoption than in other sectors such as business and commerce, aerospace, and the military. In 2009, Brailsford et al.[26] compared the use of M&S in these application fields, determining that only 8% of papers from healthcare reported analysis of a real problem with high levels of user engagement, compared to 36.5% in the defense literature, and 48.9% in commerce. It was in the last two decades when we have seen a significant scale of research in this application field, especially after 2010. This research uses Discrete-Event Simulation (DES) as the main simulation approach, although other approaches such as System Dynamics (SD), Monte Carlo, and agent-based have also been used[233].

The healthcare models can be classified into four main categories: Health and Care Systems Operation (HCSO), Disease Progression Modeling (DPM), screening modeling, and Health Behavior Modeling (HBM). Zhang et al.[233] concluded that among all the DES healthcare models developed up to 2016, 65% corresponded to HCSO-related studies. The remaining models corresponded mainly to DPM (28%), leaving only 5% for screening modeling and 2% for HBM[233]. Moreover, the most frequently analyzed medical indications are related to the circulatory system (18%), nervous system (15%), neoplasm (15%), and musculoskeletal system diseases (13%)[233].

Regarding health and care systems operation, the major part of the models (68%) represented systems corresponding with single health care units. In this category, there also existed a high interest in modeling systems such as emergency workflows, intensive care units, or health service providers. Disease progression modeling focuses on determining how diseases evolve and assist in medical decision-making. It helps to compare different treatment alternatives at a medical level in terms of resources consumed and health outcomes[44]. An example of this is Sukkar et al. research[54], which categorized Alzheimer patients into several severity levels using regression techniques. In this way, they were able to obtain information regarding performance over time on cognition, global performance, and activities of daily living. Another example is given by Holford et al.[91], who modeled the Unified Parkinson's Disease Rating Scale (UPDRS) scores collected in 800 subjects followed for 8 years, and tested the effectiveness of a specific treatment over new patients using this model as a reference. Among the screening-related modeling, most of the models correspond to breast cancer screening[43,193]. In this category also laid studies concerning other types of cancer[31,56], tuberculosis[30,82], and diabetes[48], among others. In terms of research goals, the screening modeling is used for diagnosis, assessment, or routines performance measurement. Health behavior modeling covers topics such as community organization, communication, diffusion of innovations, social marketing, information processing, stress and coping, relapse prevention, and empowerment[173]. They are mostly developed over the last decade and incorporates human behavioral factors into modeling practice. To this category belong several approaches that try to develop behavioral-based strategies to stimulate people to quit smoking[67,99].

Even though the use of M&S in healthcare has increased considerably in recent years, there is still much to be done. Among all the opportunities that have arisen in this field, personalized medicine is one of the use cases that is expected to have a greater impact on our society. In these scenarios, patients are continually monitored and different models and decision support systems are generated through the use of techniques as the BigData and Machine Learning, supporting the clinician in their diagnosis and treatment decisions. This topic is addressed with special intensity throughout this thesis.

In this chapter, several fundamental aspects of M&S were covered. First, the model life-cycle was explained, explaining the different types of models and their related V&V activities. Also, the advantages of M&S for the development of complex systems were discussed, introducing some common model-driven methodologies and simulation formalisms. This dissertation was complemented with an explanation of crucial aspects in the M&S field, as the different time bases used by simulation formalisms, an overview of how the complexity of a model can increase through several resolution dimension giving rise to Multi-Resolution Modeling (MRM), and the importance of verification and validation of models when creating reliable and robust systems. Finally, we summarized some opportunities that M&S is bringing to two application fields highly related to the contributions of this thesis, the Internet of Things (IoT) and Healthcare. A brief context was given for each of them, explaining their main features and particularities, and exemplifying the potential benefits that M&S is bringing to these fields through the enumeration of several related research projects.

Chapter 3

Discrete Event System Specification

This chapter focuses on the DEVS formalism, since it has been adopted as the driving force in the different M&S paradigms tackled in this thesis. We also describe the our contributions regarding DEVS performance analysis and model verification. DEVS is a modular and hierarchical formalism for discrete event systems modeling[230] introduced in 1976 by Bernard P. Zeigler, emeritus professor at the University of Arizona. Based on Set theory, it provides a framework for information modeling which gives different advantages to analyze and design complex systems, such as completeness, verifiability, extensibility, and maintainability. The parallel DEVS formulation (PDEVS) was introduced by Zeigler et al.[39] in 1994 as an extension of the original DEVS protocol that introduced several improvements concerning the definition of concurrent systems. Due to the high adoption of this updated version of the formalism, the term DEVS is often used to refer to PDEVS formalism. This also applies to the rest of this thesis, unless otherwise specified.

Over the years, the theory around this discrete-event formalism has been developed, emerging new formalism extending the capabilities of the original DEVS definition, and different M&S frameworks. Some examples of DEVS-based formalisms are Cell-DEVS[212], which combines the definition of Cellular Automata and DEVS for defining grid-shaped cellular models based on a set of rules, or DynamicDEVS[94], that addresses the dynamic adaptation of the models as a response to changes in the environment. Several simulators have been extensively used in the last decades for modeling and simulating systems using all these formalisms. They allow to easily implement models described in terms of the DEVS theory to a computational model using custom libraries. Often, they also include additional tools that simplify the definition, debugging, and visualization of DEVS models. Some examples are GUIs to graphically design the models, powerful logging and transducers systems, plugin support to implement new formalisms, simulation algorithms, and even some V&V tools to implement different types of tests and improve the reliability of the models. However, although these frameworks are based on the same DEVS definitions, the specific implementations can differ significantly among them in terms of capabilities and performance. This makes more important the initial simulator choice, since porting the models between simulators is a time and resource-consuming task[197]. For this reason, several efforts comparing the performance of the different publicly available DEVS simulators have been presented, providing useful insight about which simulators are good choices depending on factors like the inner complexity of the model structure.

This chapter is organized as follows. Section 3.1 gives a bit of context about the definition of DEVS models, the main simulators of this formalism, and the importance of measuring the engines' performance. Section 3.2 explains in detail the architecture and features of the xDEVS simulator, the main simulator on which we have focused our efforts. The next section explains some of the contributions made in regard to providing robust verification and performance measurement tools to DEVS simulators. Section 3.3 details the architecture and implementation of a Unit Testing tool included in the Java branch on the xDEVS framework. Section 3.4 describes the implementation of a constraint-based verification methodology implemented in the DEVS simulation layer. Finally, Section 3.6 reports a revision that we recently performed in the DEVStone benchmark, describing a metric to evaluate the performance of Discrete-Event Simulators and overcoming the lack of a common model set of the original definition.

3.1 Background

First, a formal definition of DEVS model types and structure is provided in Section 3.1.1. Second, Section 3.1.2 introduces the most popular DEVS-based M&S frameworks, placing special emphasis on xDEVS, a framework used in most of the contributions of this thesis.

3.1.1 DEVS models definition

There are two types of DEVS models, atomic and coupled. Atomic models represent the behavior of the system. They process the input events based on their current states and condition, generating output events and transition to new states. Coupled models aggregate two or more atomic and coupled models and connect them through explicit couplings. An atomic model can be formally defined by the following equation:

$$A = \langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \delta_{\text{con}}, \lambda, ta \rangle \tag{3.1}$$

where:

- X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$.
- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$.
- S is the set of states.
- $\delta_{\text{ext}} : Q \times X^b \to S$ is the external transition function. It is automatically executed when an external event arrives to one of the input ports, changing the current state, if needed.
  - $Q = (s, e) s \in S, 0 \leq e \leq ta(s)$ is the total state set, where e is the time elapsed since the last transition.
  - X^b is the set of bags over elements in X.

- $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function. It is executed right after the output (λ) function and is used to change the state S.

- $\delta_{\text{con}} : Q \times X^b \rightarrow S$ is the confluent function, subject to $\delta_{\text{con}}(s, ta(s), \emptyset) = \delta_{\text{int}}(s)$. This transition decides the next state in cases of collision between external and internal events, i.e., an external event is received and elapsed time equals time-advance. Typically, $\delta_{\text{con}}(s, ta(s), x) = \delta_{\text{ext}}(\delta_{\text{int}}(s), 0, x)$.

- $\lambda : S \rightarrow Y^b$ is the output function. Y^b is the set of bags over elements in Y. When the time elapsed since the last output function is equal to $ta(s)$, then λ is automatically executed.

- $ta(s) : S \rightarrow \Re_0^+ \cup \infty$ is the time advance function.

One the other hand, the formal definition of a coupled model is described as:

$$M = \langle X, Y, C, EIC, EOC, IC \rangle \tag{3.2}$$

where:

- X is the set of inputs described in terms of pairs port-value: $\{p \in IPorts, v \in X_p\}$.

- Y is the set of outputs, also described in terms of pairs port-value: $\{p \in OPorts, v \in Y_p\}$.

- C is a set of DEVS component models (atomic or coupled). Note that C makes this definition recursive.

- EIC is the External Input Coupling relation, from external inputs of M to component inputs of C.

- EOC is the External Output Coupling relation, from component outputs of C to external outputs of M.

- IC is the Internal Coupling relation, from component outputs of $c_i \in C$ to component outputs of $c_j \in C$, provided that $i \neq j$.

Given the recursive definition of M, a coupled model can itself be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

The main simulation algorithm of DEVS usually considers two main issues, time synchronization and message passing. Usually, the `Simulator` keeps track of the next event time registered in its atomic model, while each `Coordinator` gives access to the minimum next time among all its children simulators and coordinators. Using this approach, the main loop can easily find out the nearest event time scheduled by the different child models of the system. In this way, it is possible to improve the simulation performance by ensuring that each iteration points to a simulation time associated with the next events to be processed. The message passing usually takes advantage of the ports and couplings definition. The different `Atomic` models, depending on their state and inputs, may generate outputs for one or several output ports when executing their output function (λ). As part of the simulation loop, after executing the suitable events of a specific iteration, the values contained in these output ports are propagated to the input ports of the destination models specified by the couplings. This process is performed in two steps to assure correct propagation through the hierarchy of child models. First, the values corresponding to external output couplings (EOC) are transmitted, using a bottom-up perspective (i.e. the models present in the leaves of the hierarchy tree are considered first, and then going through the models in ascending order until reaching the root model). After this, the EIC and IC couplings are executed, using an top-down approach.

3.1.2 DEVS simulators: related work

There are several DEVS-based simulators in the state of the art, implemented in different programming languages and with different capabilities. In the following we present briefly some of the most popular, describing them and pointing out their distinctive points.

aDEVS (a Discrete EVent System simulator)[152] is a C++ library for constructing discrete-event simulations based on the Parallel DEVS and Dynamic DEVS (dynDEVS) formalisms. It was developed by James J. Nutaro, from the Oak Ridge National Laboratory (ORNL). In previous papers, this simulator has shown very good performance[175 77].

CD++[211] is a toolkit for Discrete-Event modeling and simulation created at the Advanced Real-Time Simulation Laboratory (ARS) of the Carleton University, supervised by Gabriel Wainer. It allows the definition of atomic models through the use of a C++ API. Coupled models are defined using a custom specification language, that also can be used to define Cell-DEVS models. CD++ allows the creation of DEVS models in a graphical way using the CD++Builder Eclipse plugin. It supports classic, parallel, real-time, embedded (E-CD++), and distributed simulations. A separate implementation of CD++ was created, called **CDBoost**[208]. It uses the C++ popular set of libraries Boost to optimize its performance. Although this provides good execution times, only one port per module can be used and only one data type can be used for all the messages of the model. To solve this issue they presented **Cadmium**[22], a new C++17 compliant and template-based simulator, that provides support for the PDEVS and Cell-DEVS formalisms and solves the aforementioned issues. It has integrated features to facilitate model verification and supports multiple time and message data types.

DEVSJava[183] is a Java library developed under the supervision of B.P. Zeigler, H. Sarjoughian and R. Lysecky. It provides an API for defining and simulating PDEVS, Dynamic-Structure DEVS, and Real-Time DEVS models. The current version, DEVSJava 3.1, supports some additional features as automatic pruning of system entity structure, modeling of differential equation systems, and cellular spaces. Moreover, several platforms have been developed extending DEVSJava. A significant one is DEVS-Suite[110], an M&S software that provides a GUI to define, execute, and visualize results of PDEVS and CA systems.

James-II[90] is a plugin-based framework created at the University of Rostock's Modeling and Simulation Research Group. It supports some DEVS variants (Classic, Parallel, and Parallel Dynamic DEVS), and some additional formalisms as π-calculus, cellular automata, reaction models, and reaction-diffusion models. It is written in Java and provides a GUI to define the models and run experiments. New features can be written through the development of plugins (as modeling formalisms, simulation algorithms, visualizers, etc.).

PowerDEVS[23] is a general-purpose M&S tool for discrete event system specification (DEVS) oriented to the simulation of hybrid systems. It allows to specify the behavior of atomic models in C++, while the hierarchy of models is defined in a graphical way. Based on this, the environment generates the code for the entire model and executes the simulation. PowerDEVS also allows real time simulations and it is compatible with the Scilab numerical package.

PythonPDEVS[25] is a modeling and simulation package developed by the Modelling, Simulation and Design Lab (MSDL) of the University of Antwerp, headed by Prof. Hans Vangheluwe. It is implemented in Python and supports the Classic, Parallel, and Dynamic Structure DEVS formalisms. It also allows us to perform distributed simulations, using Message Passing Interface (MPI) as middleware. DEVSimPy, a graphical interface developed at the University of Corsica, wraps this package and simplifies the PythonPDEVS model design.

xDEVS[175] is a multi-language M&S framework extensively used in the contributions of this thesis. Section 3.2 provides a detailed description of xDEVS, describing the main architecture of its DEVS simulator and several of the tools included in the framework to improve the robustness of the models and facilitate their verification.

Table 3.1 summarizes this information, comparing the programming languages, release year, supported DEVS formalisms, and their additional features.

Simulator	Programming Language	Release Year	Supported DEVS Formalisms	Additional Features
aDEVS	C++	1999	PDEVS, DynDEVS	Support for using QEMU and uCsim.
Cadmium	C++	2017	PDEVS, CellDEVS	Model validation at compile time.
CD++	C++	2002	CDEVS, CellDEVS, PDEVS, RT-DEVS	It can be emedded in single-board computers (E-CD++). Graphical eclipse plugin.
DEVSJava	Java	1998	PDEVS	Modeling of differential equation systems, and cellular spaces.
James-II	Java	2009	PDEVS, FDDEVS	Support for other formalisms as CA, attributed Pi Calculus, or ML-Rules. Plugins system. GUI.
PowerDEVS	C++	2003	PDEVS	Real time simulations. Scilab compatibility. GUI.
PythonPDEVS	Python	2001	CDEVS, PDEVS, DynDEVS	Distributed simulations. GUI,
xDEVS	C++, Java, Python	2014	PDEVS	Distributed simulations, unit testing, constraints verification layer, shared-memory ports, interoperability wrappers.

Table 3.1: Summary of DEVS-based simulators features.

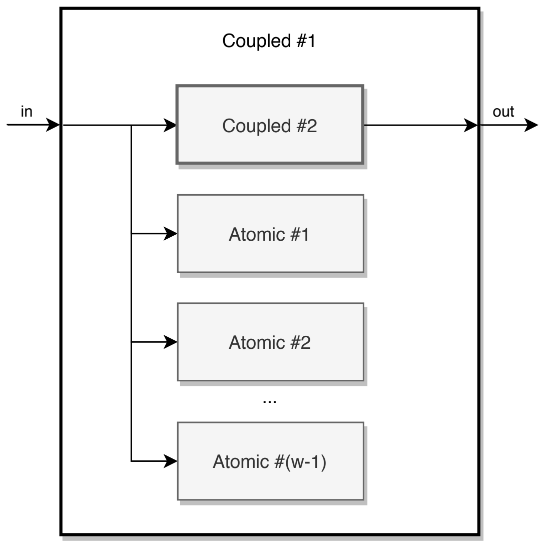
3.1.3 Performance measurement and optimization

Modeling and Simulation has been used for decades for studying and analyzing a great variety of systems, simplifying the development workflows, and reducing the overall cost of the projects. Nowadays there exist a wide variety of modeling techniques and tools, with different possibilities and specific formalism support[228]. With such a diversity of tools and formalisms, there are usually several suitable options that fit our interests and needs. The particular choice depends on different variables, such as performance, complexity, or programming language support. However, it tends to be a significant decision, as performance among tools can differ by orders of magnitude, and changing between them is usually a tough and time-expensive task. This is mainly due to the lack of interoperability standards, which tie modelers with their legacy modeling tools[197]. In this regard, some state-of-the-art proposals have been presented to provide mechanisms to enhance the compatibility and reusability of the models[145,188]. However, porting models between popular M&S frameworks nowadays keeps bringing the need for rewriting the models from scratch.

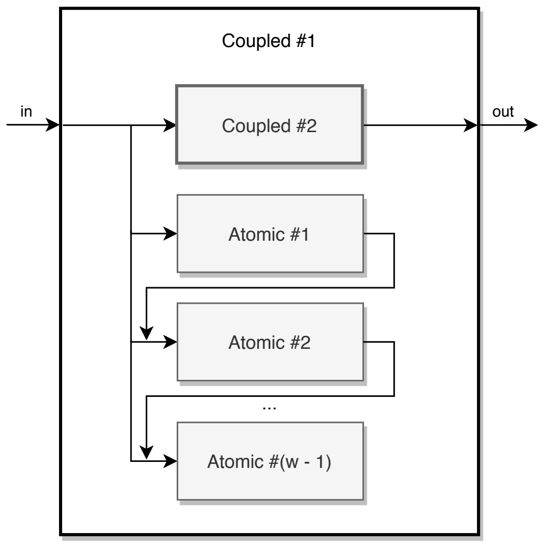
Since the definition of the DEVS formalism, the performance of DEVS simulators has been a recurring research topic. One of the first proposals to improve their performance was the Parallel DEVS (PDEVS) formalism presented Chow and Zeigler[39], opening the doors to DEVS parallel-executed simulations. Since then, several additional proposals were presented. DEVS-C++[227] is a high-performance environment that focuses on modeling large-scale systems with high resolution. The authors simulate a watershed with different degrees of detail to compare the speedup of using this tool on different High-Performance Computing (HPC) clusters. Hu et al.[93] proposed an alternative simulation engine for improving the performance of large-scale cellular DEVS models by implementing a data structure that allows less time-consuming searches of active models. The performance analysis consists of a comparison between state-of-the-art simulation engines and the new approach when simulating two examples. Muzy et al.[147] detected that the classical implementation of DEVS simulators could lead to memory inefficiencies resulting from an excessive number of nodes[147]. The authors propose different simulation algorithms to overcome these deficiencies. To illustrate the obtained speedup, each of these contributions included a performance comparative between previous simulation engines and simulation engines that implemented the proposed algorithms. These comparisons measured the time required to simulate an arbitrary model with a considerable degree of complexity. However, each contribution used a different model to illustrate this performance enhancement. Depending on the model under study, the number of events, couplings between components, and processing time of the transition functions can vary significantly. Thus, a performance comparative taking into account only one model is not enough to evaluate a DEVS simulation engine. Additionally, as each contribution used a completely different model to illustrate its performance enhancement, it is not possible to compare the performance enhancement achieved by these contributions. The DEVStone benchmark[70] was introduced in 2005 to overcome these limitations. In the following, we describe this benchmark, presenting a brief overview of its model types, and how they can be parameterized to vary its size and complexity.

The DEVStone benchmark

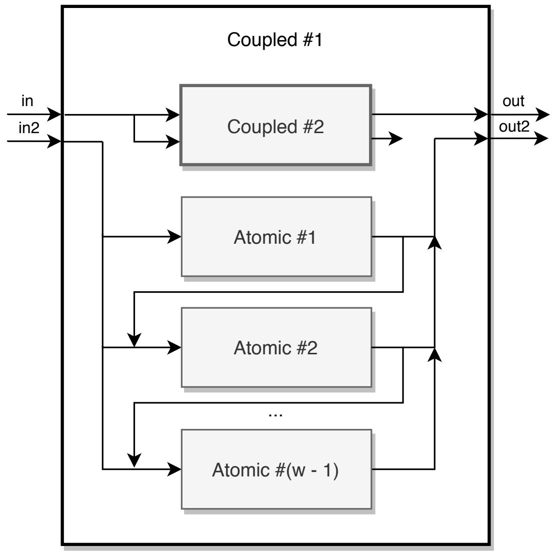
DEVStone[70] is a synthetic benchmark devoted to automating the evaluation of DEVS-based simulation approaches. It allows the generation of different types of models, each of them specialized in measuring specific aspects of the simulation. This benchmark has become popular over the years, and has been used by many researchers to evaluate and compare the performance of different DEVS simulators[175, 63, 204]. DEVStone describes several synthetic models that can be configured to vary their size and complexity. With this aim, they present a recursive structure with configurable depth where all the levels contain equivalent components and interconnections. The customization of the models is done through the use of four parameters: (i) *width*, that affects to the number of components per layer, (ii) *depth*, that specifies the number of nested coupled models, (iii) *internal transition delay*, and (iv) *external transition delay*. These two types of delays execute CPU-intensive operations during a fixed amount of time in the internal and external events of the atomic components.



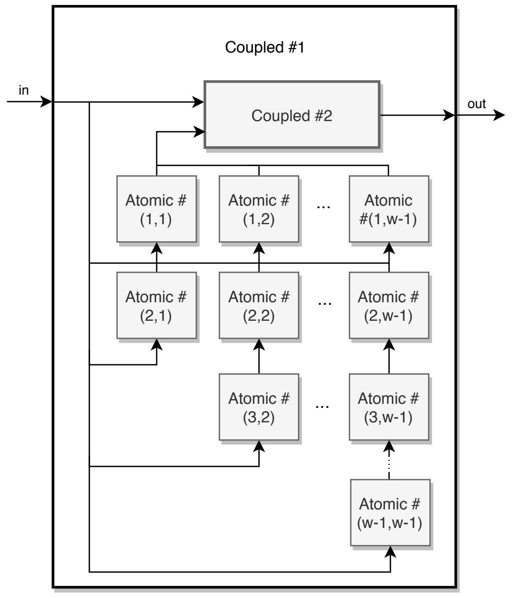
(a) Low level of Interconnections model (LI).



(b) High Input couplings model (HI).



(c) HI model with numerous Outputs model (HO).



(d) Exponential level of coupling and outputs model (HOmod).

Figure 3.1: DEVStone models internal structure.

DEVStone describes four types of models (depicted in Figure 3.1). In the following we describe its basic features and show how they scale depending of the specified depths and widths, including the number of atomic models, External Input Couplings (EIC), Internal Couplings (IC), External Output Couplings (EOC), and events. As the number of internal, external, and output events match for DEVStone models, these events indicate the number of executions of each of these event groups.

- **LI** models: are the simplest models, with a low level of interconnections in their coupled models.

$$\#Atomic = (w-1)*(d-1)+1 \tag{3.3a}$$

$$\#EIC = w*(d-1)+1 \tag{3.3b}$$

$$\#IC = 0 \tag{3.3c}$$

$$\#EOC = d \tag{3.3d}$$

$$\#Events = (w-1)*(d-1)+1 \tag{3.3e}$$

- **HI** models: similar to LI models, but increases the number of internal couplings.

$$\#Atomic = (w-1)*(d-1)+1 \tag{3.4a}$$

$$\#EIC = w*(d-1)+1 \tag{3.4b}$$

$$\#IC = \begin{cases} (w-2)*(d-1) & if w > 2 \\ 0 & otherwise \end{cases} \tag{3.4c}$$

$$\#EOC = d \tag{3.4d}$$

$$\#Events = 1+(d-1)*(w-1+\sum_{i=1}^{w-2} i) \tag{3.4e}$$

- **HO** models: variation of the HI models where all the atomic components in each coupled module are connected to the coupled output port. It is worth noting that these models present unconnected ports that may serve to detect malfunctioning in the simulators when cleaning the values of ports without couplings.

$$\#Atomic = (w-1)*(d-1)+1 \tag{3.5a}$$

$$\#EIC = (w+1)*(d-1)+1 \tag{3.5b}$$

$$\#IC = \begin{cases} (w-2)*(d-1) & if w > 2 \\ 0 & otherwise \end{cases} \tag{3.5c}$$

$$\#EOC = w*(d-1)+1 \tag{3.5d}$$

$$\#Events = 1 + (d-1)*(w-1+\sum_{i=1}^{w-2} i) \tag{3.5e}$$

- **HOmod** models: it reproduces an exponential level of coupling and outputs model.

$$\#Atomic = (w-1+\sum_{i=1}^{w-1} i)*(d-1)+1 \tag{3.6a}$$

$$\#EIC = (2*(w-1)+1)*(d-1)+1 \tag{3.6b}$$

$$\#IC = ((w-1)+(w-1)^2+\sum_{i=1}^{w-2} i)*(d-1) \tag{3.6c}$$

$$\#EOC = d \tag{3.6d}$$

$$\#Events = 1+\sum_{i=1}^{d-1}(1+(i-1)*(w-1)+\sum_{j=1}^{w-1} j+(w-1)*(w+(i-1)*(w-1))) \tag{3.6e}$$

3.2 xDEVS M&S framework

xDEVS[175] is an object-oriented M&S framework developed in the Department of Computer Architecture and Automation at the Complutense University of Madrid. Although xDEVS started as a basic DEVS implementation in Java, nowadays it counts with three equivalent implementations for the C++, Java, and Python programming languages. Moreover, xDEVS includes several additional features to facilitate the development and fine-tuning of models. Some examples are model flattening, a wrapper system to facilitate the interoperability of models, and some verification tools. During the development of this thesis, several contributions have been made to this framework. The Python branch has been created from scratch, providing it with parallel and distributed simulation mechanisms. Also, we have developed a new simulation algorithm that reduces the overhead introduced by traditional implementation approaches. This is achieved by replacing the typical message propagation present in DEVS simulators with a shared-memory approach, where the input ports directly access the source message bags present in the output ports to which they are connected. This approach has been included in the Python simulator, comparing both approaches and showing a great performance improvement. Moreover, we introduced several tools that allow the verification of models from different perspectives, including unit testing, a constraint-based simulation layer, and a metamorphic framework.

xDEVS aims to offer a common interface for developing DEVS models in several of the most popular programming languages. Therefore, a modeler who has developed models in one of these xDEVS simulators can easily adapt them or create new models in the other simulators offered by this framework. To this end, xDEVS incorporates several wrappers that allow the interaction with atomic components defined for other simulators. This interoperability effort tries to alleviate the heterogeneity of DEVS implementations and provides a way to create models reusing components developed in different DEVS-based simulators. Around this comprehensive DEVS M&S engine, we have added some utilities to improve performance, as well as alleviate the maintainability and scalability in the creation of complex models. This framework is available under the GNU Lesser General Public License v3.0 and provides support for specifying and running classic, parallel, real-time, and distributed DEVS simulations. It organizes the object-based components in a hierarchical way and provides easily interchangeable coordinators for executing different types of simulations. The different APIs keep a very similar structure and nomenclature, allowing to switch among the different implementations without additional learning time investments.

Following the DEVS formalism, xDEVS presents a clear separation between the modeling and simulation layers. A class diagram showing the relationship between these modeling and simulation layers is shown in Figure 3.2. Although this diagram shows the general structure of the simulator, there may be slight differences among the existing implementations. They also come with some additional features, described in Table 3.2. These features have been implemented based on the requirements of different projects, and are progressively being translated to all the xDEVS branches.

Table 3.2: Additional features included in the xDEVS simulator

	xDEVS C++	**xDEVS Java**	**xDEVS Python**
Constraints definition[87]	✓		✓
Distributed simulations		✓	✓
Shared-memory ports			✓
Model flattening	✓	✓	✓
Unit testing[86]		✓	

DEVS models in xDEVS are created using two main components. `Atomic` components define the behavior of the system. `Coupled` components contains other `Atomic` and `Coupled` components, creating a model hierarchy. Both of them have `Ports`, that represent input/output information points. To link two components of the model a `Coupling` can be created, selecting the source and destination `Ports`. The information of `Couplings` is contained in the `Coupled` elements that wrap the ports to be linked.

The simulation layer is based on the concept of the Abstract Simulator. Following this concept we divide the simulation entities in `Simulators` and `Coordinators`. Each `Simulator` is related with an `Atomic` component. Each `Coordinator` synchronizes their child `Simulators` and `Coordinators`. This results in an equivalent hierarchy to the one described for the modeling layer for structuring Atomic and Coupled models. Also, the `Coordinator` children management changes in the different types of execution provided in xDEVS: sequential, parallel, and real-time. These specific behaviors are defined as extensions of the base `Coordinator` class. Table 3.3 summarize the coordinators available in each of the xDEVS branches.

Figure 3.3 depicts how these layers are structured using the Experimental Frame - Processor (EFP) model as an example. The EFP model is a common reference model where a

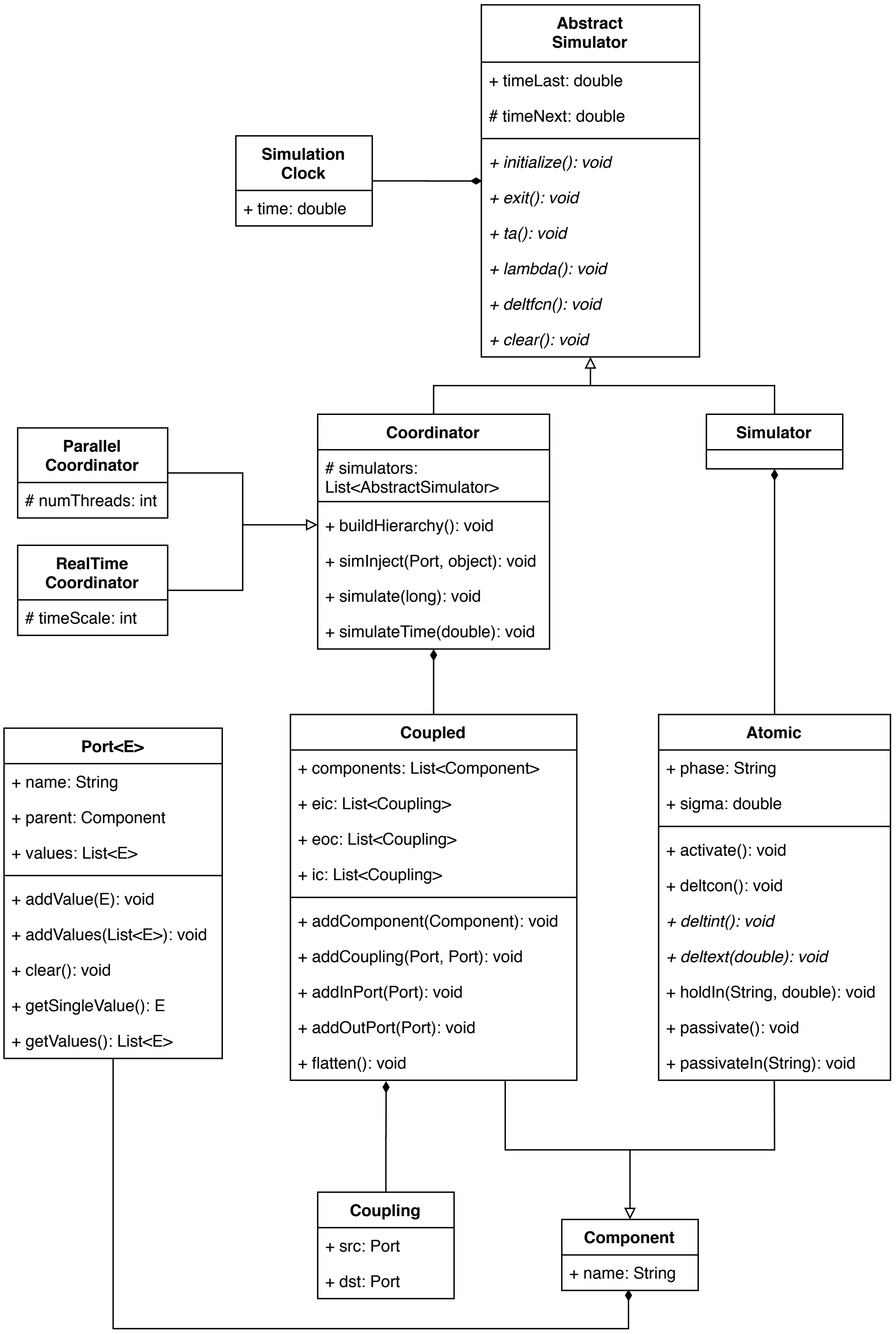


Figure 3.2: Class diagram of the xDEVS architecture.

Table 3.3: Coordinators available in the different xDEVS implementations.

	xDEVS C++	**xDEVS Java**	**xDEVS Python**
Coordinator	✓	✓	✓
Parallel Coordinator		✓	✓
Real-Time Coordinator		✓	

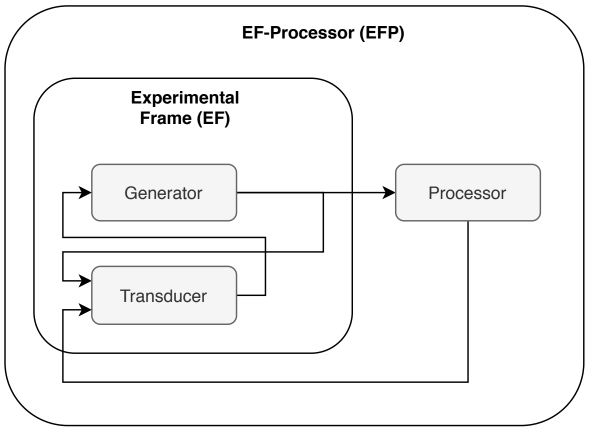
`Generator` component generates `Jobs` with specific period. The `Processor` receives these `Jobs` and simulates some internal processing. Usually, the generation time is configured to be less than the processing time in this model, and the `Processor` only accepts `Jobs` when it is in the idle state. The `Transducer` is the component in charge of counting the number of generated and processed jobs, as well as computing the ratio of processed jobs. As shown in Figure 3.3a, the `Generator` and the `Transducer` are grouped in the `EF` coupled component. Also, there is a root coupled model that contains the `EF` and the `Processor` components. This modeling hierarchy is followed also in the simulation layer (as can be seen in Figure 3.3b). A coordinator is created for each coupled component, and a simulator is created for each atomic component. The arrows depict the dependencies among the simulation entities. These dependencies between coordinators and simulators are the same that the ones expressed between the coupled and atomic components of the model.

We have added several utilities to the pure DEVS M&S functionality, enumerated below:

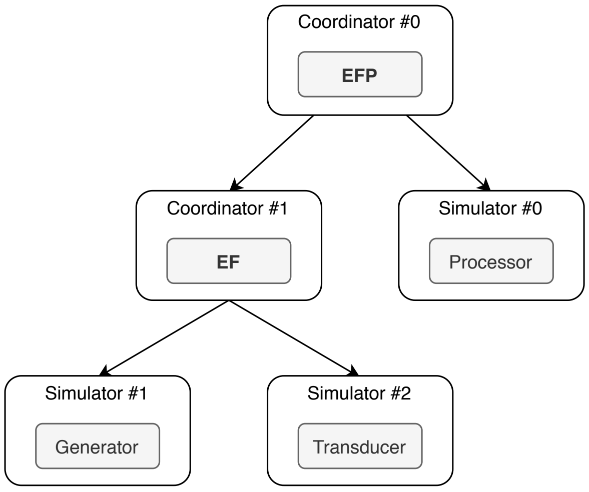
- The constraints definition syntax[87] allows checking arithmetic relations among the values of the model output ports, even combining different hierarchy levels. In this way, the reliability of the simulation is increased with the addition of intuitive JSON-based definitions.

- Both Java and Python implementations also include distributed features that allow them to interconnect models executed in different nodes through TCP network connections.

- The shared memory ports, added to the Python xDEVS implementation, notably



(a) Structure of the EFP model.



(b) Hierarchy of simulators and coordinators.

Figure 3.3: Experimental Frame-Processor (EFP) model.

decrease the simulation overhead by modifying the usual couplings definition. Instead of propagating and copying the values in the ports, they reference the memory where output values are stored in all the couplings destinations. Following this approach, it has been proved that it is possible to reduce the synchronization overhead up to 40% for this particular implementation[46].

- Model flattening is a feature that reduces communication overheads by creating a simplified and equivalent version of the models without coupled models in intermediate levels of the hierarchy. This feature is supported by all xDEVS implementations.

- Finally, the unit testing tool integrated into the xDEVS Java branch allows us to define, via XML definitions, both the internal states and the outcomes of specific model components. Based on this definition, it automatically adds *Generator* and *Transducer* modules to the model and executes it for checking the defined behavior in the proper simulation times.

xDEVS performance comparative

The performance of the xDEVS M&S framework has been compared with other similar environments over time. This comparative can be used for developers to analyze the overall performance of the simulator when dealing with specific model types, and for users when studying the best platform for modeling their systems. Initially, the Java branch of xDEVS, as the first implementation of this framework, has been compared with other simulators of the state of the art, showing a great performance[175]. More recently, during the development of this thesis, we analyzed the performance of the whole architecture, including the C++ and Python implementations, using the classic DEVStone benchmark[70] (explained in detail in Section 3.1.3).

For measuring the performance of xDEVS we have run all the DEVStone models with a wide range of depth and width for each xDEVS implementation. These experiments were run sequentially in a workstation with Ubuntu 18.04, Intel Core i7-9700, and 64GB RAM. For LI, HI, and HO, the combinations from 200 to 600 with step 20 have been generated for both of the parameters. For HOmod, due to its exponential complexity and higher execution times, we considered the parameters from 30 to 100 with step 10. The resulting simulation times have been compared against those generated by a DEVStone implementation developed in the aDEVS simulator. Table 3.4 shows information of the used engines and environments. Specifically, it includes the engine version and programming language, as well as the interpreters or compilers used to run or compile the DEVStone implementations. Also, it is worth noting that these simulation times do not include the model creation and engine set-up times. It refers only to the simulation time.

In Figure 3.4 we can see a plot matrix representing a comparison of xDEVS vs. aDEVS DEVStone simulation times. Each column corresponds to one of the xDEVS implementations (C++, Java, and Python), and each row represents a specific DEVStone model (LI, HI,

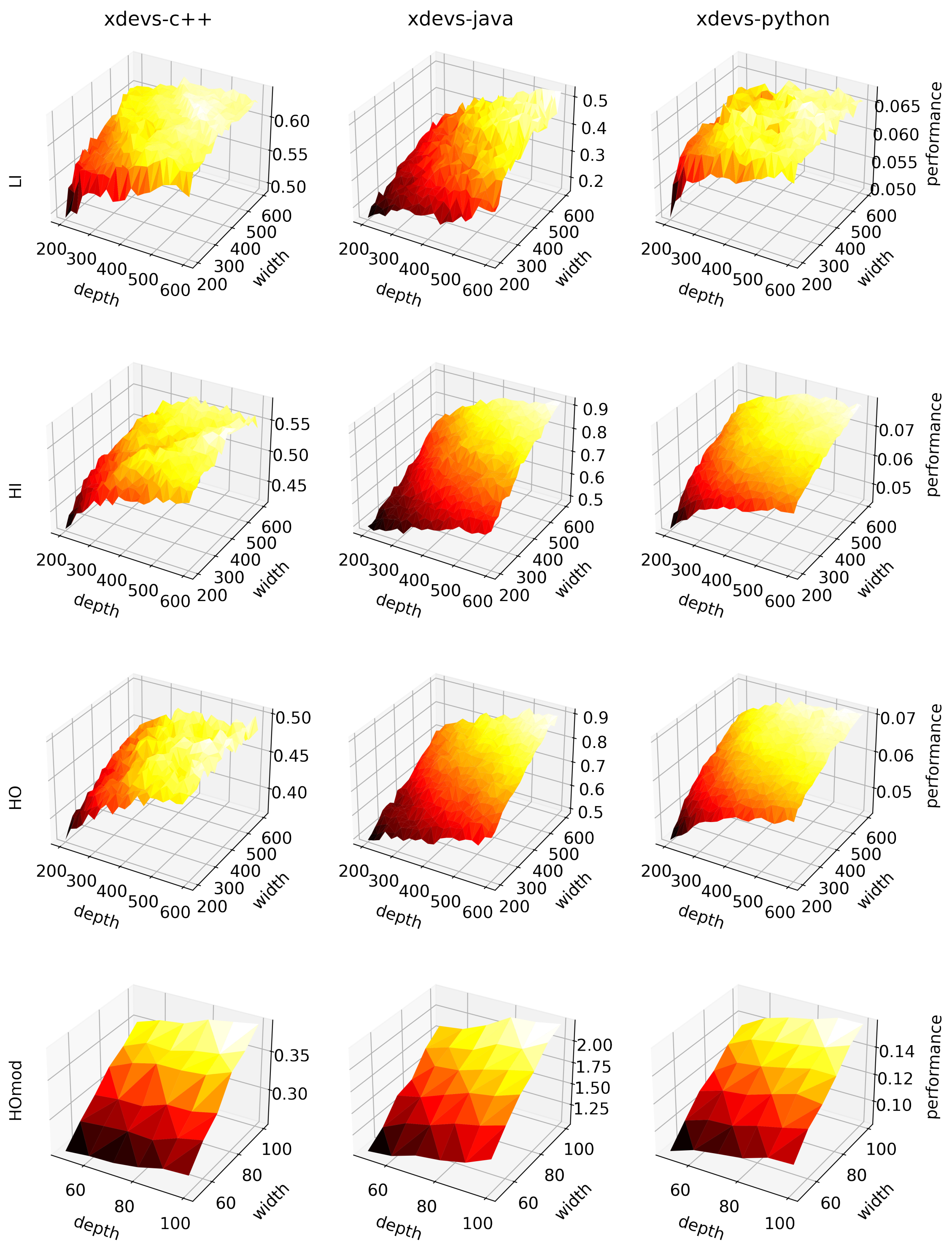


Figure 3.4: DEVStone simulation times comparison between the different xDEVS implementations and aDEVS.

Table 3.4: Engine versions and environments used for the DEVStone simulations.

Engine	**Version**	**Programming language**	**Interpreter / Compiler**
adevs	3.3	C++17	g++ 7.5 (-o3)
xDEVS (1)	1.20181115	C++11	g++ 7.5 (-o3)
xDEVS (2)	1.20200321	Java	OpenJDK 11.0.7
xDEVS (3)	1.1	Python3	CPython 3.6.9

HO, and HOmod). For each specific pair of these implementations and DEVStone models, a plot contrasting the simulation times is shown. The X and Y axes represent the DEVStone depth and width parameters used in the simulations. The Z axis corresponds to the relative performance of the xDEVS simulators in comparison with aDEVS, specified as follows:

$$RelativePerformance(RP) = \frac{T_{aDEVS}}{T_{xDEVS}}$$

This performance is calculated with aDEVS as a reference for being the most efficient DEVS simulator of the state of the art. With the use of this metric, we provide an insight of how xDEVS performs executing the different model types while softening the machine dependence of the simulation times. In the first column, we can see how the performance of the C++ xDEVS implementation is in the range of 0.3-0.6 for the major part of the selected models, obtaining better results as the models to be executed are more complex. The Java xDEVS implementation obtains better results, obtaining times similar to aDEVS in the most complex HI and HO models and improving its results in HOmod models. Finally, we can see how the Python xDEVS has a considerably lower performance than the other two xDEVS implementations, mainly due to the general differences of the programming languages. It is worth noting that Python is an interpreted language, while C++ and Java are compiled before the execution. However, we can see that the surface shapes are similar to the other ones, obtaining worse performance ratios for small models and becoming better as they get more complex. For a better understanding, Table 3.5 shows the performance values for specific balanced configurations, for each model type and implementation.

Additionally, as we focus on complex system, Table 3.6 shows a weighted average of the performance values ($\overline{RP}$) obtained for all the considered model configurations (MC), specified as follows:

$$WT(sim) = \sum_{w,d \in MC} \frac{w * d * T_{sim}(w,d)}{\sum_{w,d \in MC} w * d} \tag{3.7a}$$

$$\overline{RP} = \frac{WT(aDEVS)}{WT(xDEVS)} \tag{3.7b}$$

Table 3.5: Performance values for specific DEVStone balanced configurations.

Model	Depth	Width	xDEVS (C++)	xDEVS (Java)	xDEVS (Python)
LI	200	200	0.491	0.154	0.049
	340	340	0.592	0.325	0.063
	480	480	0.632	0.387	0.065
	600	600	0.623	0.526	0.065
HI	200	200	0.419	0.492	0.044
	340	340	0.504	0.673	0.066
	480	480	0.533	0.815	0.073
	600	600	0.539	0.921	0.077
HO	200	200	0.369	0.488	0.043
	340	340	0.458	0.667	0.060
	480	480	0.466	0.809	0.067
	600	600	0.481	0.895	0.071
HOmod	30	30	0.215	0.635	0.034
	40	50	0.237	0.914	0.075
	70	70	0.301	1.574	0.118
	100	100	0.387	2.212	0.158

Table 3.6: Weighted average of performance values

xDEVS branch	LI	HI	HO	HOmod
C++	0.619	0.546	0.474	0.354
Java	0.397	0.810	0.801	1.939
Python	0.065	0.073	0.067	0.142

3.3 DEVS Unit Testing Verification

Verification and validation play a crucial role in the development of reliable and robust simulation models. This concept is well-established in the software industry, where testing is considered a fundamental activity in the development process, and plenty of validated methodologies have been used for decades. It has been proved that an adequate testing strategy helps to improve the quality of the final product, and reduce project costs and development times. However, the simulation field is having a slow adoption of this type of V&V methodologies, and the popular modeling frameworks often lack verification tools powerful enough to be integrated effectively into the modeling workflow. As an effort to contribute to this transition, we have developed a unit testing tool upon the xDEVS M&S framework.

In the following sections, several aspects of the development of this tool are discussed. First, some implementation details are described, including its methodology and architecture. Then, the specification of test cases is briefly discussed.

Implementation

This unit testing tool has been developed upon the Java branch of the xDEVS simulator. Its implementation has been made following the principles of the Experimental Frame (EF). An EF defines a limited set of circumstances under which the system is to be observed or subjected to experimentation, including observational variables, input schedules, initialization settings, termination conditions, and specifications for data collection and compression[155].

The information of these five blocks is described following an XML syntax, which is used in the parsing phase to automatically generate some additional modules and connect them to the system. These modules are added to the system. They do not affect the behavior of the system and are only intended to inject data into the system and to obtain the resulting values. They can be of two types: (i) `Generators`, that generates the appropriate stimulus based on the test case input, and (ii) `Transducers`, that receive the response values of the different components to compare them with the expected output values. Both the `Generators` and the `Transducers` can be connected to any module of the system under test, regardless of its depth in the module hierarchy. Hence, they can be used to check the behavior of internal components without the need to isolate manually the components.

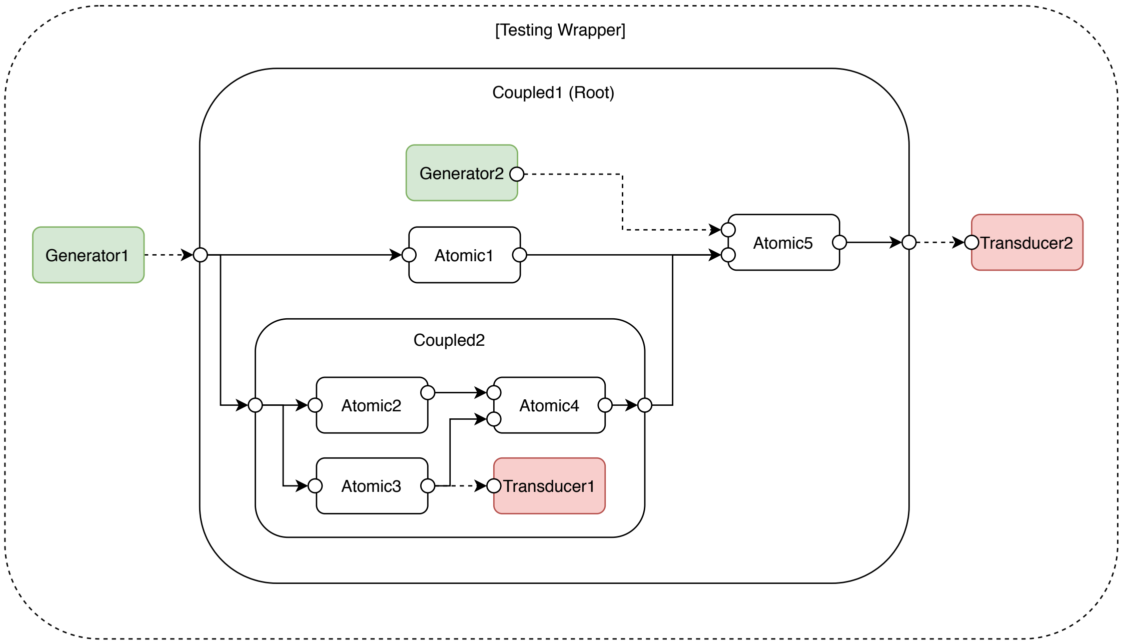
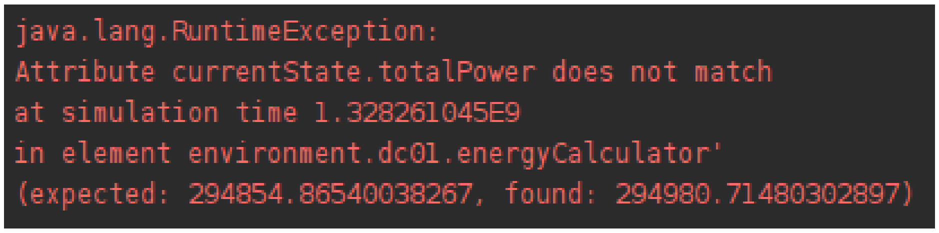


Figure 3.5: Experimental Frame approach for performing DEVS unit testing.

To perform the verification, a `UnitTester` helper class was added to the xDEVS API. This class receives as arguments an atomic or coupled module (usually the root coupled module of the system) and the XML testing file describing the test cases. Internally, this module is allocated inside a `TestingWrapper` module (as shown in Figure 3.5). This additional coupled module ensures that the root module of the test is a coupled module, and



```
java.lang.RuntimeException:
Attribute currentState.totalPower does not match
at simulation time 1.328261045E9
in element environment.dc01.energyCalculator'
(expected: 294854.86540038267, found: 294980.71480302897)
```

Figure 3.6: Exception message produced by the xDEVS unit testing framework.

includes methods to facilitate the addition of auxiliary components in all the levels of the hierarchy. After that, the different expected states specified in the XML file are processed based on their related simulation time, in ascending order. For each state, the simulation time advances according to the difference between the current simulation time and the next state to verify. For each state, the values in the transducers registering the suitable port outputs are compared with the expected ones. Moreover, the state of the different atomic modules of the system is also checked if needed. When some discrepancy is found, an exception is raised indicating the expected and actual values, the simulation time, and the component or port where the problem occurred. An example of this kind of exception message is shown in Figure 3.6.

The whole verification process described above is summarized in Algorithm 1. This algorithm shows how the testing wrapper containing the root coupled model is instantiated (lines 1-2), the specified generators are created and added to the testing frame to produce the test inputs (lines 3-6), and the suitable transducers are generated based on the monitored ports (lines 7-9). After all these needed models are created, the unit testing procedure initializes the environment (lines 10-11) and iteratively checks the specified information for each of the monitored states (lines 12-23). For each one of the states, the time difference between the current and previous state is calculated (line 13) in order to advance the simulation time (lines 14-20). Depending on the test configuration, the values registered for each port correspond either to all the outputs generated since the previous state or the ones generated at the exact state simulation time. Finally, all the port values and atomic states are verified based on the test case specification (line 21), and the state simulation time is saved as a reference for the next iteration (line 22).

Algorithm 1 Unit testing verification process

```
1: $root\_entity \Leftarrow instantiate\_root()$
2: $unit\_tester \Leftarrow instantiate\_unit\_tester(root\_entity)$
3: for $gen\_path, gen\_out\_port, model\_in\_port \in input\_generators()$ do
4:     $generator \Leftarrow instantiate\_generator(generator\_info.path)$
5:     $unit\_tester.add\_generator(generator, gen\_out\_port, model\_in\_port)$
6: end for
7: for $out\_port \in monitored\_ports()$ do
8:     $unit\_tester.add\_transducer(out\_port)$
9: end for
10: $unit\_tester.initialize()$
11: $last\_time \Leftarrow 0$
12: for $state \in monitored\_states()$ do
13:     $time\_diff \Leftarrow state.time - last\_time$
14:     if $state.accumulative$ then
15:         $unit\_tester.simulate(time\_diff)$
16:     else
17:         $unit\_tester.simulate(time\_diff - 1)$
18:         $unit\_tester.clear\_transducers()$
19:         $unit\_tester.simulate(1)$
20:     end if
21:     $check\_transducers()$
22:     $last\_time \Leftarrow state.time$
23: end for
```

Definition of test cases

In the following, the structure defined for the test case files used in this unit testing tool is presented. As shown in Figure 3.7, test cases are defined in XML files with two main sections: *Generators* and *States*. The *Generators* section defines the modules used to inject inputs into the system. Given the object-oriented paradigm used by most of the DEVS simulation engines, these generators are defined as classes in the project structure and are dynamically instantiated in the testing procedure. However, it is worth mentioning that this method can be easily adapted to other non-object-oriented simulation engines. In this case, since the target simulator is JAVA-based, each *Generator* element specifies the classpath of

```xml
<UnitTest accumulateOutputs="false">

   <Generators>
      <Generator name="generator_name" type="path.to.the.generator_class"
   port="out_port_name"
         connectTo="path.to.other.module_port" />
      <!-- ... -->
   </Generators>

   <States>
      <State time="1328227962">
         <Port name="comp1.comp2.comp3.out_port1">
            <[OutputType] attr1="val1" attr2="val2" />
            <[OutputType] attr1="val3" attr2="val4" />
            <!-- ... -->
            <[OutputType] attr1="val5" attr2="val6" />
         </Port>
      </State>

      <!-- ... -->

      <State time="1328235606">
         <Port name="comp1.out_port1">
            <[OutputType] attr1="val1" attr2="val2" />
         </Port>

         <Atomic name="comp1.comp2.comp3" phase="active" sigma="200" />
         <Coupled name="comp1.comp2.comp4" simple_attr="val1"
            obj_attr.simple_attr="val2"/>
      </State>
   </States>
</UnitTest>
```

Figure 3.7: XML-based syntax to specify test cases. It allows checking port outputs and internal attributes in all the components of the DEVS simulation.

the `Generator` module and the input port to inject the produced values. It should also be pointed out that several *Generators* can be defined. Although they are usually specified at the same level as the root coupled model, it is even possible to place them in different levels of the hierarchical design.

The *States* section includes information about the variables and outputs of a given simulation time. Each *State* can incorporate port outputs and models state variables. *Port* elements have to include in the name attribute the complete path of the port to monitor. This includes both the path of the module containing the port and the port name, in a fully qualified syntax: *component1.component2.componentN.portName*. As seen in the last state in Figure 3.7, it is also possible to inspect the values of Atomic models. It is worth mentioning that these variables can be checked even if they are private in the class design. The comparison will be made with a previous casting of the variable to a string (allowing the comparison with the string-based representation of custom objects). Moreover, inspecting attributes inside other object attributes is also possible, following a syntax like: *object1.object2.attribute_name*.

The *accumulateOutputs* of the root *UnitTest* element allows us to specify how to record the outputs in the `Transducers`. If it is set to *false*, the verification occurs over the values generated at the precise moment specified in the state. On the contrary, if it is set to *true*, the transducers accumulate all the values generated since the previous state. Moreover, this flag can also be specified in individual *State* elements to overwrite the default behavior specified in the root *UnitTest* element.

3.4 Constraint-based simulation layer for verifying DEVS models

According to Zeigler[74] and Sargent[182], the analysis of the relationship between the conceptual model and the computerized model is identified as computerized model verification. For Zeigler, the simulator ensures that a strict relation (i.e. simulation relation) exists between the conceptual model and the computerized model (Mittal and Risco-Martín 2017). In this work, we capture the idea of performing V&V at the simulation level by implementing a constraint specification architecture inside the DEVS simulation layer. In this way, the constraint evaluation is transparent to the modeler and can be defined in separated files. We allow the possibility of adding the constraints as a set of arithmetic and logical expressions, similarly to how constraints are defined in mathematical programming. Moreover, these constraints are not only related to numeric variables, but also to complex data types.

In the following, we provide additional details about both the architecture and the implementation of this constraint-based simulation layer. Moreover, we include a motivational example to ease the understanding of the tool.

Architecture

This architecture bases its operation on the constraints defined in text files. These constraints are dynamic in nature, defining arithmetic and logical expressions based on the values present in the output ports of the different modules of the model. Hence, they may not only be related to the outputs of a single component of the system. A single constraint can involve combinations of outputs of different components, even from different levels of the hierarchical model structure.

```
{
  "vars": {
    <variable_name1>: <arithmetic_expr1>,
    <variable_name2>: <arithmetic_expr2>,
    ...
    <variable_nameN>: <arithmetic_exprN>
  },
  "constraints": {
    "constraint_name1": {"expr": <logic_expr1>, "level": <"info"/"error">},
    "constraint_name2": {"expr": <logic_expr2>, "level": <"info"/"error">},
    ...
    "constraint_nameN": {"expr": <logic_exprN>, "level": <"info"/"error">}
  }
}
```

Figure 3.8: Formal definition of the set of constraints.

The set of constraints that the DEVS model must satisfy are defined through JavaScript Object Notation (JSON), as shown in Figure 3.8. These JSON definitions present two main sections, namely *vars* and *constraints*. The *vars* section is optional and includes a collection of arithmetic expressions based on the output ports of the model. Hence, each variable defined here can represent a value contained in an output port or an arithmetic combination of the values present in different output ports. The *constraints* section is mandatory and specifies the set of constraints that are checked in simulation runtime. They are verified for each simulation step and are expressed as logical expressions. These expressions can directly reference output ports, or use previously defined variables. These variables, or output port values, can involve complex data types, as long as their corresponding operators have been overloaded. Arrays of variables are also allowed. Operations over arrays require the same-length for both arrays since the logic conditions are computed element-by-element.

As shown in Figure 3.8, each variable in the vars section is expressed as a pair of $< variable\_name >:< arithmetic\_expression >$. The variable name must consist of uppercase and lowercase letters, numbers, and underscores. Output ports must be used as operators of the arithmetic expressions. These ports are identified following the full DEVS path in the following format: $coupled_1.coupled_2.....coupled_N.atomic_1.port_1$, where coupled, atomic, and ports are referenced based on the identifiers specified in the definition of the DEVS model structure. If the port used in the expression generates arrays instead of single values, it is necessary to indicate the slice of the array that is taken into account for the verification. This is done by specifying the start and the end indexes, as follows: $coupled_1.coupled_2.atomic_1.port_1[< start\_index >:< end\_index >]$.

In the constraints section, each constraint has the following specification format: $' < constraint\_name >':' expr' :'< logic\_expr >',' level' :'< info/error >'$, where:

- *constraint_name*: identifier of the constraint. It has the same restrictions as variable names. This identifier is shown in the output messages when the expression is not satisfied.

- *logic_expr*: this expression indicates the activation condition of the constraint. It can contain arithmetic ('+', '-', '*' and '/') and logical ('==', '!=', '<', '<=', '>', '>=', '&&' and '||') operators. As operands, the expression can contain the full path of a port, a numeric value or boolean literal (*true*/*false*), or a variable defined in the *vars* section. Some auxiliary functions can be used to deal with arrays, such as *sum*, *len*, *min*, and *max*.

- *level*: it specifies the severity level of the constraint. If is set to *info*, only warning messages are produced when the constraint is not accomplished. If is set to *error*, the constraint is considered critical and the simulation is stopped when it is not satisfied.

It is worth noting that constraints are evaluated when all the involved ports have produced an output. Hence, if at least one of the port outputs used as operands is not generated, the constraint is not evaluated.

Implementation

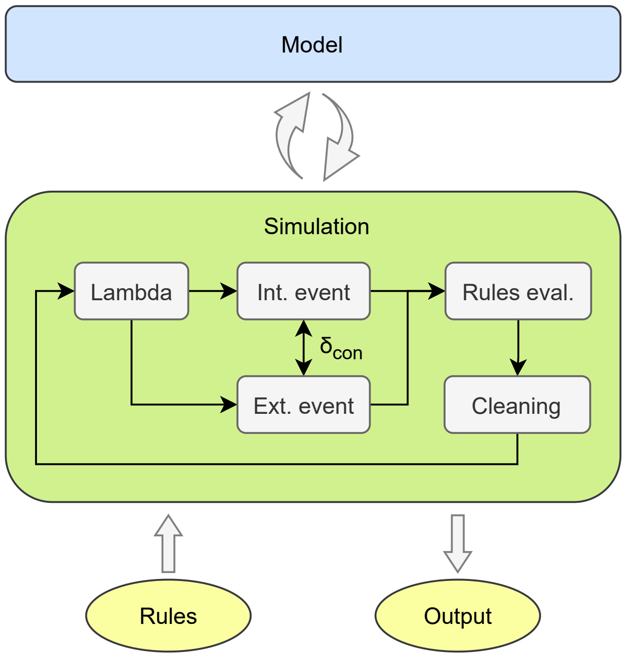


Figure 3.9: DEVS simulation layer functioning. A step for evaluating the constraint rules is introduced before the cleaning phase.

The proposed architecture has been implemented in the C++ branch of the xDEVS simulation engine. Figure 3.9 depicts a scheme of the final implementation. The set of constraints is initially given as a JSON file. When the simulation starts, the DEVS engine performs the following steps: (i) execution of all the output functions (*lambda*), (ii) propagation of the events produced and execution of the corresponding external, internal (or both) transition functions, (iii) evaluation of constraints, and (iv) cleaning of values in all the input/output ports. Finally, the simulation engine goes to step (i) and starts again, until the maximum number of iterations or the maximum simulation time limit is reached. As can be seen, the constraints are repeatedly evaluated at step (iii), when all the events have been propagated from the output ports to the corresponding input ports. At this point, the outputs of the ports implied in the specified constraints are examined. Based on these values and the constraints definition, some mathematical expressions are evaluated. When a constraint is not satisfied, warning messages are displayed or the simulation is finished (depending on the severity level of the constraint). Also, as the constraints are checked at the simulation layer, the execution is transparent to the system engineer and independent from the model definition. This aspect is very useful when verifying models, since the verification process can be tackled once the model has been completely defined. On the contrary, under a model testing approach implemented at the modeling layer, the verification process must evolve with the definition of the model, which from our point of view, is not practical.

Also, it is worth noting that all the results of the constraints evaluation are also stored in an output text file. These logs can be later examined or processed, allowing to generate statistics about conditions compliance based on the warning messages.

Algorithm 2 shows the complete implementation of the model constraints checking implementation in form of pseudocode. A maximum time of simulation is given as argument (*max_ time*). The simulation time is initialized at the beginning of the procedure (line 1) and updated at the end of each iteration (line 20). In this way, the simulation continues until the simulation time exceeds the specified maximum time. The actual xDEVS implementation allows us to specify a maximum number of DEVS iterations as well. For each iteration, all the lambda functions of the model are called recursively (lines 3-7), producing output values if necessary. Then, the values generated in the output ports are propagated to the input ports indicated by the couplings (lines 8-9). As stated in Section 2, there are three types of couplings in a DEVS model: (i) Internal Couplings (IC) connecting compo-

Algorithm 2 Constraints DEVS M&S implementation

```
1:  $sim\_time \leftarrow 0$
2:  while $sim\_time <= max\_time$ do
3:     for $comp \in components$ do
4:        if $sim\_time = comp.next\_event()$ then
5:           $comp.lambda()$
6:        end if
7:     end for
8:     $propagateOutput()$
9:     $propagateInput()$
10:    for $comp \in components$ do
11:       if $sim\_time = comp.next\_event()$ then
12:          $comp.int\_event()$
13:       end if
14:       if $comp.has\_input()$ then
15:          $comp.ext\_event()$
16:       end if
17:    end for
18:    $evaluate\_constraints()$
19:    $clear\_ports()$
20:    $sim\_time \leftarrow next\_event()$
21: end while
```

nents that share the same first parent, (ii) External Input Couplings (EIC) connecting the input ports of coupled modules to one or more of their child components, and (iii) External Output Couplings (EOC) connecting output ports of components to one or more output ports of their first parents. Following Algorithm 2, the propagation of values through these couplings is separated into two functions. Firstly, IC and EOC couplings are propagated using the *propagateOutput* function (line 8) following a bottom-up procedure. After this, values in EIC are propagated using the *propagateInput* function (line 9) in a top-down way. This separation ensures a correct propagation of values, grouping the values generated by the different components in the input ports of the coupled modules to then transmit them to the corresponding child components and allowing a correct constraint checking. Then, the suitable transition event functions are called based on the simulation time and the presence of values in input ports (lines 10-17), and the constraints are evaluated using the information present in the models' ports (line 18). Both the output and input port values are cleared to prepare the next iteration (line 19).

Motivational example

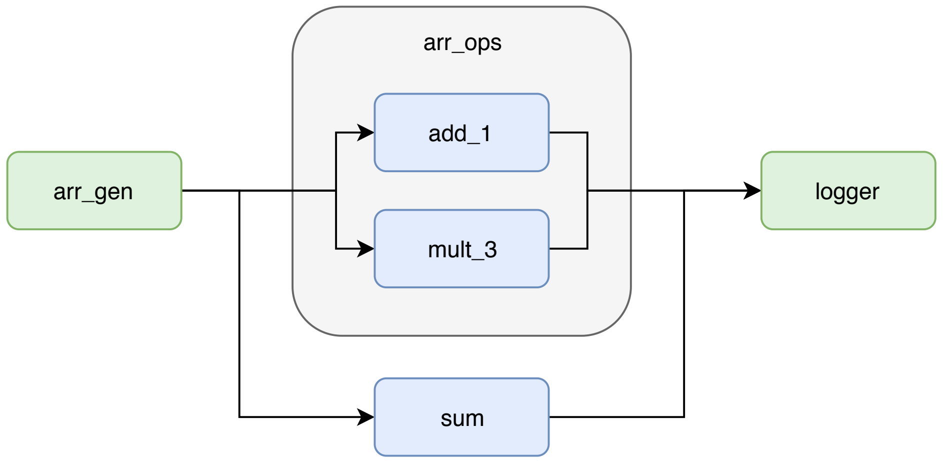


Figure 3.10: Sample DEVS model. It is composed of a generator producing integer arrays, some intermediate models performing simple calculations on them, and a logger registering the results.

Figure 3.11 depicts a complete example of a JSON constraints file. This set of constraints is applied to the DEVS model example given in Figure 3.10. The example comprises a generator (`arr_gen`), that generates arrays of size 5 with random integer numbers. Some basic operations are applied to those arrays. Inside the `arr_ops` coupled module, `add_1` and `mult_3` atomic modules, respectively, add and multiply all the elements of the input arrays by constants. The `sum` module adds up all the elements of the input arrays, returning an integer as a result. All the outputs of these last three modules are sent to a `logger` module, that shows the results.

The first two variables of the constraints file (Figure 3.11) simply get the values of the two atomic modules inside the `arr_ops` module. The third one (`gen_sum`) computes the sum of the original arrays generated by the `arr_gen` module. The fourth one (`adder_mult`) sums the arrays contained in the first two variables. The last one, (`mult_sum_0_3`), returns a scalar with the sum of the three last elements of the `mult` module output. In the constraints

```
{
    "vars": {
        "arr_adder": "arr_ops.add_1.out[0:5]",
        "arr_mult": "arr_ops.mult_3.out[0:5]",
          "gen_sum": "sum.out",
        "adder_mult": "arr_adder + arr_mult",
        "mult_sum_0_3": "sum(arr_ops.mult_3.out[2:5])"
    },
    "constraints": {
        "adder_eq_mult": {"expr": "arr_adder == arr_mult", "level": "info"},
        "ms_lt_gs": {"expr": "mult_sum_0_3 < gen_sum", "level": "info"},
        "check_mult": {"expr": "arr_mult > {59,61,3,4,5}", "level": "info"},
        "check_mult_sum": {"expr": "mult_sum_0_3 >= 100", "level": "error"}
    }
}
```

Figure 3.11: Example of a JSON-based constraints file.

section, four constraints are defined. They specify some basic rules that must be applied over the system through logical expressions, using the previously defined variables and some literals. The first three constraints are only informative, so when they are not fulfilled only warning messages will be displayed. The last one is a critical constraint, and the simulation will be terminated when it is not accomplished.

3.5 Metamorphic verification of DEVS-based systems

Although there are plenty of testing methodologies, they all examine the System Under Test's (SUT) behavior to find potential faults in its functioning. For this purpose, we usually define a set of test cases containing the system's inputs, preconditions and postconditions, and expected outputs. A fundamental aspect of generating these test cases relies on obtaining the expected outputs based on the system's input and current state. These mechanisms are usually referred to in the literature as the test oracle. It can be generated in multiple ways, including modeling, specifications, or contract-driven developments. When any technique can be applied for this test oracle's automation, we have to rely on the knowledge directly provided by a domain expert. This is not a desirable situation in all cases, as automating this test oracle is a critical part of the overall test automation, and the dependence of domain experts can become a bottleneck in the testing process[19].

Moreover, there also exist several types of systems that produce a complex output, and for which it is difficult to deduce the expected outputs for given inputs and states. Some examples of these systems are intensive numerical simulations, binary code generated by compilers, or machine learning models. This problem is referred to as the *oracle problem*, and it is recognized as one of the fundamental challenges of software testing[8,19,186]. In these cases, instead of relying on specific input-output pairs, several alternative methods are applied. For instance, in N-version testing, additional implementations are produced covering partially or totally the SUT's functionalities. Then, these equivalent implementations are executed with the same inputs, and the resulting outputs are compared. In this situation, we know that there is a failure when the outcomes of these systems differ[215]. As a black-box method, this testing technique does not allow testing the SUT flow of events[148], and cannot detect certain fault types such as coincidental correctness[27]. Moreover, its use incurs additional costs derived from the need to develop several equivalent implementations. Another approach consists of the introduction of assertions in the SUT source code[18]. In this way, we assure several constraints or behaviors in the form of boolean expressions that are checked during the execution of the SUT. When the assertion expression is not accomplished an error is raised, reporting a potential failure in the implementation. However, although this methodology can be more natural than developing oracles from other approaches, it results in the introduction of overheads in the system execution. It also harms the readability and maintainability of the source code. In Statistical Hypothesis Testing (SHT), the SUT is executed several times to obtain a set of outputs, which is aggregated using summary statistics (e.g. mean, variance)[164]. These aggregated values are compared to values that delineate the expected distribution. If there are significant differences we detect that there is some malfunction. On the contrary, if they do not yield significant differences we can suppose that it is behaving correctly. The generalisability of this method is quite limited since it is only applicable to non-deterministic systems[133] and we must know the expected output distribution[132].

Metamorphic testing[186] (MT) is a technique that determines the relation between the input and outputs of a system instead of determining specific input and output pairs. For instance, for a system reproducing the sin(x) function, instead of determining the exact value for a specific x, we can take advantage of mathematical property for defining that sin(x) == sin (pi - x) for each value x. In a system calculating the minimal graph distance between two nodes A and B, the output must be equal to the case calculating the distance in inverse order (i.e., from B to A). In an online shopping cart, the total price must increase when we add an article. With this kind of relations, metamorphic testing not only alleviates the oracle problem but allows straightforward automation of the testing process.

The basic process for the application of metamorphic testing can be summarized as follows[186]:

1. Construction of the metamorphic relations: identifying the properties that relate the inputs and outputs of the SUT, or several of its outputs. In some cases, these relations may also include a set of preconditions that must be given before verifying the property.

2. Generation of source test cases: creating a set of test cases based on the identified metamorphic relations. For this purpose, we can use several approaches, such as random testing, fuzz testing, or specification-based techniques.

3. Execution of metamorphic test cases: execution of the SUT, checking pre-established relations for the system's outcomes. If a case violates the metamorphic relation, the metamorphic test case is said to have failed.

In the last two decades, there has been a growing interest in metamorphic testing, applying it for the verification and validation of systems. Based on a survey developed by Segura et al.[186], around half of the contributions on this topic are oriented to the application of metamorphic testing to different problem domains. Among the rest, the most popular topics address the construction of metamorphic relations (19%), integration with other testing techniques (10%), and assessment of metamorphic testing (6%). From a software perspective, MT has been successfully applied to perform V&V in a great variety of use cases, such as bioinformatics programs[36], machine-learning classifiers[221], bug detection for cybersecurity[37], or web services[195]. In the M&S field MT has had less presence, although increasing research on this topic is being presented in the last years. For instance, Sim et al.[189] applied metamorphic testing to verify the correctness of the physics equations on a casting simulation. Lindvall et al.[121] developed a framework for automated testing of a simulated autonomous drone system using metamorphic testing principles combined with model-based testing. Olsen et al.[154] propose a framework and several guidelines to apply MT for simulation validation, focusing on agent-based and discrete-events simulation models. Following these research efforts, we describe in the next section our implementation of a multi-simulator metamorphic testing tool oriented to verify DEVS models.

3.5.1 M&S metamorphic verification tool

To deal with the oracle problem in developing simulation models, we have designed and implemented a flexible metamorphic verification tool that uses simulators' traces to check different metamorphic relations specified with Python functions or external test case files. Figure 3.12 represents the functioning of this verification tool.

First, the simulator executes the model with a predefined set of inputs. Consequently, the outcomes are generated and stored in log files following the simulator's specific traces syntax. As this metamorphic tool bases its operation on input and output files, it is completely independent of the simulator. It can be used for any simulator as long as it can store in external files the simulation outcomes alongside the related simulation times. Hence, it is neither restricted to DEVS-based environments, allowing the integration of non-DEVS simulators simply adding a suitable IO parser. However, this flexibility also comes with a

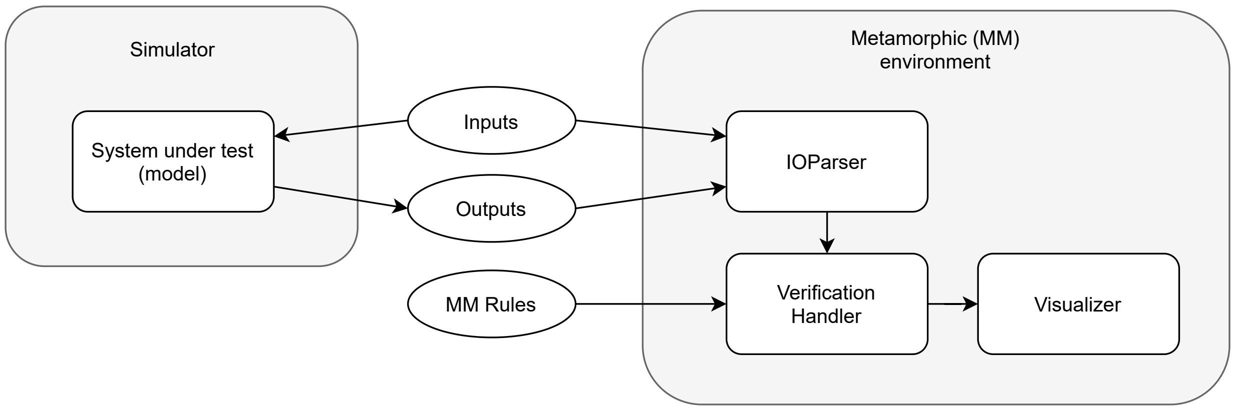


Figure 3.12: Metamorphic verification tool.

drawback: generating the simulation's output files must be completed before starting the metamorphic checking. This is especially relevant when executing large input sets, even though it can be alleviated by creating smaller input sets and parallelizing the simulator executions and the metamorphic testing.

Also, we have to define the metamorphic relations. This task that can be done in two different ways (illustrated in Figure 3.13). On the one hand, we can define the relations through Python functions, as shown in Figure 3.13a. Each of these functions receives two dictionaries as arguments, containing the input and outputs of a specific simulation step. These dictionaries have the port names as keys and their outcomes as values. This information is extracted directly from the simulation traces with specific `IOParser` modules. The modeler declares one or several assertions based on this information, covering a specific relation's scope. To improve readability and maintainability, each relation should be declared in a different function. To cover only specific cases, the user can filter the checking requests based on the input and outputs, and even store different auxiliary information in a per-relation cache. This cache is optionally received as a third parameter and keeps relevant information among request calls. To save cache changes, the modeler only has to return the updated dictionary as the output of the function. On the other hand, if the relations do not need an intermediate cache among requests, we can define them in an external file as shown in Figure 3.13b. In this case, we use rules in the form: *Condition -> Assertion*. Both of these terms are specified as boolean expressions, using the boolean operators and operands in the form *[in/out]:[port_name]*. The first part of these operands defines when the rule has to be checked. The second part defines the assertion itself, verifying specific properties of inputs and output relations.

```
def relation_name(inputs, outputs):
   # assertion1
   # assertion2
   # ...
   # assertionN

def relation_name2(inputs, outputs, cache):
   # assertion1
   # assertion2
   # ...
   # assertionN

   cache["somekey"] = somevar
   return cache
```

(a) Python function.

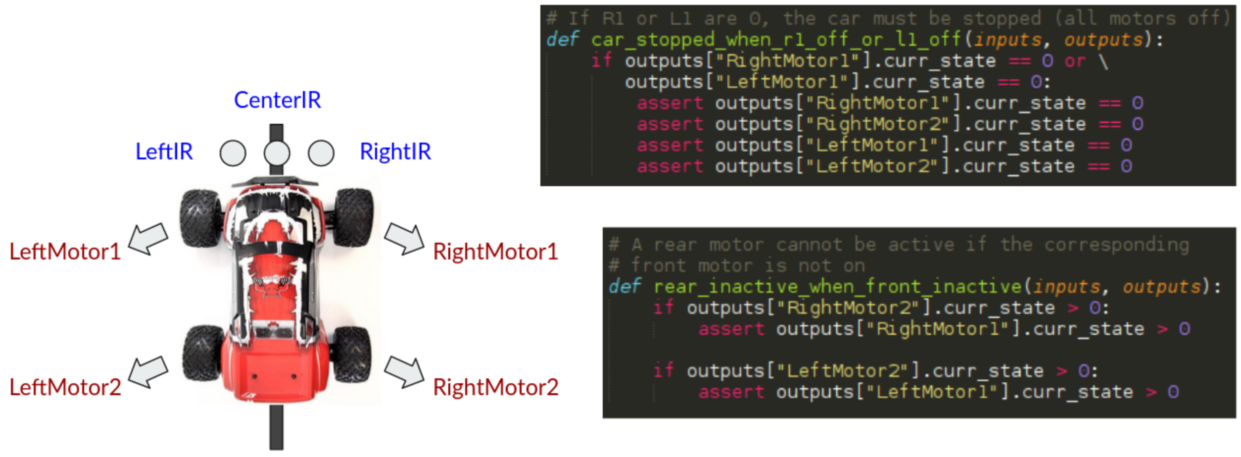
```
# alarm_when_some_red_led
out:e1l1 == 1 or out:e2l1 == 1 -> out:alarm == 1

# lights_ir_dependencies_without_alarm
out:alarm == 0 -> out:r1l1 == in:ir1 and out:r2l1 == in:ir2
out:alarm == 0 and out:r1l2 == 1 -> out:r1l1 == 1
out:alarm == 0 and out:r1l2 == 1 -> out:r1l1 == 1
```

(b) Custom specification format.

Figure 3.13: Syntax for the definition of metamorphic relations.

A sample example using the declaration of metamorphic relations using this tool is shown in Figure 3.14. It illustrates a simple model developed in CD++ to control a line follower robot car with four InfraRed (IR) sensors to detect the path and four motors in the wheels to control the movement. The registered motors' output indicates their relative intensity, from 0 to 1. Moreover, this robot's model is defined so that both of the front motors are activated at half speed to go straight. To turn, both of the motors of a side are activated at full speed, and the contrary front motor is activated at half speed. Hence, when the robot moves, neither of the front motors can be stopped. The first relation captures this behavior, checking with four assertions that all the motors are stopped when at least one of the front motors is off. The second rule shows another relation example, checking that rear motors are only activated when the corresponding front motors are active.



```python
# If R1 or L1 are 0, the car must be stopped (all motors off)
def car_stopped_when_r1_off_or_l1_off(inputs, outputs):
    if outputs["RightMotor1"].curr_state == 0 or \
       outputs["LeftMotor1"].curr_state == 0:
        assert outputs["RightMotor1"].curr_state == 0
        assert outputs["RightMotor2"].curr_state == 0
        assert outputs["LeftMotor1"].curr_state == 0
        assert outputs["LeftMotor2"].curr_state == 0
```

```python
# A rear motor cannot be active if the corresponding
# front motor is not on
def rear_inactive_when_front_inactive(inputs, outputs):
    if outputs["RightMotor2"].curr_state > 0:
        assert outputs["RightMotor1"].curr_state > 0

    if outputs["LeftMotor2"].curr_state > 0:
        assert outputs["LeftMotor1"].curr_state > 0
```

Figure 3.14: Sample example of metamorphic rules definition through Python functions.

This tool provides a simple and flexible way to define metamorphic tests when we face the oracle problem. Moreover, its design allows compatibility with different simulators. For integrating additional simulators, the modeler only has to implement specific `IOParser` modules extracting the input and outputs from the traces files. Moreover, a `Visualizer` interface is provided to simplify the visualization of simulation' status over time, using Python modules like Pillow, Matplotlib, or Seaborn. Several examples of the use of this tool are available in the Github repository of the project[84].

3.6 Extending the DEVStone definition to objectively evaluate the performance of Discrete-Event Simulators

Since its introduction in 2005, DEVStone models have been used for analyzing plenty of DEVS-based simulators. Some authors have used it as a metric to evaluate their DEVS implementations[204,208]. Others used it for measuring the impact of original proposals for performance improvement[62]. Several works have also compared some of the most relevant DEVS-based simulators of the state of the art using this benchmark[63,175]. It can also help new modelers by giving them a performance insight of the most popular DEVS simulators, facilitating the task of selecting the simulator that fits best for their interests. Moreover, it can be used for developers to compare their implementations and to evaluate the performance of new proposals and methods for reducing the overhead introduced by the simulation tools.

However, despite the growing popularity of this benchmark, a common metric has not been proposed yet. As DEVStone defines different models and allows to parameterize them to vary their size and complexity, authors have to select specific combinations of models and parameters for comparing their implementations. In this way, some of them opt to explore all the combinations in a predefined range. Others establish the reference using a small set of heterogeneous models. Different DEVStone model variations have also been presented to explore specific simulation aspects. Risco et al.[175] proposed an HOmod model variation called HOmem, that presents a straightforward mathematical way of incrementing the traffic of events with respect to the three simpler DEVStone models. Van Tendeloo and Vangheluwe[205] introduced a HI model variation that removes the recursiveness present in the DEVStone definitions. This alternative model is composed of four atomic models connected to every other atomic model. Thus, they increment the number of interconnections and show a quadratic growth in the number of events. In this model, a single parameter is provided for defining whether or not collisions should happen, being able to measure the

Model	Depth	Width
LI	200	200
	200	40
	40	200
HI	200	200
	200	40
	40	200
HO	200	200
	200	40
	40	200
HOmod	20	20
	20	4
	4	20

Table 3.7: DEVStone unit model set. All the models are configured to have 0 internal and external transition delay.

bag merging algorithms if this is activated.

This variety of references makes it difficult the comparison between works and is contrary to the concept of the benchmark as such. For overcoming this inconvenience, we performed a revision of the DEVStone benchmark, in collaborative research with the ARS laboratory of the Carleton University. This revision aims to solve the heterogeneity of references by defining a common metric for evaluating DEVS-based simulators. For this purpose, the concept of DEVStone as a performance unit is introduced, measuring the number of seconds that a specific DEVS simulator takes for executing a custom model set in a particular workstation. Table 3.7 summarizes the different model configurations included in this model set. It contains three model configurations per each DEVStone model type. For the LI, HI, and HO models there is a first model with a balanced shape that is configured to have the same value, for both the *depth* and the *width* parameters. Moreover, two additional unbalanced model configurations are included: (i) a deep model, that reduces the *width* of the balanced model to have the 20% of the original *width*, and (ii) a wide model, that reduces the *depth* of the balanced model to have the 20% of the original *depth*. The three HOmod models follow a similar pattern. However, due to the higher complexity of these models, the parameters are reduced to 10% compared to the previous ones. Therefore, they include a reduced balanced version, and deep and wide versions reducing to 20% each one of these variables. In all these models, no internal or external transition delays are established. The execution of each one of the models is triggered by inserting a single integer value in all the input ports of their main coupled models. Also, it is worth noting that only the simulation time is taken into account, discarding the model creation and engine set up time.

DEVS-based simulators comparative using the DEVStone benchmark

This section uses the benchmarking definition aforementioned to evaluate the performance of some popular DEVS-based simulators. For each of them, a DEVStone implementation has been implemented.

- **aDEVS**: useful as a reference, as it usually gets the best performance results.
- **Cadmium**: last simulator presented by the ARS research group, after CD++ and CDBoost.
- **PyPDEVS**: it has two simulator implementations: (i) the main simulator that allows more configurations, and (ii) the minimal version, that restricts the simulation functioning to the basics and presents a higher performance. As recommended by the authors, the execution has been made with the PyPy Python implementation.
- **xDEVS**: as it has support for different programming languages, three different implementations have been developed. These languages are C++, Java, and Python. For the Python implementation, we show the simulation times for the basic simulation mode and include a recent feature for applying shared-memory techniques in the model ports (the so-called Chained DEVS simulator).

Engine	Version	Language	Interpreter / Compiler	Events container	Components container
aDEVS	3.3	C++17	g++ 7.5.0	array	std::set
Cadmium	0.2.5	C++17	g++ 7.5.0	std::vector	std::vector
PythonPDEVS	2.4.1	Python3	Pypy 7.3.1	dict	list
xDEVS (1)	1.20181115	C++11	g++ 7.5.0	std::list	std::list
xDEVS (2)	1.20200321	Java	OpenJDK 11.0.7	LinkedList	LinkedList
xDEVS (3)	1.1	Python3	CPython 3.6.9	deque	list

Table 3.8: Summary of simulators versions, main data types and used interpreters / compilers.

All these simulators present port-based implementation. Therefore, the models include message entry/exit points (ports) that are linked by specifying source-destination links. (couplings). Some additional details about the simulators and interpreters/compilers used for executing the simulations are shown in Table 3.8. The *Events container* column refers to the data type used by the different simulators to store the set of new messages in the ports (message bag). The *Components container* column refers to the data type used to store the different children components in the coupled models.

DEVStone implementations for all these simulators were run in an Ubuntu 18.04, Intel Core i7-9700, and 64GB RAM workstation, following the model set presented before. Ten simulations were performed sequentially (using a single core), with no internal and external transition delays. The results are shown in Table 3.9. The second column shows the accumulated simulation time for all the models of a DEVStone model set, with a 95% confidence interval. The third column shows the corresponding DEVStones per minute (i.e. complete DEVStone model sets that can be executed per minute for each simulator).

The separated times, for each model in the model set, are shown in Tables 3.10 (LI and HI models) and 3.11 (HO and HOmod models). The times of both tables are also depicted in Figure 3.15, for a better comparison of the results (only excluding the non-minimal version of PyPDEVS in the LI models' subplot due to the high scale difference in comparison with

Engine	Average seconds / DEVStone	DEVStones / minute
aDEVS	2.827 ± 0.016	21.226
Cadmium	77.208 ± 0.292	0.777
PythonPDEVS	130.002 ± 0.482	0.462
PythonPDEVS (min)	30.146 ± 0.11	1.990
xDEVS (C++)	7.61 ± 0.041	7.884
xDEVS (Java)	6.477 ± 0.052	9.264
xDEVS (Python)	73.249 ± 0.171	0.819

Table 3.9: DEVStone results for several popular DEVS M&S simulators.

the rest of the simulators). Based on these times, we can see how the simulators perform for the different model configurations. aDEVS obtain the best performance in all the included model configurations, followed by the C++ implementation of xDEVS in the major part of the configurations. The non-minimal PyPDEVS simulator gets the worse time for LI models, where performs up to 96.01 slower than the Python implementation of xDEVS (second-worst time for this model type) and up to 129.55 slower than its minimal simulator implementation. Also, due to the simple configuration of LI models, we can see that all the simulators listed here obtain similar times for the deep and wide models. These comparable results change in HI models, where wide models get times several times higher due to the extra internal couplings. In the balanced configuration of HI models, we can see how the C++ and Java implementations of xDEVS get times near the ones produced by aDEVS. At the other end, we have again the base Python implementations (PyPDEVS and Python xDEVS), with a reduced time difference for this model. In HO models, consequently with the specification, wide models also get higher times compared to deeper configurations. However, we can see how the Java implementation of xDEVS performs better than its C++ equivalent implementation for the most complex models, including all the HO and HOmod configurations. This increase in complexity also causes Cadmium to generate times proportionally higher than the ones obtained in other model types, getting times comparable with the ones obtained by the base Python implementations.

In summary, the extended DEVStone benchmark allows to generate individual and objective performance ratings for specific pairs of DEVS-based simulators and workstations. This ratings are based in the DEVStone unit, that corresponds to the sum of the simulation times of a fixed and heterogeneous model set. Also, with the breakdown of simulation times it is possible to highlight the strengths and weaknesses of the different simulators, which can help both in the development processes of new implementation strategies and in the selection of the most appropriate simulators for a specific project.

	LI-200-200	LI-200-40	LI-40-200	HI-200-200	HI-200-40	HI-40-200
engine						
aDEVS	0.010 ± 0.000	0.001 ± 0.000	0.001 ± 0.000	1.137 ± 0.011	0.019 ± 0.000	0.099 ± 0.000
Cadmium	0.149 ± 0.001	0.030 ± 0.000	0.029 ± 0.000	20.299 ± 0.152	0.725 ± 0.010	3.587 ± 0.021
PythonPDEVS	18.914 ± 0.034	0.786 ± 0.001	0.771 ± 0.002	46.792 ± 0.309	1.543 ± 0.006	4.354 ± 0.023
PythonPDEVS (min)	0.146 ± 0.001	0.021 ± 0.000	0.020 ± 0.000	11.846 ± 0.044	0.330 ± 0.001	1.590 ± 0.008
xDEVS (C++)	0.020 ± 0.000	0.003 ± 0.000	0.003 ± 0.000	2.739 ± 0.020	0.055 ± 0.001	0.297 ± 0.002
xDEVS (Java)	0.062 ± 0.003	0.022 ± 0.000	0.021 ± 0.001	2.480 ± 0.035	0.099 ± 0.003	0.277 ± 0.003
xDEVS (Python)	0.197 ± 0.001	0.038 ± 0.000	0.036 ± 0.000	25.574 ± 0.204	0.959 ± 0.006	4.576 ± 0.019
xDEVS (Python, ch.)	0.156 ± 0.001	0.029 ± 0.000	0.029 ± 0.000	17.648 ± 0.101	0.642 ± 0.003	3.181 ± 0.023

Table 3.10: Simulation times for each one of the LI and HI configurations in the DEVStone model set.

	HO-200-200	HO-200-40	HO-40-200	HOmod-20-20	HOmod-20-4	HOmod-4-20
engine						
aDEVS	1.284 ± 0.004	0.022 ± 0.000	0.116 ± 0.001	0.135 ± 0.000	0.000 ± 0.000	0.002 ± 0.000
Cadmium	38.502 ± 0.262	1.038 ± 0.007	7.129 ± 0.026	5.573 ± 0.021	0.034 ± 0.000	0.114 ± 0.002
PythonPDEVS	46.669 ± 0.191	1.546 ± 0.006	4.374 ± 0.027	4.147 ± 0.016	0.027 ± 0.000	0.080 ± 0.001
PythonPDEVS (min)	12.028 ± 0.088	0.329 ± 0.002	1.613 ± 0.012	2.171 ± 0.022	0.012 ± 0.000	0.039 ± 0.000
xDEVS (C++)	3.448 ± 0.023	0.066 ± 0.001	0.369 ± 0.001	0.598 ± 0.004	0.003 ± 0.000	0.011 ± 0.000
xDEVS (Java)	2.709 ± 0.054	0.107 ± 0.003	0.317 ± 0.005	0.332 ± 0.005	0.020 ± 0.001	0.030 ± 0.001
xDEVS (Python)	29.841 ± 0.136	1.140 ± 0.003	5.492 ± 0.040	5.230 ± 0.047	0.045 ± 0.000	0.121 ± 0.000
xDEVS (Python, ch.)	17.771 ± 0.178	0.647 ± 0.013	3.170 ± 0.022	3.028 ± 0.023	0.025 ± 0.000	0.072 ± 0.001

Table 3.11: Simulation times for each one of the HO and HOmod configurations in the DEVStone unit model set.

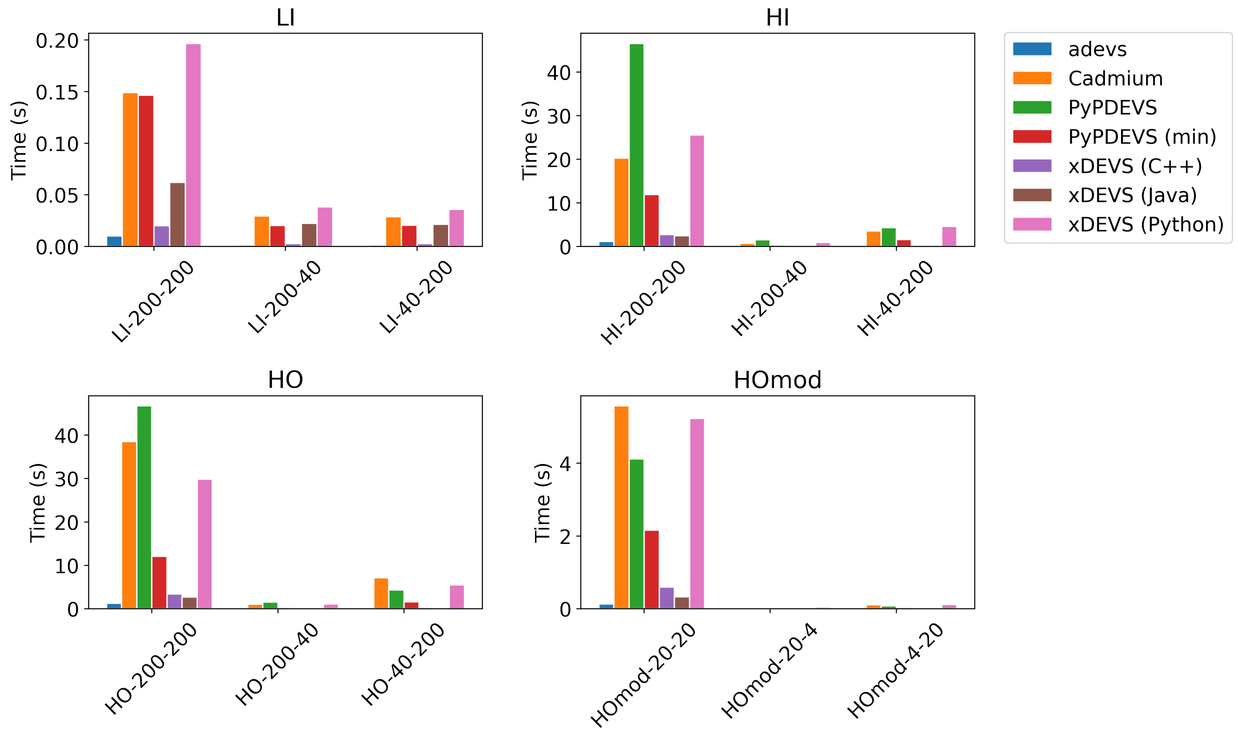


Figure 3.15: Comparison of DEVStone simulation times for several simulators.

For a better understanding of how the different DEVStone model types contribute to the results obtained for each simulator, we also show in Figure 3.16 the accumulated contribution of each model type in the DEVStone benchmark results. In line with the previous results, we can see how LI execution times represent a small percentage of the total time for all the simulators except the PyPDEVS simulator (due to the high initialization and processing time consumed by its non-minimal simulator). In the same simulator, we also see a small HOmod time proportion, probably due to that high overhead introduced in the execution of all the models. On the other hand, Cadmium presents both the minimal HI time and the maximum HO time among all the simulators. From this fact, it could be deduced that the increase in EOC couplings has a greater impact on this simulator (possibly related to its template-based model definition).

In this chapter, we covered different aspects of Discrete Event System Specification (DEVS), a simulation formalism present in the major part of this thesis contributions.

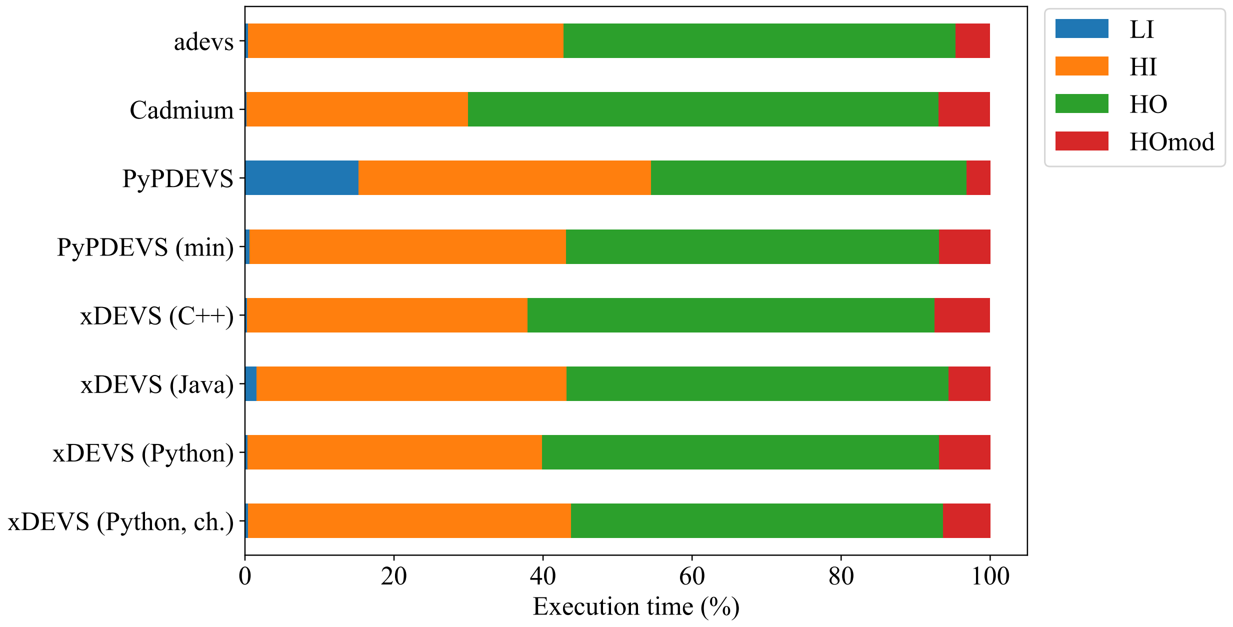


Figure 3.16: Accumulated contribution of the different model types in the DEVStone benchmark results.

First, a formal definition of DEVS was provided, explaining how the behavior of atomic models is controlled by a finite set of events, and how the models are combined through coupled models to specify hierarchical structures. The main DEVS M&S frameworks of the state of the art were presented, providing a comparison of their main features. Among them, we dedicated a section to xDEVS, a M&S toolkit on which several contributions of this thesis have been developed. We discussed its architecture, its different simulators, and its complementary tools aimed at increasing the reliability and performance of their models. Among these tools, there are several verification tools developed during the development of this thesis. These tools bring different methodologies originally proposed in the software industry to the M&S world, allowing the development of unit testing, constraints-based verification, and metamorphic testing over simulation models. Finally, we described how the DEVStone benchmarking definition was extended, enabling an objective analysis and comparison among different DEVS-based simulator implementations.

Chapter 4

Model-driven development of IoT environments

The Internet of Things is a technological revolution that is expected to have a great impact on the future of computing and communications. Nowadays, different application fields such as mobility, industry, energy solutions, healthcare, or agriculture are already benefiting from this revolution[78,109,123]. Aided by the constant evolution of Smart Cities and the progressive introduction of intelligent devices in the daily life of individuals, the number of connected devices is growing at a steady pace. International Data Corporation (IDC) estimates that there will be 41.6 billion connected IoT devices, or *things*, generating 79.4 zettabytes (ZB) of data in 2025[45].

However, this massive introduction of devices also leads to the appearance of new challenges. IoT environments are characterized by their heterogeneity, and the *things* have often limited resources, including battery capacity, storage resources, or computational capabilities. We can see simple devices like sensors, actuators, and RFID tags, but also complex devices such as computers, self-driving vehicles, and autonomous robots[13]. This heterogeneity, along with runtime adaptability, reusability, interoperability, data mining, security, abstraction, automation, privacy, middleware, and architectures are just some of the aspects we need to consider at both design time and runtime[40].

The development of successful IoT scenarios requires previous extensive analysis. Suboptimal application design can reduce the usability of the entire infrastructure, leading to inconveniences as high latencies or overuse of edge devices[76]. Controlled simulation environments allow us to explore different aspects such as application designs, deployment architectures, and resource management policies.

In this chapter, several aspects of these kinds of simulations are discussed. Section 4.1 describes some of the IoT simulators available in the state of the art. Section 4.2 describes our collaboration in the development of the SFIDE data center simulator, as part of the collaboration with the Embedded Systems Laboratory at the EPFL. This simulator focuses on studying the impact of workload allocation and cooling strategies on the overall power consumption of the data centers. As a result of the collaboration, three main contributions have been done: (i) a complete Java-based redesign of the environment, (ii) compatibility support with the SLURM workload manager, and (iii) extension of the SFIDE environment, enabling the specification of IoT environments to study the energy footprint derived of the network communications and final devices. Finally, Section 4.3 describes an optimization IoT scenario, where we study the impact that the specific locations of Micro Data Centers have on the energy consumption of the infrastructure. We describe in detail the whole workflow, including the extraction of the building layouts of the selected urban area, the modeling of the population behavior, and the implementation of the IoT scenario itself using a data stream-oriented IoT framework.

4.1 Related work

In this section, we summarize some of the most popular IoT simulators, that allow us to describe IoT scenarios from many different perspectives.

- **CloudSim**[29] is an open-source simulation framework developed at the Cloud Computing and Distributed Systems (CLOUDS) laboratory of the University of Melbourne. It allows modeling, simulation, and experimentation of Cloud computing infrastructures and application services. It also includes an end-to-end Cloud network architecture that utilizes BRITE topology[135] for modeling link bandwidth and associated latencies. This framework has been used as a basis to develop a multitude of specialized frameworks.

- **EdgeCloudSim**[192] is an edge-oriented simulator developed at the Department of Computer Engineering at the Bogazici University. It extends the functionality of CloudSim so that it can be used for Edge Computing scenarios. Through a modular architecture, it provides support for a variety of functionality such as network modeling specific to WLAN and WAN, device mobility model, realistic and tunable load generator. It also supports the definition of scenarios with several Edge server layers, properly coordinated with different cloud resources. To facilitate these definitions, it also includes orchestration modules to model the organization of the different types of resources.

- **EmuFog**[134] is an an extensible emulation framework for the definition of Fog Computing scenarios. It allows the definition of fog infrastructures and emulates real applications and workloads by embedding Docker images in the scenario nodes. All components of EmuFog are extensible and replaceable by custom-built components designed for specific scenarios. Also, it allows loading the designs performed in network topology generators as BRITE[135], facilitating the import of real-world topology datasets.

- **FogNetSim++**[168] is a toolkit for the modeling and simulation of distributed fog environments. It is built on the top of OMNeT++, a discrete-event simulator oriented to the development of network simulators. It allows us to incorporate customized mobility models and fog node scheduling algorithms, and manage handover mechanisms. It bases the construction of IoT scenarios on three main types of modules: (i) end devices, (ii) fog nodes, and (iii) brokers. In these scenarios, the end devices contain the actual sensors and generate the processing requests to upper layers. The broker receives these requests and sends them to the suitable fog nodes. The broker nodes are also connected to a backbone network which connects them to cloud data centers. FogNetSim++ allows researchers to incorporate their request scheduling and handover algorithms, simplifying the study of the energetic impact of different delivery strategies.

- **FogTorch**[28] is a Java tool for the definition of QoS-aware IoT applications to Fog infrastructures. It divides the definition of the scenario into three levels: (i) IoT devices, (ii) one or more layers of Fog computing nodes, and (iii) one or more cloud data centers. The Cloud concept is simplified in the FogTorch simulation model, understood as a virtually unlimited amount of hardware capabilities. This limitation constrains the scenario definition, but eliminates the need to describe particular cloud topologies and infrastructures and simplifies the definition of any SaaS, PaaS, or IaaS service. FogTorch allows specifying different Quality of Service (QoS) profiles based on pairs of latency and bandwidth values, associating them with specific network links.

- **GloudSim**[53] is a distributed cloud simulator that aims to reproduce the Google cloud environment, allowing to define its cluster infrastructures and simulate different types of events. It allows defining dynamic resource consumption and priority levels for the emulated jobs and reproducing kill/evict events. It is compatible with the Google traces format and includes several interfaces and scripts to extract the information from these CSV trace files. Among other parameters, it allows us to obtain the CPU and memory consumption, the number of simultaneous active jobs, and the workload processing ratio. GloudSim has been published under the GNU GPL v3 license.

- **iCanCloud**[151] is a simulation platform aimed to model and simulate cloud computing systems. The main objective of iCanCloud is to predict the trade-offs between cost and performance of a given set of applications executed in specific hardware, and then provide users helpful information about such costs. It allows modeling the cloud architecture, includes a hypervisor module for managing and comparing different brokering policies, and enables the extraction of detailed energy consumption information for each hardware component of the whole infrastructure. We can also customize our policies to analyze the impact of energy consumption on the overall system performance, facilitating the study of trade-offs between performance and energy consumption.

- **iFogSim**[76] is a toolkit for modeling and simulation of resource management techniques in the Internet of Things, derived from the CloudSim framework. It allows the study of different resource management policies applicable to fog environments concerning their impact on latency (timeliness), energy consumption, network congestion, and operational costs. It simulates edge devices, cloud data centers, and network links to measure performance metrics. One of its main contributions is the *Sense-Process-Actuate*, which allows defining scenarios where sensors publish data periodically or based on events, and devices in the fog layer subscribe to these data streams.

- **IoTSim**[231] is a CloudSim-based IoT simulator that allows to specify and execute IoT big data scenarios. It organizes this specification in six layers: (i) the Core Simulation Engine Layer provides core functionalities as the creation of cloud elements, communication among components, and management of the simulation time, (ii) the CloudSim Simulation Layer provides support for modeling and simulation of Cloud-based simulation environments, (iii) the Storage Layer includes different storage types as Amazon S3, Azure Blob Storage, and HDFS, (iv) the Big Data Processing Layer the processing of the data generated by the IoT devices through Map-Reduce or a streaming computing model, and (v) the User Code Layer includes several utilities to facilitate the definition and validation of IoT scenarios.

- **Mercury**[32] is a Modeling, Simulation, and Optimization framework to analyze the dimensioning and the dynamic operation of real-time fog computing scenarios. It has been developed by the Integrated Systems Laboratory at the Technical University of Madrid, upon the Python xDEVS API. It allows to specify 2D Mobility scenarios and includes a 5G-based model. It organizes the scenario definition in six layers: (i) IoT devices layer, (ii) edge federation layer, (iii) Access Points (APs) layer, (iv) radio interface layer, (v) core network layer, and (vi) Crosshaul layer. It also includes utilities to ease the process of selecting the APs and Edge Data Centers (EDCs) optimal location and generating different output plots to study the results of the simulations. We have used this framework to analyze the power consumption of several data centers receiving several processing requests of IoT devices in a WBSN. This contribution is detailed in Section 4.3.

- **SFIDE**[166] is a simulation framework developed by the Embedded Systems Laboratory at the École polytechnique fédérale de Lausanne (EPFL, Switzerland). It allows the customization of the data centers architecture, including both the servers arrangement in the room, the cooling equipment, and the characterization of the workloads to be executed. The real benefit of SFIDE resides in its capability to implement, test, and assess arbitrary workload allocation strategies and cooling control policies. During the development of this thesis we have contributed to the development of this simulator, adding several new features. We detail these contributions in Section 4.2.

- **YAFS** (Yet Another Fog Simulator)[116] is a fog computing simulator developed at the Department of Mathematics and Computer Science of the University of the Balearic Islands (Spain). It is implemented through Simpy, a simulator for the generation of discrete-event scenarios, and focuses on the analysis of applications design and deployment through the use of customized and dynamical strategies. YAFS allows representing the relationships among applications, network connections, and infrastructure elements, enabling the integration of application modules, workload location strategies, and path routing and scheduling. The resulting computational and transmission results can be exported as CSV files.

Table 4.1 summarizes these IoT simulators, showing some of their features and capabilities. After the simulator name, it is shown the programming language in which they are implemented, and its support for defining Edge, Fog, and Cloud computing infrastructures. The next columns show if they include a graphical GUI to graphically design the scenarios, if they include Map-Reduce computing strategies, and if they allow defining the network components and links. Finally, the last two columns indicate if the simulators allow extracting raw data or visualizations related to the power consumption and network communication latency, respectively.

Table 4.1: Comparison of Fog/Cloud simulators for defining IoT applications and scenarios.

Simulator	Programming Language	Edge	Fog	Cloud	GUI	Map-Reduce	Network configuration	Power consumption analysis	Communication latency analysis
CloudSim	Java	✗	✗	✓	✗	✗	✗	✓	✓
CloudExp	Java	✓	✗	✓	✓	✓	✓	✓	✓
EdgeCloudSim	Java	✓	✗	✓	✗	✗	✓	✓	✓
EmuFog	Python	✓	✓	✗	✗	✗	✓	✗	✓
FogNetSim++	C++	✓	✓	✓	✓	✗	✓	✓	✓
FogTorch	Java	✓	✓	✓	✗	✗	✗	✗	✓
GloudSim	Java	✗	✗	✓	✗	✗	✓	✗	✗
iCanCloud	C++	✗	✗	✓	✓	✗	✓	✗	✗
iFogSim	Java	✗	✓	✓	✓	✗	✓	✓	✓
IOTSim	Java	✗	✗	✓	✗	✓	✓	✗	✓
Mercury	Python	✓	✓	✗	✗	✗	✓	✓	✓
SFIDE	Java	✓	✓	✓	✗	✗	✓	✓	✓
YAFS	Python	✓	✓	✓	✗	✗	✓	✓	✓

4.2 SFIDE: a simulation infrastructure for data centers

The DEVS-based SFIDE (*Simulation Framework and Infrastructure for Data cEnters*) simulator[166] allows researchers to track and analyze the power consumption and thermal behavior of the servers and data centers when running different workload types. SFIDE allows the customization of the data centers architecture, including the server arrangement in the room, the cooling equipment, and the characterization of the workloads to be executed. For defining custom servers, we can extend some of the predefined models or create new models from scratch. These models define how the servers behave in terms of performance and power consumption based on the workload type. SFIDE also allows simulating the cooling both inside the data center room and the overall facility level.

The real benefit of SFIDE resides in its capability to implement, test, and assess arbitrary workload allocation strategies (i.e., algorithms to decide the specific allocation of incoming jobs to servers) and cooling control policies. These two factors dramatically impact the overall energy consumption of the facility and have been widely analyzed in the literature[4,117,163].

Inside the SFIDE simulation model, there are two main coupled models: (i) the *Room*, representing all computing infrastructure and the allocation policies, and (ii) the *Cooling*, controlling the temperature of the whole data center and establishing cooling policies. Apart from that, there are three atomic modules in the root module of the simulator: (i) the *Job-Generator*, that loads the characterization of computational jobs, (ii) the *Weather* module, that simulates the room temperature variances, and (iii) the *EnergyCalculator*, that groups the efficiency and performance stats to generate a report. This components and its interconnections are represented in Figure 4.1.

The jobs generated in the *JobsGenerator* model are directed to the *Allocator*, inside the Room. This module has the responsibility of allocating incoming jobs to specific servers. SFIDE has several types of *Allocator* modules, each one following a different allocation policy. It is also possible to implement custom *Allocator* modules when none of the available

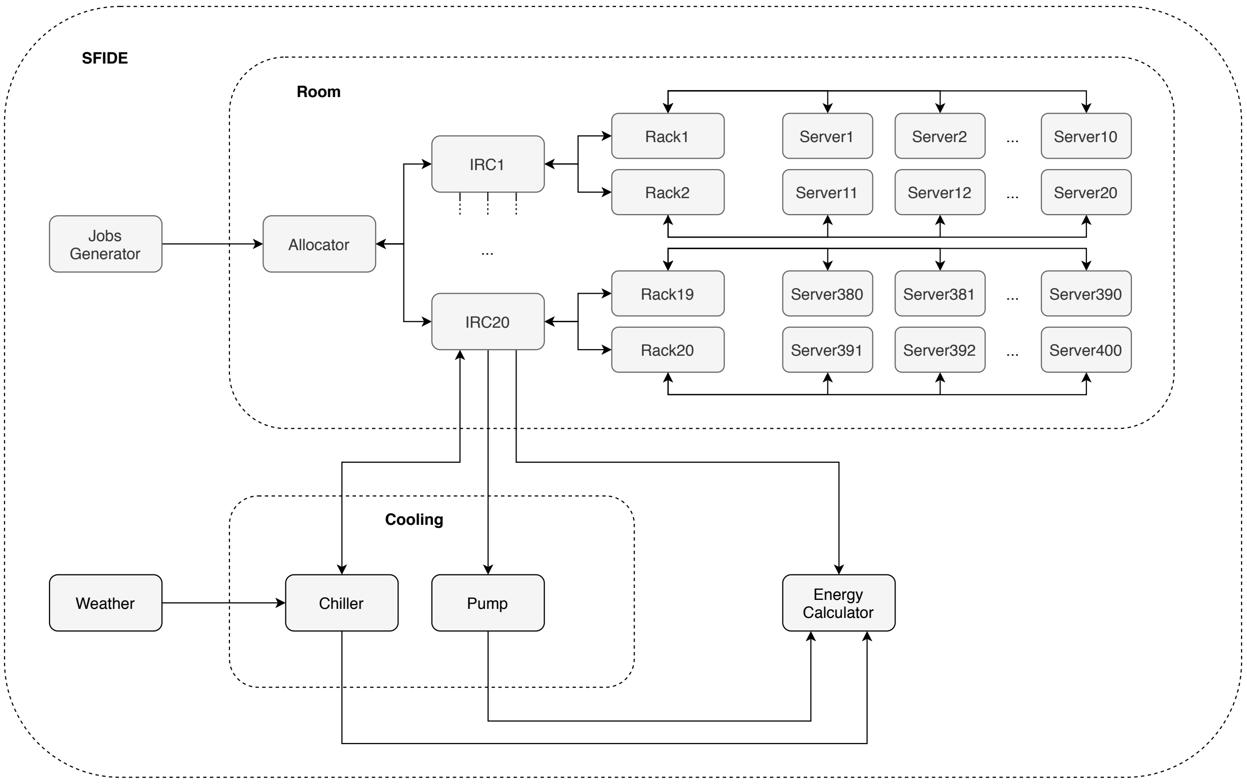


Figure 4.1: Root view of the SFIDE simulator model.

policies suit a specific use case. When a job is assigned, it goes through the corresponding *IRCs* (In-Row Cooling) and *Racks* until reaching the suitable *Servers*. There, the consumption is computed, changing parameters as CPU temperature or airflow accordingly. All the information about the status of the servers is recovered by the *Rack* modules and is communicated to the *Cooling* model. With this grouped information, it takes actions to stabilize the temperature when needed. For that, it disposes of two Atomic models: a *Chiller* and a *Pump*.

All the data produced both in the Room and the Cooling modules are grouped by the *EnergyCalculator*. This model generates reports with the status of the data center for each simulation step. In this way, it allows analyze and compare several allocation policies and architecture configurations straightforwardly, without having to test them directly in a real environment.

Originally, the SFIDE simulator was written in the C++ programming language. However, due to specific requirements of some related projects, we reimplemented the whole model architecture in Java. With this new version, several design aspects were improved, and two major features were introduced in the simulator. The following sections describe in detail these improvements. Section 4.2.1 describes how we designed support for allocating incoming computational tasks via SLURM, one of the most popular workload managers clusters and data centers. Section 4.2.2 shows how we extended the SFIDE original definition, allowing the specification of IoT scenarios, connecting the data centers and end nodes through a network layer. In this way, we can include models representing the intermediate communication protocols and the behavior of the end nodes. As a result, we obtain more detailed reports, including information as the power consumption and the data rate in the intermediate communications, or the tasks requested and processed by the different IoT devices.

4.2.1 SLURM Workload Manager Integration

SLURM is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters[185]. Its key functions include (i) resource access allocation to specific users for a certain duration of time, (ii) a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes, and (iii) contention for resources arbitration by managing a queue of pending work. Moreover, its great variety of plugins can be used to extend its basic functionality.

As part of the collaboration with the Embedded Systems Laboratory at EPFL, SLURM compatibility was added to the SFIDE simulator. This compatibility has been developed with a co-simulation perspective, using a SLURM simulator extending the original workload manager[190] and synchronizing the simulation times of both simulators. Taking advantage of the workload allocation encapsulation of the SFIDE environment, a new type of allocator was created to perform the communications with the SLURM simulator and interpret its results. The overall process to perform SLURM-based workload allocations is depicted in

Figure 4.2. First, a SLURM instance has to be configured. For greater ease, a Docker container with an operative SLURM instance was included in the SFIDE repository. Once started, this SLURM instance needs to reflect the same data center architecture as the one present in the SFIDE simulator. This conversion is made through a custom Python script. The files generated by this script are simply uploaded to the SLURM container, in the proper SLURM configuration directory.

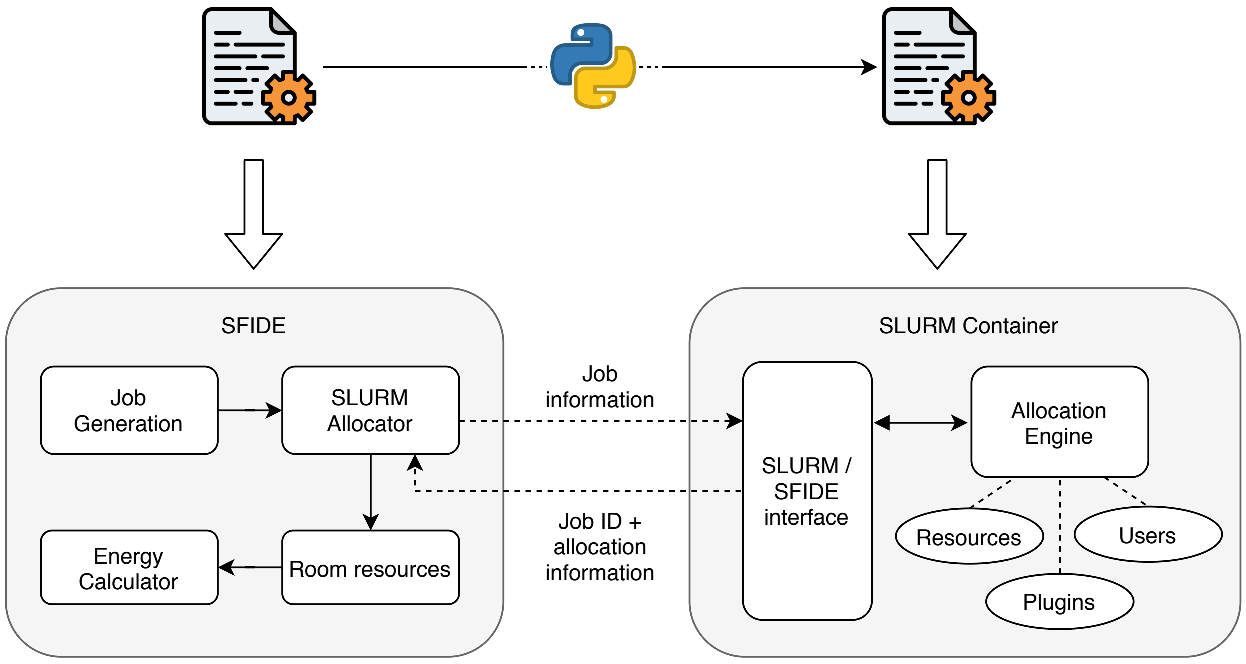


Figure 4.2: SLURM workload allocation support in the SFIDE data center simulator.

The communication between the simulators is done through sockets. The open-source SLURM simulator was modified to listen to workloads specifications in a particular port. The SFIDE SLURM allocator is configured to send the jobs to that particular port. These jobs are translated in the SLURM simulator to a proper format and introduced in the allocation engine, previously configured with a suitable architecture based on the converted file. The identifier of the workloads, as well as the allocation information, is sent back to the SFIDE simulator. Based on it, SFIDE allocates the workloads to the specific servers. The rest of the functions of the SFIDE simulator works as usual, regardless of the chosen allocation strategy. If the workloads can not be allocated in a specific simulation time, the SLURM simulator advances it until enough resources are freed, communicating the new simulation time for synchronization purposes.

This SLURM compatibility enriches the SFIDE simulator, extending it with the complex allocation strategies that can be specified in SLURM. In this way, further performance and power consumption studies can be performed in SFIDE, selecting the best cooling and server architectures. Moreover, the use of this SLURM module is straightforward, only having to follow three simple steps: (i) generate the SLURM configuration files with a Python script based on the SFIDE architecture specification, (ii) configure the SLURM instance with these files, and (iii) configure the SLURM-based allocator in the SFIDE datacenter.

4.2.2 IoT environment for defining flexible and scalable Edge/Fog/-Cloud scenarios

As aforementioned, SFIDE was originally developed as a data center simulator. However, during our collaboration, we proposed to extend this perspective with a more detailed scenario, where IoT devices and data centers can communicate over the network. Therefore, we distinguish three separate layers: (i) a Data Center Layer, (ii) a Network Layer and, (iii) a Things Layer. This separation is depicted in Figure 4.3. In this way, we allow the definition of multiple data centers and end devices, and get some insights relative to the intermediate communications over the network.

This architecture also allows proposing more complex scenarios where the processing of the tasks produced by the end nodes is distributed between Edge and Cloud computing. In this way we can study the performance of the overall system by calibrating the ratio of tasks sent over the network, analyzing aspects as the IoT device battery life, or the usability of the service. We can even specify several Micro Data Centers (MDCs) to be used in hybrid Fog-Cloud scenarios. This allows us to specify scenarios where IoT devices send processing tasks to the Fog, which in turn may forward some of them to the Cloud, based on some infrastructure or power requirements restrictions. The intercommunication

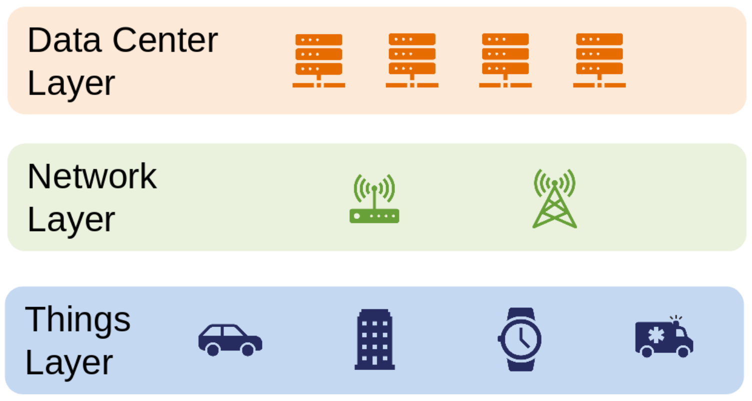


Figure 4.3: Layers of the SFIDE simulation.

of the *things* and data centers is represented through a network layer. We can configure this network with different types of connections, that represent different links related to specific network protocols. This concept is depicted in Figure 4.4. In this example, when Device 1 sends a task to DataCenter 1, it has to go through the C3 and C1 connections. Hence, the generated delay and energy consumption values will be the sum of the ones generated by these two connections. Although there are already many available connections in the framework related to high-level representations of well-known technologies as Bluetooth or Ethernet, the modeler can customize them or even create new connection definitions that fit better in a particular use case by using the interfaces provided by SFIDE.

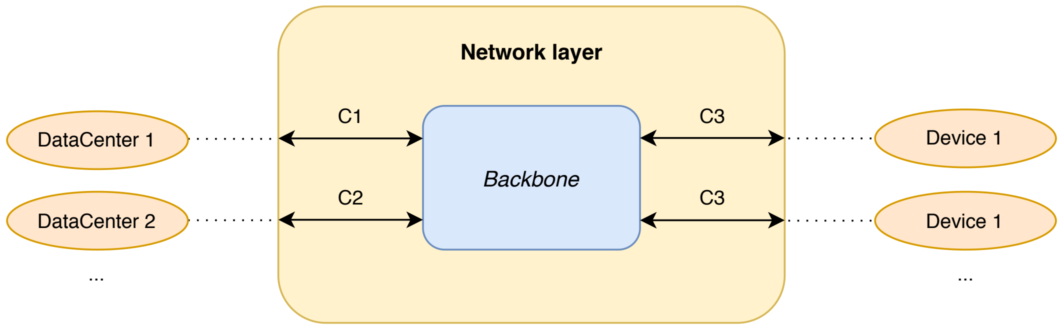


Figure 4.4: Network interaction among components of the Data Center and Things layer.

In this way, each data center or device is connected to the backbone of the network with a connection representing a specific link type and network protocol. In this representation, the `Backbone` model is only responsible of managing the incoming and outcoming data, and registering the delays and power consumption relative to the communication links. In this SFIDE IoT environments, all the information sent through the network (represented by the `NetModel` component) is encapsulated as network packets (`NetPacket` components). These packets accept any type of payload and include useful tracing information as packet identifiers, source and destination nodes, and packet sizes. In a real network, a packet going from a source node to a destination node goes through several connections. For the sake of simplification, in this model, a single connection is specified for each end-point node. These connections link the end nodes, either devices in the Things layer or data centers, with the network backbone. This `Backbone`, although is not taken into account in terms of delays and energy consumption, serves as a centralized point for packet routing and statistics calculation.

Under this scheme, we can straightforwardly communicate data centers and final devices among them. In this way, an intermediate fog data center can forward specific tasks to the cloud when they require further computational resources or the resources are overloaded. Two end devices of the Thing layer can also connect to each other defining the proper communication protocols (as Bluetooth, ZigBee, SigFox, etc.), exchanging information without the need of intermediate data centers. If an IoT end device interacts with both a data center and other devices, several separate connections have to be defined in the network model representing the different network links.

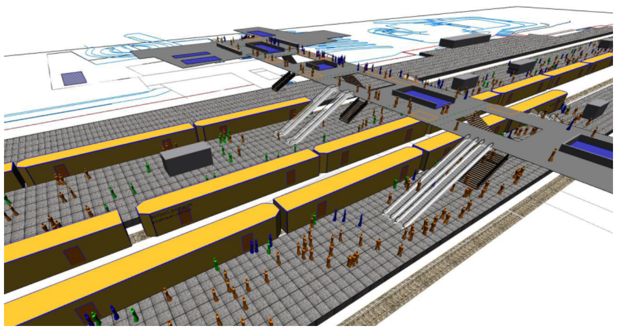
For the development of the behavior of the *things* in SFIDE, the modeler has to extend an interface defining the basic behavior of these devices. It is also possible to extend some of the predefined IoT devices included in SFIDE as illustrative examples. This *things* are highly dependent on the use case and usually have to be customized based on the scenario specifications. A typical scenario workflow would imply a periodic generation of tasks in the devices belonging to the things layer. Some examples could be smart cars in a Advanced Driver-Assistance Systems (ADASs), or monitored patients in Wireless Body Sensor Networks (WBSNs). When the end devices want certain tasks to be executed at data centers, they encapsulate them as packets using the SFIDE specifications and send them over the network including the destination device. The data center receives the packet, which is then forwaded to the server selected by the `Allocator` model. After processing the task, the server can generate a response to the source device. For this, the modeler has to specify a custom behavior in the server definitions based on the workload type and source originator of the task, overriding a specific method available in the `Server` interface for this purpose. As a response, we can also generate multiple packets to notify additional devices of the scenario of specific events, creating more complex scenarios. The resulting packets would go back through the network in a similar way to the suitable end devices. All these network interactions are recorded by the `Backbone` of the network model, generating reports with specific information as the simulation time, source and destination devices, packet sizes, delays, and power consumption.

4.3 Healthcare Infrastructure Optimization Scenarios

When considering the usability and performance of an IoT scenario, several parameters may be considered. From an end-user point of view, some aspects like the perceived latency, the battery life of the nodes, and the overall cost, have a critical impact on the acceptance of IoT products. The optimization of these key aspects can be tackled when developing the IoT scenario by optimizing the power consumption, the processing workflows, and the required bandwidth. The application placement, communication protocols, or job allocation policies are some of the decisions that can alter the usability and performance of the final product. For instance, assigning more processing responsibilities to the end-nodes can highly reduce the latency perceived by the users, but significantly impact the battery life and cost of the product. On the contrary, delegating processing to cloud services can lead to substantial cost savings and higher reliability and performance, but its use implies the acceptance of higher latency and more complex system architecture. Usually, fog computing offers a good trade-off between these two approaches, and it is especially suitable when developing low latency or real-time applications. A good example is healthcare IoT systems, which are usually latency-sensitive, show low response time, and produce a large amount of data[146]. As a result, intensive use of fog computing has been made in this area. For instance, Tuli et al.[199] presented a fog-based smart healthcare system for automatic diagnosis of heart diseases using deep learning techniques. The data coming from different IoT devices is processed through a fog service for deducing the health status of patients and identify heart disease severity. Sahoo et al.[181] designed a stochastic prediction model to foresee the future health condition of the groups of correlated patients based on their current health status. Aazam et al.[2] presented a resource management model for IoTs based on MDCs, covering resource prediction, customer type based resource estimation and reservation, and pricing estimations. Jararweh et al.[101] developed a software-defined based framework to allow mobile cloud computing (MCC) services to integrate different Software-Defined Systems (SDSys) in a Mobile Edge Computing context. This software-based specification of systems abstracts the complexity of large infrastructure deployments and is especially useful for content delivery networks, crowdsourcing, traffic management, or E-health, among others.

In IoT healthcare systems, one of the most important involved methods is monitoring[146]. A typical paradigm in this kind of scenario is the M&S of crowds. The representation of the behavior of a population or a group of individuals allows the development of realistic scenarios as a basis for evaluating new technologies or methodologies. Multiple techniques are used[235] to represent this behavior, such as agent-based models, flow-based models, or particle system models. Currently, several platforms exist to simplify this crowd representation. Simulation of Urban MObility (SUMO)[20] (shown in Figure 4.5b) is an open-source traffic simulation package including net import and demand modeling components. It provides direct compatibility with OpenStreetMap shapefiles and allows modeling of vehicles, public transport, and pedestrians. Pedestrian Dynamics[191] is a simulation environment designed to model pedestrian infrastructures or environments. It allows importing buildings geometry from multiple formats and generating 2D, 3D, and VR virtual environments.

Figure 4.5a shows an scenario example developed with this software. Simulation of such large scenarios involving crowds usually requires intensive computational resources to be processed. These resources, in some cases, far exceed the capabilities of single workstations, being more suitable for its execution in clusters or data centers.


(a) Pedestrian Dynamics.

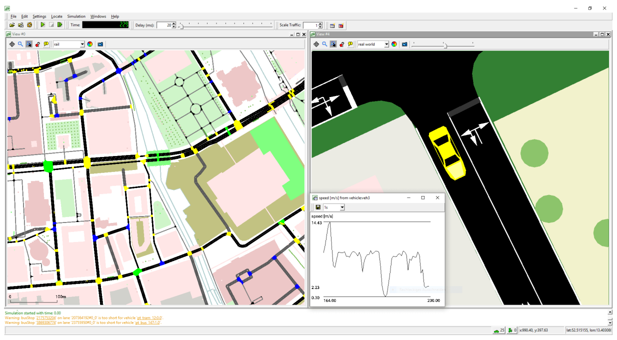

(b) Simulation of Urban MObility (SUMO).

Figure 4.5: Example scenarios designed with urban simulation packages.

4.3.1 Use Case: Optimizing Micro Data Centers location for a Wireless Body Sensor Network implementation

We consider an ambulatory monitoring system scenario where a group of migraine patients wearing health monitoring devices sends periodically some hemodynamic information to Micro Data Centers (MDCs). This information is processed by the MDCs, which calculates the probability of new pain phases through previously uploaded predictive models. The main goal of this research consists in the study of the impact that different MDCs geographical localizations have on the overall power consumed by the data centers.

The architecture of the ambulatory monitoring scenario is represented in Figure 4.6. The monitoring devices carried by migraine patients periodically obtain several hemodynamic variables and send them to their smartphones. Through an app and with a predefined frequency, these data are sent periodically to the MDCs to get the predictions. This connection is performed through a network of Access Points that provides coverage to an entire metropolitan area. In this way, each phone sends the data through the nearest AP, and

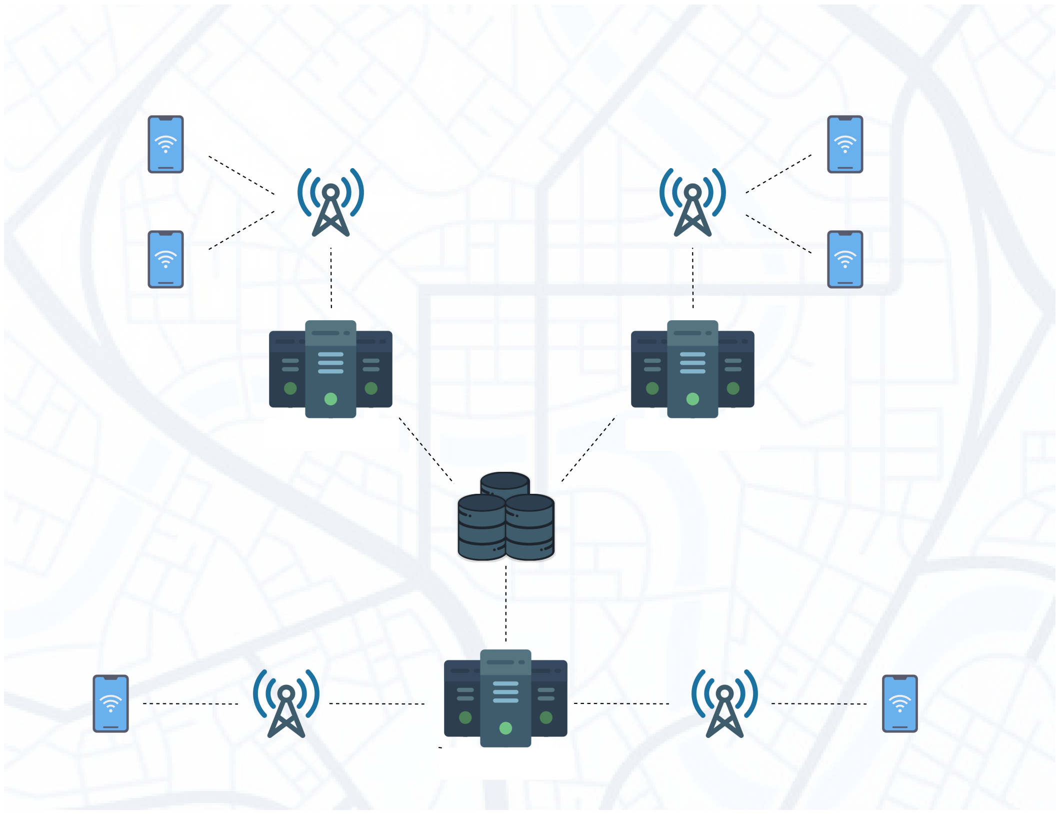


Figure 4.6: Scenario architecture. A network of Edge Data Centers communicated with a distributed database provide connectivity to end devices through several Access Points.

each AP is connected with the nearest MDC. The MDCs evaluate the data packet using per-patient custom models and return the probability of a new migraine pain phase. When this probability exceeds a threshold, the patient is alerted through the monitoring device. Moreover, there exists a shared distributed database when the different customized models are stored. Hence, when an MDC receives a prediction request corresponding to a new patient it loads the suitable model and performs the inference over it.

Each MDC consists of a set of racks, each one containing several servers. Sometimes, as part of the efforts to reduce the high energy consumption of data centers, idle servers are shut down until end-users request extra resources. In our simulation, for simplicity purposes, we do not use these types of strategies. This facilitates the interpretation of the results, allowing us to focus on the energy dynamically consumed by the processing units.

For a realistic characterization of the predictive models, we set some scenario parameters based on the research conducted by Pagan et al.[162] In their study, several types of custom migraine models have been developed, using algorithms like Subspace State Space System Identification (N4SID)[160] and Grammatical Evolution[161] for training the models. For the generation of these per-patient models, they perform ambulatory monitoring of the patients during a period from two weeks up to a month. The five variables implied in this process are: (i) electrodermal activity, (ii) heart rate, (iii) oxygen saturation, (iv) surface skin temperature, and (v) subjective pain level for each migraine event. This last variable is registered through an app, where the patient tracks the progress of migraine. The rest of them are registered with a portable medical monitoring device. After the model is trained in this *offline* phase, it is ready to be uploaded to the servers and start generating runtime predictions of the migraines.

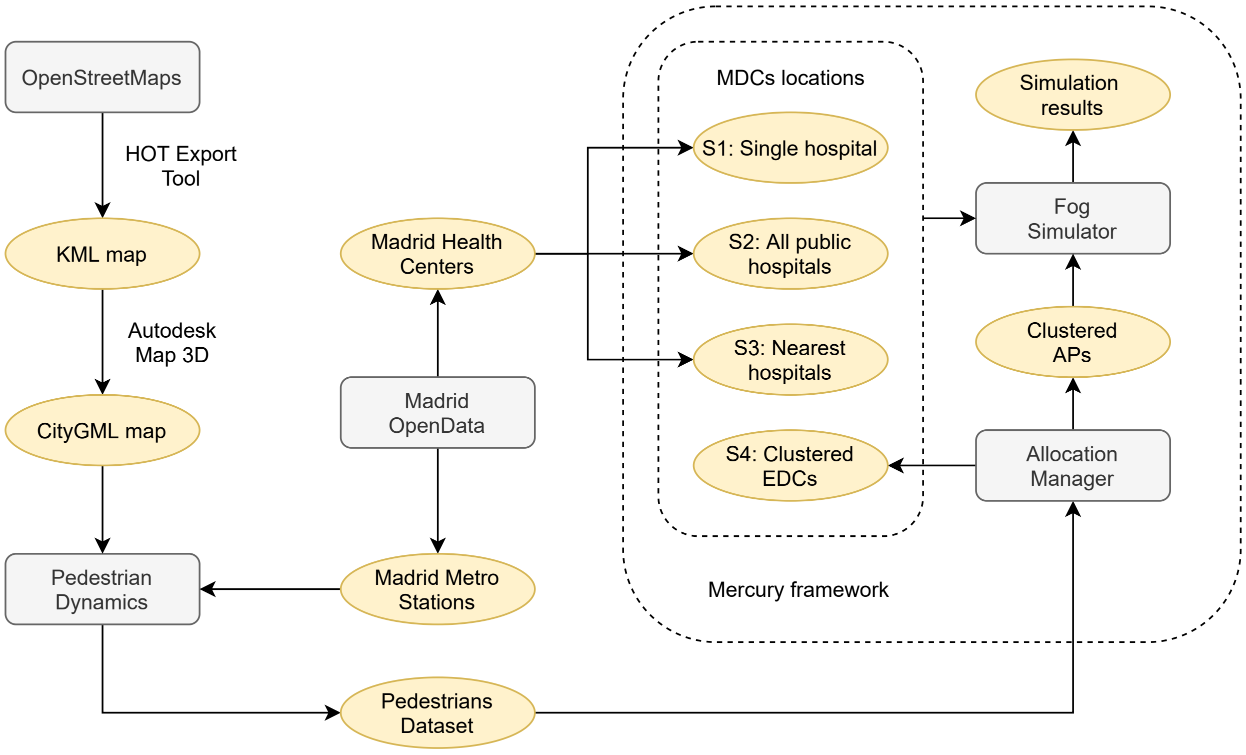


Figure 4.7: Information workflow used in the process of modeling and simulating the pedestrians scenario.

Figure 4.7 summarizes the information workflow followed in the modeling and simulation of these scenarios. We start extracting an urban area for our scenario from OpenStreetMaps (OSM)[1]. This collaborative project gives access to detailed information of map data across much of the world, containing information such as the layout of roads and paths, buildings topology, place names, and points of interest. For these scenarios, we have selected a metropolitan area of $75km^2$ belonging to Madrid, Spain's capital city. Although OSM provides a native mechanism to extract its data, it is limited to 50000 map nodes. As this scenario corresponds to a large metropolitan area, we have used the web platform *HOT Export Tool* to obtain a dataset containing the layout of all the buildings in the selected area as a KML map file. This map is then converted for compatibility reasons to CityGML[113], an open standardized data model and exchange format to store digital 3D models of cities and landscapes, using Autodesk Map 3D. This CityGML definition is then loaded into Pedestrian Dynamics, a simulation environment designed to model pedestrian infrastructures or environments.

The information of these buildings is used in the simulations as obstacles that the pedestrians have to avoid to reach their goals. The main streets and parks of the remaining space are manually marked as activity areas. These areas are used by Pedestrian Dynamics to determine the locations to which the pedestrians go during the simulation. Moreover, 139 stops of the Madrid metro network extracted from the Madrid Open Data portal were included in the map as entry/exit points. With these infrastructures already loaded into Pedestrian Dynamics, we configured the pedestrians' simulation as follows. Every 180 seconds, 200 pedestrians agents are instantiated in the entry/exit points, simulating the arrival of new trains in the metro stations. Each of these pedestrians is configured to walk until a random location included in the main streets or parks defined before, wait for a few seconds, and exit the scenario for the nearest metro stop. This simulation is performed until 10 hours of simulation time are completed. As a result, we obtain a dataset containing the XY coordinates of each pedestrian agent for each simulation time step, that can be used as

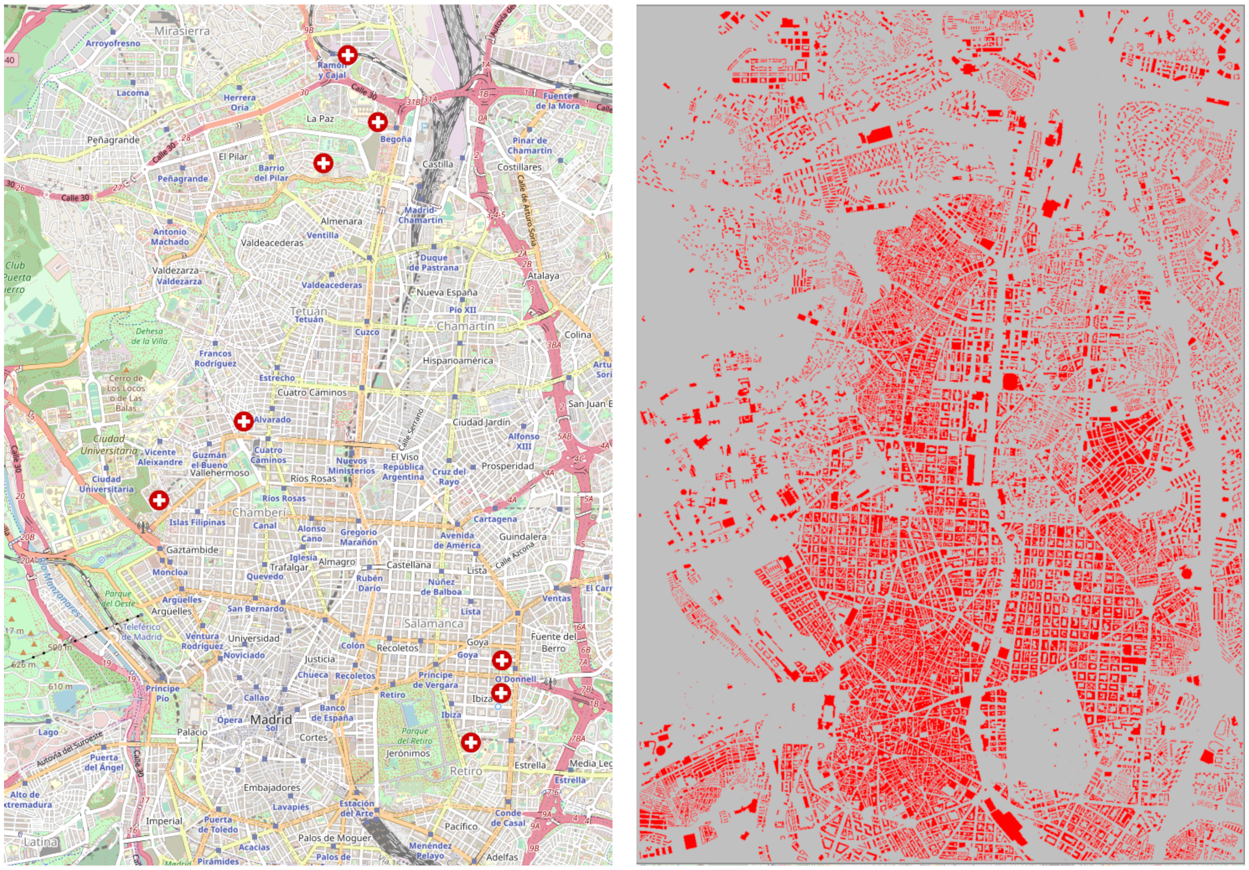
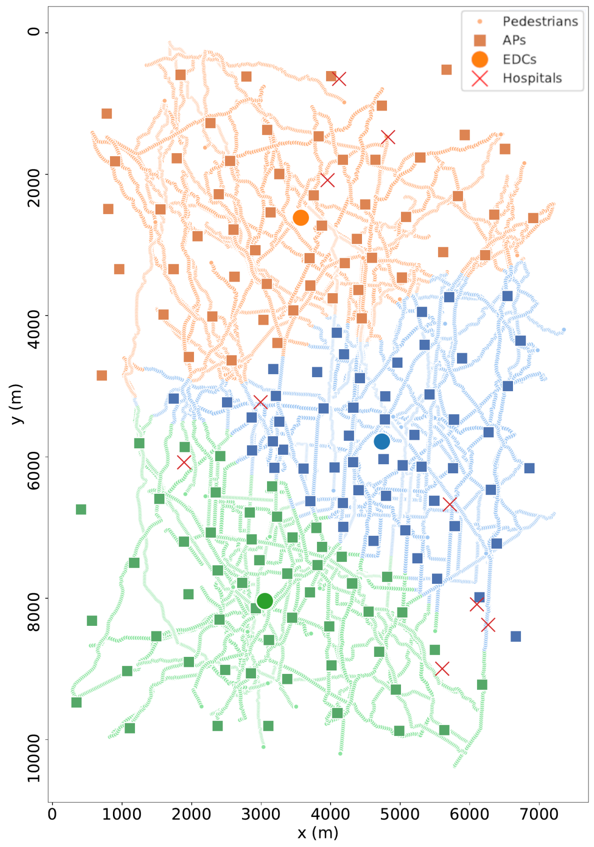

(a) Map view. (b) Simulation scenario.

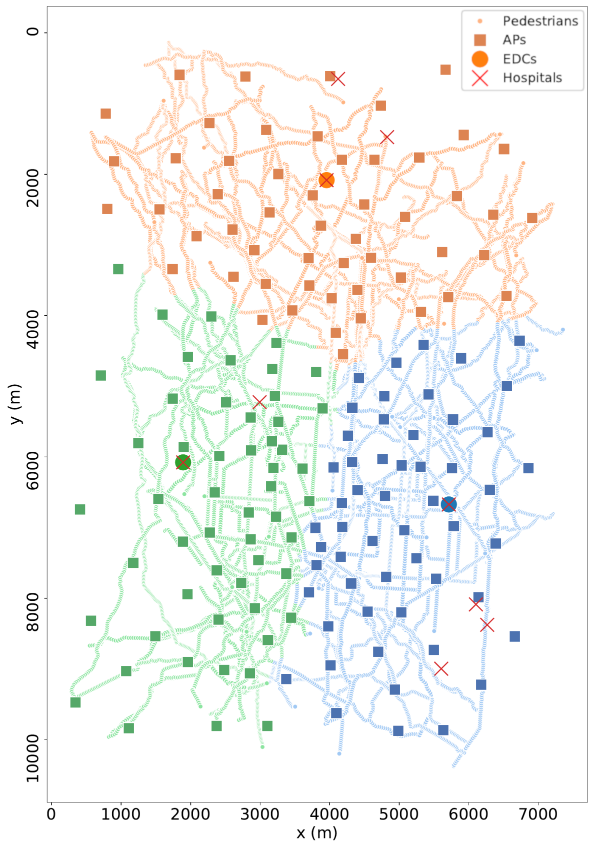
Figure 4.8: Geographical area considered for the simulation. It corresponds to Madrid (Spain), has an area of around $75km$, and includes 8 hospitals.

a reference to model the behavior of the population.

The 10-hour dataset generated using Pedestrian Dynamics is then loaded into the Mercury DEVS framework[32]. Apart from the modules involved in the IoT scenarios simulation itself, this framework also includes some basic tools to generate datasets with optimized locations of APs and MDCs. Firstly, this allocation method partitions the scenario map using a grid, registering the maximum presence of pedestrians in each cell considering a specific time window and grid resolution. Then it sets the location of the APs and MDCs applying KMeans clustering over this grid. We use this method for placing the APs location, which is common to all the scenarios. For this study, we define four alternative scenarios:



(a) C3 scenario: APs and MDCs are allocated using KMeans clustering.



(b) H3 scenario: MDCs are moved to the closest hospitals to the C3 clusters.

Figure 4.9: Scenarios considered in the simulation.

- Clustering scenario (C3): both the APs and the MDCs are allocated based on the KMeans clustering. The resulting locations of these elements are depicted in Figure 4.9a. This Figure depicts the paths chosen by the pedestrians with points of different colors, corresponding with the coverage area of each MDCs. The different APs representing the mobile communication network are drawn as squares and the MDCs as circles. As a reference, public hospitals of this metropolitan area are marked with crosses. In this scenario we consider a fixed amount of 3 MDCs to illustrate the validity of the analysis.

- Central hospital (H1): only one MDC is placed in the scenario, corresponding with the nearest hospital to its center. As a result, all the tasks generated by the monitorization devices are sent to this single data center in this scenario.

- Nearest hospitals (H3): the MDCs of the clustering scenario (C3) are moved to the nearest hospitals. Figure 4.9b represent the resulting scenario.
- All the hospitals (H9): an MDC is placed in each of the nine hospitals present in the selected metropolitan area.

As we aim to compare how the power consumption is distributed among the MDCs of these scenarios, we preserve the same configuration for all the MDCs. Each MDC is composed of 10 processing units (PUs), allocated in a single rack. Each of these PUs represents an Intel Core i9900K processor, operating at 3.6 GHz and composed of 8 cores (16 threads). For the calculation of the power consumption of the PUs, we selected the *IdleActive* power model included in the Mercury framework. This simple model considers two different power consumption. The idle power consumption applies when no task is being executed in a specific PU. The active power consumption specifies the power consumption registered when the PU is executing one or more tasks. In this case, according to the selected processor, the idle power is set to 47W and the active power to 95W. Moreover, it is worth noting that the selected dispatching strategy searches sequentially the first PU with enough free resources to allocate new tasks. Therefore, a PU does not receive a task until the previous PU is unable to allocate it. Besides, although Mercury allows configuring the shutdown of inactive processing units, all the PUs are powered on during all the simulations to reduce variability and facilitate the interpretation of the results.

Table 4.2: Statistics of the monitoring devices services.

Service	Generation period	PU Util. (%)	Operation time (s)	Packet payload size (B)	Total packet size (B)
Inference	60s	6.25	1.17	65	119
Training	unif(1s, 1d)	6.25	18	20	74

For each pedestrian in the 10-hour dataset, an agent is created in the simulation. For this, Mercury sorts the entry points of all the agents and creates and destroys them dynamically so that only the active agents are loaded in memory. We consider that all these pedestrian agents are carrying a monitoring device, and configure two types of services. The inference service requests periodically to the nearest MDC an updated estimation of the probability of a new onset of pain, based on the previously trained predictive models. The training service requests a new training of the patient model and aims to iteratively generate more efficient models with the updated patient data. Table 4.2 shows more detailed information of these services. The inference service requests each minute an updated probability of pain onset, while the next training request is calculated with a uniform distribution from 1 second to 1 day. Both services use 6.25% of a PU, since they fully use one of its 16 threads. The operation times for the inference and training services, 1.17s and 18s respectively, were extracted from real training times of migraines models using similar processors. The packet payload size is determined according to the BSON encoding of the dictionaries exemplified in Figure 4.10. Both of them contain the patient identifier and a boolean indicating to the data center if new training is needed. Additionally, the inference packet contains the average values of the last minute for the implied hemodynamic variables. The total packet size of Table 4.2 simply corresponds to the sum of the payload size and the headers included in the TCP packet (fixed to 54B).

```json
{
   "id":348345,
   "train":false,
   "spo2":98,
   "eda":19.7,
   "temp":34.23,
   "hr":67
}
```

(a) Inference service sample payload.

```json
{
   "id":348345,
   "train":true
}
```

(b) Training service sample payload.

Figure 4.10: Sample packets generated by the simulation services.

The four scenarios defined in the previous section were executed in the ETNA cluster of the Arcuitecture and Technology of Computing Systems (ArTeCS) research group of the Complutense University of Madrid. As a result, we obtained a report with the evolution of the MDCs power consumption and utilization ratios. Figure 4.11 shows how the average utilization of each MDCs evolves as simulation time progresses, and includes shaded areas whose edges indicate the minimum and maximum utilization over time. In this Figure, the solid lines represent the evolution of the utilization factor, and the dashed line represents the evolution in the number of simultaneously monitored pedestrians. We can see how the H1 hospital gets quickly overloaded, being unable to serve all incoming requests. This situation is depicted in Figure 4.12, showing its number of rejected sessions over time. H3 and C3 scenarios produce very similar results, being its evolution lines overlapped in the plot. Finally, because the requests are distributed among a greater number of MDCs, the H9 scenario obtains the least average utilization factor.

For a better understanding of how the requests are distributed over the MDCs, we represent in Figure 4.13 the percentage of utilization of the individual MDCs with respect to the total utilization of each scenario. In the case of H3 and C3 scenarios, we can see an even distribution of the requests, sending about a third of them to each MDC. As expected, the percentage of individual usage differs considerably in the H9 scenario since the distance among data centers is not optimized. As a result, a third of data centers have less than 3% usage while the two most used data centers add up to around 44%.

Figure 4.14 depicts the mean power consumption for the different MDCs after 15000s of simulation time, when the number of simultaneous pedestrians is stabilized. The blue bars indicate the power consumption derived from the infrastructure itself, while the orange ones represent the consumption due to the execution of the training and inference services. Consequently with the utilization ratios, we can see how H3 and C3 scenarios present an equivalent power consumption. Also, we can see how around a third of the power consumption corresponds to the execution of the services. While looking at the H1 scenario, it is

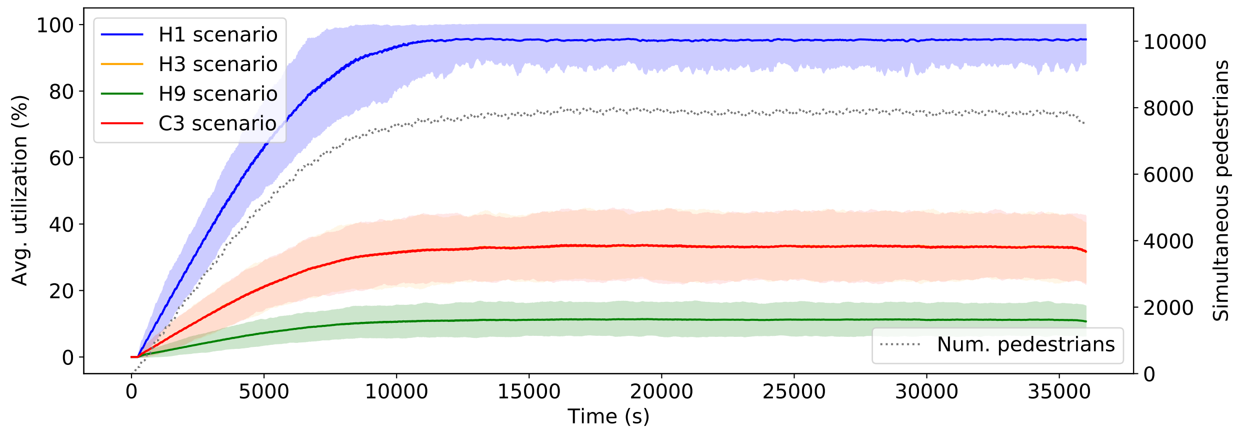


Figure 4.11: MDCs average utilization for the different scenarios, compared with the number of simultaneous pedestrian agents. H3 and C3 produce similar results (overlapped).

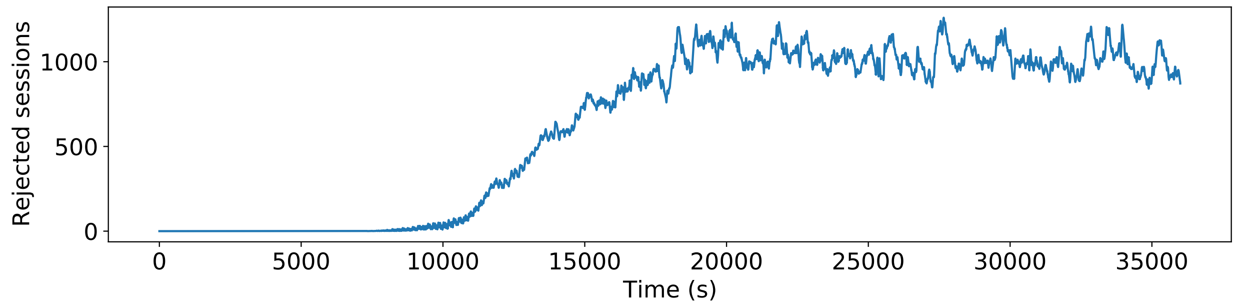
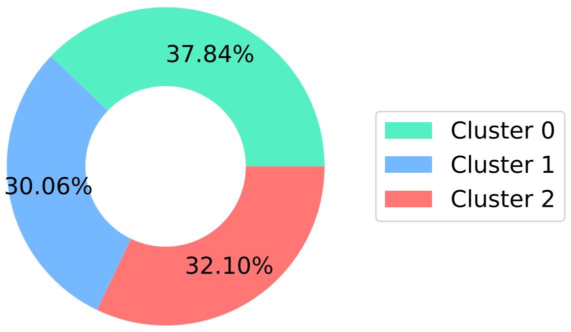


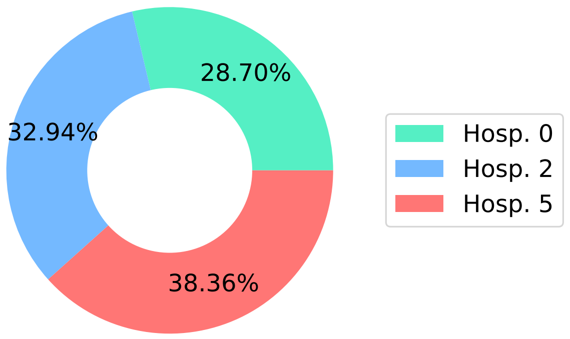
Figure 4.12: Sessions rejected by the MDC of the H1 scenario due to resource overload .

worth noting that even though the energy consumption only represents around 47% compared to the H3 and C3 scenarios, its MDC gets quickly overloaded before reaching a third of the total simulation time. Hence, this scenario does not provide a suitable service to the training and inference tasks and does not accomplish the requirements of the scenario. The H9 scenario is not a good solution either, since it presents a total consumption almost 2.5 times higher than the scenarios with 3 MDCs. Moreover, as its processing units present a less exhaustive use, the resulting non-idle power consumption is 45% than the H3 and C3 scenarios.

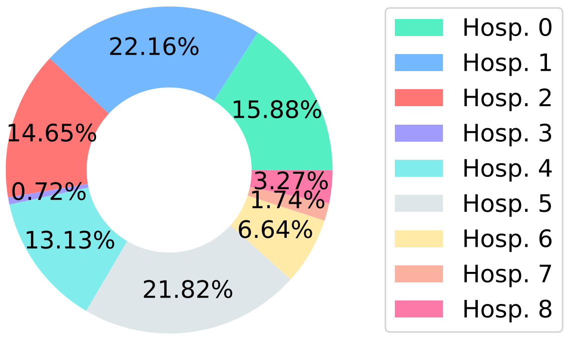
The presented M&S-driven methodology can be used as a tool for analyzing and deploying IoT infrastructures for healthcare systems. It is particularly useful when dealing with moving agents generating computational tasks, as is the case of a monitored patient network.



(a) C3 scenario.



(b) H3 scenario.



(c) H9 scenario.

Figure 4.13: Distribution of power consumption over the different Micro Data Centers.

We have simulated the movement of patients using a well-known crowd simulator, over an urban scenario based on actual building layouts and metro stops. The structure and behavior of data centers has been modeled through the Mercury framework, defining specific APs over the city to serve as intermediate connection infrastructures for the communication between monitoring devices and data centers. We have compared different scenarios, analyzing the differences in terms of energy consumption among several clusterized and hospital-based MDC locations. As a result, we have obtained several comparisons of utilization ratios and power consumption, providing insights that can help in decision-making when optimizing the location of IoT infrastructures.

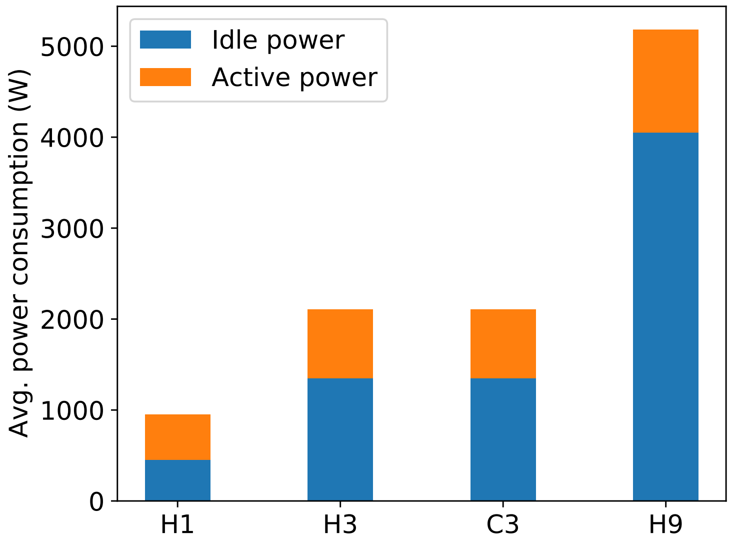


Figure 4.14: Average power consumption of the different scenarios after the number of simultaneous pedestrians is stabilized (after 15000s of simulation time).

In this chapter, several aspects regarding IoT scenarios analysis and deployment were discussed. First, we presented a comparison of some of the most popular environments for the simulation of IoT environments. Also, we present two research projects aiming to provide tools and methodologies to study the large-scale deployment of predictive models. One of them implies our contributions in the development of the SFIDE simulator, allowing it to interoperate with the popular SLURM workload manager from a co-simulation perspective, and define IoT scenarios where different devices send workload requests to a set of data centers through a basic network definition modeling the latency and energy consumption of the intermediate network links. On the other hand, we developed an optimization scenario aiming to find the best location for a set of Micro Data Centers (MDCs) receiving several types of processing workloads in a healthcare monitoring context. We build an urban scenario upon actual data of building layouts and metro stops, reproducing the pedestrians' behavior using the Pedestrian Dynamics software, and simulating the workload distribution and execution through the Mercury framework.

Chapter 5

Modeling and Simulation in Healthcare

In the last years, we have seen how modeling and simulation gained popularity for the development of healthcare applications, services, and infrastructures, while the systems to develop became more and more complex. This M&S acceptance, supported by the development of other complementary technologies like the Internet of Things and Machine Learning, enabled the deployment of multiple applications that seemed infeasible some decades ago. Nowadays we can quickly respond to events as unexpected as a pandemic, quickly developing models to study and analyze the best procedures for mitigating its spread, estimating the outcomes and the socioeconomic impact, and developing optimal vaccination strategies. We also have Decision Support Systems (DSS) capable to diagnose some diseases with the same or even better success ratio than the medical staff, or the development of auxiliary models able to estimate the outcomes and key features of some diseases shortly after the patient is connected to a medical unit. The development of Cloud and Fog infrastructures opened the door to more independent scenarios, where the patients can be constantly monitored using unintrusive devices generating periodical requests to these infrastructures with high computing power. This methodology comes with new opportunities, as the development of predictive systems able to detect key events in symptomatic diseases and the automated creation of knowledge bases allowing more detailed studies of different phenomena and illnesses. All this results both in a significant reduction in medical expenses and a considerable improvement in the quality of life of the population.

This section discusses some contributions made in the definition of models for the study, analysis, and optimization of healthcare scenarios. It is organized as follows. Section 5.1 introduces some relevant concepts developed in the rest of the section. Section 5.2 describes the development of a highly-configurable epidemic model developed following the DEVS and cellular automata principles. Section 5.3 describes a M&S-driven Field-Programmable Gate Array (FPGA) implementation of a healthcare monitoring system able to anticipate the pain phases of migraine patients. Finally, Section 5.4 discusses a modular methodology that aims to simplify the creation of predictive modeling systems, using the related framework to face the categorization and outcome prediction in the early stages of stroke crises.

5.1 Background

In this section, we discuss some contextual background concerning the healthcare scenarios and tools developed during the thesis. Section 5.1.1 describes some common approaches to model epidemiology diseases. The SIR model is described and some common variants are explained. Section 5.1.2 describes some relevant factors concerning the development of Healthcare Monitoring Systems (HMS), including the desired features of a successful HMS and the importance that the use of FPGAs has had in the development of these devices.

5.1.1 Numerical methodologies for modeling epidemiology diseases

Epidemiology models use several kinds of methodologies for modeling infectious diseases. These methodologies include different approaches, such as deterministic and stochastic dynamics, discrete and continuous time, or agent-based or compartmental models. Susceptible-Infected-Recovered (SIR) models are one of the most popular. They classify the individuals of a population into a finite number of mutually-exclusive disease stages. These compartmental models have their origin in the work of Kermack and McKendrick[108], who classify the population into three main groups:

- S(t): Population susceptible to the virus in time t. They have no immunity, and therefore can be infected if they are exposed to infectious agents.

- I(t): Population infected with the virus in time t. They can transmit the infection to individuals of the susceptible group when they are in contact.
- R(t): Population removed in time t. It includes both the individuals that have acquired immunity to the virus and the ones deceased because of the infection.

According to these definitions, the total population is constant and satisfies the following equation for any time t: N(t) = S(t) + I(t) + R(t). In these models, usually, two more parameters are considered: (i) an infection rate (λ), which specifies how much the infection is transmitted in each step, and (ii) the recovery rate (γ), which determines the portion of the infected individuals that recover from the disease. Hence, the progression of the infection can be expressed formally as:

$$S(t+1) = S(t) - \lambda * S(t) \tag{5.1a}$$

$$I(t+1) = I(t) + \lambda * S(t) - \gamma * I(t) \tag{5.1b}$$

$$R(t+1) = R(t) + \gamma * I(t) \tag{5.1c}$$

Due to the success of this model, different authors have proposed several extensions. The Susceptible-Exposed-Infected-Recovered (SEIR) model[118] adds an extra group (Exposed) that introduces infected individuals unable to transmit the disease. Susceptible-Infected-Recovered-Deceased (SIRD) models[130] separate explicitly the recovered individuals, who acquire immunity to the disease, and the ones that passed away. SIS models[35] add the possibility of becoming susceptible again after being infectious. MSIR models introduce maternally-derived immunity (passive immunity), where new-born babies remain immune to the disease for the first few months of life due to protection from maternal antibodies. The combination of these extended models results in a great variety of specifications (e.g. SEIIR, SIRS, SEIRS, SIRDS, SEIRDS, MSEIR, or MSEIRS), in which the population individuals are translated from one class to another based on differential equations.

This type of model has been used frequently for the study of epidemics, adapting the models to the particularities of each disease and the socioeconomic and governmental contexts. For instance, Acemoglu et al.[3] presented a multi-risk SIR model (MR-SIR) in which the infection, hospitalization, and fatality rates vary between a limited set of age groups. This categorization allowed them to perform finer grade simulations and find optimal policies differentially targeting risk/age groups. Fernández-Villaverde et al.[59] adapted a SIRD model to study the spread of the COVID-19 in different countries, establishing time-varying contact rates to capture behavioral and policy-induced changes associated with social distancing. As a result, they obtained daily deaths per million evolution forecasts for different USA states and countries. Biswas et al.[24] studied the vaccination planning based on compartmental SEIR models with the introduction of state variables constraints. They obtained several models estimating the optimal vaccination schedules and control strategies based on cost-effectiveness metrics.

5.1.2 Model-Driven development of Healthcare Monitoring Systems

There are multiple types of Healthcare Monitoring Systems (HMSs) supporting medical and healthcare services and applications. These devices can be used as a tool to automatically collect data from patients, which can be used by medical staff for patient monitoring. This results in lower working loads of physicians and increased efficiency in patient management. Also, grouping historical data of a multitude of patients opens the door to extract hidden patterns and conclusions related to the development and outcomes of different diseases.

In the development of a successful HMS several factors must be considered. To be well accepted by patients, these devices should be small, easy-to-use, lightweight and portable. Moreover, a critical development factor is the cost-effectiveness of the proposal. To reduce the costs, devices should limit their components based strictly on the expected functions, and consider using accessible, widely used, and fully configurable components. Among these, programmable components like Field-programmable gate arrays (FPGAs) are a great choice when developing an HMS. These devices allow reconfiguring their internal functional logic using Hardware Description Languages (HDLs). Based on these specifications, the interconnections of the FPGAs programmable logic blocks are altered. It results in low-level representations of the systems, implying reduced power consumption and high efficiency. This covers one of the fundamental challenges of HMSs. These systems require a high level of integration due to the necessary portability. High power efficiency is needed for systems that are truly portable and thus battery-operated.

For these reasons, FPGAs have proven to be more convenient than the traditional microprocessors-based implementations in a variety of applications[38,170,223]. Also, the increase in the complexity of systems over the years has reduced the suitability of microcontrollers in some aspects. Unfortunately, simply switching to a bigger microcontroller only helps to a certain extent as the focus is usually on more memory and general-purpose pins rather than I/O modules. Adding more microcontrollers complicates the overall system design due to the communication overhead. FPGAs, in contrast, do not have these limitations. Also, FPGAs allow high parallelization rates, helping to speedup a lot of implementations whose performance would be more limited in other architectures.

5.2 CellDEVS-based SIRDS model for studying the evolution of epidemics

During the collaboration with the ARS group at the Carleton University, several epidemiology models were developed. This section describes one of these, a SIRDS model, developed with the Cell-DEVS formalism. This variant of DEVS combines cellular automata mechanisms with the discrete-event specification. It allows representing grid-based scenarios whose behavior is defined by a finite set of rules. Also, it allows linking the behavior of specific cells to external DEVS models, expanding the possibilities offered by pure cellular automata models. First, the CD++ modeling toolkit was used to develop the model (as it was the main platform supporting the implementation of Cell-DEVS models). However, the model was later ported to Cadmium, the latest DEVS-based simulator developed by the ARS group (as it added also Cell-DEVS support, extending the functionalities offered by CD++).

This particular SIRDS model aims to be a highly-configurable tool for defining and studying epidemiology scenarios, extending and adapting other model configurations of the state of the art[216,234]. It represents the population as a grid of cells. At time t, each cell (i, j) has a number of individuals $N_{i,j}$ and a several ratios indicating the percentage of individuals present in the different categories of the SIRDS model: *susceptible* ($S^t_{i,j}$), *infected* ($I^t_{i,j}$), *recovered* ($R^t_{i,j}$), and *deceased* ($D^t_{i,j}$). At the beginning of the simulations, the majority of cells population is susceptible to the disease, including only a small infected percentage in one or a few cells. When *infected*, individuals remain ill from 1 to T_I days. After this infection, they become immune for a fixed period of T_R days, and then they return to the *susceptible* category.

Both the *infected* and the *recovery* categories are broken down in different states, indicating how many days have been in that category. Hence, ($I^t_{i,j}$) and ($R^t_{i,j}$) are calculated as follows:

$$I^t_{i,j} = \{I^t_{i,j}(p) | p \in \{1, ..., T_I\}\} \tag{5.2}$$

$$R^t_{i,j} = \{R^t_{i,j}(p) | p \in \{1, ..., T_R\}\} \tag{5.3}$$

, where p is the number of consecutive days that the population has been in each state.

These states are depicted in Figure 5.1. As shown, for each *infected* state p, individuals have a probability $\gamma(p)$ of recovering from the disease. Individuals that do not recover in previous states reach the immunity after the last *infected* state. Moreover, there is a probability $\varphi(p)$ of decease while being infected. As the γ and φ probabilities can be specified for each infected state, this model can easily be configured for adding the exposed or latent category. This category, often included in other epidemiology models of the state of the art, usually describes the first stage of infection, where people become infected but

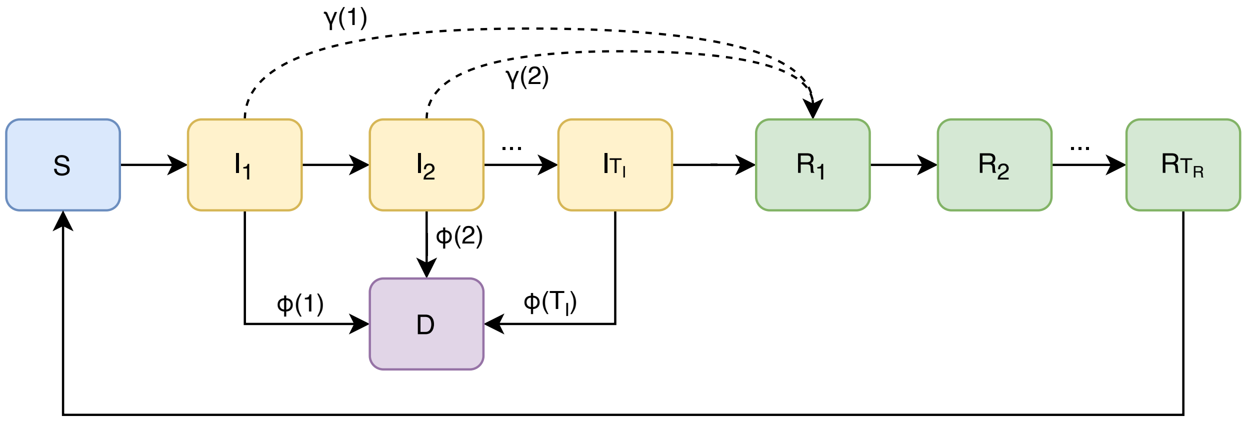


Figure 5.1: States flow of the SIRDS model.

are not infectious yet. Also, by calibrating these probabilities, it is possible to configure more complex scenarios where the probability of infections evolves over time.

The ratio of new infected individuals (i.e. individuals in the first infected state) at time t ($i^t_{i,j}$) is described in Equation 5.4. This ratio directly depends on the proportion of infected individuals in the neighboring cells and the population density ratio between neighboring cells and the origin cell.

$$i^t_{i,j} = min(S^{t-1}_{i,j}, S^{t-1}_{i,j} \cdot \sum_{(\alpha,\beta)\in V, p\in\{1,\ldots,T_I\}} (c^{(\alpha,\beta)}_{i,j} \cdot m^{(\alpha,\beta)}_{i,j} \cdot \lambda(p) \cdot \frac{N_{i+\alpha,j+\beta}}{N_{i,j}} \cdot I^{t-1}_{i+\alpha,j+\beta}(p)) \cdot (1 - \varphi(p)))) \tag{5.4}$$

As show in this equation, this model also considers the following parameters:

- Connectivity factor ($c_{i,j}$): factor representing how many connections are between two neighboring cells, including mobility infrastructures and transportation facilities.
- Mobility factor ($m_{i,j}$): the probability of an individual in a cell $(i + \alpha, j + \beta)$ to move to cell (i, j).
- Infection rate ($\lambda(p)$): probability of susceptible individuals becoming infected when in contact with the infection.

Equation 5.4 takes into account the subgroups of individuals that surpass the disease in each neighbor cell. Consequently, the rest is the ratio of new *deceased* individuals, defined as follows:

$$d_{i,j}^{t} = \sum_{p \in \{1,\ldots,T_I\}} (I_{i,j}^{t-1}(p) \cdot \varphi(p)) \tag{5.5}$$

Regarding to the *susceptible* population, the calculation of the next step is represented in Equation 5.6. The new *infected* individual ratio is subtracted from the *susceptible* population of the previous step, while the population that stop being immune are added.

$$S_{i,j}^{t} = S_{i,j}^{t-1} - i_{i,j}^{t} + R_{i,j}^{t-1}(T_R) \tag{5.6}$$

The states transition for the *infected* steps is calculated as shown in Equation 5.7. For the first *infected* state, the value corresponds to the ratio of newly infected individuals. For each transition, this value is propagated to the next *infected* states, removing the suitable percentage of deceased and recovered individuals for each step (based on the γ and φ parameters).

$$I_{i,j}^{t}(p) = \begin{cases} i_{i,j}^{t} & ,if\ p = 1 \\ I_{i,j}^{t-1}(p-1) \cdot (1 - \varphi(p-1)) \cdot (1 - \gamma(p-1)) & ,if\ 1 < p \leq T_I \end{cases} \tag{5.7}$$

Similarly, the first *recovered* state is calculated for each transition as the sum of the recovered individuals in each *infected* state. In the last infected state, I_{T_I}, γ is always considered as 1 (i.e. all the individuals in the last *infected* step are forced to move to the first *recovered* state). For the rest of the *recovered* states, the ratio is simply taken from the previous *recovered* state, until reaching the last one. After this, the population is again considered to be susceptible to the disease. The calculation of *recovered* population is represented in Equation 5.8.

$$R_{i,j}^{t}(r) = \begin{cases} \sum_{p \in \{1,\ldots,T_I\}} (\gamma(p) \cdot I_{i,j}^{t-1}(p)) & ,if\ r = 1 \\ R_{i,j}^{t-1}(r-1) & ,if\ 1 < r \leq T_R \end{cases} \tag{5.8}$$

The output of these models can be visualized in several ways. For generating evolution plots, several Jupyter notebooks were developed. A sample of these visualizations can be seen in Figure 5.2. It shows three subplots, considering: (i) the evolution of the three main categories (*susceptible*, *infected*, and *recovered*), (ii) the ratios of new *infected* and *recovered* individuals, and (iii) the evolution of deceased population.

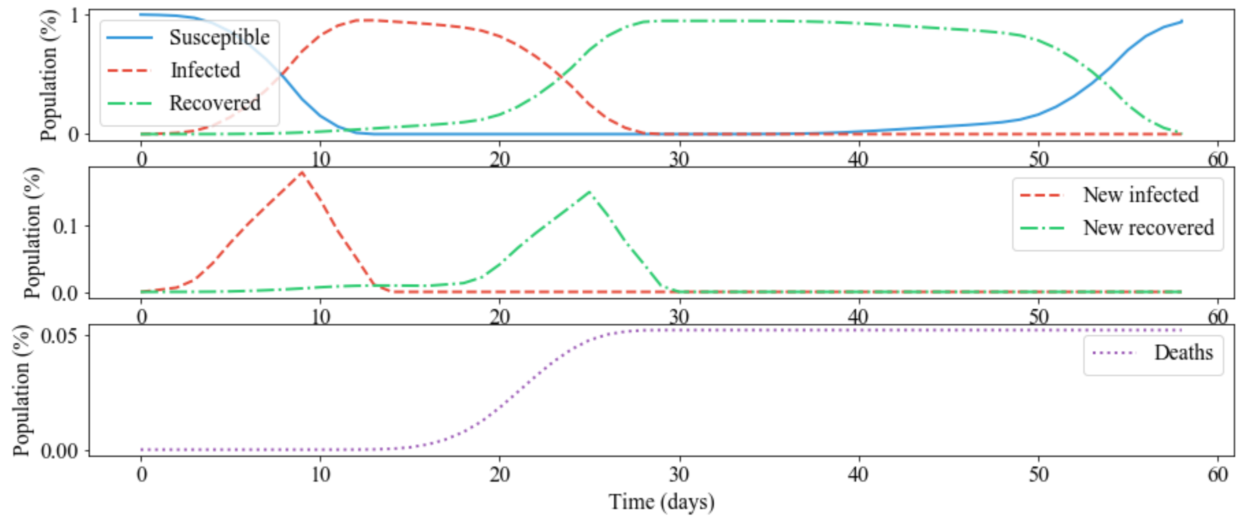


Figure 5.2: Sample visualization generated based on the SEIRD model results.

Moreover, Cadmium simulations produce logs containing states and outputs traces that can be used to study graphically the results of the simulation. The online tool Cell-DEVS Web-viewer[115] allows us to easily visualize these results by uploading the definition and log files and displays cell information and activity according to our style definitions. Figure 5.3 shows some sample visualizations produced with this tool. These four scenarios correspond to the same epidemic model and initial state, but vary parameters as the infection rate and the virulence of the epidemic. As a result, the number of immune (represented in green) and deceased (represented in black) population changes drastically for the same period. These visualizations can help us to analyze and understand the spread of the epidemics, becoming a great tool for the study of palliative measures and the estimation of epidemics' repercussions.

The model presented here can be easily adapted to represent the behavior of complex

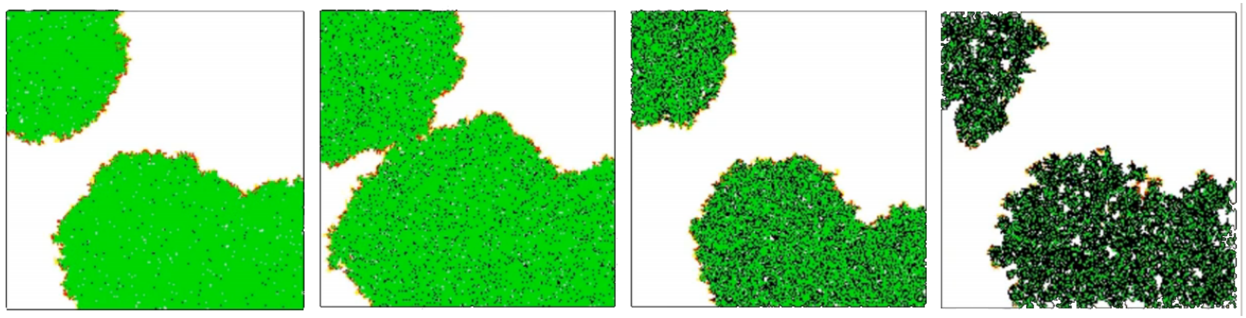


Figure 5.3: Sample Cell-DEVS Web-Viewer visualization. It corresponds to four scenarios of the same epidemic model in the same simulation time but with different virulence and infection rates. The colors of the cell grid corresponds to the susceptible (white), infected (yellow), immune (green), and deceased (black) population.

epidemic scenarios, reflecting government measures and changes in the population behavior. Along with the presentation of the model[33], we showed an example of dynamic scenario implementation, proposing a sample scenario using some data extracted from the spread of the SARS-CoV-2 in South Korea (including actual infection and recovery rates). We configured a 50x50 grid, with a population of $N_{i,j} = 100$ individuals per cell. We set the connectivity factor at 1 and the mobility factor to 0.6 for the neighboring cells. We used the cell in the middle to trigger the epidemic (with an infection ratio $I_{25,25} = 0.3$, a susceptible ratio $S_{25,25} = 0.7$, and a recovery ratio $R_{25,25} = 0$). We set the infection phase length T_I to 22 days. The individuals experience the first symptoms on the 4th day and isolate themselves on the 8th day. Until this event, we establish a fixed infection rate $\lambda = 0.15$. For the rest of the period they are considered isolated, and their λ is reduced to 0.01. The recovery rate γ is set to 0.07 for all the infected states. Figure 5.4 show the evolution of this spread in the cellular automata-based grid scenario. At the beginning of the progression (Figure 5.4a), we can see a majority of susceptible individuals (darker grey) and the core of the epidemic, representing the first infected (black) and exposed individuals (lighter grey). In the next captured simulation time (Figure 5.4b), we can see how the infection spreads, leaving behind a trail of immunized individuals. As no data regarding the immunity period were available for the COVID-19 at the moment of developing the model, this field was decided to be filled with a small value, producing a single infection wave and making the individuals rapidly transition to the susceptible state again. It is worth noting that this decision aims to describe how the model can be configured to describe short-immunity epidemics and does not try to reproduce the actual SARS-CoV-2 immunity period. Finally, Figures 5.4c and 5.4d, show how the epidemic grows until reaching the edges of the scenario, causing the end of the epidemic wave (facilitated by the immune individuals' barrier).

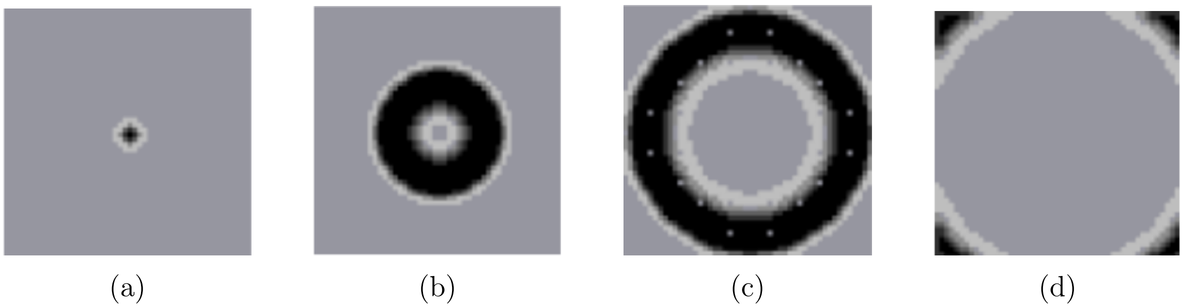


Figure 5.4: Infected rate reported by simulations at day (a) 5, (b) 11, (c) 42, and (d) 60.

The implementation of this model is publicly available[178], and several simulation scenarios, including the ones presented here, can be accessible online[114].

A different use example is depicted in Figure 5.5. In this case, we used the epidemic model (before adding the deceased mechanisms) to study how the disobedient population (i.e. people not following the government protection measures against the epidemic) impacts the ratio of infected population, and the distribution of the infected population over time. For this, we implemented a 10-4 model[106], where the population follows a cyclic schedule of 4-day work and 10-day lockdown. This restriction tries to prevent the resurgence of the epidemic while providing part-time employment, considering that a patient is not contagious to others for some days after they are infected. In the two top plots in Figure 5.5 we see how a specific epidemic scenario behaves when no restrictions are set over the population. We can see how the epidemic, although it is present for a shorter time, has a high infection ratio peak. Such a ratio would probably result in the collapse of medical infrastructures, being one of the main concerns when facing an epidemic as it would unnecessarily increase the number of resulting deaths. The next pair of plots show how the 10-4 model helps to mitigate the impact of the epidemic, maintaining considerably lower rates of the infected population overtime at the cost of lengthening the duration of the epidemic. In the following cases, we increment the disobedience ratio to 20%, 50%, and 80%. Comparing the outcomes for these scenarios, it is possible to see how as disobedience increases, the infected curve looks more and more like the scenario without measures.

These examples show some insights of the value that these models have in the decision-making and the development of countermeasures when a new epidemic appears. This was again checked with the recent COVID-19 outbreak, after which a large number of models were developed around the world to face the spread[10,11,88,102,222]. Through them, several critical factors as the lockdown policies, mobility restrictions, or vaccination strategies were studied, resulting in more effective procedures and a better evolution of the epidemic.

5.3 HMS Model-Aided Design

This section describes the development of a model-aided design of a VHDL implementation of a Healthcare Monitoring System (HMS) able to predict pain episodes in migraine crises up to 45 minutes before the onset of pain. Section 5.3.1 discusses the socioeconomic impact that the migraine disease has in our societies, as well as various relevant peculiarities of this disease that had to be taken into account when developing the system. Section 5.3.2 explains in detail the VHDL implementation of the system, described the adapted system architecture and the synchronization mechanisms of the device.

5.3.1 Migraine disease impact

Migraine is one of the most disabling neurological diseases. It affects around 10% of population worldwide[120] and 15% in Europe[194]. Also, migraine sufferers are more prone to suffer from other diseases such as fatigue, anxiety, or cardiovascular problems, which leads to high costs for private and national health systems. In Europe, it is estimated that migraine leads

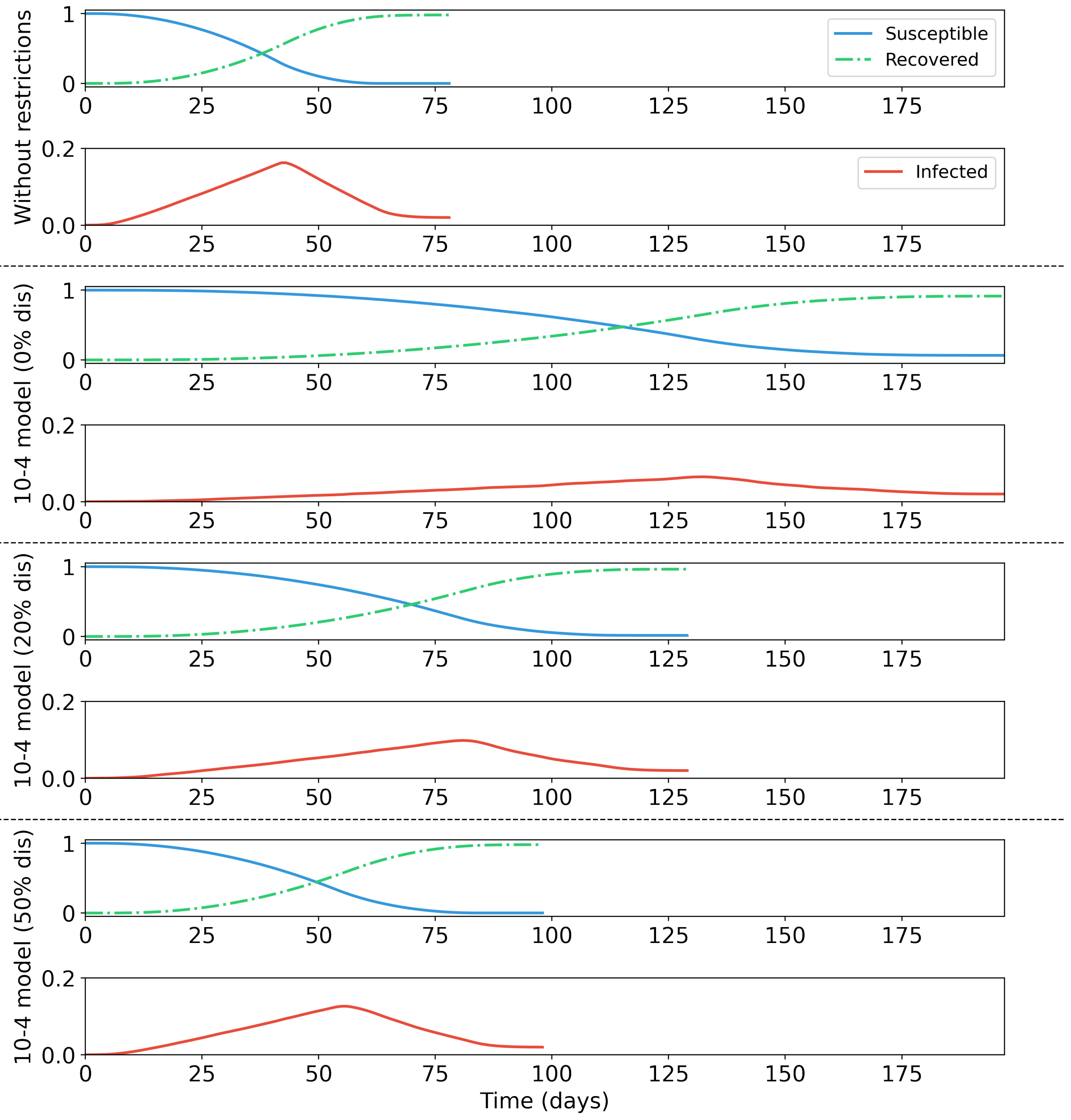


Figure 5.5: Comparison of the progression of a sample epidemic scenario, showing the evolution differences without applying any restriction, and imposing a 10-4 strategy with different disobedience levels.

to direct and indirect costs of €1,222 per patient per year[120].

Apart from the pain phase, migraine disease includes other less-known symptoms. Premonitory or prodromic symptoms may occur from three days to hours before the pain starts[69]. They are subjective, varied, and include changes in mood, appetite, sleep, etc. Auras occur in one-third of the cases[153] and appear within 30 minutes before the onset of pain. It consists of a short period of visual disturbance. Postdromes are symptoms that occur after the headache. Some of the most common are tiredness, head pain, or cognitive difficulties. They are present in 68% of the patients and they have an average duration of 25.2 hours[107].

It is difficult to estimate the onset of pain to make an effective intake of drugs. The time response of the pharmacokinetics of the drugs (the mechanisms of absorption and distribution of substances in an organism) does not match the long times of the vague predictive symptoms, or the short times of the urgent auras. So, most migraine sufferers wait until the onset of pain to take the rescue medication. The delayed intake reduces the effectiveness of the treatment.

5.3.2 Use Case: Migraine Prediction System

In this section, we show the model-driven development of an HMS capable of predict migraine pain episodes. The implementation of the HMS firmware was developed in VHDL using a Zynq-7000 FPGA as the target device, and it is based on the model presented by Pagan et al.[159]. This simulation describes a system able to predict pain episodes in migraine crises up to 45 minutes before the onset of pain, following the predictive methodology described in[85]. The modular and hierarchical nature share by the simulation formalism and the hardware specification language makes suitable this model-driven workflow, simplifying the development and reducing the implementation times. In this way, only a few changes have been introduced to adapt the simulation model to the hardware implementation, including the addition of a module to centralize shared data and a replacement of the N4SID prediction algorithm for an Auto-Regressive model with eXogenous inputs (ARX) model. It is worth noting that, because of the modular structure of the system, this change of prediction algorithm does not imply changes in the predictive methodology. The prediction module is implemented as an easily interchangeable component and was only changed in the hardware implementation because the chosen ARX model presents a lower complexity and power consumption compared to the N4SID algorithm.

The following sections describe the development of this HMS. First, original predictive simulation on which we base the work is summarized. Then, the particularities of the VHDL implementation are presented, including a description of the main components and synchronization signals of the system. Finally, we show how the system was validated, using real data of migraine patients.

Migraine Prediction Simulation Model

This model-driven implementation was developed based on the DEVS simulation model presented by Pagan et al. for migraine prediction[159]. The system allows the capture of different hemodynamic variables and generates alarms to warn the migraine patients of the proximity of pain episodes up to 45 minutes in advance. Moreover, this robust model can continue generating these alarms even if errors are present in one of the input signals. For this, it trains different sets of models: one using all the variables, and some extra models trained to work without one of them. The system has eight inputs. Four of them correspond to the hemodynamic variables. They are autonomous nervous system-related signals, captured through in-body sensors. They measure the body temperature, electrodermal activity, heart rate, and oxygen saturation. The remaining four inputs correspond to restoration buttons, activated by the users when the sensor activity is restored after an error appears. The system can detect three types of errors: (i) saturation, (ii) fall, and (iii) noise. To study the response of the system to these errors, the simulation model includes `ErrorInductor` components that emulate these failures. These components receive the actual sensors data stored in log files and modify them simulating this kind of error, before entering the prediction system. These error inductors adapt the signals to reproduce several issues that affect sensors in real life (noise, saturation, and disconnections). As outputs, it has the alarm signal itself and four LEDs indicating the presence of any error in the input signals.

The work presented by Pagan et al. included the possibility of using different types of predictive mathematical models such as Grammatical Evolutionary algorithms and state-space models. In the implementation of the HMS, we considered the latter, a Subspace State Space System Identification (N4SID) model to generate the prediction of the new pain onset probability. N4SID is a state-space-based algorithm[203], which describes immeasurable states and specifies differential equations that relate future outputs with current and past inputs. It is formally described in Equations 5.9 and 5.10:

$$x_{k+1} = Ax_k + Bu_k + w_k \tag{5.9}$$

$$y_k = Cx_k + Du_k + v_k \tag{5.10}$$

where u_k are the $U = 4$ hemodynamic inputs—body temperature, sweating, heart rate, oxygen saturation—at time k. y_k is the output at time k. In this project, it corresponds to the predicted pain level. A, B, C, and D are the state-space matrices. v_k and w_k represent white immeasurable noises[160].

From DEVS prototype to VHDL implementation

Figure 5.6 shows the root component resulting to adapt this DEVS system[159] as an VHDL implementation. Its main components include:

- `Drivers`: able to read the input of the sensors (interpreting the appropriate protocols), calculate the corresponding physical magnitude, convert the obtained measurements to the appropriate data type and send them to the synchronizer (`Sync`).

- `Sync`: it packs the input values of the different sensors into a unified data structure. Each minute it receives a pulse signal and averages the values received for each variable. It receives 180 samples of temperature and sweating and 60 of heart rate and oxygen saturation per minute. That information is sent to the Sensor Status Detectors (`SSDs`).

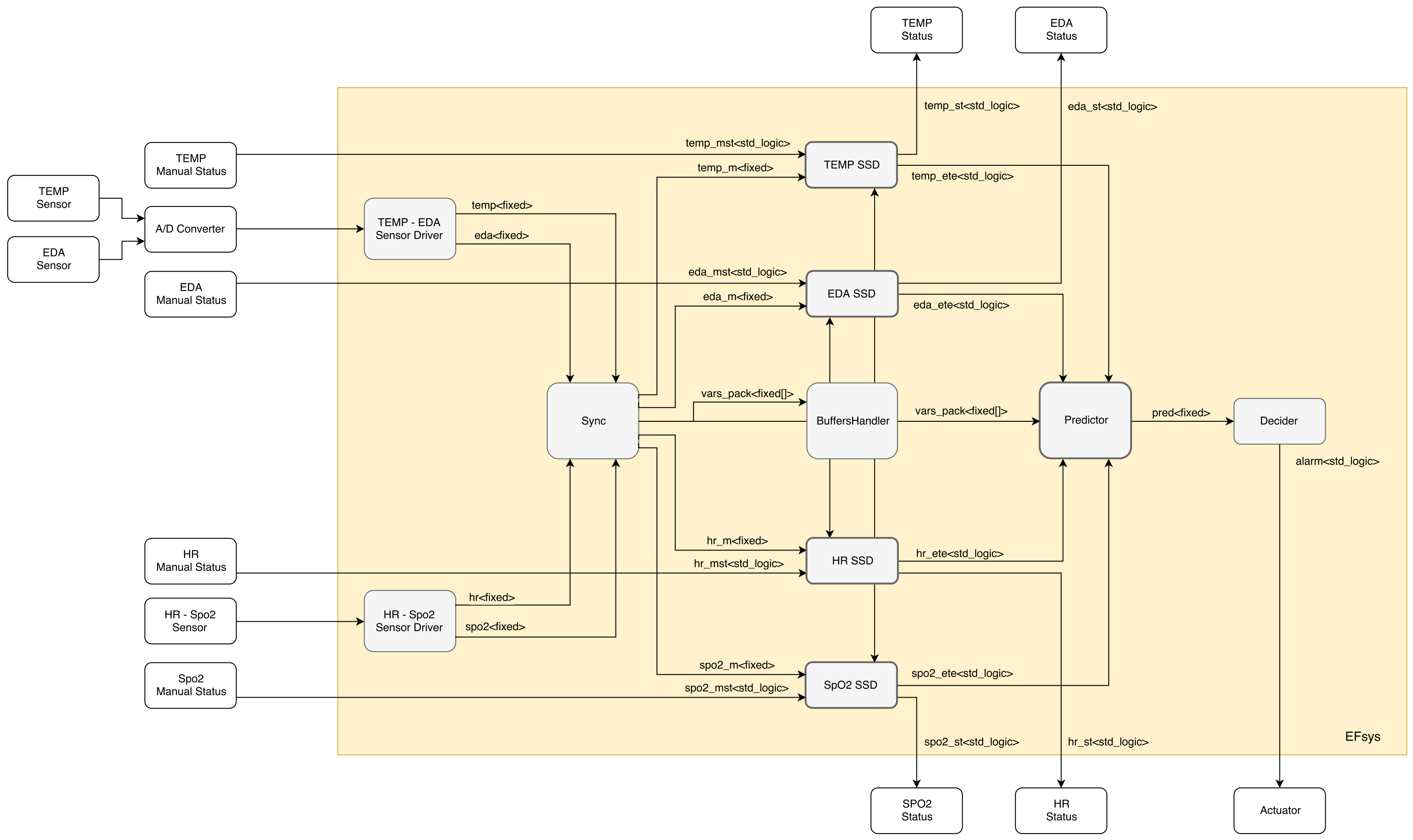


Figure 5.6: Root component of the migraine pruning system implemented in an FPGA with VHDL.

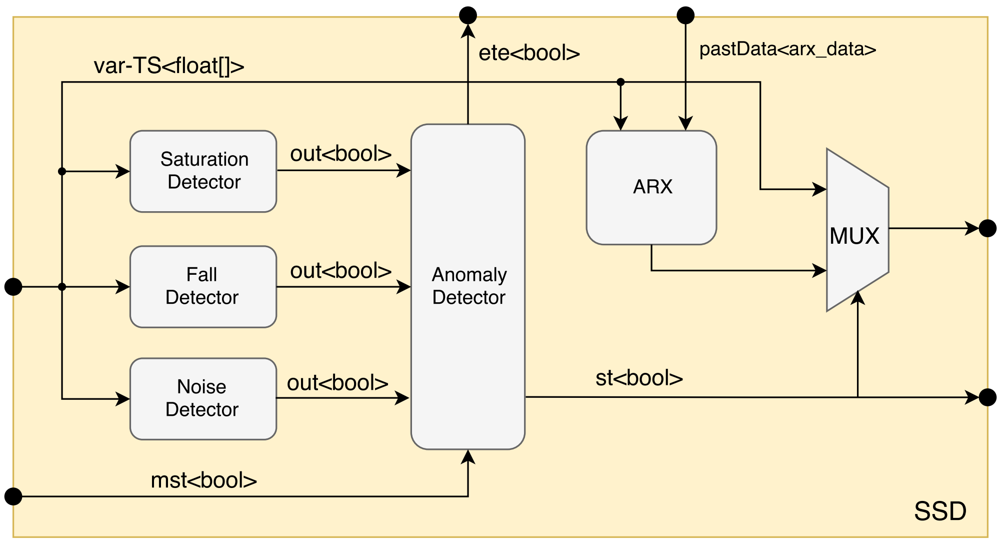


Figure 5.7: `SSD` module. It detects errors in the input signal and recovers it (with the `ARX` module) while it is not restored.

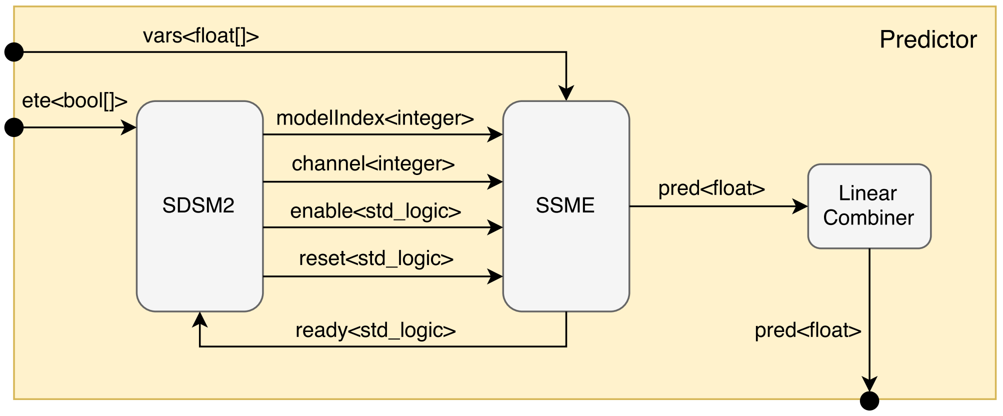


Figure 5.8: `Predictor` module. It generates predictions of new pain episodes. The `SDMS2` module controls the predictions generation and the `LinearCombiner` group them together to produce a single output.

- `SSDs` (Figure 5.7): they check if different types of errors are present in the input signal (saturation, fall or noise). If one of them is detected, the status signal raises, so that the patient can restore the sensor. While the sensor is not restored, the system tries to repair the signal temporarily. An `ARX` module is used for that purpose. It generates estimations of the input using previous samples of both the controlled variable and the exogenous ones. When it has passed too much time since the error detection, an *Elapsed Time Exceeded* (ETE) signal is raised. That signal points out that the variable implied can not be used reliably to generate predictions, so the `Predictor` module will discard it. The `SSDs` are connected to the output of the `Sync` , instead of going after the `Drivers` . That is because error detection and signal repair capabilities have to operate over data separated by minutes. In the DEVS simulation, the input data was already stored by minutes. Conversely, in the VHDL implementation, the synchronizer is in charge of doing this task.

- `BufffersHandler`: it stores and handles previous inputs of the system, used by the different `ARX` modules present in the SSDs.

- `Predictor` (Figure 5.8): it generates a prediction of occurrence of a new pain episode. For that, it contains 5 sets of models[160]. Each one of them is related to a different group of three or four input variables. Each set has 3 predictive models whose results are combined to refine the prediction. These models are trained for different prediction horizons. In this way, when a sensor fails or recovers, the suitable set of models used to generate predictions is selected. In this way, the system can generate predictions when all the variables are fine or when there is an error in only one of them. With less than three variables the prediction is not considered representative and is not supported by the system. The management of the state-space models is done by a unique model in the Predictor (`SSME`). In this way, the `SDMS2` module is responsible for selecting the correct models and requesting the generation of the three predictions, which will be carried out sequentially.

- `Decider`: activates the alarm when the output generated by the `Predictor` module exceeds a certain threshold (trained previously with several hours of data).

System synchronization is controlled by a set of clock and pulse signals. These can be seen in Figure 5.9 and are the following:

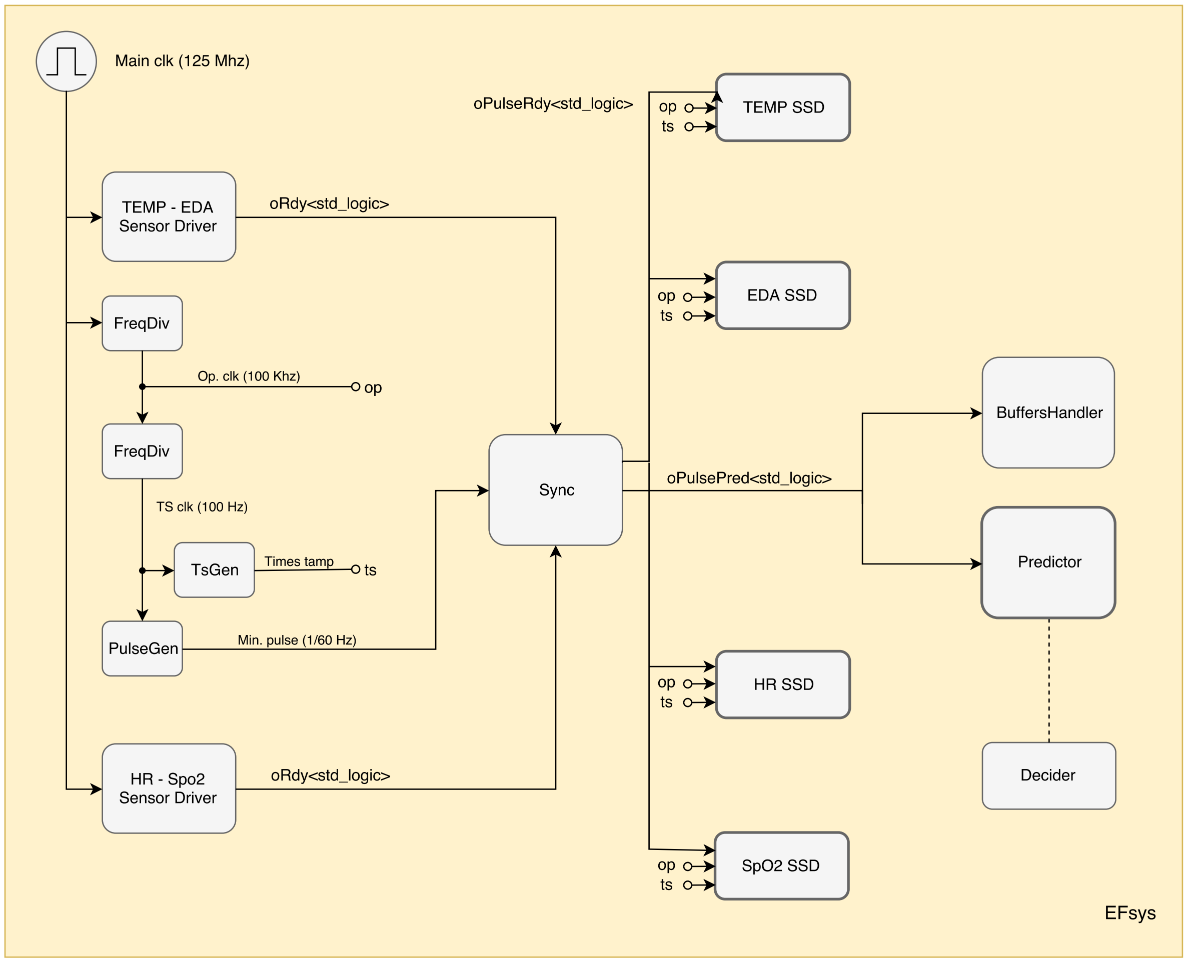


Figure 5.9: Clock and pulse signals used to synchronize the prediction system.

- Main clock: reference clock used in the `Drivers` for controlling the communications and in the generation of all the remaining clocks and pulses.
- Operations clock: used for the synchronous components of the system to control its operation (except `Drivers`). The clock frequency is 100 kHz.
- Timestamp clock: used to generate pulses each minute. That pulses are used to cause the `Sync` to generate new packets.
- `Drivers` pulses: used to notify the `Sync` of a new reading.
- `Sync` pulses: after the preparation of a new packet a pulse is generated to communicate

that fact to the `SSD`. Later, it waits a given number of cycles and generates another pulse that informs the `Predictor` and `BuffersHandler` modules of the presence of the new data. That delay corresponds to the processing time spent in the `ARX` modules to regenerate the signal (if necessary).

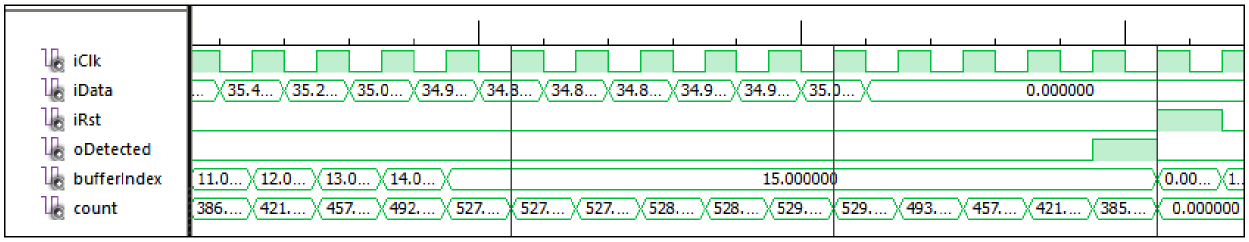
To deal with decimal numbers in VHDL the FLOAT32 data type was firstly used, as in the original DEVS model. However, when the system was synthesized it needed too many resources to handle the operations. For this reason, a fixed precision data type was used instead. For assuring that all the operations can be supported without the appearance of overflows, auxiliary fixed data types were used. The size of these customized data types is adapted taking into account the different operations that are performed in the system, thus avoiding the appearance of overflows.

Validation of the HMS

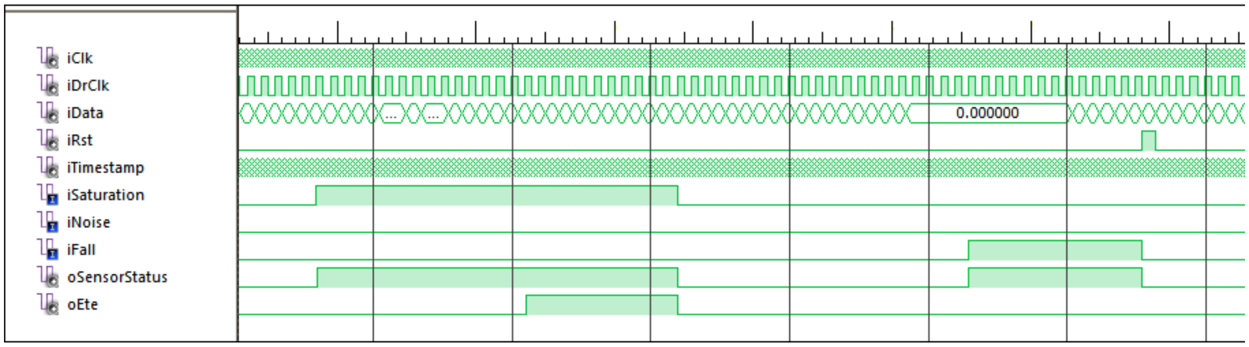
To implement the migraine prediction system, the design software Xilinx ISE 14.7 has been used, and a Zynq-7000 FPGA has been established as target FPGA (XC7Z010 device, CLG400 package). The components of the system were simulated using the ISE simulator (ISim).

To validate the system, data acquired from real patients are used. They were monitored in ambulatory conditions with a Wireless Body Sensor Network (WBSN), as described in[160]. They are saved so that a value for each of the variables is available per minute.

Figure 5.10 reflects how the errors are managed in the SSD modules. Figure 5.10a shows how the `FallDetector` module behaves. When it detects a fall, the signal *oDetected* is raised, notifying the new error. Once this signal has been activated, it remains in a high state until the data in the buffer are valid or the reset signal (*iRst*) is raised. When the module resets, the buffer is emptied and has to be refilled before detecting new errors in the signal. This situation occurs when the patient presses a restoration button after being informed of the failure of one of the signals. The `SaturationDetector` and `NoiseDetector`



(a) `FallDetector`, which detects falls in the signal



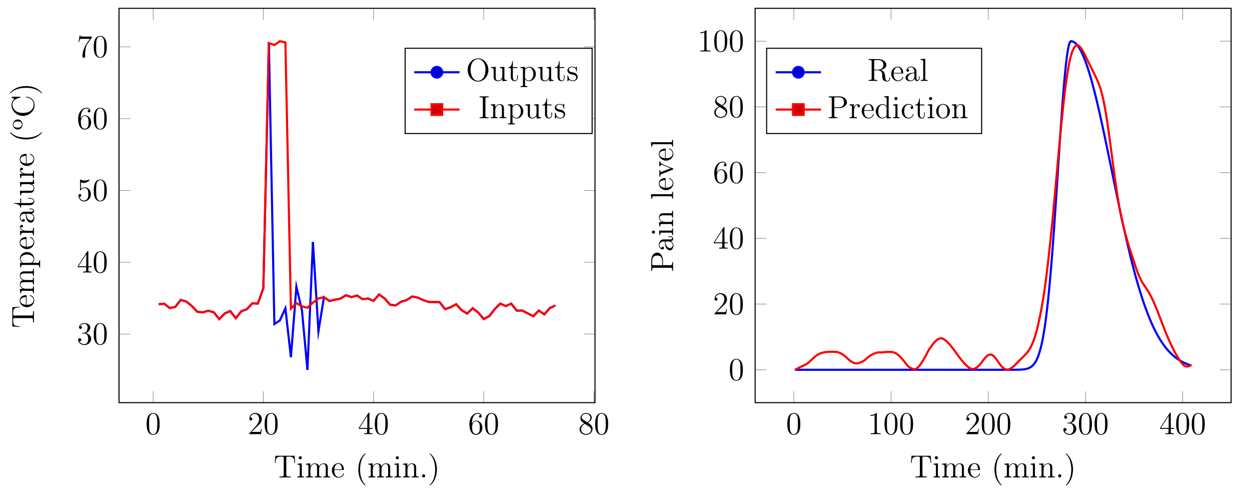
(b) `AnomalyDetector`, which manages the errors provided by the detectors

Figure 5.10: Error handling in the *SSD*.

modules present a similar behavior. The outputs of those three detectors are attached to the `AnomalyDetector` module. That module manages two output signals: *oSensorStatus*, which will be activated when at least one error is detected in the signal, and *oETE*, which will be activated when an error lasts for a certain time. Figure 5.10b shows the aforementioned situation.

When the *oSensorStatus* output of the `AnomalyDetector` is raised, the `ARX` enables. Figure 5.11a depicts the response of this module to a saturation error. Once the error is detected, the `ARX` module corrects the signal and approximates the values the variable should take if it were fully operational.

These signals, after going through the `SSD` modules, reach the `Predictor`. In that module, a set of three state-space models are selected based on the *ETE* signals of the four SSDs. After that, three predictions are generated, one for each model of the selected model set. Those predictions are unified by making an average. Figure 5.11b illustrates the predictions generated by the system in response to a known episode. Those predictions



(a) SSD outputs after an error detection (b) Comparison between real and predicted pain levels

Figure 5.11: Outputs of the predictive system.

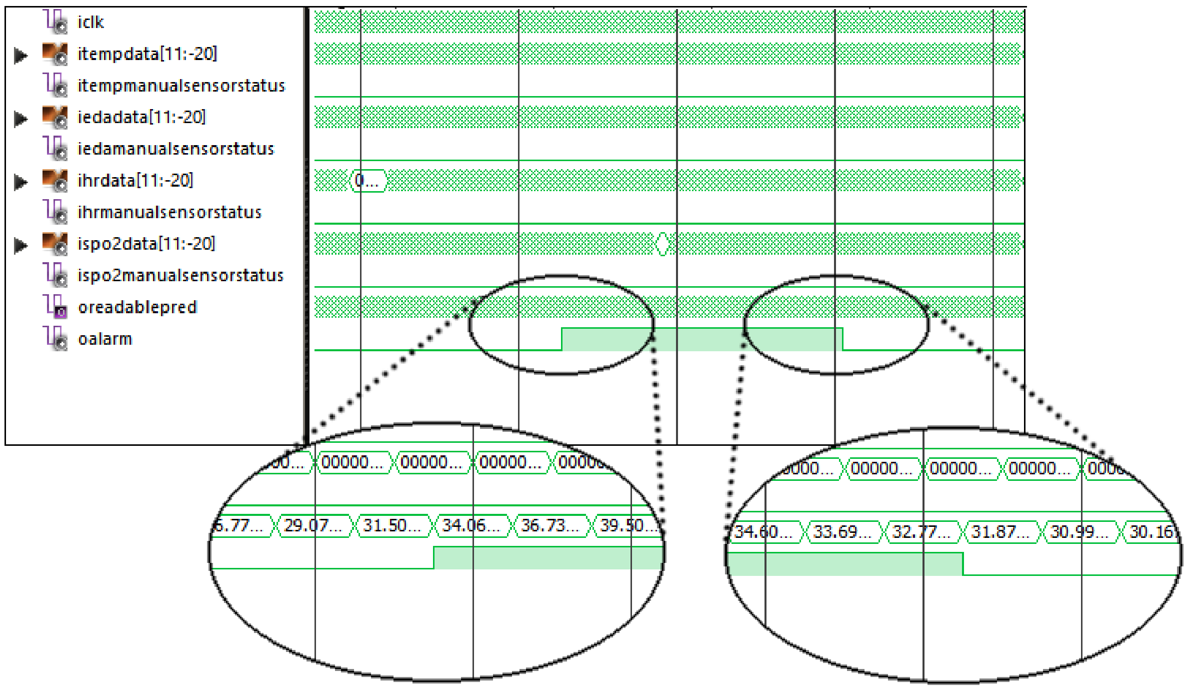


Figure 5.12: General view of the system simulation.

are compared with the subjective pain curve, generated with patient data. The prediction oscillates over the reference curve and triggers an alarm when the migraine event occurs. This trigger is based on a threshold over the generated prediction, obtained by studying the previous data. When the prediction goes over that threshold it is considered that the pain episode will occur in the next few minutes. The generated prediction curve has a fit of 83.63% with respect to the original one.

Finally, Figure 5.12 shows a simulation of the root component of the system. It shows the four hemodynamic variables data corresponding to a real episode, stored previously, and the prediction generated by the system. In this way, it can be seen how the *oAlarm* signal is raised when the prediction provided by the corresponding predictors set exceeds a threshold (32 in this example). Consequently, the alarm signal is set low when the value goes back below that threshold.

5.4 Modular DEVS-based methodology for developing robust prediction systems

Over the years, more and more data-collection devices have been created and acquired by both individuals and institutions. These data are usually accumulated in different types of data stores and, adequately combined and processed, can lead to useful insights and conclusions. However, before having the information in a unified format, several critical tasks have to be performed. This includes importing, filtering, normalizing, and merging the data coming from heterogeneous data sources. Such procedures lead to the emergence of several challenges. In these cleaning stages some data are discarded or transformed, having to deal with specific problems as duplicated or contradictory values, naming conflicts, inconsistent timings, or structural inconsistencies[9,171].

Once all the heterogeneous data sources are processed and unified, the resulting information can be adapted and used to generate specific models and systems. For that, different machine learning techniques can be applied, including kernel methods, classification, regression, and neural networks. Despite the number of variants, there is a well-known and widely used subset of machine learning techniques. They are usually present in the literature and implemented in a variety of machine learning software and libraries[165,219]. When applying these techniques to a specific use case, several machine learning algorithms are chosen based on the data types and problem description. This process of training, validating, evaluating, and comparing different sets of models is usually a repetitive and time-consuming task. By applying appropriate M&S techniques, the whole process can be automated, allowing better control and analysis of the parameters involved in each model.

Aiming to contribute to the automation of these tasks, a model-based methodology has been created, encapsulating the main filtering, processing, and modeling operations. In the following sections, some of its features and possibilities are explained. First, the main aspects of the structure and components of the methodology are discussed. Second, some data importing mechanisms are outlined. Then, the process of describing the system through XML files is explained. Finally, a use case applying this methodology for predicting stroke types and outcomes is shown.

5.4.1 Prediction systems development workflow

The goal of the methodology is the automatic creation of predictive models, based on specification files. Figure 5.13 shows the common phases followed when creating these types of models. First, data from one or several data sources are collected (`Data sources` element in Figure 5.13). These data are initially filtered following some common criteria, to assure their quality and discard wrong or incomplete records. Also, feature selection procedures can be performed in this stage if necessary. Both tasks are performed in the `Filter` module. Second, normalization operations are applied to the `Normalize` module according to the specific needs of the use case. This stage is especially important when heterogeneous data sources are taken into account. It includes unifying the units of the variables, categorization tasks, and adjustments in the implied values or distributions. At the end of this stage, all the sources have to be unified in a single format. Hence, each data source may need different processing to convert their data to this common view. Third, some transformations may have to be applied in the pre-processing stage to adapt the data format to the use case (`Pre-Process` module in Figure 5.13). For example, in the health domain, single samples coming from a patient monitoring device can be too granular to be directly related to the event to predict. In this situation, it is common to split clinical data into time-windows of a predefined size so that a certain number of consecutive samples matches with the class to predict. Next, the `Pre-Process` module separates the input data into several datasets: train datasets, which are used to fit the model, and validation/test datasets, which are used to adjust the models and select the best ones. Once the separation is done, different sets of models are generated with the training dataset using the suitable machine learning algorithms in the `Train` module. The set of algorithms to use depends as much on the problem characteristics as on the data types. They can include classification and regression algorithms, neural networks, and kernel methods, among others. Once trained, models are verified and compared using the previously separated validation/test datasets in the `Validation/Test` module. The selected models will be subsequently evaluated to generate valuable information of the domain to be modeled. New samples can be used both to evaluate the models and to feed the data sources. This allows us to generate new models periodically so that their accuracy can be increased over time.

In the presented M&S methodology, the main operations performed in each stage are encapsulated and implemented in the set of modules described above. Each of these modules has a set of parameters that can be modified to adapt the transformations performed according to the nature of both the input data and the predictive models. In this way, the processing of the data goes through a chain of modules, starting from the data sources and finishing with the generation and evaluation of the models, as Figure 5.13 depicts. Hence, the design and implementation of the proposed methodology include a set of instances of these modules (with the suitable parameters) and a set of couplings. Within a DEVS context, these couplings link output and input ports, which represent the input/output points of the modules.

All the stages involved in the data processing and model definition depicted in Figure 5.13

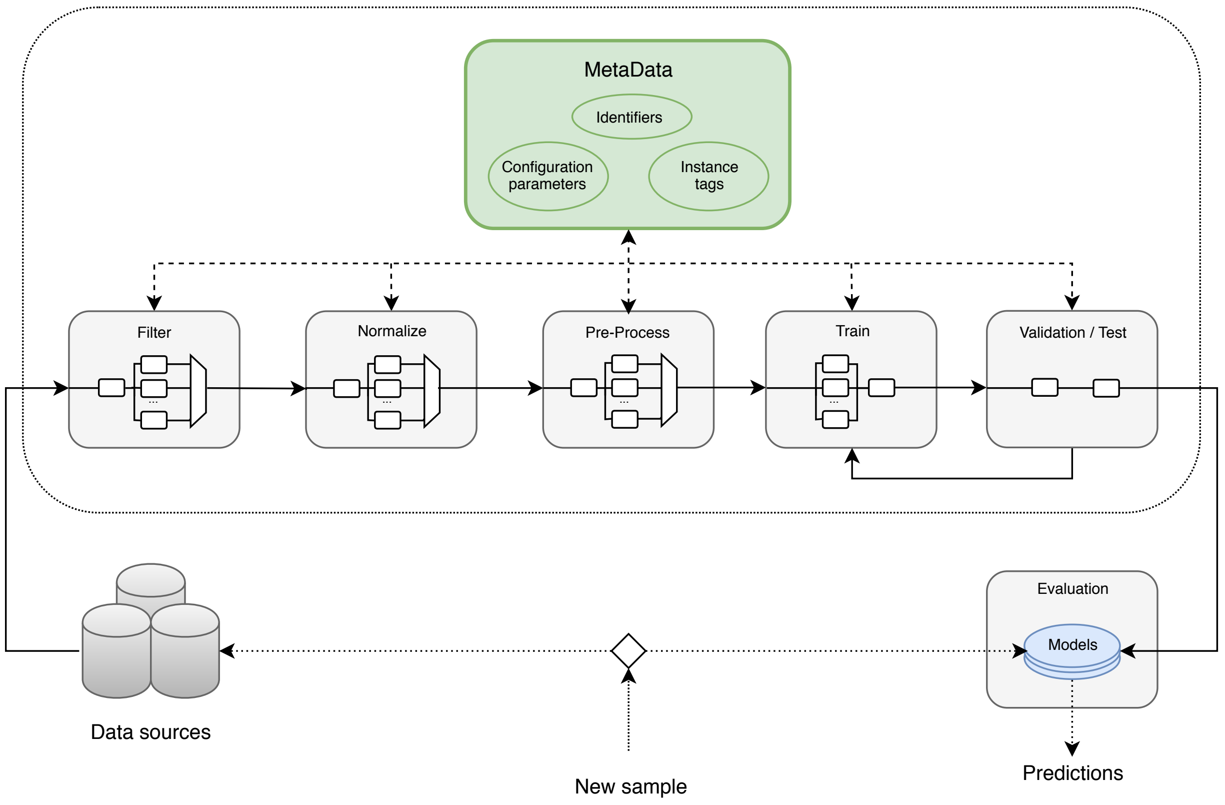


Figure 5.13: Global view of the modular methodology oriented to the generation of predictive models.

are configured with metadata related to the corresponding dataset, represented at the top of the Figure. This includes datasets and modules identifiers, specific configurations established for each of the modules to adjust the performed transformations, and custom tags reflected in the specification.Through this metadata, all the modules of the system are aware of all the transformations made in each dataset. Following this idea, modules can auto-adapt their operations based on the transformations performed by previous modules and other metadata (or even request configuration changes in other modules).

As a result, this modular methodology favors the reusability of the systems and modules. It also facilitates the creation of subsystems and, therefore, the generation of distributed systems, for scalability purposes. Furthermore, the main techniques and procedures are encapsulated in predefined modules, allowing us to specify, using configuration files, the structure and operation of the system, instead of directly coding its behavior.

In particular, the definition of this M&S framework for a given neurological disease is performed through the support of XML files. These files include the structure and behavior of the abstract M&S method illustrated in Figure 5.13, but focused on a particular disease, with its features and objectives. We have developed a framework that parses these XML files, and automatically generates the set of DEVS models that describes the behavior depicted in Figure 5.13, but for the specific disease. All the information propagated through the whole system is encapsulated into a special custom structure called *MetaFrame*. It contains both a table of data and metadata accumulated by all the modules included in the data path.

5.4.2 Importing data from different heterogeneous sources

To ease the process of integrating heterogeneous data from a variety of data sources, the framework includes an XML-based definition that allows importing and unifying data of multiple sources. This mechanism, depicted in Figure 5.14, creates and fills a database based on two XML specifications files. The first one defines the attributes and restrictions present in the dataset that the user wants to use. Here it is possible to introduce the data types for each attribute and some basic restrictions for their values. In the second one, these attributes are mapped to the actual data sources, allowing to create of the resulting datasets with the data present in several data sources. Among them, it allows importing data from Comma-Separated Values (CSV), Microsoft Excel spreadsheet files (XLS/XLSX), and several relational databases engines (as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, or SQLite). Moreover, several popular protocols are supported for retrieving the information from files, including File Transfer Protocol (FTP), Secure Shell (SSH), or Hypertext Transfer Protocol (HTTP). It is worth to note that the mechanism represented in Figure 5.14 is based on a set of Python scripts, and is independent of the DEVS-based workflow manager presented in Section 5.4.3.

As aforementioned, the specification files define the structure of the information that

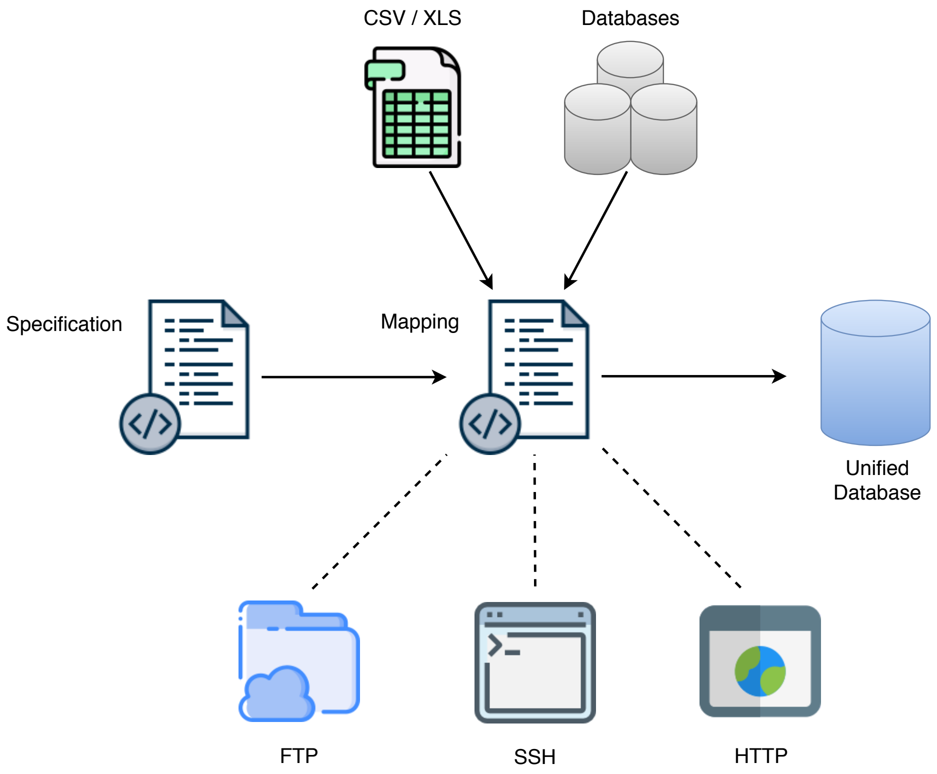


Figure 5.14: Importing data from heterogeneous sources. The specification file describes the information and the mapping file connect it with the actual data sources.

is contained in the database. This database can be used later as the main information source in the modeling part. As shown in Figure 5.15, this structure is divided in *datasets*. Each *dataset* can define one or more *attributes*, and corresponds to a database table. These attributes can include several restrictions and correspond to table columns. The following restrictions can be included:

- *key*: indicates if an attribute is part of the dataset key. Key attributes allow to uniquely identify registers in the dataset and are specified as primary keys in the resulting database.

- *type*: specifies the data type of an attribute. When mapping the data, the different imported values are transformed to the suitable data types based on these types.

- *min*, *max*: minimum and maximum values for an attribute, respectively. They are

```
<datasets name="datasets">
   <dataset name="dataset1" key="key_attribute_name" key_type="int">
      <var name="attribute_name1" type="text" />
      <var name="attribute_name2" type="date" format="%d/%m/%Y" required="true" />
      <var name="attribte_name3" type="enum" values="H,M" />
      <var name="attribte_name4" type="int" min="0" max="25" />
      <var name="attribte_name5" type="boolean" />
   </dataset>

   <dataset name="dataset2" extension="dataset1">
      <!-- ... -->
   </dataset>

   <dataset name="dataset3" parent="dataset2">
      <!-- ... -->
   </dataset>
</datasets>
```

Figure 5.15: Dataset specification file format. It allows to define the attributes of the datasets that the user wants to create, including data types and restrictions.

used to introduce constraints in the database tables definition.

- *required*: indicates if an attribute must always have a value. When it is set to *false* the corresponding table column accepts *NULL* values.
- *date_format*: specifies the format of the input dates.
- *enum_values*: specifies the different values of an *enum* structure, represented with a comma-separated string.

Additionally, when specifying the different datasets, several keywords can be used to include dependence relations (exemplified in Figure 5.16). The *extension* attribute allows extending a previous dataset definition. Thus the extended dataset contains all the variables defined in both datasets. The *parent* attribute allows to specify a parent dataset. By specifying it, the information in the child dataset is identified by a combination of both datasets.

After defining the datasets, an XML mapping file is specified for linking them to the actual data sources. The structure of this mapping file is shown in Figure 5.16. Inside

```
<mapping>
   <connections>
      <connection name="database" type="db" host="..." user="..." pwd="..."
   db_name="..."/>
      <connection name="ssh_server" type="ssh" host="..." user="..." pwd="..."/>
      <connection name="ftp_server" type="ftp" host="..." user="..." pwd="..."/>
   </connections>

   <sources>
      <!-- Local sources -->
      <source name="source1" type="csv" path="path/to/the/file.csv"/>
      <source name="source2" type="xls" path="path/to/the/file2.xls"/>

      <!-- Remote sources-->
      <source name="source3" type="db_table" conn="db" table_name="..." />
      <source name="ftp_test" type="csv" conn="ftp"
          path="/remote/path/to/the/file.csv" />
      <source name="ssh_test" type="csv" conn="ssh"
          path="/remote/path/to/the/file2.csv" />
   </sources>

   <datasets>
      <dataset name = "dataset1" source="source1,source2"
   join_on="source1.attr1=source2.attr2">
         <var name="attribute_name1" from="source1.attr1" />
         <var name="attribute_name2" from="source2.attr2" />
         <var name="attribute_name3" from="attr3" />
         <!-- ... -->
      </dataset>

      <dataset name = "dataset2" source="source2">
         <!-- ... -->
      </dataset>
   </datasets>
</mapping>
```

Figure 5.16: Mapping file format. It defines the relation between the datasets definition and the actual data sources.

a root *mapping* element, there are three main children: (i) *connections*, for defining the remote access points, (ii) *sources*, for defining specific locations of both local and remote data sources, and (iii) *datasets*, which links the information defined in the specification file with the one contained in the data sources.

The *connections* element contains zero or more *connection* children, with the following attributes:

- *name*: identifies the connection. This name is referenced in the *sources* section to use specific connections.
- *type*: specifies the type of the connection. It supports FTP servers (*ftp*), SSH servers (*ssh*), and relational databases (*db*).
- *host*: specifies the server related with the connection, with the corresponding IP or host name.
- *port* (optional): specifies the port used in the connection. If not specified, the default one is used for each protocol (21 for FTP, 22 for SSH, and 3306 for databases).
- *user*: user name to login in the server.
- *pwd*: password to login in the server.
- *db_name* (only for databases): selects a specific database in the server.

The *sources* element contains one or more *source* children, and specifies the location of specific local and remote data sources. Each *source* element contains the following attributes:

- *name*: identifies the data source. This name is used in the *datasets* section to reference specific sources.
- *type*: specifies the data source type (*csv*, *xls*, or *db_table*)
- *conn* (optional): specifies a remote connection, with the name used in one of the *connection* elements. If no connection is specified, it is assumed to be a local data source.
- *path* (only for data files): path of the data source in the file system.
- *table_name* (only for databases): name of a specific table of the selected database.

Finally, the *datasets* element maps datasets attributes (as defined in the specification file) with fields present in the data sources. It contains one or more *dataset* elements, whose *var* children corresponds to the attributes defined in the datasets description. Each *dataset* defines one or more sources in the *source* element. If more than one source is specified, a join condition has to be added in the *join_ on* attribute (as it can be seen in the *dataset1* element in Figure 5.16. For each *var* element, two attributes are required:

- *name*: identifier of the attribute, as defined in the specification file.
- *from*: source field used to fill that attribute. If only one source is used, or the field name is unique for among several sources, the field name is enough. However, if the same field name appears in different sources, it is required to define it in the following format: *source_ name.field_ name*. This can be seen in the first two *var* elements in the *dataset1* (Figure 5.16).

Both the specification and the mapping files are validated using XML Schema Definitions (XSD). This format, recommended by the World Wide Web Consortium (W3C), allows to formally describe the elements in an XML document. In this way, it is possible to warn the user when errors are found in the input files and improve the usability of the system.

5.4.3 Defining subsystems to format the input data and generate predictive models

The different systems composing a predictive platform are also described in this framework through XML specifications. In these specifications, the configuration of the different components is described, as well as the appropriate couplings connecting them. The framework includes multiple modules for each of the stages shown in Figure 5.13, facilitating the process of data processing and model training. Also, it provides a programming interface to implement new modules when needed, which can be easily reused in later projects. Moreover, due to the distributed capabilities of the xDEVS Python implementation, it also allows the communication of the different subsystems over the network in a straightforward way.

Figure 5.17 shows the XML structure for the definition of a predictive system. As it can be seen, it has four main sections: (i) *Nodes*, describing the access points of other subsystems, (ii) *Modules*, including all the modules that compose the system and their configurations, (iii) *Couplings*, which specify how these components are connected, and (iv) *ServedPorts*, which enumerates the ports of the modules exposed over the network to receive information from other subsystems.

```
<Root>
    <Nodes>
        <Node name = "node1" host = "..." port = "..." />
    </Nodes>

    <Modules>

        <[ModuleName] name = "module1"
             param1 = "..."
             param2 = "..."
             tag = "..." />

        <[ModuleName] name = "module2"
             param1 = "..."
             param2 = "..." />

    </Modules>

    <Couplings>
        <Coupling src = "module1.out_port" dst = "module2.in_port" />
        <!-- ... -->
    </Couplings>

    <ServedPorts>
        <Port name = "module1.in_port" />
        <!-- ... -->
    </ServedPorts>

</Root>
```

Figure 5.17: Systems specification format in the modular predictive framework. It includes nodes, modules, and couplings specification.

A system described using this framework is typically distributed into several subsystems. Communications among them can be performed directly with socket-based connections, or with a database as an intermediary. By breaking down the different main tasks of the processing workflow, it is possible to easily assign the execution of specific tasks to certain levels of the IoT infrastructure. In this way, the initial processing and the model training could be assigned to a powerful workstation in the Cloud, while the inference could be located in the Fog or Edge layers. For communicating with each other, the different subsystems include a *Nodes* section specifying the access points of the nodes to which they send information. These access points are specified with *Port* elements in the *ServedPorts* section, as show in Figure 5.17.

The *Modules* section declares different types of processing modules. As described in Section 5.4.1, these modules process and transform the information using a data structure called *MetaFrame*, which combines the popular DataFrame of the Python *pandas* module with additional auxiliary metadata related to the previous processing and characteristics of the contained data. Among these metadata, we found *MetaFrame* identifiers, user-defined tags (introduced by specific modules), and the transformations performed over the *MetaFrame* by previous modules.

Among the predefined modules, the framework includes input/output modules (to import data from different data sources, or to export the resulting information or models), visualization modules (for generating plots with the results), processing modules (to perform operations like filters, transformations, splits, joins, etc.), and training/testing modules (for generating and validating the models). Although their behavior is customizable through parameterization, the modeler can create its own modules if needed. A complete list of the modules contained in this framework, as well as the parameters available for each one, can be accessed in the official repository documentation[83].

Finally, the *Couplings* section specifies how to link the modules of the system. This specification is done with *Coupling* elements containing pairs of source/destination ports (as shown in Figure 5.17). When communicating with remote systems, a local output module port can be communicated with a remote input port following the notation: *[Re-*

moteNodeName].[ModuleName].[PortName].

5.4.4 Use case: Anticipating stroke types and outcomes for new crises

In this section, our modular predictive framework is applied to address the specification of a stroke prediction system, similar to the one manually developed by García-Temza et al.[65]. First, a brief context is given about stroke disease and its impact on our society. Second, a description of the data flow and the architectures of the subsystems that compose the predictive system is presented. Finally, some results generated by this example system are shown.

Impact of the stroke disease

Stroke is the third most common cause of death in developed countries, exceeded only by Coronary Heart Disease (CHD) and cancer[218]. Annually, fifteen million people worldwide suffer a stroke. Among these people, one third die, and another third became dependent, placing a burden on family and community. Survivors can experience loss of vision and/or speech, paralysis, and confusion, among others[218]. There exist two main types of strokes. Ischemic stroke appears as a result of an obstruction within a blood vessel supplying blood to the brain. It corresponds to 87% of all stroke cases. On the other hand, a hemorrhagic stroke occurs when a weakened blood vessel ruptures[14]. The Stroke Alliance For Europe (SAFE) estimated the total stroke cost in 2015 at €45 billion. Of them, €20 billion corresponds to direct costs for in-hospital care and therapy. The rest corresponds to indirect costs from informal care costs and productivity loss[57]. Multiple techniques are used to diagnose stroke. However, they are usually complex and are conducted by the medical staff. Hence, they are not accessible everywhere. This is especially important in non-urban areas. Moreover, for stroke diseases, time is critical[89]. These special situations make necessary to find alternatives that allow the generation of accurate diagnoses without incurring a significant delay in the treatment. We propose a system that generates predictions based on different sets of models, generated using several Machine Learning algorithms.

Data flow and system architecture

For this system, a dataset obtained from the Stroke Care Unit of the Princess University Hospital (Madrid, Spain) is used. It includes information about 118 stroke patients, collected from March to July 2017. Of them, 104 correspond to ischemic strokes, and the remaining 14 to hemorrhagic strokes. This coincides with the usual proportion of ischemic and hemorrhagic strokes. Following the general scheme provided in Figure 5.13, two types of models are generated: one to diagnose the stroke type and other to predict the *exitus* risk, both in the early stages after the episode. These predictions are intended to be generated when monitoring new stroke attacks, before conclusive tests have been done. In this way, the system could be used in ambulances, during the patient transportation from a rural area to his reference hospital, or upon arrival at the health center, and serve as a first diagnosis to facilitate the treatment of patients.

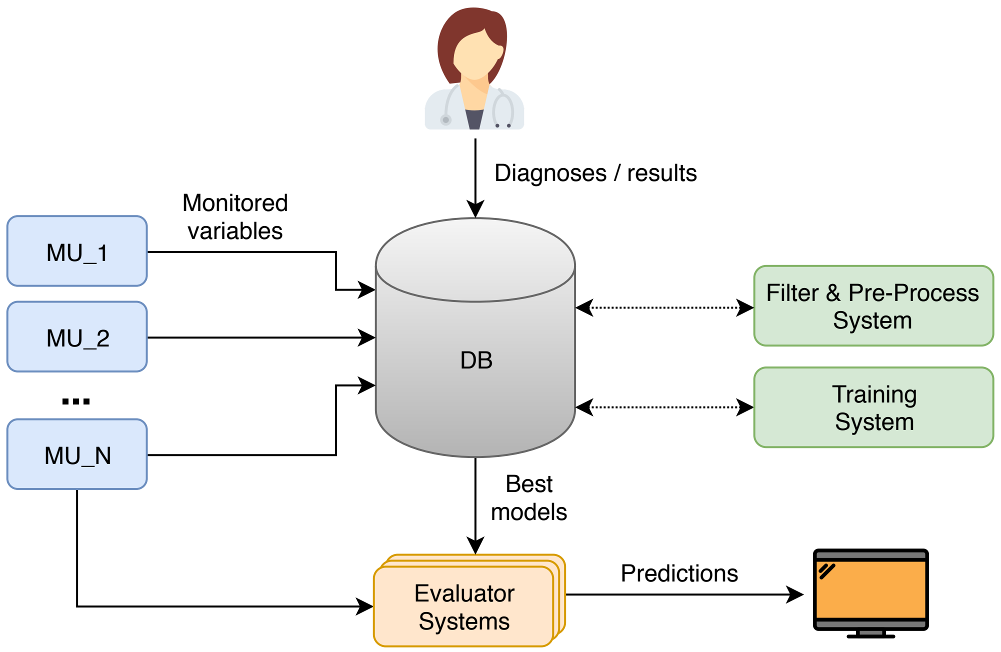


Figure 5.18: Top-View of the stroke prediction system. Both the monitoring units and the medical staff add the information to the system. The DEVS-based subsystems preprocess it and generate the models and the predictions.

A top view of the data flow of this system is shown in Figure 5.18. This system centralizes all the information in a database. It is used to store the information generated by the data sources, Monitoring Units (MU) and the medical staff, as well as to store the intermediate data and the models generated by the subsystems. These MUs are associated with each one of the hospital stretchers and include a Philips IntelliVue Information Center iX (PIIC iX). This device collects information related to ECG signal, perfusion, respiratory rate, and oxygen saturation. These data are sent to the central database (labeled as DB in Figure 5.18). When the diagnosis and the resulting status of the patients are clear, the medical staff introduces additional information into the database. These data are associated with each stroke event and include variables like the recurrence, the stroke type, and the *exitus*. This categorization is used to train predictive models, which could be used as a reference in the early stages of stroke events suffered by other patients.

Following the general view illustrated in Figure 5.13, this particular system needs five of the six subsystems to separate the processing stages. In this use case, the module of data normalization was not considered necessary due to the use of a single data source and the intrinsic characteristics of the input data. First, both the `Filter & Pre-Process systems` (labeled with the same names but as a single block in Figure 5.18) discards the episodes that do not accomplish several fixed criteria to assure the quality of input data. Some corrections and transformations are also applied in this stage to adapt the original data to the next procedures. Second, the `Training System` generates different models that can be used to generate predictions. It considers variations in the input parameters and uses several well-known classification algorithms. These models are uploaded to the database, including information about their fitness and error. Finally, the `Evaluator Systems` load the models with the best accuracy from the common database and generate predictions. An `Evaluator System` can be deployed for instance into an ambulance monitoring equipment or in a patient stretcher. The resulting predictions are sent to the medical staff and may be used to facilitate the diagnosis and treatment of the stroke events.

In order to manage all the information, several intermediate database tables are automatically created (represented in Figure 5.19). First, the `Filter & Pre-Process` system loads the *Raw data* table, which contains the data gathered by the Information Centers. It adequately formats them and recovers damaged signals (if any). Then, it saves the results in several intermediate tables (*Processed data*). Finally, the Training System combines these tables with the diagnoses of the medical staff, also stored in the database (*Diagnoses* table), to generate both stroke type and *exitus* sets of predictive models.

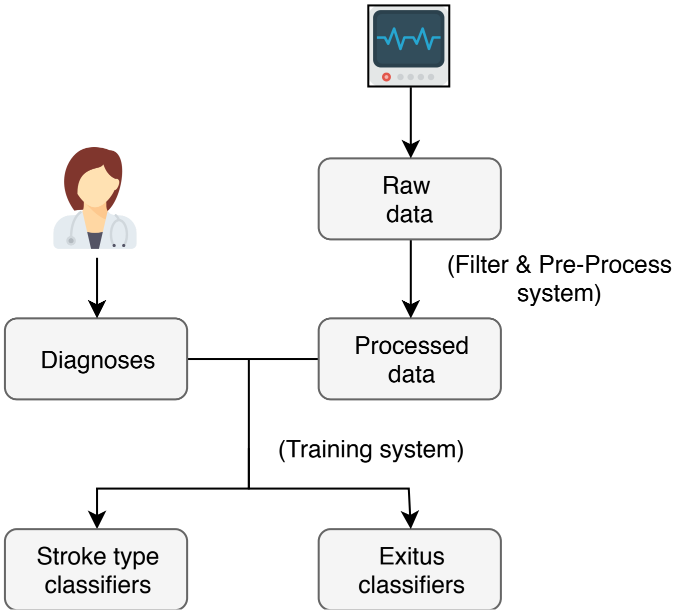


Figure 5.19: Tables implied in the database of the stroke prediction system.

Figure 5.20 shows the structure of the Filter & Pre-Process system more in detail. This module loads the original data generated by the MUs and splits them so that a *MetaFrame* is generated for each patient. Then, invalid patients are discarded using the `Filter` module. A patient is considered to be invalid when he/she does not have at least 45 minutes of monitoring data or when the percentage of null values in some of the variables of the first 45 minutes exceeds 20% of the total. On the used dataset, 14 of the 118 patients are discarded in accordance with the filtering rules. After the filter, the first 45 minutes of each valid patient's data are selected and they are re-directed to the `Filler` module of Figure 5.20. In it, null values that appear in the monitoring variables are filled based on previous and next valid data.

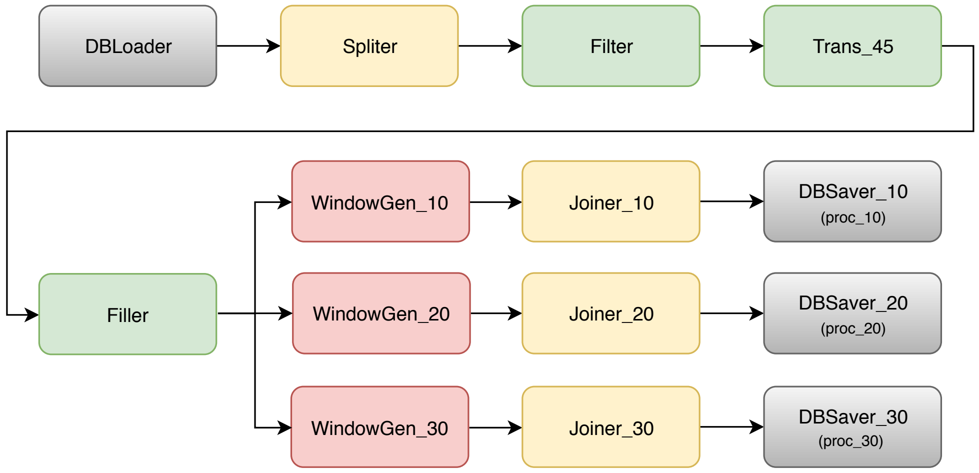


Figure 5.20: Filter & Pre-Process system. It pre-process the monitoring data and repairs possible failures.

Next, as Figure 5.20 shows, each resulting *MetaFrame* goes through three `WindowGen` modules. These modules group the patient's data using windows. An example of this process is illustrated in Figure 5.21, where a generic window structure of size 3 is created. That means that each row of the resulting structure is composed of three samples of the original one and the class related to the last one. In the case of the presented system, the variables on the left are the periodic measurements of a specific patient and the class (in the right) is the stroke type or the *exitus* value. Each one of the `WindowGen` modules is configured to group the packet with a different window length (10, 20, and 30 samples per row). The use of these structures is due to the data nature. Each monitoring sample is too granular to perform predictions separately, but a consecutive set of samples can represent a significant trend that can be used to recognize patterns. After that, the resulting structures are joined in the `Joiner` module to group the data relative to each patient in new *MetaFrames*.

Finally, this information is saved in three tables of the central database (one for each window length). It is worth mentioning that our framework generates several predictive models in parallel (using different algorithms, window sizes, cross-validation type, etc). Thus, all

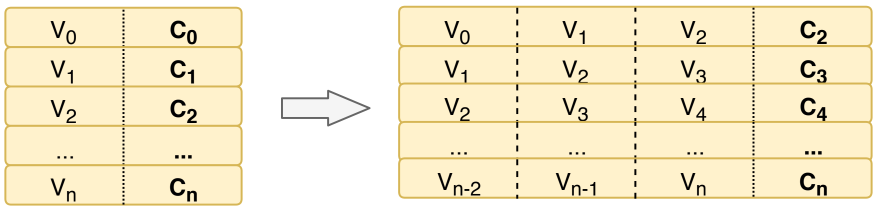


Figure 5.21: Example of grouping information using windows.

data are automatically pre-processed in all the needed formats to avoid the execution of this procedure each time that new models are being created. That subsystem is executed every 12 hours, to check if there are newly available data buffered to process. After several tests, we have concluded that 12 hours is enough to gather new relevant data and update existing models or generate new ones.

Details of the Training System are depicted in Figure 5.22. This module is in charge of generating updated models periodically. It is executed every 24 hours and replaces all the models stored in the database if new data are available (i.e. when the Filter & Pre-Process system adds new data to the intermediate tables or the medical staff tag previous episodes in the *Diagnoses* table). It also sends pulses to all the active Evaluator Systems to update their models if needed.

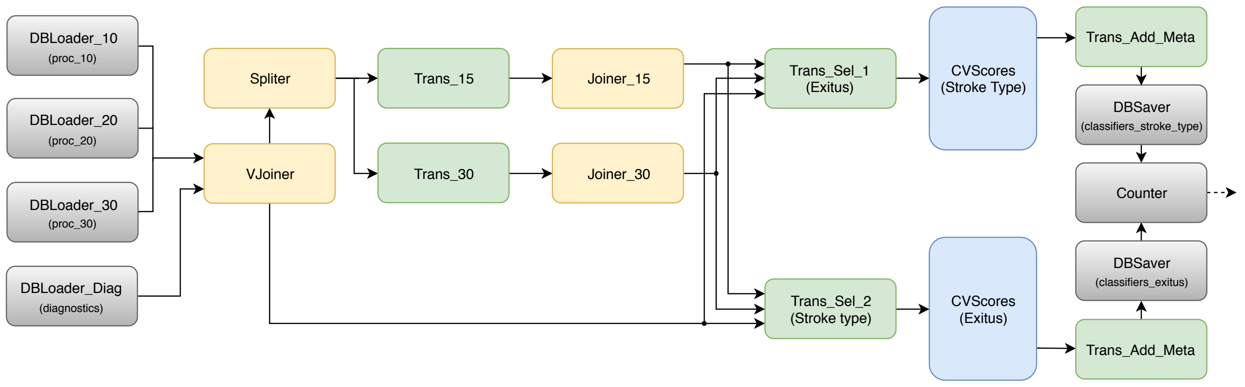


Figure 5.22: Training System. It generates models based on the processed patients info generated by the Filter & Pre-Process system.

This system starts loading the pre-processed data generated for the Filter & Pre-Process system. Each one of the three resulting *MetaFrames* is joined with the information present in the *Diagnoses* table, using a `VJoiner` module. With that, all the tagged episodes are recovered. The patient's data that have not been still tagged by the medical staff are discarded in this module.

Next, each one of the *MetaFrames* is used to generate three new ones: one taking into account the first 15 minutes of monitoring, a second that uses the first 30 minutes, and a third one that uses the first 45 minutes. To this end, a `Splitter` module is used to separate the patients and two `Transformers` are used to cut the information in pieces of 15 and 30 minutes. The group of 45 minutes does not need to be cut because it was already saved with this length in the Filter & Pre-Process system.

After that, the resulting structures are joined in the `Joiner` module to group the data relative to each patient in new *MetaFrames*. Finally, this information is saved in three tables of the central database (one for each window length). As above, it is saved in this pre-processed format to avoid executing this procedure each time that new models are being created. That subsystem is executed every 12 hours, to check if there are new available valid data to process.

At this point, nine packets are generated combining three different window sizes and three sample lengths. Each packet is used to create several custom prediction models, using well-known classification algorithms. For doing that, they are injected into two models in charge of selecting the classes of the data (one of them selects the stroke type, and the other the *exitus* risk). Each output goes to the corresponding `CvScores` module. These coupled components generate sets of models using several classifiers included in the *sklearn* Python library. Specifically, the following Machine Learning algorithms are used: *Ada Boost*, *Bagging*, *Extra Tree*, *Decision Tree*, *Gradient Boosting* and *Random Forest*. As output, a *MetaFrame* is obtained containing the name of each classifier, the generated models, its score, and its error. The score corresponds to the average fit values obtained in a 5-fold cross-validation process. Next, this information is tagged with the window size and the monitoring length used in each combination (using a `Transformer` module and getting this information from the metadata of the packets). Finally, the models are saved in two tables of the central database (*classifiers_stroke_type* and *classifiers_exitus*) to be used by the Evaluator System. The `Counter` is used to notify the Evaluator Systems of the existence of updated models. This is done by generating and sending a pulse after the 9 sets of models have been uploaded.

The Evaluator Systems are the ones in charge of loading the most suitable models from the database and use them to generate predictions about the stroke type and the *exitus* of new stroke attacks, evaluating the models with the real-time monitoring of the patient. The structure of this system can be seen in Figure 5.23. It is centered two `DBClassifiers` modules, which start loading the best models among the ones trained with samples of 15 minutes. This model will be updated when one of these two conditions happen: (i) it arrives a pulse from the Training system, indicating that the models have been updated, or (ii) the `Counter` modules detect that enough data have been collected to use another set of models. This occurs when the monitoring device collects data for 30 and 45 minutes. When the model has been updated, the `DBClassifiers` modules generate a *MetaFrame* with the information of the new model. This packet is modified with a `Transformer` module to generate a special packet that alters the window length used in the corresponding `WindowBuffer` module. These modules are responsible for generating the appropriate packets to be evaluated in the models, grouping the input data using windows of the same length as the ones used in training.

The input of that system is generated by a `StreamLoader` module, which simulates the operation of the monitoring units. In this way, each output packet contains only one value for each measured variable (as they were measured in real-time). In the following `Transformer` the suitable variables are selected (the ones that are used in the training phase). The `Logger` module receives the predictions generated by the two `DBClassifiers` and display them so that it can be seen by the medical staff.

As a result of the execution of the system, several models are generated and stored in

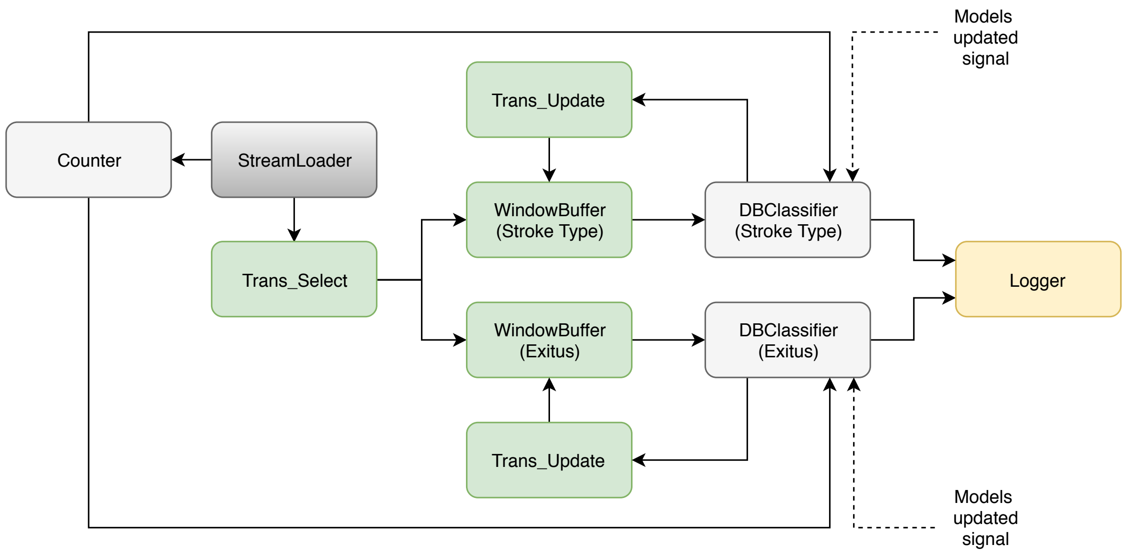


Figure 5.23: Evaluator system. It downloads and evaluates the best models and display the predictions.

the database. Specifically, 54 models are generated to diagnose the stroke type (Figure 5.24) and other 54 models correspond to *exitus* predictions (Figure 5.25). The fit score of each one of them, generated with different combinations of sample size in minutes (m) and window lengths (s), can be seen in both Figures. With regard to stroke type predictions, the best fit (0.824 $\pm$ 0.001060) is achieved by a *Gradient Boosting* model with a sample size of 30 minutes and a window length of 30 samples. The best *exitus* prediction model corresponds to a *Random Forest* with a sample size of 45 minutes, and a window length of 30 samples It achieves a fit of 0.936 $\pm$ 0.000342.

The presented abstract methodology can be used to provide support to estimate future risks in several neurological diseases, or other clinical scenarios where the patient is continuously being monitored. By using this methodology, it is possible to straightforwardly generate predictive models centered on key aspects of the diseases. Moreover, it helps to reduce developing time and costs, and can be easily integrated as a part of decision support systems to provide assistance to the work of the medical staff.

In this chapter, we exemplify the benefits that the use of M&S methodologies can bring to

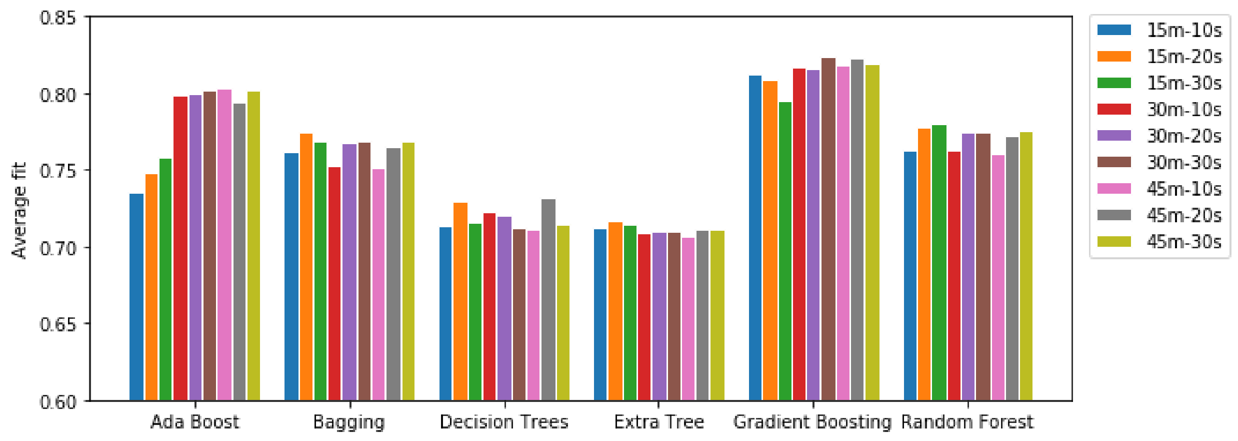


Figure 5.24: Fit of the stroke type prediction models, generated with different combinations of sample size in minutes (m) and number of samples used in the size of the window (s).

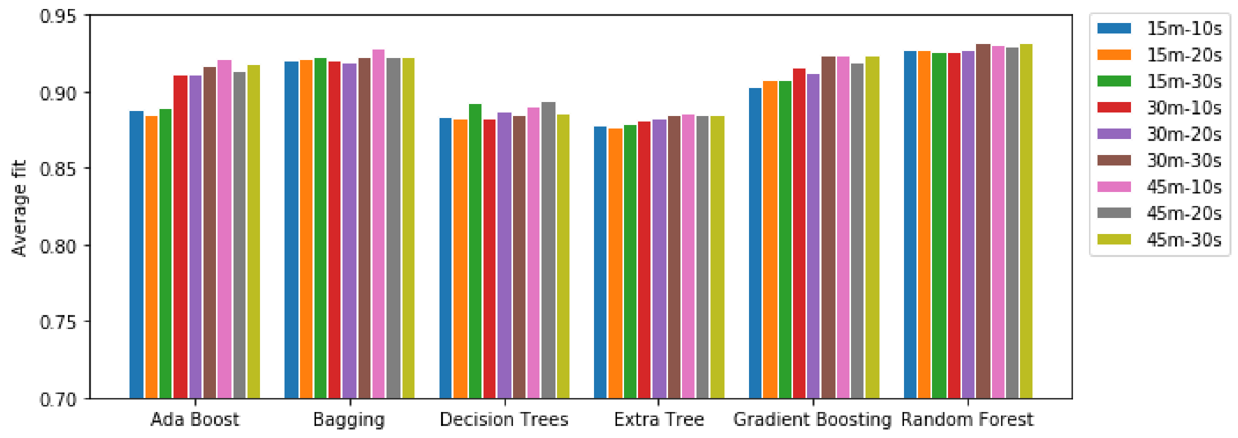


Figure 5.25: Fit of the *exitus* prediction models, generated with different combinations of sample size in minutes (m) and number of samples used in the size of the window (s).

the development of predictive healthcare systems and models with the presentation of diverse modeling use cases and utilities. First, we describe some background aspects regarding numerical methodologies for modeling epidemiology diseases and model-driven development of healthcare monitoring systems. Then, we describe the related contributions, including a Cell-DEVS SIRDS epidemiological model, a model-driven design of a healthcare monitoring system, and a modular methodology to automate predictive modeling workflows. Also, the development of these heterogeneous systems also serves to show the potential of DEVS-based developments, and present the benefits of model-driven designs.

Chapter 6

Conclusions and future work

The main goal of this thesis was to study and optimize the design and implementation of complex systems in an IoT-healthcare context. M&SBSE has been used as a driving force to achieve this goal, using discrete-event modeling and simulation techniques to increase the reliability and robustness of our work. The resulting contributions can be grouped into three main categories: (i) optimization and improvement of DEVS-based M&S frameworks, (ii) MBSE-based development of predictive systems, (iii) scaling and deployment of these systems in an IoT context.

6.1 Conclusions

The main conclusions drawn as a result of this research are listed below, categorized into the main topics of this dissertation.

Modeling and simulation of complex systems

- With the dawn of the data era, we have seen how data generated worldwide is increasing exponentially, and more devices are connected to the network every year. The complexity of systems increases accordingly, incorporating many heterogeneous components and combining knowledge of several disciplines. This trend has also been favored with the rise of different technologies as Machine Learning, the Internet of Things, and Big Data. M&S formalisms have proven to have an essential role in managing this complexity, providing clear, reusable, and unambiguous specifications that help overcome its inner challenges. During the thesis development of the thesis, we contributed to the development of a DEVS framework, xDEVS, with (i) the implementation of a Python simulator, (ii) the development of an algorithm to reduce the overhead introduced by the traditional DEVS implementation approach, and (iii) the incorporation of several V&V tools.

- Verification and Validation techniques play a crucial role in the development of complex systems. Although the software industry places great emphasis on the inclusion of testing procedures in their development flow, most M&S frameworks still present a lack of robust tools that can be used to perform testing over the simulation models. Their V&V is often performed with ad-hoc methodologies and without any standardization. We contributed to overcoming this problem, developing several tools for the verification of discrete-events models. These tools have been designed so that the testing specifications do not affect the model's structure itself, which allows its definition in parallel or even before the model development. These contributions enable verifying DEVS models through the unit and metamorphic testing, and a constraints simulation layer, allowing to check arithmetic properties over the model output ports values.

- There are many DEVS-based simulators implemented in different programming languages and with different possibilities. Sometimes, when selecting the suitable M&S toolkit for developing specific scenarios or introducing implementation changes in these simulators, it is required to analyze and compare their performance objectively and with a common metric. These comparisons have been traditionally performed by designing custom sample models, executing them in several simulators, and comparing their execution times. This approach complicates the comparison of results between studies and supposes a subjective metric dependent on the chosen model's specific characteristics. As part of this thesis, we have presented an extension of the DEVStone benchmark, defining a specific model set covering heterogeneous model configurations to be used as a reference for comparing DEVS-based simulators.

Internet of Things deployments

- The Internet of Things is a technological revolution that will greatly impact the future of computing and communications. From now, we have to optimize the processes and infrastructures to deal with the huge amount of processing and storing needs that this paradigm brings. Controlled IoT simulation environments widely help in this task, allowing us to explore different aspects of application design, architecture deployment, and resource management policies optimization. In this thesis, we contributed to one of these simulation environments, SFIDE, which helps implementing, testing, and assessing workload allocation strategies for data centers. Also, it allows to track and analyze the power consumption and thermal behavior of data center infrastructures when running different workload types. We have redesigned the entire environment, adding compatibility support with the popular SLURM workload manager and extending the simulator's original perspective to implement IoT environments where the devices can communicate over a network layer.

- When designing and implementing an IoT environment, we need to focus on critical aspects such as power consumption, the processing workflows, and the required bandwidth. Based on them, we must optimize the application placement, communication protocols, and job allocation policies to improve the final product's usability and performance. A common trend in the last years consists of using Fog Computing to obtain a trade-off between the processing capabilities traditionally given by the Cloud and the small latency of Edge computing. We have followed this trend in the development of an IoT scenario aiming to study the impact that the location of Micro Data Centers (MDCs) has on the overall power consumption of the system. These MDCs receive processing requests from a network of connected migraine patients intended to predict their next pain phases. We extracted the layout of the buildings in a central area of Madrid, Spain, and the location of the metro stops. We performed a crowd simulation with this information, where migraine patients use the metro stops as entry/exit points and perform certain tasks in the city's main passable areas. Finally, we have modeled the MDCs, characterized the training and inference services, and performed several simulations comparing the optimal locations of the data centers with several hospital configurations to assess the impact of the location on the resulting power consumption of the system.

Healthcare modeling trends

- M&S and IoT are having an increasing acceptance in healthcare. Although a wide range of applications, services, and infrastructures are being developed, healthcare trends with more potential involve patients' continuous monitoring through non-intrusive networked sensors. This approach makes possible the collection of useful data that can be used to obtain relevant conclusions at the early stages of the diseases, favoring the prognosis over the classical post-facto diagnose-and-treat reactive paradigm. In this regard, we developed a M&SBSE-aided HMS implementation from a DEVS-based migraine prediction model. The resulting VHDL system was loaded into an FPGA, and can continuously monitor four hemodynamic variables of migraine patients from medical-precision sensors and generating predictions of the proximity of new pain phases. This use case highlighted the suitability of DEVS for designing and developing complex systems, especially when the final implementation is intended to be based on a modular language as VHDL.

- The huge data generation ratio in this data era opens the door for many possibilities and comes with some drawbacks. One of these is the heterogeneity of the generated data. It is common to see similar information in plenty of different formats and with different scales and units. When integrating several external datasets to create knowledge, we must implement an intermediate step to unify and normalize their data before creating models and platforms. As part of this thesis, we have developed a framework that facilitates the implementation of entire predictive scenarios based on XML specifications, covering the extraction of data from a multitude of sources, their filtering, integration and transformation, as well as the generation and verification of classification and regression predictive models. We have illustrated its use with the development of a decision support system to determine the stroke type and *exitus* probability at early stages of new crises, based on previous stroke patient monitoring data and diagnoses provided by the medical staff.

6.2 Future work

In the following, we discuss some limitations in these research lines that enable additional future work:

- The multi-platform metamorphic testing tool presented in this thesis provides a flexible way to integrate new simulators and define the metamorphic relations. However, the tester still has to generate the inputs of the system manually. To fully automate the metamorphic testing process, several additional testing strategies can be incorporated into this tool to cover the generation of input test cases. Some examples are (i) random testing, which produces random and independent test inputs, (ii) fuzzy testing, intended to generate invalid and unexpected input data to verify the correct response of the system in edge cases, and (iii) specification-based testing, where a description of the system is used to create input test cases based on its expected values.

- The DEVStone benchmark offered a standard specification for defining testing models and studying the DEVS-based simulators' performance. We extended the original concept, providing a standard model set to deal with the current heterogeneity of selected configurations and enabling the direct performance comparison among studies.

However, this metric is centered on measuring the overhead introduced by the simulator itself, avoiding the introduction of delays in the internal and external transitions of the atomic modules. In subsequent versions, it would be convenient to include the possibility of executing several types of computational workloads in the transition functions, allowing us to represent the performance of programming languages when dealing with specific tasks, and giving the modelers a better idea of which simulators are most convenient for designing new systems.

- One major concern in M&S is the interoperability among simulation models. Nowadays, there exist significant number of M&S frameworks, including a wide range of features and tools, implemented in different programming languages and providing diverse APIs. However, they usually do not include compatibility mechanisms to integrate external models. Hence, models tend to be implemented as islands of knowledge, hard to reuse and interconnect. xDEVS has made some efforts to improve this interoperability, developing a SOA interface to communicate different models. We want to extend this concept, developing wrappers for the most popular DEVS-based simulation frameworks and improving its web-service-based interconnection, further facilitating the creation of hybrid models.

- The modular healthcare framework presents a way to create information systems and predictive scenarios based on XML specifications. Although the first can be done automatically, the modeler still has to manually specify the modules and connections for creating processing, training, and inference systems. We want to automatize this process further, generating full architectures based on the disease and patient information.

- The SFIDE environment allows the study of workload location strategies in data centers. For deploying a scenario, both the data center architecture and the workload characterization must be specified in configuration files. Also, the modeler has to instance and link all the network and computing models when specifying an IoT environment. Hence, it would greatly benefit from integrating network simulators such as OMNeT++, allowing it to leverage the modeler task and graphically design both the data center architecture and the scenario components structure.

Appendix A

xDEVS models definition and simulation

This section provides some implementation examples in the different xDEVS branches. It presents some basic components implementations, both atomic and coupled, and details the particularities of the different APIs.

A.1 Atomic components

The behavior of the DEVS models is encapsulated in the atomic components. Each component has a *phase*, which is the label of the current state of the component, and a *sigma*, that is the duration of the current state. The atomic components base their operation on events. The response to each of these events is defined by implementing specific abstract methods of the `Atomic` class. The main events controlled by these components are the following ones:

- External event (*deltext*): it is activated when one or more messages arrive at any of the input ports of an atomic component.
- Internal event (*deltint*): it is triggered after the lifetime of the present state has been consumed (specified by *sigma*).
- Confluent event (*deltcon*): it is activated when both the internal and external events are scheduled for a specific simulation time. The corresponding method is already implemented with the most common expected behavior (execute the internal event

method first, and then the external event method). Hence, it only has to be implemented when an alternative behavior is expected.

- Output function (*lambda*): it is activated before every internal event. All the output values have to be sent through the output ports in this function.

Also, following the xDEVS Atomic interface, modelers must implement two additional methods to control the initialization and destruction of the objects: *initialize* and *exit*, respectively.

In the following, we present the implementations of the `Processor` component of the `EFP` example (shown in the previous section) for the three available xDEVS APIs. This component starts with a *passive* state (set in the *initialize* method) and waits until a `Job` arrives to its input port. When that happens, the external event method is activated. In this method, if the `Processor` is idle at that simulation time, the component changes its state to *active* and sets its sigma to the processing time. All the jobs received while the `Processor` is in this state are discarded. When the time specified in sigma is consumed, the output function (*lambda*) is activated and the `Job` is sent through the output port. Right after that, the internal event method is invoked, which changes the status to *passive* again, indicating that it is available for processing new jobs.

We can see the `Processor` in the Java branch of xDEVS in Listing A.1. It can be seen how the ports are added to the atomic component in the constructor, and a *processingTime* parameter is received. This parameter is saved and used later in the external event method (*deltext*) to specify the duration of the *active* state. When this time is consumed, the *lambda* and *deltint* methods are called. First, *lambda* outputs the suitable values (in this example, the original input job), and then *deltint* calls the *passivate* method. This is a shortcut for specifying the *passive* phase with an infinity duration. In this way, the component only can be activated again due to an external event. It is worth to note that both the *active* and *passive* phase do not have any special behavior, and are used in some auxiliary methods only for usability reasons.

In the Python implementation of Listing A.2, we can see how both the structure and the nomenclature are equivalent to the Java one. However, the format of the method names is changed to snake case to comply with the well-accepted nomenclature conventions of the Python language. Also, the *lambda* output method is renamed to *lambdaf* to avoid overwriting the Python *lambda* keyword. In C++ (Listing A.3), although it keeps the camel case nomenclature, for the API methods, it introduces the *Event* additional object for the message passing. This wrapper object creates a shared pointer to the memory address of the actual message to release it when it will no longer be used, simplifying the memory management of the values.

```java
public class Processor extends Atomic {

    protected Port<Job> iIn = new Port<>("iIn");
    protected Port<Job> oOut = new Port<>("oOut");
    protected Job currentJob = null;
    protected double processingTime;

    public Processor(String name, double processingTime) {
        super(name);
        super.addInPort(iIn);
        super.addOutPort(oOut);
        this.processingTime = processingTime;
    }

    @Override
    public void initialize() { super.passivate(); }

    @Override
    public void exit() {}

    @Override
    public void deltint() { super.passivate(); }

    @Override
    public void lambda() { oOut.addValue(currentJob); }

    @Override
    public void deltext(double e) {
        if (super.phaseIs("passive")) {
            currentJob = iIn.getSingleValue();
            super.holdIn("active", processingTime);
        }
    }
}
```

Listing A.1: Atomic module definition in xDEVS (Java)

```python
class Processor(Atomic):
   def __init__(self, name, proc_time):
      super().__init__(name)

      self.i_in = Port(Job, "i_in")
      self.o_out = Port(Job, "o_out")

      self.add_in_port(self.i_in)
      self.add_out_port(self.o_out)

      self.current_job = None
      self.proc_time = proc_time

   def initialize(self):
      self.passivate()

   def exit(self):
      pass

   def deltint(self):
      self.passivate()

   def deltext(self, e):
      if self.phase == PHASE_PASSIVE:
         self.current_job = self.i_in.get()
         self.hold_in(PHASE_ACTIVE, self.proc_time)

   def lambdaf(self):
      self.o_out.add(self.current_job)
```

Listing A.2: Atomic module definition in xDEVS (Python)

```cpp
class Processor : public Atomic {
protected:
   Event nextEvent;
   double processingTime;
public:
   Port iIn;
   Port oOut;
   Processor(const std::string& name, double processingTime):
      Atomic(name), nextEvent(), processingTime(processingTime), iIn("in"), oOut("out") {
      this->addInPort(&iIn);
      this->addOutPort(&oOut);
   }

   ~Processor() {}
   virtual void initialize() { Atomic::passivate(); }
   virtual void exit() {}
   virtual void deltint() { Atomic::passivate(); }
   virtual void lambda() { oOut.addValue(nextEvent); }

   virtual void deltext(double e) {
      if (Atomic::phaseIs("passive")) {
         nextEvent = iIn.getSingleValue();
         Atomic::holdIn("active", processingTime);
```

```
        }
    }
};
```

Listing A.3: Atomic module definition in xDEVS (C++, header)

A.2 Coupled components

Coupled components encapsulate another atomic and coupled components and define the couplings among them. This grouping facilitates the reusability of the models and allows us to define the hierarchy of the system. As an example of coupled components implementation in xDEVS, we show the definition of the `EF` component shown in Figure 3.3a. In Listing A.4 we can see Java implementation of this component. It does not have to implement special methods, so only the constructor is defined. The internal components (`Generator` and `Transducer`) are instantiated and added as part of the coupled component. After that, the suitable links are established using the *addCoupling* method. This actions are repeated in the Python (Listing A.5) and C++ (Listing A.6) version with no remarkable changes.

```java
public class Efp extends Coupled {

    public Efp(String name, double generatorPeriod, double processorPeriod, double
         transducerPeriod) {
        super(name);

        Ef ef = new Ef("ef", generatorPeriod, transducerPeriod);
        super.addComponent(ef);
        Processor processor = new Processor("processor", processorPeriod);
        super.addComponent(processor);

        super.addCoupling(ef.oOut, processor.iIn);
        super.addCoupling(processor.oOut, ef.iIn);
    }
}
```

Listing A.4: Coupled module definition in xDEVS (Java)

```python
class Efp(Coupled):
    def __init__(self, name, generator_period, processor_period, transducer_period):
        super().__init__(name)

        ef = EF("ef", generator_period, transducer_period)
        proc = Processor("processor", processor_period)
        self.add_component(ef)
        self.add_component(proc)

        self.add_coupling(ef.o_out, proc.i_in)
        self.add_coupling(proc.o_out, ef.i_in)
```

Listing A.5: Coupled module definition in xDEVS (Python)

```
class Efp : public Coupled {
protected:
   Ef ef;
   Processor processor;
public:
   Efp(const std::string& name, const double& generatorPeriod, const double& processorPeriod,
       const double& transducerPeriod): Coupled(name),
        ef("ef", generatorPeriod, transducerPeriod),
        processor("processor", processorPeriod) {
      Coupled::addComponent(&ef);
      Coupled::addComponent(&processor);
      Coupled::addCoupling(&ef, &ef.oOut, &processor, &processor.iIn);
      Coupled::addCoupling(&processor, &processor.oOut, &ef, &ef.iIn);
   }

   ~Efp() {}
}
```

Listing A.6: Coupled module definition in xDEVS (C++, header)

A.3 Simulation layer

This layer defines all the simulation entities that are necessary to carry out the simulation with the previously defined models. Although the modeling and simulation layers communicate with each other to perform the simulation, it is worth noting that they are completely decoupled. In this way, the simulation entities only keep references of the models to activate the proper events and propagate the outputs of the components through the couplings of the model.

```
Efp efp = new Efp("efp", 1, 3, 1000);
Coordinator coordinator = new Coordinator(efp);
coordinator.initialize();
coordinator.simulate(Long.MAX_VALUE);
coordinator.exit()
```

Listing A.7: Launching a simulation in xDEVS (Java)

```
efp = Efp("efp", 1, 3, 1000)
```

```
coord = Coordinator(efp)
coord.initialize()
coord.simulate_time(INFINITY)
coord.exit()
```

Listing A.8: Launching a simulation in xDEVS (Python)

```
Efp efp("efp", 1, 3, 1000);
Coordinator coordinator(&efp);
coordinator.initialize();
coordinator.simulate((long int)10000);
coordinator.exit();
```

Listing A.9: Launching a simulation in xDEVS (C++)

xDEVS is available at a public repository[128] under the GNU LGPL license. Moreover, it contains an example models set that can be used to facilitate the learning of models definition.

Bibliography

[1] Open Street Maps. https://www.openstreetmap.org/. [Online; accessed 25-November-2020].

[2] Mohammad Aazam and Eui-Nam Huh. Fog computing micro datacenter based dynamic resource estimation and pricing model for iot. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 687–694. IEEE, 2015.

[3] Daron Acemoglu, Victor Chernozhukov, Iván Werning, and Michael D Whinston. A multi-risk sir model with optimally targeted lockdown. Technical report, National Bureau of Economic Research, 2020.

[4] Abdulla M Al-Qawasmeh, Sudeep Pasricha, Anthony A Maciejewski, and Howard Jay Siegel. Power and thermal-aware workload allocation in heterogeneous data centers. *IEEE Transactions on Computers*, 64(2):477–491, 2013.

[5] Farman Ali, Shaker El-Sappagh, SM Riazul Islam, Daehan Kwak, Amjad Ali, Muhammad Imran, and Kyung-Sup Kwak. A smart healthcare monitoring system for heart disease prediction based on ensemble deep learning and feature fusion. *Information Fusion*, 63:208–222, 2020.

[6] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[7] Elvio Gilberto Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. 30 years of greatspn. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 227–254. Springer, 2016.

[8] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[9] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 586–597. Elsevier, 2002.

[10] Alex Arenas, Wesley Cota, Jesús Gómez-Gardenes, Sergio Gómez, Clara Granell, Joan T Matamalas, David Soriano-Panos, and Benjamin Steinegger. A mathematical model for the spatiotemporal epidemic spreading of covid19. *MedRxiv*, 2020.

[11] MK Arti and Kushagra Bhatnagar. Modeling and predictions for covid 19 spread in india. *no. April*, 2020.

[12] W Ross Ashby. *An introduction to cybernetics.* Chapman & Hall Ltd, 1961.

[13] Kevin Ashton et al. That 'internet of things' thing. *RFID journal*, 22(7):97–114, 2009.

[14] American Heart Association. Types of stroke. https://www.strokeassociation.org/STROKEORG (accessed 27-October-2020), 2018.

[15] Jang Won Bae and Tag Gon Kim. Devs based plug-in framework for interoperability of simulators. In *Proceedings of the 2010 Spring Simulation Multiconference*, pages 1–7, 2010.

[16] Santiago Balestrini-Robinson, Dane F Freeman, and Daniel C Browne. An object-oriented and executable sysml framework for rapid model development. *Procedia Computer Science*, 44:423–432, 2015.

[17] Carolina Tripp Barba, Miguel Angel Mateos, Pablo Reganas Soto, Ahmad Mohamad Mezher, and Mónica Aguilar Igartua. Smart city for vanets using warning messages,

traffic statistics and intelligent traffic lights. In *2012 IEEE intelligent vehicles symposium*, pages 902–907. IEEE, 2012.

[18] Luciano Baresi and Michal Young. Test oracles. Technical report, Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and ..., 2001.

[19] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[20] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. Sumo–simulation of urban mobility: an overview. In *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind, 2011.

[21] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Håkansson, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. 2006.

[22] Laouen Belloli, Damian Vicino, Cristina Ruiz-Martin, and Gabriel Wainer. Building devs models with the cadmium tool. In *2019 Winter Simulation Conference (WSC)*, pages 45–59. IEEE, 2019.

[23] Federico Bergero and Ernesto Kofman. Powerdevs: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87(1-2):113–132, 2011.

[24] Md Haider Ali Biswas, Luís Tiago Paiva, and MDR De Pinho. A seir model for control of infectious diseases with constraints. *Mathematical Biosciences & Engineering*, 11(4):761, 2014.

[25] Jean-Sébastien Bolduc and Hans Vangheluwe. A modeling and simulation package for classic hierarchical devs. *MSDL, School of Computer McGill University, Tech. Rep*, 2002.

[26] Sally C Brailsford, Timothy Bolt, Con Connell, Jonathan H Klein, and Brijesh Patel. Stakeholder engagement in health care simulation. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1840–1849. IEEE, 2009.

[27] Susan S Brilliant, John C Knight, and PE Ammann. On the performance of software testing using multiple versions. In *Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pages 408–409. IEEE Computer Society, 1990.

[28] Antonio Brogi and Stefano Forti. Qos-aware deployment of iot applications through the fog. *IEEE Internet of Things Journal*, 4(5):1185–1192, 2017.

[29] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[30] Jonathon R Campbell, James C Johnston, Victoria J Cook, Mohsen Sadatsafavi, R Kevin Elwood, and Fawziah Marra. Cost-effectiveness of latent tuberculosis infection screening before immigration to low-incidence countries. *Emerging infectious diseases*, 25(4):661, 2019.

[31] Leslie Anne Campbell, John T Blake, George Kephart, Eva Grunfeld, and Donald MacIntosh. Understanding the effects of competition for constrained colonoscopy services with the introduction of population-level colorectal cancer screening: A discrete event simulation model. *Medical Decision Making*, 37(2):253–263, 2017.

[32] Román Cárdenas, Patricia Arroba, Roberto Blanco, Pedro Malagón, José L Risco-Martín, and José M Moya. Mercury: A modeling, simulation, and optimization framework for data stream-oriented iot applications. *Simulation Modelling Practice and Theory*, 101:102037, 2020.

[33] Román Cárdenas, Kevin Henares, Cristina Ruiz-Martín, and Gabriel Wainer. Cell-devs models for the spread of covid-19. In *Cellular Automata: 14th International Conference on Cellular Automata for Research and Industry, ACRI 2020, Lodz, Poland, December 2–4, 2020, Proceedings 14*, pages 239–249. Springer International Publishing, 2021.

[34] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[35] Carlos Castillo-Chavez and Abdul-Aziz Yakubu. Discrete-time sis models with complex dynamics. *Nonlinear Analysis, Theory, Methods and Applications*, 47(7):4753–4762, 2001.

[36] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics*, 10(1):1–12, 2009.

[37] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55, 2016.

[38] Chia-Ching Chou, Wai-Chi Fang, and Hsiang-Cheh Huang. A novel wireless biomedical monitoring system with dedicated fpga-based ecg processor. In *2012 IEEE 16th International Symposium on Consumer Electronics*, pages 1–4. IEEE, 2012.

[39] Alex Chung Hen Chow and Bernard P Zeigler. Parallel devs: A parallel, hierarchical, modular modeling formalism. In *Proceedings of Winter Simulation Conference*, pages 716–722. IEEE, 1994.

[40] Federico Ciccozzi, Ivica Crnkovic, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Romina Spalazzese. Model-driven engineering for mission-critical iot systems. *IEEE software*, 34(1):46–53, 2017.

[41] Thomas M Cioppa, Thomas W Lucas, and Susan M Sanchez. Military applications of agent-based simulations. In *Proceedings of the 2004 Winter Simulation Conference, 2004.*, volume 1. IEEE, 2004.

[42] Cisco. Global 2021 Forecast Highlights. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_2021_Forecast_Highlights.pdf. [Online; accessed 01-January-2021].

[43] Mercè Comas, Arantzazu Arrospide, Javier Mar, Maria Sala, Ester Vilaprinyó, Cristina Hernández, Francesc Cots, Juan Martínez, and Xavier Castells. Budget impact analysis of switching to digital mammography in a population-based breast cancer screening program: a discrete event simulation model. *PLoS One*, 9(5):e97459, 2014.

[44] Sarah F Cook and Robert R Bies. Disease progression modeling: key concepts and recent developments. *Current pharmacology reports*, 2(5):221–230, 2016.

[45] International Data Corporation. The Growth in Connected IoT Devices . https://www.idc.com/getdoc.jsp?containerId=prUS45213219. [Online; accessed 28-October-2020].

[46] Román Cárdenas, Kevin Henares, Patricia Arroba, Gabriel Wainer, and José L. Risco-Martín. A devs simulation algorithm based on shared memory for enhancing performance. In *Proceedings of the 2020 Winter Simulation Conference (WSC'20)*, 2020.

[47] Paulo Salem da Silva and Ana CV de Melo. On-the-fly verification of discrete event simulations by means of simulation purposes. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 238–247, 2011.

[48] R Davies, P Roderick, C Canning, and S Brailsford. The evaluation of screening

policies for diabetic retinopathy using simulation. *Diabetic Medicine*, 19(9):762–770, 2002.

[49] Paul K Davis and Robert H Anderson. Improving the composability of department of defense models and simulations. Technical report, RAND CORP SANTA MONICA CA, 2003.

[50] Paul K Davis and James H Bigelow. Experiments in multiresolution modeling (mrm). Technical report, RAND CORP SANTA MONICA CA, 1998.

[51] Hidde De Jong. Modeling and simulation of genetic regulatory systems: a literature review. *Journal of computational biology*, 9(1):67–103, 2002.

[52] Anind K Dey, Gregory D Abowd, and Daniel Salber. A context-based infrastructure for smart environments. In *Managing Interactions in Smart Environments*, pages 114–128. Springer, 2000.

[53] Sheng Di and Franck Cappello. Gloudsim: Google trace based cloud simulator with virtual machines. *Software: Practice and Experience*, 45(11):1571–1590, 2015.

[54] Rachelle S Doody, Valory Pavlik, Paul Massman, Susan Rountree, Eveleen Darby, and Wenyaw Chan. Predicting progression of alzheimer's disease. *Alzheimer's research & therapy*, 2(1):1–9, 2010.

[55] Eclipse. Papyrus. https://www.eclipse.org/papyrus/. [Online; accessed 28-September-2020].

[56] Fatih Safa Erenay, Oguzhan Alagoz, Ritesh Banerjee, and Robert R Cima. Estimating the unknown parameters of the natural history of metachronous colorectal cancer using discrete-event simulation. *Medical Decision Making*, 31(4):611–624, 2011.

[57] Stroke Alliance For Europe. The burden of stroke in europe report: the challenge

for policy markers. http://strokeeurope.eu/media/download/ (accessed 15-November-2020), 2018.

[58] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[59] Jesús Fernández-Villaverde and Charles I Jones. Estimating and simulating a sird model of covid-19 for many countries, states, and cities. Technical report, National Bureau of Economic Research, 2020.

[60] David Fone, Sandra Hollinghurst, Mark Temple, Alison Round, Nathan Lester, Alison Weightman, Katherine Roberts, Edward Coyle, Gwyn Bevan, and Stephen Palmer. Systematic review of the use and value of computer simulation modelling in population health and health care delivery. *Journal of Public Health*, 25(4):325–335, 2003.

[61] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[62] Romain Franceschini and Paul-Antoine Bisgambiglia. Decentralized approach for efficient simulation of devs models. In *IFIP International Conference on Advances in Production Management Systems*, pages 336–343. Springer, 2014.

[63] Romain Franceschini, Paul-Antoine Bisgambiglia, Luc Touraille, Paul Bisgambiglia, and David Hill. A survey of modelling and simulation software frameworks using discrete event system specification. In *2014 Imperial College Computing Student Workshop*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[64] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[65] Luis García-Temza, José L Risco-Martín, José L Ayala, Gemma Reig Roselló, and Juan M Camarasaltas. Comparison of different machine learning approaches to model stroke subtype classification and risk prediction. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–10. IEEE, 2019.

[66] Gordana Gardašević, Mladen Veletić, Nebojša Maletić, Dragan Vasiljević, Igor Radusinović, Slavica Tomović, and Milutin Radonjić. The iot architectural framework, design issues and application domains. *Wireless personal communications*, 92(1):127–148, 2017.

[67] Denis Getsios, Jenő P Marton, Nikhil Revankar, Alexandra J Ward, Richard J Willke, Dale Rublee, K Jack Ishak, and James G Xenakis. Smoking cessation treatment and outcomes patterns simulation: a new framework for evaluating the potential health and economic impact of smoking cessation interventions. *Pharmacoeconomics*, 31(9):767–780, 2013.

[68] Daniele Gianni, Andrea D'Ambrogio, and Andreas Tolk. *Modeling and simulation-based systems engineering handbook*. CRC Press, 2014.

[69] NJ Giffin, L Ruggiero, Richard B Lipton, SD Silberstein, JF Tvedskov, J Olesen, J Altman, Peter J Goadsby, and A Macrae. Premonitory symptoms in migraine: an electronic diary study. *Neurology*, 60(6):935–940, 2003.

[70] Ezequiel Glinsky and Gabriel Wainer. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments. In *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 265–272. IEEE, 2005.

[71] Peter W Glynn. A gsmp formalism for discrete event systems. *Proceedings of the IEEE*, 77(1):14–23, 1989.

[72] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. Introduction to iot. *International Advanced Research Journal in Science, Engineering and Technology (IARJ SET)*, 5(1), 2018.

[73] Maureen S Golan, Laura H Jernegan, and Igor Linkov. Trends and applications of resilience analytics in supply chain modeling: systematic literature review in the context of the covid-19 pandemic. *Environment Systems and Decisions*, 40:222–243, 2020.

[74] Kim Tag Gon, Bernard P Zeigler, and Herbert Praehofer. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. 2000.

[75] Siva Leela Krishna Chand Gudi, Suman Ojha, Benjamin Johnston, Jesse Clark, and Mary-Anne Williams. Fog robotics for efficient, fluent and robust human-robot interaction. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–5. IEEE, 2018.

[76] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[77] Marcelo Gutierrez-Alcaraz and Gabriel Wainer. Experiences with the devstone benchmark. In *Proceedings of the 2008 Spring simulation multiconference*, pages 447–455. Society for Computer Simulation International, 2008.

[78] Badis Hammi, Rida Khatoun, Sherali Zeadally, Achraf Fayad, and Lyes Khoukhi. Iot technologies for smart cities. *IET Networks*, 7(1):1–13, 2017.

[79] VSKV Harish and Arun Kumar. A review on modeling and simulation of building energy systems. *Renewable and sustainable energy reviews*, 56:1272–1292, 2016.

[80] Moeen Hassanalieragh, Alex Page, Tolga Soyata, Gaurav Sharma, Mehmet Aktas, Gonzalo Mateos, Burak Kantarci, and Silvana Andreescu. Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges. In *2015 IEEE International Conference on Services Computing*, pages 285–292. IEEE, 2015.

[81] Eileen Head. ASSIST: A Simple SImulator for State Transitions. http://www.cs.binghamton.edu/~software/. [Online; accessed 28-September-2020].

[82] Joseph A Heim, Hao Huang, Zelda B Zabinsky, Jane Dickerson, Monica Wellner, Michael Astion, Doris Cruz, Jeanne Vincent, and Rhona Jack. Design and implementation of a combined influenza immunization and tuberculosis screening campaign with simulation modelling. *Journal of evaluation in clinical practice*, 21(4):727–734, 2015.

[83] Kevin Henares. Modular data framework for generating predictive models. https://github.com/khvilaboa/HealthDataFramework/. [Online; accessed 22-October-2020].

[84] Kevin Henares. mmpy: Methamorphic verification tool. https://github.com/khvilaboa/mmpy (accessed 26-October-2020), 2020.

[85] Kevin Henares, Josué Pagán, José L Ayala, Marina Zapater, and José L Risco-Martín. Cyber-physical systems design methodology for the prediction of symptomatic events in chronic diseases. *Complexity Challenges in Cyber Physical Systems: Using Modeling and Simulation (M&S) to Support Intelligence, Adaptation and Autonomy*, pages 223–253, 2019.

[86] Kevin Henares, José L Risco-Martín, José L Ayala, and Román Hermida. Unit testing platform to verify devs models. In *Proceedings of the 2020 Summer Simulation Conference*, pages 1–11, 2020.

[87] Kevin Henares, José L Risco-Martín, and Marina Zapater. Definition of a transparent constraint-based modeling and simulation layer for the management of complex systems. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–12. IEEE, 2019.

[88] Andres Hernandez-Matamoros, Hamido Fujita, Toshitaka Hayashi, and Hector Perez-Meana. Forecasting of covid19 per regions using arima models and polynomial functions. *Applied Soft Computing*, 96:106610, 2020.

[89] Michael D Hill. Diagnostic biomarkers for stroke: a stroke neurologist's perspective, 2005.

[90] Jan Himmelspach and Adelinde M Uhrmacher. Plug'n simulate. In *40th Annual Simulation Symposium (ANSS'07)*, pages 137–143. IEEE, 2007.

[91] Nicholas HG Holford, Phylinda LS Chan, John G Nutt, Karl Kieburtz, Ira Shoulson, Parkinson Study Group, et al. Disease progression and pharmacodynamics in parkinson disease–evidence for functional protection with levodopa and other treatments. *Journal of pharmacokinetics and pharmacodynamics*, 33(3):281–311, 2006.

[92] Diego A Hollmann, Maximiliano Cristiá, and Claudia Frydman. Adapting model-based testing techniques to devs models validation. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, page 6. Society for Computer Simulation International, 2012.

[93] Xiaolin Hu and Bernard P Zeigler. A high performance simulation engine for large-scale cellular devs models. In *High Performance Computing Symposium (HPC'04), Advanced Simulation Technologies Conference*, pages 3–8. Citeseer, 2004.

[94] Xiaolin Hu, Bernard P Zeigler, and Saurabh Mittal. Dynamic reconfiguration in devs component-based modeling and simulation. *SIMULATION: Transactions of the Society of Modeling and Simulation International*, 2003.

[95] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management.* John Wiley & Sons, 2007.

[96] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 633–642. ACM, 2011.

[97] Moon Ho Hwang and Bernard P Zeigler. A modular verification framework based on finite & deterministic devs. *SIMULATION SERIES*, 38(1):57, 2006.

[98] IDC. Iot growth demands rethink of long-term storage strategies. *https://www.idc.com/getdoc.jsp?containerId=prAP46737220*, 2020.

[99] Ataru Igarashi, Rei Goto, Kiyomi Suwa, Reiko Yoshikawa, Alexandra J Ward, and Jörgen Moller. Cost-effectiveness analysis of smoking cessation interventions in japan using a discrete-event simulation. *Applied health economics and health policy*, 14(1):77–87, 2016.

[100] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[101] Yaser Jararweh, Mohammad Alsmirat, Mahmoud Al-Ayyoub, Elhadj Benkhelifa, Ala' Darabseh, Brij Gupta, and Ahmad Doulat. Software-defined system support for enabling ubiquitous mobile edge computing. *The Computer Journal*, 60(10):1443–1457, 2017.

[102] Jiwei Jia, Jian Ding, Siyu Liu, Guidong Liao, Jingzhi Li, Ben Duan, Guoqing Wang, and Ran Zhang. Modeling the control of covid-19: impact of policy interventions and meteorological factors. *arXiv preprint arXiv:2003.02985*, 2020.

[103] Irena Jovanović. Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research*, 30, 2006.

[104] Jongtack Jung and Hwangnam Kim. Mr-cloudsim: Designing and implementing mapreduce computing model on cloudsim. In *2012 International Conference on ICT Convergence (ICTC)*, pages 504–509. IEEE, 2012.

[105] George-Dimitrios Kapos, Vassilis Dalakas, Anargyros Tsadimas, Mara Nikolaidou, and Dimosthenis Anagnostopoulos. Model-based system engineering using sysml: Deriving executable simulation models with qvt. In *2014 IEEE International Systems Conference Proceedings*, pages 531–538. IEEE, 2014.

[106] Omer Karin, Yinon M Bar-On, Tomer Milo, Itay Katzir, Avi Mayo, Yael Korem, Boaz Dudovich, Eran Yashiv, Amos J Zehavi, Nadav Davidovitch, et al. Cyclic strategies to suppress covid-19 and allow economic activity.

[107] L Kelman. The postdrome of the acute migraine attack. *Cephalalgia*, 26(2):214–220, 2006.

[108] William Ogilvy Kermack and Anderson G McKendrick. A contribution to the mathematical theory of epidemics. *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*, 115(772):700–721, 1927.

[109] Abhishek Khanna and Sanmeet Kaur. Evolution of internet of things (iot) and its significant impact in the field of precision agriculture. *Computers and electronics in agriculture*, 157:218–231, 2019.

[110] Sungung Kim, Hessam S Sarjoughian, and Vignesh Elamvazhuthi. Devs-suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–7. Citeseer, 2009.

[111] Ernesto Kofman and Sergio Junco. Quantized-state systems: a devs approach for continuous system simulation. *Transactions of The Society for Modeling and Simulation International*, 18(3):123–132, 2001.

[112] Andreas Kohne, Marc Spohr, Lars Nagel, and Olaf Spinczyk. Federatedcloudsim: a sla-aware federated cloud simulation framework. In *Proceedings of the 2nd International Workshop on CrossCloud Systems*, pages 1–5, 2014.

[113] Thomas H Kolbe, Gerhard Gröger, and Lutz Plümer. Citygml: Interoperable access to 3d city models. In *Geo-information for disaster management*, pages 883–899. Springer, 2005.

[114] ARS Laboratory. ARS simulation visualizations playlist. https://bit.ly/3aiDM4j. [Online; accessed 22-January-2021].

[115] ARS Laboratory. Cell-devs web viewer tool. https://simulationeverywhere.github.io/CD-WebViewer-2.0, 2020.

[116] Isaac Lera, Carlos Guerrero, and Carlos Juiz. Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, 7:91745–91758, 2019.

[117] Li Li, Wenli Zheng, Xiaodong Wang, and Xiaorui Wang. Coordinating liquid and free air cooling with workload allocation for data center power minimization. In *11th International Conference on Autonomic Computing ({ICAC} 14)*, pages 249–259, 2014.

[118] Michael Y Li and James S Muldowney. Global stability for the seir model in epidemiology. *Mathematical biosciences*, 125(2):155–164, 1995.

[119] Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for devs formalism implementations. In *SpringSim (TMS-DEVS)*, pages 183–188. Citeseer, 2011.

[120] Mattias Linde, Anders Gustavsson, Lars Jacob Stovner, Timothy J Steiner, Jessica Barré, Zaza Katsarava, JM Lainez, Christian Lampl, Michel Lantéri-Minet, D Ras-

tenyte, et al. The cost of headache disorders in europe: the eurolight project. *European journal of neurology*, 19(5):703–711, 2012.

[121] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 35–41. IEEE, 2017.

[122] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: a mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.

[123] Yi Liu, Chao Yang, Li Jiang, Shengli Xie, and Yan Zhang. Intelligent edge computing for iot-based energy management in smart cities. *IEEE Network*, 33(2):111–117, 2019.

[124] Mordor Intelligence LLP. Internet of Things (IoT) Market - Growth, Trends, Forecasts (2020 - 2025) . https://www.reportlinker.com/p05893117/Internet-of-Things-IoT-Market-Growth-Trends-Forecasts.html?utm_source=GNW. [Online; accessed 12-January-2021].

[125] Knud Lasse Lueth. State of the iot 2018: Number of iot devices now at 7b–market accelerating. *IoT Analytics*, 8, 2018.

[126] Mukhtar ME Mahmoud, Joel JPC Rodrigues, Kashif Saleem, Jalal Al-Muhtadi, Neeraj Kumar, and Valery Korotaev. Towards energy-aware fog-enabled cloud of things for healthcare. *Computers & Electrical Engineering*, 67:58–69, 2018.

[127] Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 7–13, 1997.

[128] José Luis Risco Martin. Github repository. https://github.com/dacya/xdevs, 2014 (accessed October 15, 2020).

[129] Marcelo Masera, Ettore F Bompard, Francesco Profumo, and Nouredine Hadjsaid. Smart (electricity) grids for smart cities: Assessing roles and societal impacts. *Proceedings of the IEEE*, 106(4):613–625, 2018.

[130] Maba Boniface Matadi. The sird epidemial model. *Far East Journal of Applied Mathematics*, 89(1):1, 2014.

[131] Friedemann Mattern and Christian Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. Springer, 2010.

[132] Johannes Mayer. On testing image processing applications with statistical methods. *Software Engineering 2005*, 2005.

[133] Johannes Mayer and Ralph Guderlei. Test oracles using statistical methods. *Testing of component-based systems and software quality*, 2004.

[134] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6. IEEE, 2017.

[135] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 346–353. IEEE, 2001.

[136] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[137] Saurabh Mittal. Model engineering for cyber complex adaptive systems. In *European Modeling and Simulation Symposium, Bordeaux, France*, 2014.

[138] Saurabh Mittal and Scott A Douglas. Devsml 2.0: The language and the stack.

Technical report, AIR FORCE RESEARCH LAB WRIGHT-PATTERSON AFB OH, 2012.

[139] Saurabh Mittal, Umut Durak, and Tuncer Ören. *Guide to simulation-based disciplines: Advancing our computational future*. Springer, 2017.

[140] Saurabh Mittal and José L Risco Martín. *Netcentric system of systems engineering with DEVS unified process*. CRC Press, 2013.

[141] Saurabh Mittal and José L Risco-Martín. Devsml studio: a framework for integrating domain-specific languages for discrete and continuous hybrid systems into devs-based m&s environment. In *Proceedings of the Summer Computer Simulation Conference*, pages 1–8, 2016.

[142] Saurabh Mittal and José L Risco-Martin. Simulation-based complex adaptive systems. In *Guide to Simulation-Based Disciplines*, pages 127–150. Springer, 2017.

[143] ModelioSoft. Modelio. https://www.modelio.org/categories/about-modelio-2.html. [Online; accessed 28-September-2020].

[144] Youssoufa Mohamadou, Aminou Halidou, and Pascalin Tiam Kapen. A review of mathematical modeling, artificial intelligence and datasets used in the study, prediction and management of covid-19. *Applied Intelligence*, 50(11):3913–3925, 2020.

[145] Alejandro Moreno, José L Risco-Martín, Eva Besada, Saurabh Mittal, and Joaquín Aranda. Devs/soa: Towards devs interoperability in distributed m&s. In *2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 144–153. IEEE, 2009.

[146] Ammar Awad Mutlag, Mohd Khanapi Abd Ghani, Net al Arunkumar, Mazin Abed Mohammed, and Othman Mohd. Enabling technologies for fog computing in healthcare iot systems. *Future Generation Computer Systems*, 90:62–78, 2019.

[147] Alexander Muzy and James J Nutaro. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. In *1st Open International Conference on Modeling & Simulation (OICMS)*, pages 273–279, 2005.

[148] Paulo Augusto Nardi, Márcio Eduardo Delamaro, et al. Test oracles associated with dynamical systems models. 2011.

[149] Euclides C Pinto Neto, Gustavo Callou, and Fernando Aires. An algorithm to optimise the load distribution of fog environments. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1292–1297. IEEE, 2017.

[150] Michael J North and Charles M Macal. *Managing business complexity: discovering strategic solutions with agent-based modeling and simulation*. Oxford University Press, 2007.

[151] Alberto Núñez, Jose L Vázquez-Poletti, Agustin C Caminero, Gabriel G Castañé, Jesus Carretero, and Ignacio M Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[152] Jim Nutaro. Adevs (a discrete event system simulator). *Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson. Availableathttp://www.ece.arizona.edu/nutaro/index.php*, 1999.

[153] Headache Classification Subcommittee of the International Headache Society et al. The international classification of headache disorders. *cephalalgia*, 24(1):1–160, 2004.

[154] Megan Olsen and Mohammad Raunak. Increasing validity of simulation models through metamorphic testing. *IEEE Transactions on Reliability*, 68(1):91–108, 2018.

[155] Tuncer I Ören and Bernard P Zeigler. Concepts for advanced simulation methodologies. *Simulation*, 32(3):69–82, 1979.

[156] Jessica Oueis, Emilio Calvanese Strinati, and Sergio Barbarossa. The fog balancing: Load distribution for small cell cloud computing. In *2015 IEEE 81st vehicular technology conference (VTC spring)*, pages 1–6. IEEE, 2015.

[157] Min Ouyang. Review on modeling and simulation of interdependent critical infrastructure systems. *Reliability engineering & System safety*, 121:43–60, 2014.

[158] Dale K Pace. Modeling and simulation verification and validation challenges. *Johns Hopkins APL Technical Digest*, 25(2):163–172, 2004.

[159] Josué Pagán, José M. Moya, José L. Risco-Martín, and José L. Ayala. Advanced migraine prediction simulation system. In *Summer Computer Simulation Conference (SCSC)*, 2017.

[160] Josué Pagán, De Orbe, M Irene, Ana Gago, Mónica Sobrado, José L Risco-Martín, J Vivancos Mora, José M Moya, and José L Ayala. Robust and accurate modeling approaches for migraine per-patient prediction from ambulatory data. *Sensors*, 15(7):15419–15442, 2015.

[161] Josué Pagán, José L Risco-Martín, José M Moya, and José L Ayala. Grammatical evolutionary techniques for prompt migraine prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 973–980, 2016.

[162] Josué Pagán, Marina Zapater, and José L Ayala. Power transmission and workload balancing policies in ehealth mobile cloud computing scenarios. *Future Generation Computer Systems*, 78:587–601, 2018.

[163] Ehsan Pakbaznia and Massoud Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 145–150, 2009.

[164] Krishna Patel and Robert M Hierons. A mapping study on testing non-testable systems. *Software Quality Journal*, 26(4):1373–1413, 2018.

[165] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[166] Ignacio Penas, Marina Zapater, José L Risco-Martín, and José L Ayala. Sfide: a simulation infrastructure for data centers. In *Proceedings of the Summer Simulation Multi-Conference*, pages 1–12, 2017.

[167] Carl Adam Petri. Kommunikation mit automaten. 1962.

[168] Tariq Qayyum, Asad Waqar Malik, Muazzam A Khan Khattak, Osman Khalid, and Samee U Khan. Fognetsim++: A toolkit for modeling and simulation of distributed fog environment. *IEEE Access*, 6:63570–63583, 2018.

[169] Luis Rabelo, Kiyoul Kim, Tae Woong Park, John Pastrana, Mario Marin, Gene Lee, Khalid Nagadi, Bibi Ibrahim, and Edgar Gutierrez. Multi resolution modeling. In *2015 Winter Simulation Conference (WSC)*, pages 2523–2534. IEEE, 2015.

[170] Chandrakant S. Ragit, Sameer Subhash Shirgaonkar, and Dr. Sanjay Badjate. Reconfigurable fpga chip design for wearable healthcare system. *International Journal of Computer Science and Network*, 4(2):208–212, 2015.

[171] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[172] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In

International Conference on Application and Theory of Petri Nets, pages 450–462. Springer, 2003.

[173] Colleen A Redding, Joseph S Rossi, Susan R Rossi, Wayne F Velicer, and James O Prochaska. Health behavior models. In *International Electronic Journal of Health Education*. Citeseer, 2000.

[174] David Reinsel, John Gantz, and John Rydning. Data age 2025: the digitization of the world from edge to core. *Seagate, https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf*, 2018.

[175] José L Risco-Martín, Saurabh Mittal, Juan Carlos Fabero Jiménez, Marina Zapater, and Román Hermida Correa. Reconsidering the performance of devs modeling and simulation environments using the devstone benchmark. *Simulation*, 93(6):459–476, 2017.

[176] José L Risco-Martın, Alejandro Moreno, JM Cruz, and Joaquın Aranda. Interoperability between devs and non-devs models using devs/soa. In *Proceedings of the 2009 Spring Simulation Multiconference on ZZZ*, page 147. Citeseer, 2009.

[177] Susan H Rodger and Thomas W Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.

[178] Gabriel Wainer Román Cardenas, Kevin Henares. COVID19 Cell-DEVS models (ACRI-2020). https://github.com/Simulation-Everywhere-Models/COVID-Cell-DEVS-ACRI2020. [Online; accessed 22-January-2021].

[179] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[180] Hesham Saadawi and Gabriel Wainer. Verification of real-time devs models. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–8, 2009.

[181] Prasan Kumar Sahoo, Suvendu Kumar Mohapatra, and Shih-Lin Wu. Analyzing healthcare big data with prediction for future health condition. *IEEE Access*, 4:9786–9799, 2016.

[182] Robert G Sargent. Verification and validation of simulation models. In *Proceedings of the 2010 winter simulation conference*, pages 166–183. IEEE, 2010.

[183] Hessam S Sarjoughian and BR Zeigler. Devsjava: Basis for a devs-based collaborative m&s environment. *Simulation Series*, 30:29–36, 1998.

[184] Indranil Sarkar and Sanjay Kumar. Fog computing based intelligent security surveillance using ptz controller camera. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–5. IEEE, 2019.

[185] SchedMD. Slurm workload manager. https://slurm.schedmd.com/overview.html (accessed 26-October-2020), 2020.

[186] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[187] Peter M Senge and Jay W Forrester. Tests for building confidence in system dynamics models. *System dynamics, TIMS studies in management sciences*, 14:209–228, 1980.

[188] Chungman Seo and Bernard P Zeigler. Devs namespace for interoperable devs/soa. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1311–1322. IEEE, 2009.

[189] KY Sim, WKS Pao, and C Lin. Metamorphic testing using geometric interrogation technique and its application. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 1(2):91–95, 2005.

[190] Nikolay Simakov. Slurm simulator. https://github.com/ubccr-slurm-simulator/slurm_simulator (accessed 18-June-2021), 2021.

[191] InControl Simulation Software. Pedestrian Dynamics. https://www.incontrolsim.com/software/pedestrian-dynamics/. [Online; accessed 25-December-2020].

[192] Cagatay Sonmez, Atay Ozgovde, and Cem Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[193] Natasha K Stout, Marjorie A Rosenberg, Amy Trentham-Dietz, Maureen A Smith, Stephen M Robinson, and Dennis G Fryback. Retrospective cost-effectiveness analysis of screening mammography. *Journal of the National Cancer Institute*, 98(11):774–782, 2006.

[194] Lars Jacob Stovner and Colette Andree. Prevalence of headache in europe: a review for the eurolight project. *The journal of headache and pain*, 11(4):289, 2010.

[195] Chang-Ai Sun, Guan Wang, Baohong Mu, Huai Liu, Zhaoshun Wang, and Tsong Yueh Chen. Metamorphic testing for web services: Framework and a case study. In *2011 IEEE International Conference on Web Services*, pages 283–290. IEEE, 2011.

[196] Daniel Tokody, Imre János Mezei, and György Schuster. An overview of autonomous intelligent vehicle systems. In *Vehicle and Automotive Engineering*, pages 287–307. Springer, 2017.

[197] Andreas Tolk. Their implications for distributed simulation: Towards mathematical foundations of simulation interoperability. In *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 3–9, 2013.

[198] Luc Touraille, Mamadou K Traoré, and David RC Hill. A mark-up language for the storage, retrieval, sharing and interoperability of devs models. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 1–6, 2009.

[199] Shreshth Tuli, Nipam Basumatary, Sukhpal Singh Gill, Mohsen Kahani, Rajesh Chand Arya, Gurpreet Singh Wander, and Rajkumar Buyya. Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated iot and fog computing environments. *Future Generation Computer Systems*, 104:187–200, 2020.

[200] CARNEGIE MELLON UNIVERSITRY. The" only" coke machine on the internet, 2018.

[201] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[202] Wil MP van der Aalst, Poul JN de Crom, Roy RHMJ Goverde, Kees M van Hee, Wout J Hofman, Hajo A Reijers, and Robert A van der Toorn. Exspect 6.4 an executable specification tool for hierarchical colored petri nets. In *International Conference on Application and Theory of Petri Nets*, pages 455–464. Springer, 2000.

[203] Peter Van Overschee and Bart De Moor. N4sid: Subspace algorithms for the identification of combined deterministic-stochastic systems. *Automatica*, 30(1):75–93, 1994.

[204] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the python (p) devs simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation-DEVS*, pages 387–392, 2014.

[205] Yentl Van Tendeloo and Hans Vangheluwe. An evaluation of devs simulation tools. *Simulation*, 93(2):103–121, 2017.

[206] Hans Vangheluwe. Foundations of modelling and simulation of complex systems. *Electronic Communications of the EASST*, 10, 2008.

[207] Hans LM Vangheluwe. Devs as a common denominator for multi-formalism hybrid systems modelling. In *Cacsd. conference proceedings. IEEE international symposium on computer-aided control system design (cat. no. 00th8537)*, pages 129–134. IEEE, 2000.

[208] Damián Vicino, Daniella Niyonkuru, Gabriel Wainer, and Olivier Dalle. Sequential pdevs architecture. 2015.

[209] Change Vision. Astah SysML. https://astah.net/products/astah-sysml/. [Online; accessed 28-September-2020].

[210] William Voorsluys, James Broberg, Rajkumar Buyya, et al. Introduction to cloud computing. *Cloud computing: Principles and paradigms*, pages 1–44, 2011.

[211] Gabriel Wainer. Cd++: a toolkit to develop devs models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[212] Gabriel A Wainer and Norbert Giambiasi. Application of the cell-devs paradigm for cell spaces modelling and simulation. *Simulation*, 76(1):22–39, 2001.

[213] Fangxin Wang, Feng Wang, Xiaoqiang Ma, and Jiangchuan Liu. Demystifying the crowd intelligence in last mile parcel delivery for smart cities. *IEEE Network*, 33(2):23–29, 2019.

[214] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE mobile computing and communications review*, 3(3):3–11, 1999.

[215] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[216] S Hoya White, A Martín Del Rey, and G Rodríguez Sánchez. Modeling epidemics using cellular automata. *Applied mathematics and computation*, 186(1):193–202, 2007.

[217] Timothy M White and Thomas P Way. jfast: A java finite automata simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 384–388, 2006.

[218] WHO. The atlas of heart disease and stroke. https://www.who.int/cardiovascular_diseases/resources/atlas/en/ (accessed 13-October-2020), 2018.

[219] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical machine learning tools and techniques with java implementations. 1999.

[220] Wayne Wymore. A mathematical theory of systems engineering: the elements. 1967.

[221] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[222] Xintian Xu, Ping Chen, Jingfang Wang, Jiannan Feng, Hui Zhou, Xuan Li, Wu Zhong, and Pei Hao. Evolution of the novel coronavirus from the ongoing wuhan outbreak and modeling of its spike protein for risk of human transmission. *Science China Life Sciences*, 63(3):457–460, 2020.

[223] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78. IEEE, 2015.

[224] Terry Young, Julie Eatock, Mohsen Jahangirian, Aisha Naseer, and Richard Lilford.

Three critical challenges for modeling and simulation in healthcare. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1823–1830. IEEE, 2009.

[225] Greg L Zacharias, Jean Ed Macmillan, and Susan B Van Hemel. *Behavioral modeling and simulation: From individuals to societies.* National Academies Press, 2008.

[226] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

[227] Bernard P Zeigler, Yoonkeon Moon, Doohwan Kim, and George Ball. The devs environment for high-performance modeling and simulation. *IEEE Computational Science and Engineering*, 4(3):61–71, 1997.

[228] Bernard P Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of modeling and simulation: discrete event & iterative system computational foundations*. Academic press, 2018.

[229] Bernard P Zeigler, Alexandre Muzy, and Ernesto Kofman. Open research problems: Systems dynamics complex systems. *Theory of Modeling and Simulation*, pages 641–658, 2019.

[230] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2 edition, 2000.

[231] Xuezhi Zeng, Saurabh Kumar Garg, Peter Strazdins, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, and Rajiv Ranjan. Iotsim: A simulator for analysing iot applications. *Journal of Systems Architecture*, 72:93–107, 2017.

[232] Ahmet Zengin and Muhammed Maruf Ozturk. Formal verification and validation with devs-suite: Ospf case study. *Simulation Modelling Practice and Theory*, 29:193–206, 2012.

[233] Xiange Zhang. Application of discrete event simulation in health care: a systematic review. *BMC health services research*, 18(1):1–11, 2018.

[234] ShaoBo Zhong, QuanYi Huang, and DunJiang Song. Simulation of the spread of infectious diseases in a geographical environment. *Science in China Series D: Earth Sciences*, 52(4):550–561, 2009.

[235] Suiping Zhou, Dan Chen, Wentong Cai, Linbo Luo, Malcolm Yoke Hean Low, Feng Tian, Victor Su-Han Tay, Darren Wee Sze Ong, and Benjamin D Hamilton. Crowd modeling and simulation technologies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):1–35, 2010.

[236] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 346–351. Software Engineers Association Xian, China, 2004.

Glossary

ANS Autonomic Nervous System. 29

AP Access Point. 94

API Application Programming Interfaces. 4, 21, 30, 42, 50, 60, 94, 170

ARX Auto-Regressive model with eXogenous inputs. 132

BSON Binary JSON. 113

CA Cellular Automata. 42, 44

CDEVS Classic DEVS. 4, 44

CSV Comma-Separated Values. 94, 146

DES Discrete-Event Simulation. 32

DEVS Discrete Event System Specification. 20, 24, 37, 39, 41, 49, 59, 64, 78, 97, 120, 123, 133, 144, 173

DEVSML DEVS Modeling Language. 6

DPM Disease Progression Modeling. 32

DSS Decision Support System. 119

DynDEVS Dynamic DEVS. 4, 44

ECG ElectroCardioGram. 29

EDA ElectroDermal Activity. 29

EDC Edge Data Center. 94

EEG ElectroEncephaloGram. 29

EF Experimental Frame. 6, 53, 59

EFP Experimental Frame - Processor. 51

EIC External Input Couplings. 41, 47, 69

EOC External Output Couplings. 41, 47, 69

FDDEVS Finite and Deterministic DEVS. 4, 6, 44

FPGA Field-programmable gate array. 120, 122, 132, 139, 168

FSA Finite State Automata. 24

FTP File Transfer Protocol. 146, 150

GSMP Generalized Semi-Markov Process. 26

GUI Graphical User Interface. 38, 42, 43, 95, 96

HBM Health Behavior Modeling. 32

HCSO Health and Care Systems Operation. 32

HDL Hardware Description Language. 123

HMS Healthcare Monitoring System. 10, 120, 122, 123, 130, 168

HPC High-Performance Computing. 45

HTTP Hypertext Transfer Protocol. 146

IaaS Infrastructure as a Service. 30

IC Internal Couplings. 41, 47, 68

IDC International Data Corporation. 1, 28, 89

IoT Internet of Things. 3, 7, 27

JSON JavaScript Object Notation. 53, 65

M&S Modeling and Simulation. ix, 2, 3, 9, 10, 19, 22, 27, 37, 39, 45, 49, 74, 115, 119, 143, 144, 165

M&SBSE Model & Simulation Based Systems Engineering. 2, 20

MBSE Model-based systems engineering. 2, 20

MDC Micro Data Center. 101, 106, 167

MPI Message Passing Interface. 43

MRM Multi-Resolution Modeling. 17

MT Metamorphic testing. 73

N4SID Subspace State Space System Identification. 134

NFC Near Field Communication. 28

NIST National Institute of Standards and Technology (United States). 29

ODE Ordinary Differential Equations. 15

OSPF Open Shortest Path First. 5

PDEVS Parallel DEVS. 4, 37, 44

QVT Query/View/Transformation. 27

RFID Radio Frequency IDentification. 28

SD System Dynamics. 32

SFIDE Simulation Framework and Infrastructure for Data cEnters. 9, 90, 97, 167, 170

SIR Susceptible-Infected-Recovered (epidemiological model). 120

SIRDS Susceptible-Infected-Recovered-Deceseaded-Susceptible (epidemiological model). 123

SOA Service Oriented Architecture. 4, 170

SpO2 Peripheral oxygen saturation. 29

SQL Structured Query Language. 146

SSH Secure Shell. 146, 150

SUT System Under Test. 71

SysML Systems Modeling Language. 27

UML Unified Modeling Language. 27

UPDRS Unified Parkinson's Disease Rating Scale. 33

V&V Verification and Validation. 21

VHDL VHSIC Hardware Description Language. 130, 132, 134, 168

WBSN Wireless Body Sensor Network. 94, 139

WSDL Web Services Description Language. 4

WSN Wireless Sensor Networks. 28

xDEVS Cross-platform DEVS simulator. 49

XLS/XLSX Microsoft Excel Spreadsheet. 146

XML Extensible Markup Language. 4, 6, 11, 54, 60, 143

XSD XML Schema Definition. 151