Markus Schuß, BSc

Design and Implementation of a Distributed Simulation Framework based on Cloud Computing

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Ralph Weissnegger, BSc

Institute for Technical Informatics
Graz University of Technology

Graz, March 2016

Kurzfassung

Während UML (Unified Modeling Language) ein etabliertes Werkzeug im Bereich der Softwareentwicklung ist, setzt es sich in anderen Anwendungsdomänen erst nach und nach durch. Der Ansatz Systeme lediglich zu modellieren reicht jedoch in diesen Domänen oft nicht aus, dies gilt vor allem für die Elektronik- und Automobilindustrie. Darum drängen diese Branchen dazu, UML bereits in früheren und vor allem mehreren Phasen der Entwicklung einzusetzen, als es in der Softwareentwicklung der Fall ist. Diese Arbeit befasst sich daher mit dem Schritt der Simulation von modellierten Systemen (mit einem Fokus auf ein Beispiel aus der Automobilindustrie) unter Verwendung der massiven Rechenleistung der Cloud. Hierfür wurde das SHARC Framework erstellt, welches sowohl als Modellierungswerkzeug als auch als lokale Simulationsumgebung dient. Mit ihm können die erstellten Modelle zur Simulation an einen Scheduler zur Verteilung in die Cloud eingereicht werden. Diese Arbeit beschreibt die Struktur und den Aufbau dieses Frameworks und die Ressourcen und Infrastruktur welche benötigt werden, um es zu verwenden.

Abstract

While UML (Unified Modeling Language) serves as a well-established tool for modeling systems in the domain of software development it is still being established in other domains. Modeling a system is not always sufficient therefore the industry, especially electronics and automotive companies, push to establish UML in earlier and more steps during development than software development. This holds especially true for simulation which is not at all a part for typical software development workflow. This thesis investigates simulation of a system using on an automotive example utilizing the vast computational resources available in the cloud. For this the SHARC framework was created which serves as a modeling tool as well as a local simulation environment. Simulation tasks can be submitted to a scheduler distributing these tasks among workers running on compute instances in the cloud. This work describes the structure and design of this framework and the resources and infrastructure required to deploy it.

Danksagung

Diese Masterarbeit wurde im Jahr 2015/2016 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Ich möchte mich an dieser Stelle beim gesamten Institut bedanken, vor allem jedoch bei Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger, welcher mich bei dieser Arbeit betreut hat. Außerdem möchte ich mich bei Dipl.-Ing Ralph Weissnegger BSc. für die gute Unterstützung und Betreuung bedanken. Besonderer Dank gilt an dieser Stelle auch der CISC Semiconductor GmbH durch welche mir diese Arbeit erst ermöglicht wurde. Zuletzt möchte ich noch meinem Mentor Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eugen Brenner für die gute Betreuung während meines Studiums danken.

Dank gilt hier auch meiner Mutter, für Ihr Verständnis und Unterstützung. Auch allen Freunden, vor allem Manuel Weber BSc. und Christina Tanja Ballek, die mich auf diesem Weg begleitet und unterstützt haben, möchte ich von ganzem Herzen danken.

Graz, im März 2016 Markus Schuß

Contents

Chapter 1

Introduction

This work was created as part of the eRamp (Excellence in Speed and Reliability for More than Moore Technologies) project which is funded by a joint undertaking of several European countries. This document introduces the SHARC framework, which uses a SystemC AMS (a SystemC variant with additional support for analog and mixed systems) based simulation core in a cloud computing environment. It enables the simulation of mixed signal systems modeled via UML (Unified Modeling Language) diagrams designed with the support of tools commonly used in academia as well as industry such as Papyrus, an Eclipse [1] plugin and part of the Eclipse Modeling Tools. The main goal of this work was to create a viable flow from the initial design, created from specification, to verification using compute resources in the cloud. While the generation of testbenches using an UVM like (Unified Verification Methodology [2]) approach was integrated into the SHARC framework the details of this feature will be covered in future work. The core used for the distributed simulation was created in prior work (as part of a project at the university which was also part of the the eRamp project) and has been adapted to better suit the needs of the cloud based simulation approach. The basic functionality is briefly covered as well but it is not intended as a full technical manual for the simulation core. The core was initially designed to be extensible as well as portable with a focus on memory footprint so it required only a few adaptations to enable it for a cloud environment.

1.1 Motivation

While simulation is becoming a viable solution for verification and certification, as well as an integral part of today's design flows, starting in earlier phases than ever before, the scale of these simulations is also getting larger. Once it was once enough to simulate the theoretical model of a control system or an electrical schematic upon completion. Nowadays simulation is used to create and test requirements, to test against fault injections and reliability, to help make decisions for allowed tolerances of individual components and much more. This also implies that not only the size of the individual simulations is getting larger but also the number of runs as well as the amount of different simulation types is growing fast. This often results in time wasted on tool migration due to the different standards used by various tool vendors and products. In order to reduce the time required for simulation and the number of tools in the design steps overall the SHARC framework was created. It is based on UML which is already a tool commonly used by academia and industry in the design of software and is becoming increasingly relevant in hardware design due to extensions such as MARTE [3] and EAST-ADL [4]. These extensions, called annotations, are also a key reason why UML was chosen for the proposed framework as nearly any step in the design flow can be modeled using UML and anything not yet part of UML to date can be added via annotations. This also means that the tool or tools used for simulation can only consider the currently relevant annotations and diagrams while simply ignoring the rest. Therefore several design steps can now run in parallel. This means, for example, one designer already starts to partition the existing function blocks into hard- and software while another can already find possible faults attacks on these blocks while a third verifies a given design against the original requirements. This breaks away from a simple waterfall model where each step depends upon the completion of the last and therefore reduces the time required for the design process. In addition any changes required do not necessarily result in a complete redesign of the system. This means that the time required for simulation may now heavily influence the time to market for a given product. The focus of this work was therefore on how to speed up simulation using either a public or private cloud. While the preliminary user interface based on Eclipse was also created as part of this work the final version will be part of future works and is only briefly described where needed in order to give a better understanding of the overall process.

1.2 Goals

This section elaborates the basic goals achieved by the simulation framework. As the task was to create a complete solution for the entire design flow from modeling to verification there are many components playing together. The four major components that make up the framework in its current form are:

- **The SysCore binary:** This part is used as the simulation core for the framework. It is written in SystemC AMS and uses an UML file to describe the model. It is used on the workers in the cloud as well as part of the locally installed IDE (Integrated Development Environment).
- **The Eclipse product:** This part is the main user interface and created in cooperation with Martin Schachner. It relies on a set of existing and custom plugins merged into a single Eclipse product. It serves as a development environment with an integrated UML editor and a trace file viewer. From it simulations can be started locally as well as in the cloud. Testbenches for models can be created using an UVM inspired procedure, as there is currently no implementation of UVM standard for SystemC AMS.
- **The Worker queue:** This part serves as a load balancer and task scheduler for all the workers. This includes mechanisms for robustness as workers may be added or removed at any time. Each worker reserves one task in the queue and it is only deleted after successful upload of the simulation results to the master. The worker side implementation consists of a Python based daemon while the master is an AMQP (Advanced Message Queuing Protocol) based RabbitMQ server.

- **The Web-interface:** This part currently serves two purposes: one is to allow the user to view the results of all prior simulations as well as to download the resulting trace files and the workpackage, which contains all files used for simulation, for further inspections. The other is meant for machines in the form of a webservice. It allows clients to submit data for simulation tasks and workers to upload the results. It can also be used for integration into tools such as Eclipse or custom web frontends by developers.

1.3 Structure

Chapter 2 covers the state of the art as well as all works that influenced this project or cover related topics. Chapter 3 elaborates the basic structure of the core components as well as the use cases of the SHARC framework. This is done using UML syntax where applicable. It also elaborates on the underlying technologies and concepts used in the creation of the framework itself. Chapter 4 describes internal structure of the framework as well as the infrastructure required to run SHARC and gives an example for the workflow described in the design chapter in the form of an automotive control system. In chapter 5 the results of this work will be evaluated and options for further development are explained. The appendix contains (source code) examples for some of the technologies covered by this work.

Chapter 2

Related works

The field of EDA tools is mostly dominated by a few large companies such as Synopsis[1], Cadence[2] and Mentor Graphics[3]. While there exists some extend of academic research much is still financed by these companies or in cooperation with them. Therefore this chapter is split into two sections: the first mainly focuses on the state of the art in the industry while the second contains the related works that originate mostly from academic research. As this work was written over an extended period of time newer releases or products of these vendors may address shortcomings or add features missing in here. An example for this is `systemvision.com` which launched late during this work and is a cloud/web based EDA tool from Mentor Graphics. While currently lacking many features of their commercial tools the nature of the software shows promise and there may be versions of it for enterprise customers that address these shortcomings. Due to these rapid changes only works that existed prior to the start of the project are listed here as they influenced key decisions in the creation of the SHARC framework.

2.1 Industry

This section gives a brief overview of how the major companies in the EDA (Electronic design automation) handle the emergence of cloud computing. While one can always just roll out existing software on a virtual machine on an IaaS provider and use a remote desktop session to use the design tools (if the license even permits such use) this section covers more fundamental changes required and proposed by the industry.

2.1.1 Synopsys

Synopsys published in their in-house info bulletin [5] an overview of their approach to EDA in the cloud. While in 2011 the cloud computing was already rather well established tool vendors still struggled to apply their know-how and tools to this platform. Even though this no longer holds true today this article still holds relevance today as it nicely highlights the varying requirements in the different design phases. Figure 2.1 highlights a concept

[1]`http://www.synopsys.com/`
[2]`https://www.cadence.com`
[3]`https://www.mentor.com/`

Synopsys calls "surge compute resources". This is also used in the SHARC framework on a dynamic level as resources can be added (or removed) at any given time. While this bulletin gives little detail it serves well as a starting point for Synopsys' efforts into this topic.

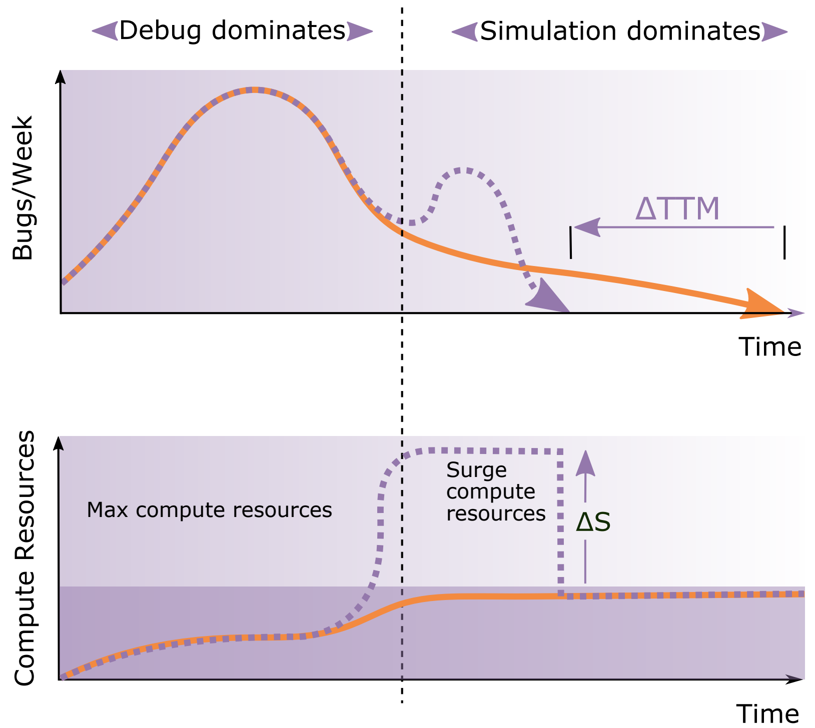


Figure 2.1: Synopsys' early vision of cloud computing (adapted from [5]).

In [6], an official publication by Synopsys from the same year, the author highlights the advantages of a scalable and flexible cloud infrastructure. It also shows that even at the time cloud computing offered benefits over conventional in-house server farms due to a number of limiting factors highlighted in fig. 2.2. A main focus of the paper is that a flexible solution such as a public cloud is especially useful if unforeseen delays in one projects threaten the schedule of different projects due to conflicts in resource allocation. While this was previously handled by over-provisioning server farm capabilities or accept delays cloud computing offers a way to only add additional resources when required.

Sadly Synopsys had not made any noteworthy publications since then and no resources are publicly available on how to deploy their tools to the cloud.

2.1.2 Cadence

Unlike Synopsys Cadence appears on little to no publications on the topic of cloud based EDA but offers a glimpse on their strategy in [7]. This blog posting shows the problems established EDA tool vendors have with the move to the cloud. Their current licensing models are still based on a per seat model and they only envision providing existing tools as "Apps". This article still serves as a good example against a web-based IDE as stated in the article one has to remove functionality in order to simplify support. A local IDE

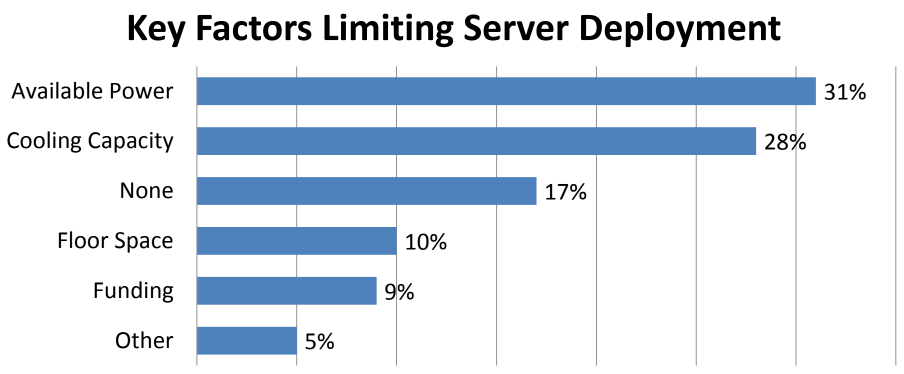


Figure 2.2: Key limiting factors for on-site server farms (adapted from [6]).

serves the role thin-clients had in the era of mainframe computing as it is a relatively weak machine that only serves the user-interface while the compute intense tasks are executed somewhere else. [8] gives an example of Cadences SaaS offering in the context of integrated circuit design. Figure 2.3 shows the mapping of the typical layers of SaaS to the specific scenario with a remote desktop connection for the user interface. While this also allows for a thin-client solution it is highly dependent on the latency and performance of the internet connection. While such a solution was preferable when desktop computers where much less capable even the lowest offerings in today's PC market are more than capable of rendering the user interface locally.

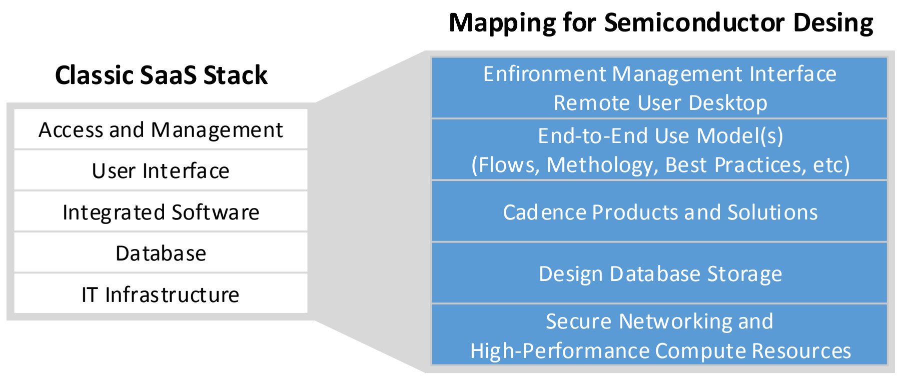


Figure 2.3: SaaS in the context of integrated circuit design (adapted from [8]).

2.1.3 Mentor Graphics

As mentioned in the introduction Mentor Graphics recently unveiled their own cloud based offering. While Mentor Graphics also has little no now publications of their own they when it comes to cloud computing [9] is highly applicable to this topic. Instead of using many machines to work on a collective task-set this work focuses on a multi-processor approach. This allows for a better utilization of all cores of a machine working on a single simulation. Mentor's licensing model allows for shared use and seems generally more permissive in terms of geographic or machine requirements (e.g., a single license may be used on any site within Europe but not on multiple machines at once). This approach is in stark contrast to the SHARC framework which uses only a single processor for each simulation (as the underlying SystemC is not thread safe yet) but on many low powered virtual machines. While mentors licensing model seems to allow cloud based deployments using shared licenses this still means that the software has to be maintained by the user and there is no SaaS (Software as a Service) offering from Mentor Graphics either.

2.1.4 Summary

A recent article [10] published to an electronics blog summarizes the current state of EDA in the cloud as a SaaS solution. While tool vendors are starting to offer some tools as SaaS they mostly expect the customer to purchase licenses and deploy and manage them themselves. From the article it is also clear that these solutions are usually based on remote desktop or similar technologies which rely on a constant and low latency connection for a proper user experience. While they do mention a pay per use licensing model their websites at the time of this writing did not represent this model. This article still offers a unique perspective on the current development and highlight that these vendors are embracing the cloud, while slower than expected considering the first public statements were made in 2011. Another mayor problem highlighted in this article is that a full EDA design flow consists of around 30 tools (which usually come with their own user interface) but the majority of compute resource is apparently consumed in the verification steps (according to Christopher Porter, IBM's HPC Cloud Product). The fact that these tools are meant to be run only on HPC (high performance computing) as opposed to general purpose computing resources such as AWS' t2 tier which greatly limits the number of sites to host the tools.

2.2 Academia

The majority of publications on the topic of cloud based EDA and its applications so far have been proposed by academic institutions or are affiliated with such. This section will highlight a few that inspired concepts gave reasons for decisions on the overall concept behind the SHARC framework.

2.2.1 Cloud-EDA: a PaaS Platform Architecture and Application Development for IC Design & Test

In [11] the author details a similar architecture as proposed in this work. In this paper C. Man et. al. elaborate the use IaaS but add an additional ”App-Store” to the design in order to offer a proper PaaS (Platform as a Service) solution. While this approach could be used for the monetization of the SHARC framework this is outside of the scope of this work. Figure 2.4 shows the current state for IaaS deployments as perceived by the authors while 2.5 shows the approach proposed by them. This structure allows for better management an monetization of the resources but it is also shows the clear difference between ”reselling” IaaS obtained from a Service Provider as is and leaving the task management and prioritization to a Manager. This layered design was also incorporated

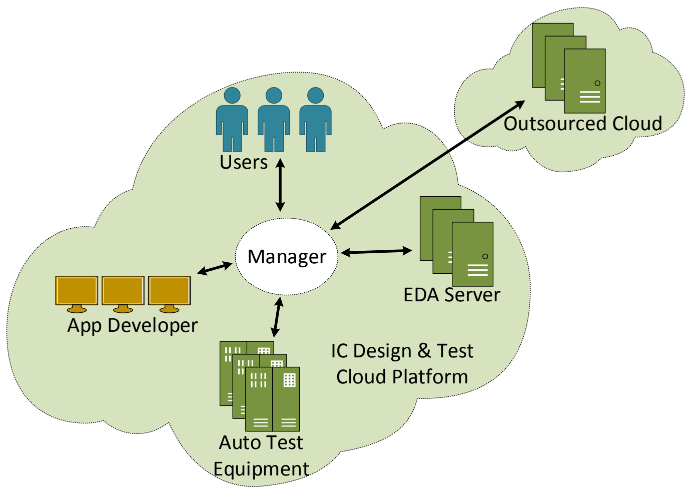


Figure 2.4: Architecture of Cloud-EDA Platform (adapted from [11]).

into the SHARC framework which differentiates between (framework-) developers which use an Eclipse CDT installation to develop C models distributed to customers which in turn use a modified Eclipse IDE for model design. Both do not need any information about the underlying EDA servers and tool other than the agreed upon interfaces. One major difference is the fact that our solution integrates the task management into the scheduler at the server level and the task queues are currently not manageable by the user. This could be added as a feature in future work though.

2.2.2 Experiences with a Private Enterprise Cloud: Providing Fault Tolerance and High Availability for Interactive EDA Applications

In [12] several engineers from Intel provide an overview of their design solution utilizing a private cloud for EDA. It is mostly based on mainframe/cluster technologies and serves as an alternative architecture. During the development our focus shifted towards a reliable and fault tolerant infrastructure on commodity due to the fact that our computation tasks are not spread among several machines. In this paper the authors show that the utilization between different types of servers varies widely. They classify their tasks into three main groups:

- **Batch computing**: Has a high and nearly constant utilization of more than 80% of the overall compute resources. This category covers mainly the simulation tasks.
- **Interactive computing**: The workload and therefore the utilization in this category is defined by user input. Systems in this category constitute for only 15% of the overall computing resources.
- **Long running and resource intensive workloads**: While this is seen as a category which will see rising use at Intel it is currently not relevant for our application.

These categories are also different in their needs for high availability therefore the authors offer different solutions (at least the first two categories). The proposed solution for

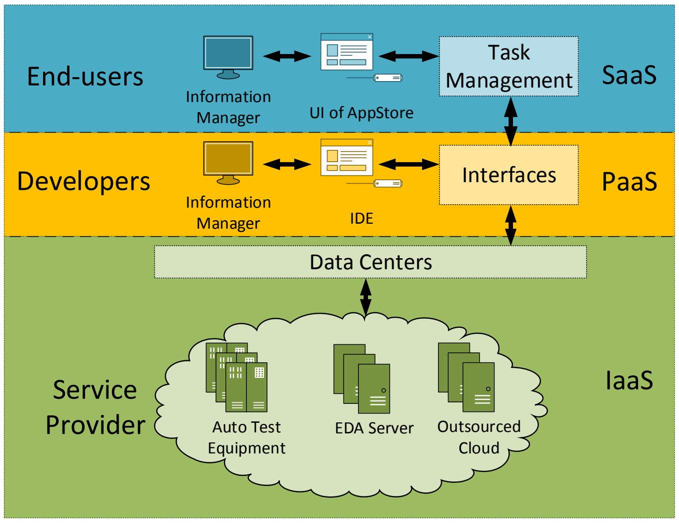


Figure 2.5: The hierarchy layered design of Cloud-EDA Platform (adapted from [11]).

batch computing is a checkpointing system which allows the task to be resumed on failure. This is done by DMTCP (Distributed MultiThreaded Checkpointing) which allows the system to create snapshots of a single binary with all its threads without any modifications to the operating system. According to the paper DMTCP claims to have support for OpenMPI which was considered as a solution for the distribution of tasks among multiple machines. While this is currently not supported by the SHARC framework it is a viable topic for future works.

Figure 2.6 gives a good understand of how the use the migration features in conjunction with VNC for remote access to provide a fault tolerant work environment. Combining these two aspects - the checkpointing of the batch pool and the seamless migration for the interactive pool - they are able to provide a high reliability for their users. While checkpointing was not used in the design of the SHARC framework the need for clear separation of interactive parts of the framework (the SHARC IDE based on Eclipse) and the processing parts became clear. In order to avoid an availability issues of a remote desktop based IDE we decided to base our design on an existing platform that is easy to decouple from the actual computation task. As the end users and developers are not bound to a hosted IDE in this design one of the two points of failure could be removed early in the design phase.

2.2.3 AzureBOT: A Framework for Bag-of-Tasks Applications on the Azure Cloud Platform

In [13] D. Agarwal and S. K. Prasad show a framework for a Bag-of-Tasks Application on a specific cloud platform. This approach utilizes the API of Microsoft Azure platform but the core problem is the same. A ”Bag” of unrelated tasks that need to be processed on a (smaller) number of workers. Figure 2.7 shows an overview of the architecture proposed by the authors. This work has a very similar design to our final framework but heavily relies on the API of a single vendor. In our approach we decided to use standards available from

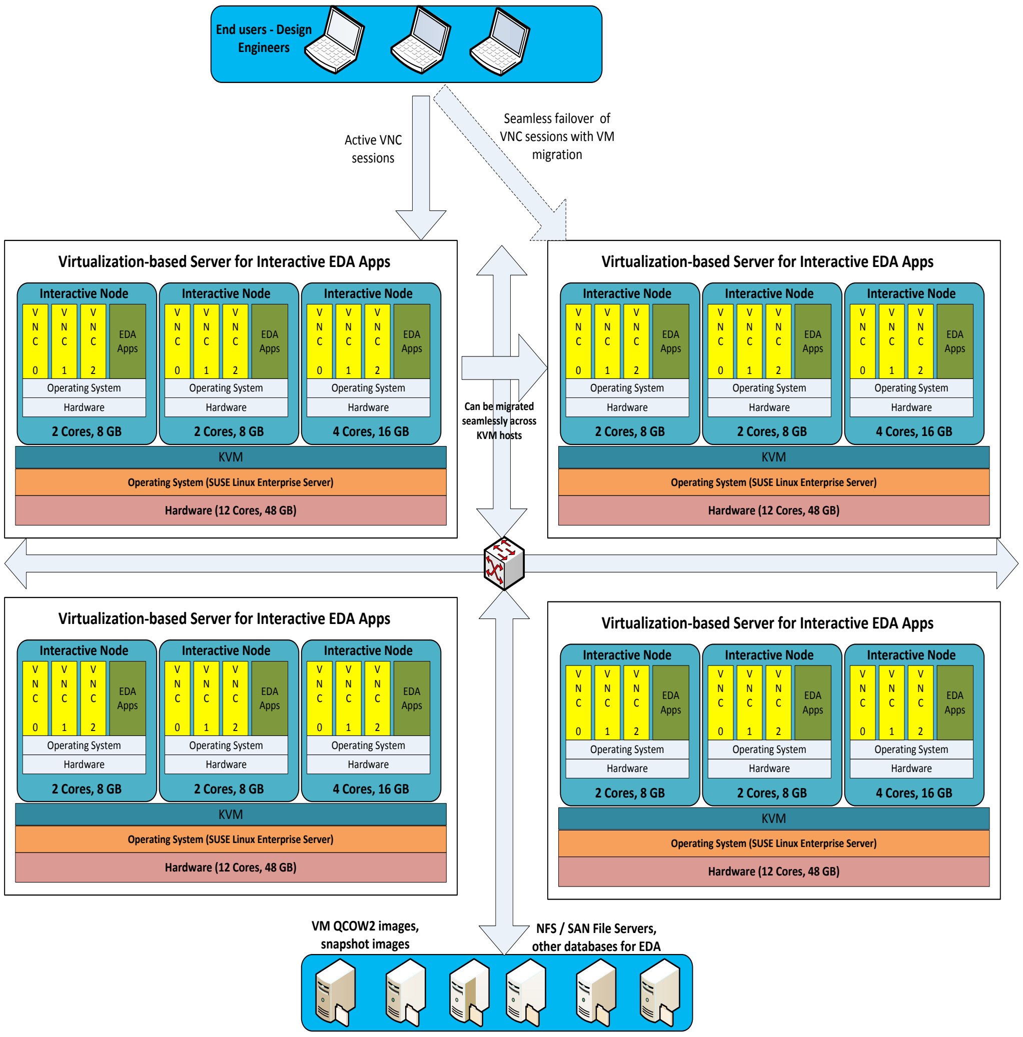

End users - Design Engineers
Active VNC sessions
Seamless failover of VNC sessions with VM migration
Virtualization-based Server for Interactive EDA Apps
Interactive Node
VNC 0
VNC 1
VNC 2
EDA Apps
Operating System
Hardware
2 Cores, 8 GB
4 Cores, 16 GB
KVM
Operating System (SUSE Linux Enterprise Server)
Hardware (12 Cores, 48 GB)
Can be migrated seamlessly across KVM hosts
VM QCOW2 images, snapshot images
NFS / SAN File Servers, other databases for EDA

Figure 2.6: System Architecture for providing Fault Tolerance to EDA applications at Intel (extracted from [12]).

multiple vendors in order to avoid a vendor lock-in. Other than that the main concepts remains, the producer in their diagram is the SHARC IDE which creates a number of tasks and stores the data required on the server. The task is then put in a queue which is in turn consumed by workers which fetch external data from the server and compute the results. These results are then uploaded to the server. Figure 2.8 highlights that the main scaling issue (for their implementation) is the fact that a single master can only supply a finite number of workers with data. This influenced the decision to use only very small payloads for our jobs and use a high performance message queuing server which is capable of a throughput of several thousand messages per second (for small payloads) as well as spreading a single queue among several masters in order to scale beyond the limitations of a single machine (implemented by RabbitMQ via federated queues).

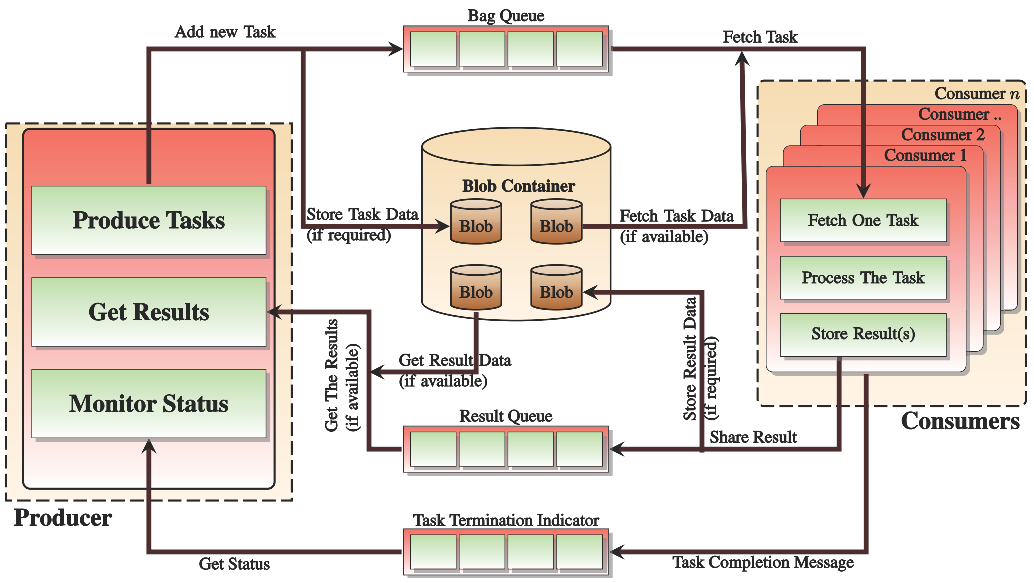


Figure 2.7: A typical setup for a bag-of-tasks application employing AzureBOT framework (extracted from [13]).

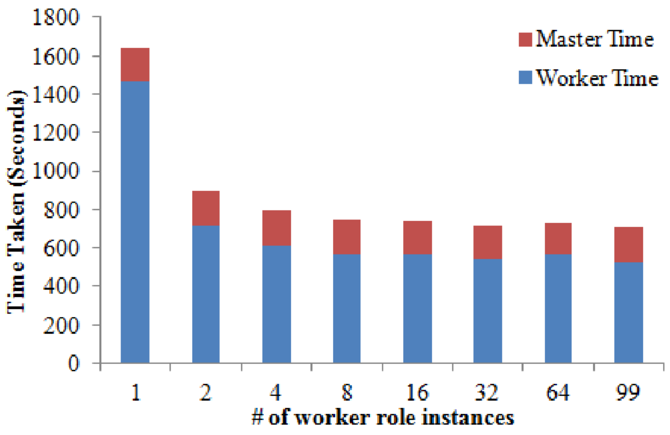


Figure 2.8: Performance of an example application over varying number of Azure worker role instances (extracted from [13]).

While the custom API proposed by the paper was replaced by established and proven technologies early in the design phase the concept remained an inspiration during the design. The use of standardized technologies such as HTTP(S), SQL and AMQP allow for a number of different implementations of these protocols (such as Apache or Nginx for HTTP etc.) to be used on any IaaS solution in order to avoid vendor lock-ins at all. All tools used are able to use load balancing techniques in order to avoid the problems encountered in the paper. This also greatly reduced the development time of the worker queue and due to a large number of deployments of the individual components ensures that security updates are provided and bugs are fixed quickly. All tools used also come with build in security features such as authentication as well as encryption.

2.2.4 TerraME HPA: Parallel Simulation of Multi-agent Systems over SMPs

[14] proposes an extension to the LUA programming language for creating models simulated on a distributed environment. For this the TerraME annotations are used to implement different parellization strategies and a bag-of-tasks approach to provide load balancing over multiple processors or machines. While the use of a scripting language has the advantage of being easier to learn for the end user as it hides the specifics of the scientific and signal processing libraries employed by the interpreter such interpreted languages usually sacrifice performance as the code has to be read and executed by a separate program (the interpreter). The work itself focuses on a multi-model approach to simulate an environmental model but all models are created using the same tool (no co-simulation with existing tools). The overall architecture of the resulting framework is illustrated in fig. 2.9 and their bag-of-tasks distribution scheme in fig. 2.10.

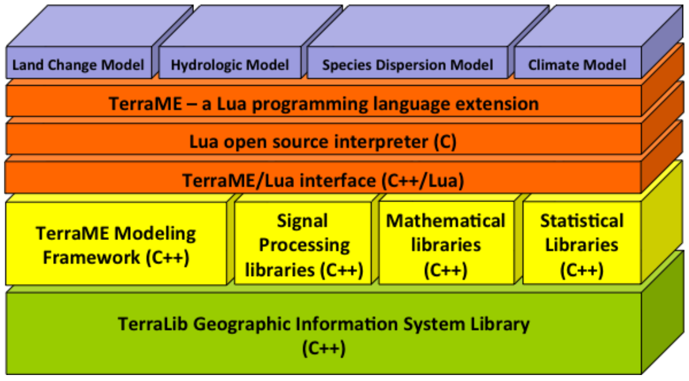

Land Change Model
Hydrologic Model
Species Dispersion Model
Climate Model
TerraME – a Lua programming language extension
Lua open source interpreter (C)
TerraME/Lua interface (C++/Lua)
TerraME Modeling Framework (C++)
Signal Processing libraries (C++)
Mathematical libraries (C++)
Statistical Libraries (C++)
TerraLib Geographic Information System Library (C++)

Figure 2.9: TerraME architecture (extracted from [14]).

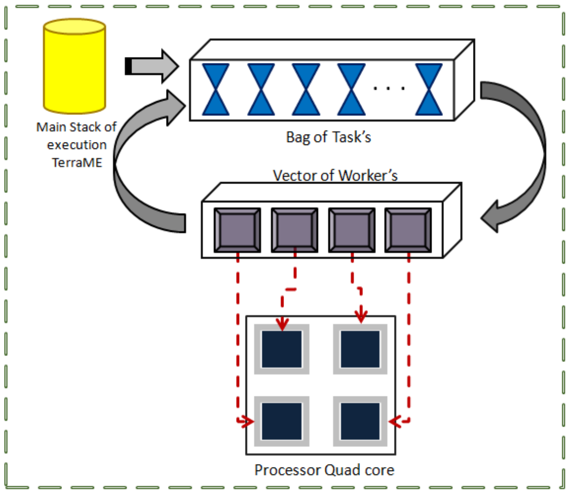


Figure 2.10: TerraME HPA parallelization mechanism (extracted from [14]).

Using this design the authors could show an impressive, near linear speedup on a simple pray-predator model shown in fig. 2.11 while the speedup observed on the original TROLL forest grow model only resulted in a 35% speedup for two and a speedup of 3 for eight cores. This is however due to the large number of synchronization points of such a complex model. Given the results of this model and the impact of the added overhead of spreading the computation among multiple cores/processors the SHARC framework was designed to run only on a single core (until the underlying SystemC kernel becomes properly thread safe). This result is especially interesting compared to fig. 2.8 which clearly shows that an inefficient framework can have devastating effects on the performance.

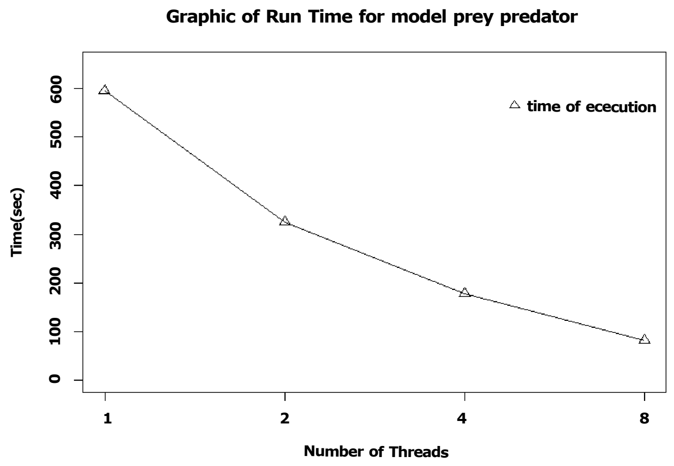


Figure 2.11: Prey-Predator simulation runtime (adapted from [14]).

2.2.5 Summary

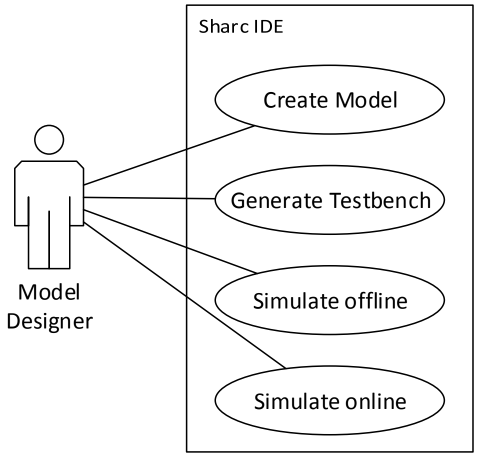
While there has been some research on parallelization for HPC and multi-core environments most papers ignore the general purpose compute cloud. As a distributed model is heavily impacted by network latency and throughput if there is a large degree of synchronization distribution of a single model already has a large impact on a simple multi-core system as shown in [14]. Network latency and throughput only affect a bag-of-tasks approach to the degree the latency delays the fetching of a task or resource which usually only occurs before the task is started. Due to this property it is perfectly suited for cloud applications as has been shown in [13]. Overall most approaches add a PaaS or SaaS layer on top of the IaaS obtained from the service provider which allows easy migration of data centers and providers without the need of any adaptation of the framework. [11] best shows the advantage (and need) for a layered design as it ensures a fair distribution of resources without the need of a central manager while maintaining the option to prioritize certain jobs by using a (managed) queue. [12] perfectly highlights the need for a robust and fault tolerant solution especially on a private cloud where the compute resources are fixed. While this requirement can be relaxed to some degree on a public cloud the need for robustness is apparent and was a key aspect in the design of the SHARC framework.

Chapter 3

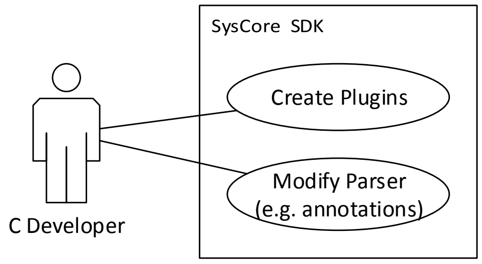
Design

This chapter highlights the use cases of the SHARC framework which was created as part of this work. Due to the extensive nature of the framework these features can easily be extended by the use of the APIs provided as part of this work (as well as via DLLs for the simulation core). Therefore the following sections will also contain use cases for future developers wishing to extend the framework to their specific needs.

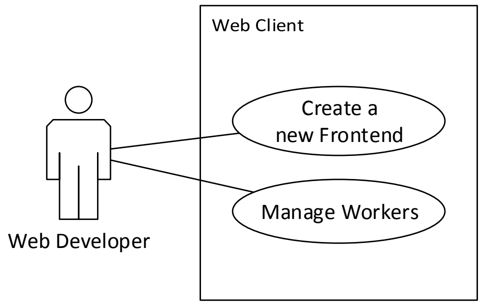
3.1 Workflow



(a) Use Case diagram for a model designer.



(b) Use Case diagram for a C developer.



(c) Use Case diagram for a web developer.

Figure 3.1: UML use cases for the SHARC framework.

The framework created in this work can be seen from the point of view of three actors. Each actor has its own development environment and workflow.

3.1.1 Use Cases

This section contains the full descriptions of the use cases shown in figure 3.1.

3.1.1.1 Use Case: Create Model

Description

This use case describes how a designer creates a model that describes the behavior of a (physical) system with the goal of simulation in mind.

Actors

Model Designer
A person skilled in the modeling of (physical) systems as a set of well-defined, abstract models. Does not need programming skills.

Preconditions

A (physical) system exists which ideally can be broken down into a number of well-defined systems which interact to exhibit the same behavior as the system for the desired aspect.

Basic Flow of Events

Using the Papyrus plugin of SHARC a new model is created. As SHARC uses UML for the modeling approach the individual parts will be models as classes. This can be done with a flat or hierarchical structure (with well-defined ports between the sub models). These classes represent the fundamental components of system and need not necessarily be atomic in their nature (for example a PID controller, a battery and so on). This can be done in a Class Diagram or by creating these classes in the Model Explorer at any desired location.
Then using a Composite Structure Diagram a top class for the system (usually named after the system like vehicle, SoC etc) is created and the models are drawn in from the model explorer to the desired position. Next the connections between the classes are created according to the specification and if necessary signals required for testing and debugging are brought out to the ports on class itself. For an example of such a diagram see fig. 4.16.

Alternative Flow

Using an existing IP library (usually from a previous project) the model is created skipping portions of or the entire first section of the basic flow (design and reuse approach).

Post-conditions

A model has been created that can be given to the C developer in order to create the functionality behind the building blocks or may need refinement into a submodel to enable the reuse of existing IP blocks.

3.1.1.2 Use Case: Generate Testbench

Brief Description

This use case describes how a designer creates uses SHARC to create a testbench for the previously designed model of a (physical) system.

Actors

Model Designer
A person skilled in the modeling of (physical) systems as a set of well-defined, abstract models. Does not need programming skills.

Preconditions

A model of a (physical) system has been created (usually by the model designer himself).

Basic Flow of Events

In order to verify operation of this model a testbench can be generated by the SHARC IDE. The testbench attaches stimuli generators to the inputs which usually use a CSV (comma separated values) format for the stimuli and monitors to any port (input and output) available on the model. These inputs should cause the system to exhibit a known behavior which can additionally be verified with a success validator component (which reads a CSV file containing the expected output of the system ± a margin of error).

Alternative Flow

The model designer may also opt to manually create a testbench for the model in in order to model feedback and/or add other models which are required for the operation of the model under test.

Post-conditions

A testbench for the model has been created that can now be simulated.

3.1.1.3 Use Case: Simulate offline

Brief Description

This use case describes how a designer simulates a testbench (or model) previously created in order to verify operation or generate data.

Actors

Model Designer
A person skilled in the modeling of (physical) systems as a set of well-defined, abstract models. Does not need programming skills.

Preconditions

A model of a (physical) system and a testbench has been created (usually by the model designer himself).

Basic Flow of Events

Using the simulation core shipped with the SHARC IDE the created model is simulated in the testbench. This way proper operation of the model can be verified.

Alternative Flow

The model is embedded in a testbench which is used to extract new data from the model (e.g., to test the expected behavior under conditions that deviate from the original).

Post-conditions

Data for the simulation run is available locally and can be evaluated using the build in Impulse plugin of the SHARC IDE.

3.1.1.4 Use Case: Simulate online

Brief Description

This use case describes how a designer simulates a testbench (or model) previously created in order to generate new data from a verified model (usually using constrained random or a Monte Carlo style approach).

Actors

Model Designer
A person skilled in the modeling of (physical) systems as a set of well-defined, abstract models. Does not need programming skills.

Preconditions

A model of a (physical) system and a testbench has been created (usually by the model designer himself) and typically verified offline using a set of known inputs and outputs.

Basic Flow of Events

The UML files and stimuli are compressed and stored as a so called workpackage. This file is the same for all simulation runs. Additionally a XML file is created for each run containing overrides such as different stimuli and/or modifications to the parameters of the model. These files are submitted to the master (located in the cloud or on site) using the SHARC IDE.

Alternative Flow

A workpackage from a previous run is downloaded from the web-interface and parameters are modified (e.g., higher resolution timestep) and resubmitted.

Post-conditions

Data for the simulation run is available via the web-interface and can be downloaded for evaluation using the build in Impulse plugin of the SHARC IDE or proper behavior is verified using the output of a success validator block (a preview of the trace is available in the web-interface).

3.1.1.5 Use Case: Create Plugin

Brief Description

This use case describes how a C developer creates a SysCore plugin containing a single or multiple IP blocks based on the requirements document provided or developed in cooperation with a model designer.

Actors

C Developer
A person well versed in SystemC AMS (or only SystemC for digital designs) which knows which plugins are already available. This person is also familiar with the structure and interfaces of SysCore and its plugins but not necessarily the internal works of the framework (e.g., the parser).

Preconditions

A requirements document exists which contains the mathematical or functional description of an IP block. The developer has access to the required header files and is familiar with the API of SystemC, SystemC AMS and SysCore. In addition the developer has access to libSysCoreDLL.a which is the export library required for the windows version of SysCore that contains all the required information for linking. Posix systems require no such library.

Basic Flow of Events

Using the *if_sim_obj_plugin* interface a new main.cpp is created within the plugin project (usually based on plugin_template from the provided examples) that implements the functions *getName* which returns the name of the model the plugin belongs to as well as *getObj* which returns a smart pointer (c++11 unique pointer) which points to a new instance of the class contained within the plugin. As the interface inherits from *if_sim_obj* all functions described in it must be implemented as well.

- *getPortByName:* returns a pointer to a port described by the given name (same name as in the model)

- *setParameter:* changes a parameter of the model specified by the given name
- *checkParameters:* checks all parameters for validity (e.g., if the still hold their default/initial value)
- *getInstanceName:* returns the name given to this specific instance (most likely returns the name of the contained SystemC AMS object)

Any number of parameters may be given as a `std::map<std::string,std::string>` to the *getObj* function to be processed by the constructor or changed at runtime by *setParameter*. The top level ports of the model must adhere to the LSF modeling formalism while internally any formalism may be used. Any libraries required by the plugin should also be included (static) as the provided Cygwin environment of SysCore only contains the bare minimum of libraries.

Post-conditions

A plugin that contains the functionality described by the designers requirements document should now be available as a plugin (as *DLL* file for windows machines and *SO* for the workers).

3.1.1.6 Use Case: Modify the Parser

Brief Description

This use case describes how a C developer modifies a core component of the simulation core in order to add features such as custom annotations which are not part of the original design.

Actors

C Developer
A person well versed in C++. This person is familiar with the structure and interfaces of SysCore and the internal works of the framework.

Preconditions

The source core for the simulation core has been checked out from the repository. A toolchain for native C++ development is available (e.g., Cygwin with GCC for Windows) and the development version of all required plugins is installed.

Basic Flow of Events

In order to extend the functionality of the parser either a new step can be added or in an existing one modified. Changes to the parser are only required in rare cases and mostly if new methods of passing values or additional information for the factory are required. An example would be the support of annotations that are which are currently not supported (therefore ignored during parsing). In this case the parse annotations function could be amended to set the required parameters at runtime or if necessary the create model step could be modified to pass the settings from these annotations to the factory at creation.

Post-conditions

The parser is now able to parse different types of annotations or elements as required.

3.1.1.7 Use Case: Create a new Frontend

Brief Description

This use case describes how a web developer creates a custom user interface for the simulation results and/or usage statistics.

Actors

Web Developer
A person well versed a web design language/framework (HTML, Javascript, PHP, ...) and the concept of a webservice (RESTfull, JSON based).

Preconditions

A concept of the fronted and the API specification of the webservice that is hosted by the webserver are available.

Basic Flow of Events

Typically only statistics are required to store data on the webserver the new frontend is running. Everything else can be handles using HTML and Javascript to run the bulk of the interface on the client side. Using these features an interface for a Public view screen could be created that notifies the workers of the current number of simulations and the result of the most recent ones.

Post-conditions

A new frontend is available that can be used with the simulation framework hosted in the cloud or locally as well (even on the same host).

3.1.1.8 Use Case: Manage Workers

Brief Description

This use case describes how a web developer creates a program that monitors the current worker queue or currently available resources in order to modify the number of active workers.

Actors

Web Developer
A person well versed a web design language/framework (HTML, Javascript, PHP, ...) and the concept of a webservice (RESTfull, JSON based).

Preconditions

A suitable control algorithm has been designed and the OpenStack or EC2 API specifications are available (depending on the deployment).

Basic Flow of Events

As the number of available resources can vary as well as the requirements (large bursts during work hours and little load at night or holidays) may require a task that adjusts the number of workers accordingly. For this the current queue is checked using the RabbitMQ API (which is exposed through the framework acting as a proxy) as well as the current number of worker instances. As tasks can be rescheduled easily (due to heartbeat signal even unannounced termination can be handled) no special handling is required (even though the number of fetched workpackages per worker could be checked via the API for larger tasks).

Post-conditions

A automated managements software that controls the number of workers based on a custom criteria is available.

3.2 Initial Concept

Figure 3.2 show how the initial concept for the submission of a model for simulation. The model creation itself follows basic UML creation procedure and will not be covered in this work. For the simulation an UML composite diagram is used, while classes may be defined in any other type of diagram. A set of predefined base classes (such add, gain, integrate etc) are part of the SHARC IDE. Simulation can be executed locally for better debugging and once a workpackage (a set of UML files used for simulation) is ready for submission the user uploads it from the IDE to a webserver using a webservice API. This triggers a number of compute instances in the cloud to perform the simulation and upload the results. For persistent storage a database and a file server are used.

3.3 Underlying technologies

Staring from the initial concept described in section 3.2 a number of existing technologies were evaluated for their viability for the project. This section gibes a brief overview of the technologies chosen as part of the framework or relevant for future work.

This chapter takes information from the respective language specifications as well as [15] and [16] (for SystemC AMS) and explains the basic semantics of the different languages and technologies with a simple example.

3.3.1 SystemC

SystemC is a language specification released as version 1.0 by Accellera (then OSCI) in early 2000. It was accepted by the IEEE as a standard in 2005 (and a later revision in 2011). The language is implemented as a set of libraries and macros for C++ that allow

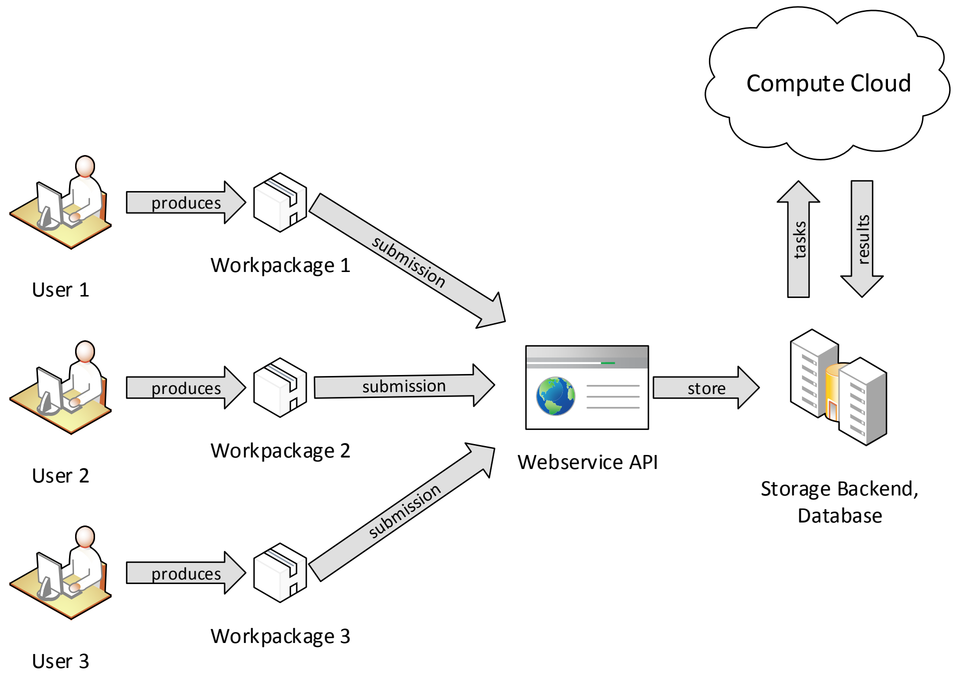


Figure 3.2: Initial concept of the task creation process.

the description of systems that are processes seemingly in parallel. The current version at the time of this writing is version 2.3.1 which was also used to implement the SysCore framework.

3.3.1.1 Key Components

The few core components which are (most likely) part of every SystemC model which will be explained in the following sections by comparing them to their counterpart in hardware description languages (HDL) or object oriented programming (OOP).

Modules

Models are the basic building block of SystemC and are the equivalent of a class in OOP. They can be created via the SC_MODULE macro or simply by declaring a class that inherits from sc_module. As with classes they serve as a container for functions and variables and can be compared to Verilog modules and similar structures in other HDL languages.

Processes

In order for a module to perform any computation SystemC employs processes. Each process is represented by one function within the code which serves as its ”main” function but it may call other functions as well. The main difference to conventional OOP is that those processes run in parallel (do not confuse with conventional threads, this is true parallelism therefore if two or more processes listen to the same event, their result will be committed to the system at same time). Processes usually are sensitive to at least one event (i.e. the system clock) via a sensitivity list. There are two principal types of processes one is declared via SC_THREAD which can wait at arbitrary positions in the code within for an another occurrence of an event in the sensitivity list (via *wait()* statement) or as SC_METHOD which are run to completion type processes which are called (anew) every time an event in the sensitivity list occurs. These processes are similar to Verilogs `always@` blocks and similar constructs in other HDLs.

Ports

In order to communicate between several modules SystemC uses Ports. While conventional in OOP one would merely call a function/method (of a different class) or use some form of IPC, ports are a common concept in HDL like inputs/outputs in Verilog.

Channels

In order to connect two or more ports a channel is used. There are many different types that are part of the SystemC specification like *signal* (basic connection), *buffer* (more advanced version of a signal, creates event on any write but does not store previous values) and *fifo* (improves uppon a buffer by also storing previous values written to it in a first in first out manner) as well as synchronization mechanisms like *mutexes* and *semaphores*. Single channels may be grouped together into a bus which can then be connected as a whole. This concept requires ports and is therefore also foreign to OOP but native to HDL like wires in Verilog.

As for the data types used inside the modules as well for communication via channels either regular C types can be used or sc_int/sc_uint. These integers are similar to the commonly used data types in regular languages except that they can be specified in an arbitrary number of bits which is similar to how wire [n:0] behaves in Verilog.

Events

As previously mentioned SystemC is based upon events (e.g., positive edge of a clock signal) which allow synchronization between blocks/processes executed in parallel. This is also the standard in HDL (like `always@(posedge Clock)` in Verilog) while similar constructs exist in OOP (events, semaphores, mutexes...). The blocks they protect are rarely executed in a truly in parallel way (except for multi processor systems) and can therefore affect each others outcome (e.g., race conditions may occur).

3.3.1.2 Modeling in SystemC

SystemC is well suited to describe hardware components and serve as a golden device or reference for lower level hardware models (like HDL models at register transfer layer RTL). While there is a synthesizable subset of the language it is not a goal of this summary to explain which features are supported. Due to its nature SystemC allows the use of other C++ libraries within its models which helps greatly when modeling. One can even create a cycle accurate model that perfectly represents the RTL (as opposed to SystemC's default transaction level modeling). While SystemC may be used in this fashion this introduction will not detail those features as they are not used in the SysCore framework at all.

3.3.1.3 First Example

It is always a good idea to take a look at a simple ”hello world” example when faced with a new language to gain a grasp of the language’s basic look and feel.

Listing 3.1: inverter.h

```cpp
#include "systemc.h"

SC_MODULE(inverter)              // module declaration
{
public:
  sc_in<bool> in;                // input port
  sc_out<bool> out;              // input port

  void do_something()            // process
  {
    out.write(not in.read());    // output equals the opposite of the input
  }

  SC_CTOR(inverter)              // constructor
  {
    SC_METHOD(do_something);     // register process do_something
    sensitive << in;             // add input in to its sensitivity list
  }
};
```

The simple module described above merely reads one input and writes the opposite value to its output. The important thing is that it does this without delay and whenever the input changes (it does not rely on a clock signal like a latch would). Unfortunately our module does nothing on its own but requires some sort of main. SystemC requires a function with a predetermined name to do so: sc_main.

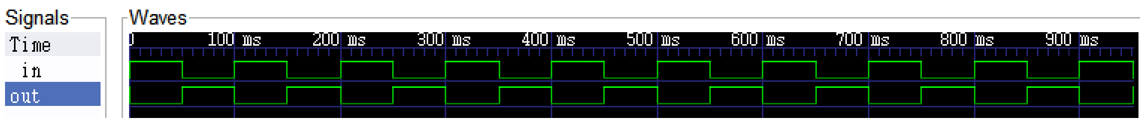


Figure 3.3: Plot of the inverter modeled in SystemC.

Listing 3.2: main.cpp

```
#include "inverter.h"

int sc_main (int argc, char* argv[]) {

  sc_clock clock("clock",100,SC_MS); // create a 100ms period clock signal
  sc_signal<bool> dummy;             // required as all ports must be bound

  inverter inv("Inverter");          // create an instance of inverter

  inv.in(clock);                     // connect our clock to the input
  inv.out(dummy);                    // and the dummy signal to the output

  //create a vcd trace file and add all signals
  sc_trace_file* file = sc_create_vcd_trace_file("trace");
  sc_trace(file,inv.in,"in");
  sc_trace(file,inv.out,"out");

  sc_start(1, SC_SEC);               // run the simulation for 1 second

  sc_close_vcd_trace_file(file);

  return 0;
}
```

When creating the instance a name can be passed to the constructor that hides behind the SC_CTOR macro. If none is given one will be generated internally. This is later used to uniquely identify the instance in question (e.g., if an error occurs with the binding of a module's ports). sc_clock is used as a simple data generator in this case and sc_start is used to run the simulation for the given amount of time. In order to save the output to a file which can later be used to plot the signal (e.g., GTKWave or Impulse) sc_trace is used.

3.3.1.4 Simulation

The advantage of SystemC is the same as with most if not all HDL simulations. The system has little to no regard for the concept of time as the state only changes when an event occurs. The only time computation is necessary for the model above is when the input changes. With the given sc_main this only happens every 100 ms (due to the clock period). This means that only ten or so events need to be processed for the entire simulation. Compared to fixed timestep simulations this usually results in better performance (while in this simple case a timestep of 100ms would result in the same performance of course) and the advantage that every event is processed immediately when it occurs (not at the next timestep when it is ”detected”).

3.3.1.5 Summary

As shown in this section SystemC tries to bridge the gap between HDL and OOP languages. This enables a designer to model both hardware and software components using the same language. While SystemC is commonly used in the context of a system on chip or to model several components (chips) in a system it is usually limited to the digital domain.

3.3.2 SystemC AMS

SystemC AMS is based upon the SystemC simulation core and was released in its 1.0 version in 2010 (while a prior proof of concept was released 2005). For the SysCore framework version 1.0.1 was used. The goal was to allow SystemC models to ”interact” with their analog physical surroundings. The language is mostly used to model embedded analog mixed signal systems such as ADCs or HF systems (radio). On top of the SystemC core SystemC AMS provides a synchronization layer (which is transparent to the user) as well as a scheduler. The most important part is the linear differential algebraic equation solver. It is used to compute the state of the system in a fixed timestep fashion while the synchronization layer takes care of communication (e.g., with discrete event systems of classical SystemC). As the language can be used in many ways three distinct modeling formalisms where created.

3.3.2.1 Timed Data Flow - TDF

TDF is still similar to SystemC itself by only considering signals in a time discrete manner while the value of the signal may have discrete (integer) or continuous (real) values. Modules created in this formalism behave similar to plain SystemC models as they contain a processing function that behaves like a process. However this process is not called upon an external event but periodically (this results in a fixed timestep simulation). To better understand the formalism please consider the following example.

Example

Listing 3.3: sine.h

```
#include <systemc.h>
#include <systemc-ams.h>

SCA_TDF_MODULE( sine ) {
  public:
  sca_tdf::sca_out<double> out; //output port of type double in TDF

  SC_CTOR(sine){                //constructor
  }

  //this function is called for each computation to generate a sine at a
  //frequency of 1kHz and (with values from 0 to 5)
  void processing(){
    out.write( 2.5+2.5*sin(sc_time_stamp().to_seconds()*(1e3*2.*M_PI)));
  }

  void set_attributes(){ //this function is called by the AMS core
    //this line configures SystemC AMS core to solve the equation every 10ns
    out.set_timestep(10.0, sc_core::SC_NS);
  }
};
```

The file above creates a simple sine source which outputs a value every 10ns. This means that the signal really is a stair function (which is hardly visible due to the small timestep). Without the use of multi-rate systems it is not possible to have more than one timestep (one can apply the same on many models) in a single cluster (set of connected modules). For TDF systems SCA_TDF_MODULE has to be used instead of the SC_MODULE of bare SystemC.

Listing 3.4: filter.h

```
#include <systemc.h>
#include <systemc-ams.h>

SCA_TDF_MODULE( filter ) {
public:
  sca_tdf::sca_in<double> in;            //input port of type double
  sca_tdf::sca_out<double> out           //output port of type double

  sca_tdf::sca_ltf_nd ltf;               //laplace tranfser function
  sca_util::sca_vector<double> num, den; //vectors for the equation

  SC_CTOR(filter) {                      //constructor
  }

  void initialize() {                    //called by the AMS core
    num(0) = 1.0;
    den(0) = 1.0;
    den(1) = 1.0/(2.0*M_PI*1.0e3);       //set the cut-off frequency
  }

  void processing()
    out.write(ltf(num, den,in.read()));  //reads in and applies the filter
  }
};
```

The filter above describes a linear time-invariant system with a transfer function in standard Laplace notation.

$$H(s) = \frac{1}{1 + \frac{s}{\omega_c}} \tag{3.1}$$

With ω_c defined as $2\pi \times 2kHz$. The equation describes a first order low-pass filter with a cut-off frequency of 2kHz while our input is 1kHz. Therefore only a little phase shift and next to no attenuation should occur for that frequency.

As with regular SystemC we need a sc_main which starts the simulation but this time two instances of SystemC AMS classes are created (filter and source) and connected via a signal of type double (sig). As with SystemC unbound outputs must be connected to a signal (with one end unconnected).

Listing 3.5: main.cpp

```
#include "filter.h"
#include "sine.h"

int sc_main(int argc, char *argv[]){

  //Signals of type double
  sca_tdf::sca_signal<double> sig;
  sca_tdf::sca_signal<double> dummy;

  filter flt("filter");    //create an instance of the filter
  sine source("source");   //create a sine source

  source.out(sig);         //connect sig to the output of the generator
  lft.in(sig);             //connect sig to the input of the filter
  flt.out(dummy);          //SystemC does not like unbound ports

  //plot input and output of the filter to a file named trace.vcd
  sca_trace_file* file=sca_create_vcd_trace_file("trace.vcd");
  sca_trace(file,flt.in,"in");
  sca_trace(file,flt.out,"out");

  sc_start(5, SC_MS);      //Start teh simulation and run for 5ms

  sca_close_vcd_trace_file(file);


  return(0);
}
```

Summary

While the above example only shows a simple TDF model the main features can be seen. Changing the timestep results in a change of the resolution of the entire simulation and can therefore be used to achieve a trade-off between time and quality. This is one of the main differences between SystemC and SystemC AMS as the former uses events while the later uses a fixed interval to compute changes. With version 2.0 of SystemC AMS (see [17]) variable timesteps will be enabled which allows better control of the above mentioned trade-off. The TDF modeling formalism provides the programmer with a set of functions

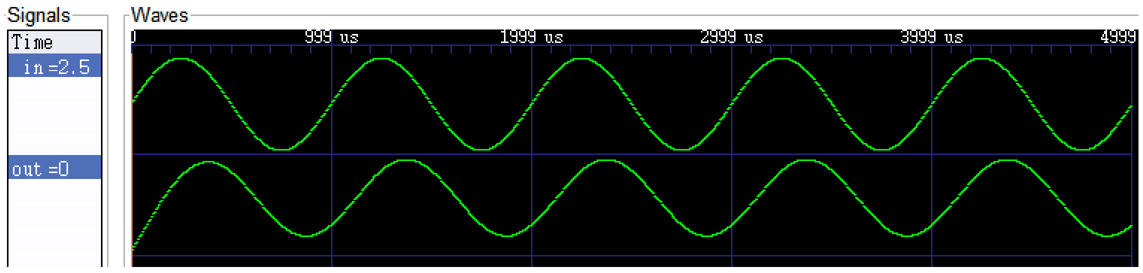


Figure 3.4: Plot of the low-pass filter modeled in SystemC AMS via TDF.

that allow better control over the behavior of the system. In the example above the *initialize()* is used to configure the parameters. While this could be done in the constructor, some features like setting the delay and initial value of the output can only be done here. *set_attributes()* is currently the only function which is allowed to change the timestep of a port. As this is still based upon SystemC interaction between AMS and regular (DE) models is easily possible with the use of `sca_tdf::sca_de::sca_in<[type]>`.

3.3.2.2 Linear Signal Flow - LSF

While TDF still describes a system that computes a result every timestep LSF uses models to describe a set of algebraic equations. All signals are continuous in time and are represented by a real value. LSF has a finite set of predefined models (integrator, adder, source...) and does not allow user models at all. It is however possible to create interactions between TDF and LSF models via `sca_lsf::sca_tdf::sca_source/sink`. One can therefore wrap TDF models in LSF and vice versa. The behavior is similar to Matlab Simulink which also comes with a predefined set of models (far numerous though) and allows the user to create models using encapsulated *.m*-files (while for Simulink other ways extensions are possible). Once again a simple example of a low-pass will server to showcase the formalism.

Example

Notice that there is no code to describe the function of the module merely its internal structure. Due to the nature of LSF all ports and signals are basically double by default.

Listing 3.6: filter.h

```
#include <systemc.h>
#include <systemc-ams.h>

SC_MODULE( filter ) {
public:
  sca_lsf::sca_in in;      //input port always real
  sca_lsf::sca_out out;    //output port always real

  sca_lsf::sca_signal sig; //internal signal

  sca_lsf::sca_dot dot1;   //dot derives the input and multiplies by k
  sca_lsf::sca_sub sub1;   //sub subtracts both inputs scaled by k1 and k2

  //constructor sets k of dot1 and connects components
  SC_CTOR(filter) : dot1("dot1", 1.0/(2.0*M_PI*1.0e3)), sub1("sub1") {

    dot1.x(out);           //connect dot into feedback path
    dot1.y(sig);

    sub1.x1(in);           //subtract the result of dot from the input
    sub1.x2(sig);          //and write to output and dot
    sub1.y(out);
  }
};
```

As with with all other SystemC code a sc_main is required in order to start the simulation. The LSF modeling formalism already contains a source model which generates a sine wave with the given parameters.

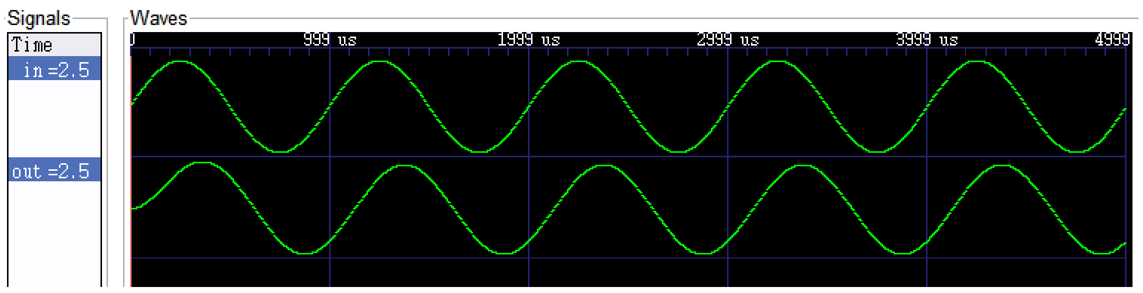


Figure 3.5: Plot of the low-pass filter modeled in SystemC AMS via LSF.

Listing 3.7: main.cpp

```
#include "filter.h"

int sc_main(int argc, char *argv[]){

  //signals
  sca_lsf::sca_signal sig;
  sca_lsf::sca_signal dummy;

  //predefined sine source and filter
  filter flt("filter");
  sca_lsf::sca_source source("source",0,2.5,2.5,1.0e3);

  //generates points at given timesteps
  source.set_timestep(10.0, sc_core::SC_NS);

  //connect
  source.y(sig);
  flt.in(sig);
  flt.out(dummy);

  //plot results to file
  sca_trace_file* file=sca_create_vcd_trace_file("trace.vcd");
  sca_trace(file,flt.in,"in");
  sca_trace(file,flt.out,"out");

  sc_start(5, SC_MS);

  sca_close_vcd_trace_file(file);

  return(0);
}
```

Summary

With the LSF modeling formalism it is not required or possible to specify the behavior of the system by any other means than by connecting the predefined models together. Unlike Matlab Simulink only the most basic functions are supported. LSF however comes with a set of common blocks like an integrator, differentiator, a two port add/subtract and even delays to model temporal behavior. For the model described above a block diagram would look like the one shown in fig. 3.6.

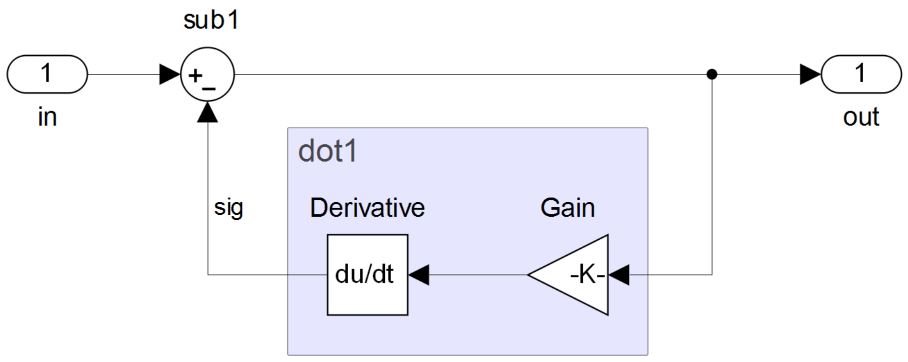


Figure 3.6: Block diagram of a low-pass filter as modeled in LSF.

3.3.2.3 Electrical Linear Networks - ELN

The third and last modeling formalism is called ELN and is best used for the description of electrical systems. While by far not as powerful as SPICE or similar simulators it is possible to include simple electrical networks into the simulation. As with LSF the solver uses a set of algebraic equations to solve the network but for ELN Kirchhoff's law is used to describe how current and voltage behave. ELN signals can also be converted to TDF and DE signals and vice versa. There is however no way to convert ELN directly into LSF. ELN is also the only formalism that has no concept of input and output ports, merely terminals which are basically a voltage/current input and output port at the same time. Internally `sca_eln::sca_node` are used to connect signals with each other while `sca_eln::sca_node_ref` represents ground (common). As with the other formalisms an example should help to illustrate the basic structure of an ELN model.

Example

The way ELN is described is similar to LSF but instead of mathematical constructs like integrators, differentiator etc. only basic electrical components like resistors and capacitors (as well as a controlled voltage and current sources) can be used to create a model. Figure 3.7 shows the modeled system as an electrical circuit.

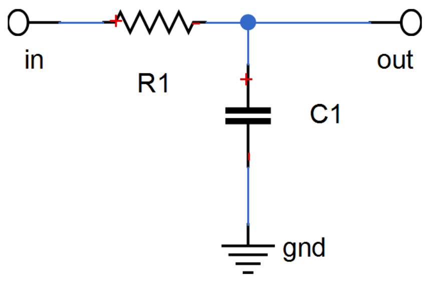


Figure 3.7: Electrical diagram of a low-pass filter as modeled in ELN.

Listing 3.8: filter.h

```
#include <systemc.h>
#include <systemc-ams.h>

SC_MODULE( filter ) {
public:
  sca_eln::sca_terminal in, out; //external terminals
  sca_eln::sca_node n_in, n_out; //internal nodes
  sca_eln::sca_node_ref gnd;     //gnd reference

  sca_eln::sca_r *r1;            //resistor
  sca_eln::sca_c *c1;            //capacitor

  SC_CTOR(filter)
  {
    r1 = new sca_eln::sca_r("r1",1e3/(2*M_PI));
    c1 = new sca_eln::sca_c("c1",1e-6,1e-6*2.5);//Q=C*U

    r1->p(in);
    r1->n(out);
    c1->p(out);
    c1->n(gnd);
    in(n_in);
    out(n_out);
  }
};
```

As usual a file containing a sc_main is used to start the simulation and capture the results. As with LSF we use a predefined ELN source to create our input signal.

Listing 3.9: main.cpp

```
#include "filter.h"

int sc_main(int argc, char *argv[]){

  filter flt("filter");
  sca_eln::sca_vsource vs("vs",0,2.5,2.5,1.0e3);
  vs.set_timestep(10.0, sc_core::SC_NS);
  vs.n(flt.gnd);
  vs.p(flt.in);

  //plot results to file
  sca_trace_file* file=sca_create_vcd_trace_file("trace.vcd");
  sca_trace(file,flt.in,"in");
  sca_trace(file,flt.out,"out");

  sc_start(5, SC_MS);

  sca_close_vcd_trace_file(file);

  return(0);
}
```

Summary

While ELN is by far the most specialized modeling formalism it clearly has its uses. Even though only the most basic of blocks exist it can be extended by modeling the behavior of

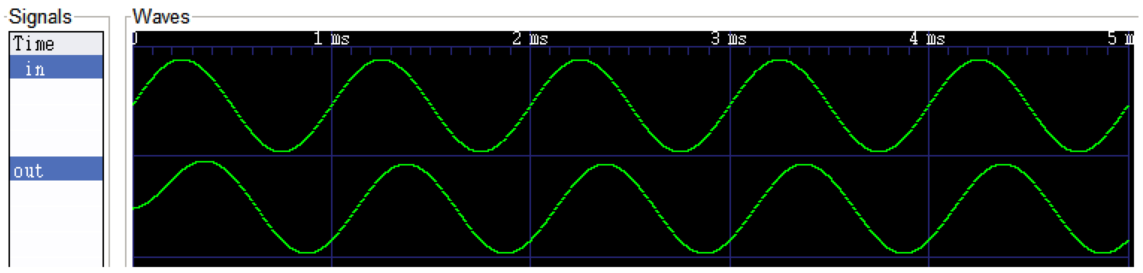


Figure 3.8: Plot of the low-pass filter modeled in SystemC AMS via ELN.

a complex part into a TDF model and encapsulate it into an ELN block. This formalism is currently not used by the SysCore framework due to its specialized nature and the lack of any electrical networks within the simulation models required by the case study.

3.3.2.4 Summary

While the SystemC AMS core handles each formalism different it is easy to create connections between two or more formalisms as well as integrate conventional SystemC models (DE) into the system. While unlike other AMS languages it is not possible to simply describe a model via an equation one can most likely model said equation using a LSF block diagram. While TDF is limited to fixed step simulation in SystemC AMS version 1.0 as of version 2.0 (see [17]) variable/dynamic steps have been included and are implemented in the current prove of concept by Fraunhofer IIS. Considering the nature of the original model (Matlab Simulink) it should by clear why SystemC AMS was chosen over plain SystemC.

3.3.3 XML

XML is short for Extensible Markup Language and was first formalized in W3Cs XML 1.0 Recommendation [18] in 1998 with the current version being 1.1 (published 2004). While XML files are written in a plain text language they are not meant to be read by humans but are intended to be parsed/interpreted by machines. With the help of XSLT it is possible to convert them into human readable form (HTTP/PDF). However this short introduction will only focus on the machine readable nature of these documents.

3.3.3.1 Structure

All XML documents are structured as a tree with one distinct root. There are no limitations in the number of children per level or the depth of the structure. The XML specification allows for two distinct methods to encode information into this tree structure:

- **Elements:** As with regular children one may simply put the desired information in a child of its own and name it accordingly (say a child named pi and a content of 3.14).

- **Attributes:** Another way of encoding information into a XML file is via attributes. One may simply create a child and in addition to the name and value of the node specify any number of additional attributes (for our previous example one may have a child named pi with an attribute value set to 3.14 and use the content of the child for something else or omit it entirely).

It is usually up to the designer to use either formalism. As with the previous tutorials for this section a short example will be used to highlight the main features of the language.

3.3.3.2 Example

Listing 3.10: sample.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<conf:attendants conf:version="9001" xmlns:conf="http://url/conf/9001">
  <person sex="male">
    <name>hank</name>
    <description>some dude named hank</description>
  </person>
  <person sex="female">
    <name>katherine</name>
    <description>some girl named katherine</description>
  </person>
</attendants>
```

It is important that each node has a begin <node> and an end </node> tag. For the sake of simplicity only one namespace is used (for the root element) while multiple namespaces (or the same) could have been chosen for the children and their elements/attributes as well. A schema could be used as well to describe the basic context of a node as well as what elements and attributes are required and what values they may have. This may be verified by the parser to ensure a valid document. For the sake of simplicity this kind of information is not used for this simple example. As mentioned earlier which properties are modeled as attributes and which are modeled as elements is up to the designer (one could have modeled sex as an element as well or the name as an attribute).

3.3.3.3 Parser

As mentioned in the beginning the main purpose of XML is being read and interpreted by a machine. A Program fulfilling these functions is called parser. While there are many distinctions between parsers (validation, codesize, language, ...) there are two main classes.

SAX Parser

SAX is short for Simple API for XML and describes a type of parser that parses a XML document sequentially. While there is no formal specification for SAX parsers they typically allow the programmer to register a callback function for different events such as the beginning and end of an element, text nodes or comments. As there is no state required in the parser they are usually smaller in code size and are easier to implement but they usually lack the ability to verify a document. These types of parsers generally come with a reduced set of features (no XPATH and XSLT support). Due to the nature of the documents created by Papyrus a DOM parser was chosen for the SysCore framework.

DOM Parser

Unlike SAX parsers a DOM parser (short for Document Object Model) creates a complete representation of a XML document in memory. Therefore any object may be accessed at any time which allows for easy validation as well as searches using XPATH or the translation of that representation into a different format (XSLT). While one could create a DOM parser from a SAX parser (updating the internal representation on each event) it is usually easier to just use a DOM parser to begin with. For the SysCore framework libxml (which is part of the Gnome project) with a wrapper for C++ (libxml++) was used. While the UML files generated with Papyrus comply with xml 1.0 they have a very distinct structure not unlike that of a database (with `xmi:id` as the primary/foreign key)

3.3.3.4 UML

Depending on the amount and type of profiles the root node of these documents may differ. For a simple UML model the root node is `uml:Model` while profiles like MARTE GCM add nodes beside the `uml:Model` requiring a new root called `xmi:XMI`. For the purpose of this tutorial a very simple UML model split into a library and an implementation with no profile applied will be used. Figure 3.9 shows the composite structure diagram which shows the connection between the classes.

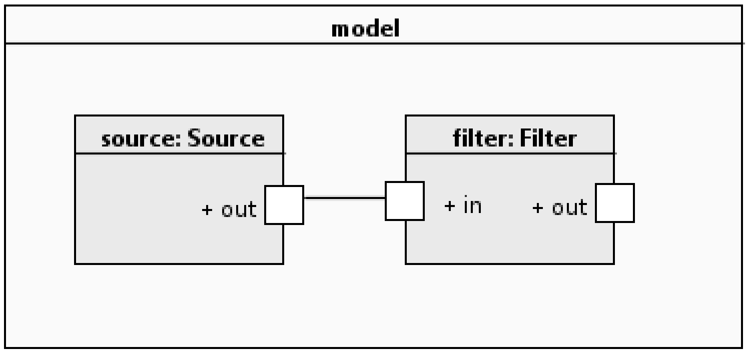


Figure 3.9: UML diagram of a filter system with attached source.

Listing 3.11: library.uml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="_0mYgACkyEeWMw-x-ManwQA" name="Library">
  <packagedElement xmi:type="uml:Class" xmi:id="_5W4akCkyEeWMw-x-ManwQA" name="Filter">
    <ownedAttribute xmi:type="uml:Port" xmi:id="__Iz88CkyEeWMw-x-ManwQA" name="in"/>
    <ownedAttribute xmi:type="uml:Port" xmi:id="_AMbWECkzEeWMw-x-ManwQA" name="out"/>
  </packagedElement>
</uml:Model>
```

As there are no profiles involved in this simple model the `uml:Model` is used as the root node. The library contains the actual class of the filter used in the model and describes the attributes such as ports and properties (not used in this example). The type element here is always an attribute that only has two values: `uml:Port` and `uml:Class`. Every element also has an unique `xmi:id` which will become important later on and a `name` (must be unique in the namespace otherwise the verification will fail in Papyrus). The structure seen here is always the same with *packagedElement* being usually a class an *ownedAttribute* describing some property that belongs to that class (while only one exists in this case, multiple *packagedElements* and *ownedAttributes* are possible). The library contains only the filter therefore it can be compared with the previous *filter.h* files.

Listing 3.12: model.uml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="_xPVrsCkyEeWMw-x-ManwQA" name="Model">
  <packagedElement xmi:type="uml:Class" xmi:id="_Pv9VYCkzEeWMw-x-ManwQA" name="model">
    <ownedAttribute xmi:type="uml:Property" xmi:id="_SmKtgCkzEeWMw-x-ManwQA" name="filter">
      <type xmi:type="uml:Class" href="library.uml#_5W4akCkyEeWMw-x-ManwQA"/>
    </ownedAttribute>
    <ownedAttribute xmi:type="uml:Property" xmi:id="_XtVz8Ck5EeWMw-x-ManwQA" name="source" type="_VxqKYCk5EeWMw-x-ManwQA"/>
    <ownedConnector xmi:type="uml:Connector" xmi:id="_fN-T0Ck5EeWMw-x-ManwQA" name="Connector1">
      <end xmi:type="uml:ConnectorEnd" xmi:id="_fN-64Ck5EeWMw-x-ManwQA" partWithPort="_XtVz8Ck5EeWMw-x-ManwQA" role="_VxqKYSk5EeWMw-x-ManwQA"/>
      <end xmi:type="uml:ConnectorEnd" xmi:id="_fN-64Sk5EeWMw-x-ManwQA" partWithPort="_SmKtgCkzEeWMw-x-ManwQA">
        <role xmi:type="uml:Port" href="library.uml#__Iz88CkyEeWMw-x-ManwQA"/>
      </end>
    </ownedConnector>
  </packagedElement>
  <packagedElement xmi:type="uml:Class" xmi:id="_VxqKYCk5EeWMw-x-ManwQA" name="Source">
    <ownedAttribute xmi:type="uml:Port" xmi:id="_VxqKYSk5EeWMw-x-ManwQA" name="out"/>
  </packagedElement>
</uml:Model>
```

The basic structure is the same as the *library.uml* but this time the one class (the model) described in a *packagedElement* has properties. Overall *model.uml* contains two classes, one is basically the top model which describes the overall architecture of the system and the other is a source for that generates the input for the filter. This equals the information that was previously present in the *main.cpp* files. A key feature of the UML format is the ability to create links from one file to another. In this example the filter is the one described by the *library.uml* therefore the `type` that was previously an attribute becomes an element of its own. This new element contains not only the type but also an attribute called `href` which is composed of the name of the file (to which the link points) followed by a # followed by the `xmi:id` of the target in that file. The other attribute of the *packagedElement* model is defined within the same file and therefore as a second *packagedElement* beside the model. The type of the *ownedAttribute* is also `uml:Property` but this time it is merely an attribute of the node with the `type` only containing the `xmi:id` of the target. The UML structure is also a good example for the need of namespaces as each *ownedAttribute* has basically two types, one is the `xmi:type` which describes the nature of the element (`uml:Property`) the other is simply called `type` and describes the type of class the property references (the `xmi:id` of the target). Either way the *ownedAttribute* does not name the ports it has as this information is only contain in the class description.

Other than the *ownedAttribute* the model also has *ownedConnectors* which link the instances of the classes together. Each *ownedConnector* has exactly two children called *end* of the `xmi:type` `uml:ConnectorEnd`. As with the `type` of properties the `role` of each end is either an attribute (for ports on local targets) or an element for ports on linked parts. The `href` on the element follows the same rules as the type of the property. Even though not modeled in this example if a port is connected to the class that contains the connector (for example if the output of the filter was connected an output on model) than no `partWithPort` is specified.

In order to keep this example reasonable small no profiles were used as their exact syntax depends on the profile in question. Basically every node follows similar rules and if for example `GCM:flowports` is applied to a port a new child of `xmi:XMI` is created and the element is linked via the attribute `base\_port` that specifies the `xmi:id` of that port. While the exact name of the attribute or element that links these elements differs it can be easily found by looking at the source code of the UML.

3.3.4 Parallel Execution

There are two distinct scenarios for which a distributed environment can be harnessed for speedup. First is the distribution of a single program on multiple Machines or several programs that work together to complete a single task. For this purpose frameworks such as Open MPI (Open Message Passing Interface) as well as more high-level frameworks such as FMI (Functional Mock-up Interface) were created over time. These frameworks usually rely on a fixed set of machines that possess a relative high availability as the loss of a single machine results in the failure of the entire system. This also means that lossy networks (such as the internet) are not well suited for this task. With the emergence of the internet and cloud services a new paradigm was required which focuses on the second scenario of parallel execution: the parallel execution of individual task (with or without dependency). This second type of parallel computing for example is used for web servers when a single machine is incapable of handling all requests. In this case a load balancer divides the requests among all the available web servers and as all requests are independent from one another there is no need for direct communication or synchronization among the servers. This also means that if a single machine fails only some requests fail, but as this is sometimes not a valid option better solutions have been developed as well. For the purpose of cloud computing a set of design pattern were introduced as enterprise integration patterns which are implemented mostly in AMQP (Advanced Message Queuing Protocol).

3.3.4.1 Open MPI

While there are MPI language specification implementations in most programming languages for this section the C++ bindings of Open MPI will be evaluated. As SystemC as well as SystemC AMS are C++ libraries as well this was the most relevant scenario for this work. An interesting aspect of Open MPI (and MPI in general) is that the user only writes a single program which at runtime distinguishes which role each specific instance should fulfill. This mapping is done beforehand by the programmer and is static for this task (in the form of a hostfile passed at program launch). It is possible to use a scheduler instead which allows for a more dynamic behavior. As for the hostfile: the programmer has to take care of this mapping this can get complicated for large systems such as a cluster of computers. The language does support concepts like rendezvous so a queue and with the help of a scheduler a more or less dynamic system could be implemented but the language itself is more suited for divide and conquer algorithms where a single problem is broken down into (a predefined amount of) smaller parts which are computed individually and later merged into the solution.

Summary

While it would be possible to use a task scheduler (such as SLURM) with software written using Open MPI there is no failure tolerance or recovery build into MPI. Due to the nature of the deployment in a cloud (especially a public cloud) Open MPI would not be feasible for the project in its current form. It would however be possible to exploit the distributed nature to bypass SystemC's own inability to handle threads and multiple processes in order to parallelize SysCore on a single machine using the loopback interface. This would mean that the current worker queue would still schedule simulations to the individual workers which in turn would distribute the SystemC simulation core to all available threads. This could be useful for situations where a few large simulations are required instead of the current scenario where a multitude of simulations with varying parameters (such as Monte Carlo style or constrained random simulations) is the expected workload.

3.3.4.2 FMI

The Functional Mock-up Interface standard (FMI) has been developed to facilitate model exchange and co-simulation using an independent tool. Tools following the FMI standard are able to create models that are packed into a single *.fmu* file which can then be distributed. These files typically contain the following files:

- **A XML descriptor:** containing information about the type of the model (co-simulation and/or model exchange) as well as knowledge internal variables and the models inputs and outputs.
- **A model as DLL:** compiled from C code for a specific architecture and operating system. It follows a well-defined interface specified in the standard.
- ***Optional:*** a Functional Mock-up Unit (FMU) can also contain the source code that was used to create the DLL as well as human readable documentation.

While this standard could be incorporated into the SysCore simulation core itself in order to allow for model exchange (extending the current plugin system or even replacing it altogether) as well as co-simulation with either other tools (the list of supported simulation around 80 tools with varying support for FMI) or other instances of the SysCore simulation core in order to bypass SystemC's limitation on threading (currently the SystemC core does not allow for proper threaded execution).

Design

As mentioned before FMI as a standard consists of two main components (model exchange and co-simulation) that can solve different problems for tool vendors. These two frameworks can be used independent of one another and while sharing some common code can be considered two independent frameworks.

The main advantage of FMI is that vendors can focus on one part of a problem where they excel while relying on other software to solve part of the problem within their field of expertise. Using such an approach discrete event models (written e.g., in VHDL running in ModelSim) and time continuous models (e.g., as differential equations modeled in Matlab) can work together to model the vastly different aspects required of the simulation of cyber physical systems. While neither framework (in this example ModelSim and Matlab) is particularly well suited for the whole system together they form composite system perfectly suited for this task.

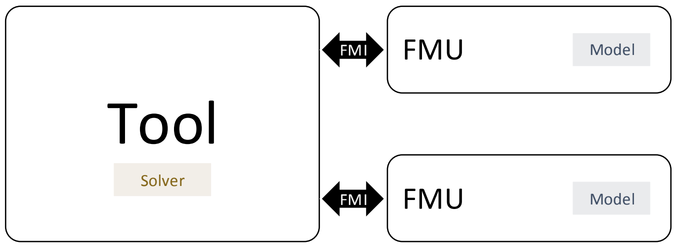


Figure 3.10: Block diagram of a system using FMI for model exchange.

Model Exchange Using the model exchange framework tool vendors can export a model (typically as a set of c-functions) without an enclosed solver. It is also possible to only share a precompiled version of the module (binary) which allows the vendor to share models with the simulation framework of other vendors while keeping the intellectual property within the models secret. The core of a model usually consists of a set of differential and/or algebraic equations with time-, state- and step-events.

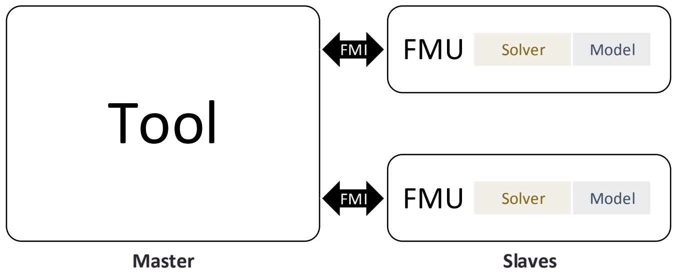


Figure 3.11: Block diagram of a system using FMI for model co-simulation.

Co-Simulation Using the co-simulation framework tool vendors can use standardized interfaces to connect with software from other tool vendors. The framework makes a distinction between master and slave subsystems. While there can be only one master (which coordinates the modules and provides the communication channel behind the interfaces described in the specification) any number of slave subsystems may exist within a system. The standard defines interfaces which allow information exchange between the models at discrete communication points only. The slave subsystems may be standalone solutions (for example binaries or code created via Matlab Simulink's code export feature) or still running within the tool they were developed. The Co-Simulation specification does not include any information on how data is actually exchanged between processes. Figure 3.12 show the simplest possible example of a master that loads the DLL file from the FMU into memory. The master then calls the functions exposed by the FMU in order to perform actual the simulation. Note that the master does not have a solver of its own but only advances the simulation time.

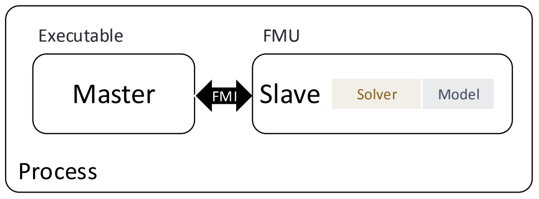


Figure 3.12: Block diagram of a simple master application using FMI for co-simulation.

If the model is for example required to run within the tool it was created (which usually comes with its own executable, user interface etc. a wrapper is required. This wrapper exposes FMI on one end and maps to a set of (usually proprietary) inter process communication functions on the other side. This means that the FMU does not really include any functionality in this example but is only used to communicate with a second piece of software which performs the simulation (containing the solver and model). A block diagram of an application following this pattern is shown in fig 3.13.

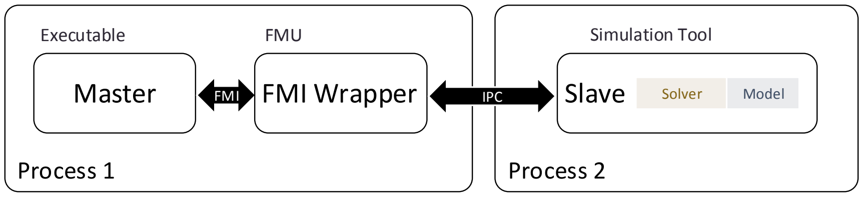


Figure 3.13: Block diagram of a master application using a FMI for co-simulation with another program.

The most interesting aspect for the goal of cloud computing is the distributed setup. Here the master sits on one computer controlling any number of slaves. Figure 3.14 shows such a setup with only one slave (for simplicity). The master from the last example has been replaced by an application server that exposes the FMI interface to the network. Any number of network protocol and frameworks could be used to establish communication between the two computers). As the state of the simulation is of course stored within the simulation tool the loss of a slave usually results in a system wide failure. Therefore this is well suited for high performance computing (HPC) but less so for cloud applications where the availability of a machine is not guaranteed.

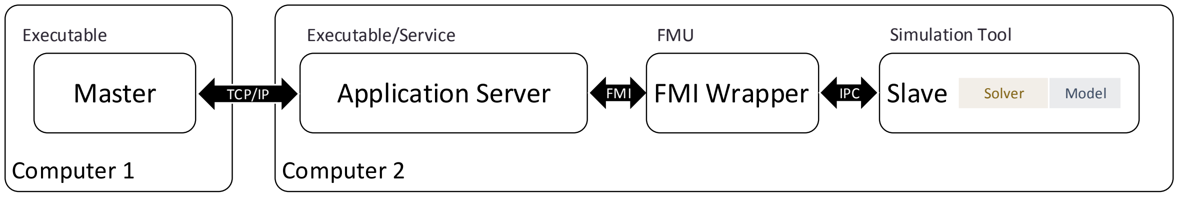


Figure 3.14: Block diagram of a master application using a FMI for co-simulation with another machine.

Summary

Aside from the advantage of having the ability to use existing simulation tools (e.g., Matlab to model the environment of the car) the extension of the current plugin system to FMI would allow better description of the variables and parameters of the models within those plugins. The best approach would be to update the current Simulation Core of SysCore to act as the FMI Master while the plugins (already in binary form) would act as slaves. Unfortunately this would mean that no equations could span more than one model and that the slowest module would determine the speed at which the entire model can be simulated (as all models have to finish computation before the timestep can advance). The second disadvantage that became apparent during the analysis was that the values needed casting/conversion at each interface if the models would be running on different nodes within a cluster (for transmission). Aside from the latency of the network communication this would be a major performance impact on the simulation. Therefore a model could not reasonably span among different machines. Unfortunately these two facts combined mean that FMI is not a suitable solution for the task of a constrained random simulation as a massive number of simulations need to be run which have the advantage of being independent of each other.

3.3.4.3 Advanced Message Queuing Protocol - AMQP

The Advanced Message Queuing Protocol is an open standard used for message-oriented middle-ware. Most of the implemented patterns are described in Enterprise Integration Pattern and can be summarized by the examples from the RabbitMQ website (the following examples will be using Pika and Python). The function of the messaging pattern is to allow for a many to many connection where only one address has to be known to all machines. This machine accepts and distributes messages, takes care of message persistence and can optionally monitor all consumers using a heartbeat signal. Retransmission is also build into this message broker. RabbitMQ was chosen as the implementation of choice due to the number of features that come out of the box including cluster fail-over (allowing high availability systems) and high performance. While brokerless MQ solutions like ZeroMQ exist it is typically easier due to the fact that only a single public IP address is required when using a broker.

Summary

Using the patterns available in AMQP (described by the examples in section A.3) a simple worker queue can be implemented which uses Topics or a routing based Exchange for status updates. For consumers of such updates would be a logging facility which writes the logs to a database for inspection and additionally send out an email to the administrator on messages declared as critical or error. Using such a system allows for easy adaptations such as publishing to a jobs topic under starting when a worker starts a task and finish upon completion. These messages could be used for pop-ups in a web-interface as well as a desktop client used for submitting tasks.

3.4 Public vs. Private Cloud

Overall there are two main approaches to cloud computing:

- **Public Cloud:** All machines and the management and maintenance of the underlying machines is done by a third party. The user typically gets access to a web-interface and pays per resource. Examples for this are Amazons Web Services or IBM's Bluemix.
- **Private Cloud:** All machines are kept in house and a local administrator takes care of maintenance and management. While solutions like OpenStack offer a similar web-interface it is not always common for the end user of the cloud to get access. The advantage is that there is no per resource cost but the initial purchase of the machines and infrastructure usually amount to a large onetime payment followed by moderate upkeep costs (usually power, support contracts with vendors, licenses etc.).

The advantage of a public cloud is that only resources that are currently in use need to be paid and machines and storage can be added or removed on demand. The advantage of the private cloud is that machines can be configured to serve the individual needs better and all data can stay in house. In case of large amounts of data a locally hosted cloud usually has the advantage of higher speeds as the data does not need to be transferred via an internet connection. Neither solution is better than the other for all purposes and hybrid solutions may also be worth considering (having some machines in house while adding machines from an Infrastructure as a service provided when needed).

3.4.1 AWS - Amazon Web Services

While there are many competitors (Microsoft, Google, VMware, IBM, Citrix, Rackspace, Amazon) in the field of cloud computing there is one clear leader according to Gartner : "Amazon's cloud is 10x bigger than its next 14 competitors, combined" (the full report is available at [19]) and similar quotes clearly show the dominance of Amazon in this field. For this work not all Amazon services are of relevance but the key services (for the current state of SysCore) will be briefly described.

3.4.1.1 EC2 - Elastic Compute Cloud

This service falls into the category of IaaS (infrastructure as a service) and provides the customer with virtual machines of a set of predefined configurations (so called flavors). There are several groups for flavors but the most interesting ones for SysCore are:

- **T2 General Purpose:** These instances have a good baseline performance and are able to use so called CPU bursts in order to temporarily increase the performance. These bursts consume so called CPU Credits which T2 instances generate when idle.
- **M3/M4 General Purpose:** Similar to the T2 series these instances offer a good balance. Each type has its own specialization though M3 instances offer few vCPUs with fast I/O while M4 offer instances with up to 40 vCPUs. Unlike T2 instances there is no burst though.
- **C3/C4 Compute Optimized:** These instances are specifically build for compute tasks and offer cluster configurations for networks. Similar to the M4 group these machines also come with their own SSDs per instance optimized for throughput.

Each of this groups has aforementioned flavors of their own allowing the customer to choose the right one for the task. While the C series would offer the benefit of cluster configurations the goal of SysCore is to operate on general purpose machines with no specific network setup (as long as they can reach the master). Therefore the T2 instances with their burst capabilities have the best price/performance for SysCore (while any type may be used of course). Table 3.1 shows the flavors available at the time of this writing. the nano and micro flavors are specifically well suited for the task as only a few megabytes of RAM are required by the simulation framework and a single worker usually saturates a single CPU (as SystemC lacks proper threading support). For the master a medium or large flavor would be beneficial as there are many components that each require considerable resources (CPU and RAM) .

3.4.1.2 S3 - Simple Storage Service

Amazon S3 offers object storage in the cloud using a RESTfull API and a SDK for many common languages for web development. Due to its build in redundancy Amazon guarantees as much as 99.999999999% availability for objects stored using their service. This

Table 3.1: Amazon EC2 T2 flavors.

Model	**vCPU**	**CPU Credits / hour**	**Mem (GiB)**	**Storage**
t2.nano	1	3	0.5	EBS-Only
t2.micro	1	6	1	EBS-Only
t2.small	1	12	2	EBS-Only
t2.medium	2	24	4	EBS-Only
t2.large	2	36	8	EBS-Only

kind of storage is well suited for the log files, workpackages (UML files required for the simulation), configurations and the resulting trace files. As there is a Python SDK for S3 it would serve well as storage back-end for SysCore once deployed using AWS.

3.4.1.3 RDS - Relational Database Service

While one can simply set up a SQL database on the master and maintain redundancy by replication with a failover master Amazon already offers a redundant database solution (with common SQL engines such as MySQL, MariaDB, PostgeSQL and many more) with multi-availability zone replication (e.g., one replica in Europe and one in the US).

3.4.1.4 SQS - Simple Queue Service

The backbone of the framework is a durable message queue from which each worker consumes messages in order to perform simulations. Amazon offers their own solution for this as well and while RabbitMQ and other AMQP implementations usually provide cluster fail-over as an option Amazons service would be a viable solution due to its high availability as well (while requiring no manual user configuration).

3.4.1.5 EB - Elastic Beanstalk

While deploying a web application (especially with modern web frameworks such as Flask) is relatively straight forward, maintaining and monitoring the instances that run the application still requires time and resources. Amazon EB allows for applications to be deployed to the aforementioned services without knowledge of the underlying infrastructure.

> With Elastic Beanstalk, you can quickly deploy and manage applications in the AWS cloud without worrying about the infrastructure that runs those applications. AWS Elastic Beanstalk reduces management complexity without restricting choice or control. You simply upload your application, and Elastic Beanstalk automatically handles the details of capacity provisioning, load balancing, scaling, and application health monitoring. Elastic Beanstalk uses highly reliable and scalable services
>
> *AWS Elastic Beanstalk, Developer Guide (API Version 2010-12-01)*

This basically means that EB takes care of most of the management tasks for the customer reducing the work for the administrator drastically.

3.4.2 OpenStack

The OpenStack project began 2010 as a joint venture of Rackspace hosting and NASA and has since grown to one of the -if not the largest- open source Infrastructure as a Service solutions. At its core OpenStack is split into sub-projects projects for each individual task which like a Linux distribution get released together under a common release name and number. OpenStack closely follows Ubuntu in their release cycle with new versions usually getting released in April and October. Most components are written in Python and the software is therefore cross platform by nature and many Linux distributions come with support for easy deployment of OpenStack. Examples for this are Ubuntu's JuJu or Redhats RDO which allow "one click" deployments of a cloud. While JuJu is meant for a deployment using multiple machines RDO offers support for an All-In-One setup. The following sections will describe the core components for an OpenStack installation that are usually present on any installation (with some exceptions). There are however far more components that are part of OpenStack than the ones listed here.

3.4.2.1 Compute: Nova

Similar to Amazons EC2 Nova offers the ability to manage virtual machines in a cloud. Nova also has machines that come in flavors and can be configured to support the same API as EC2 allowing users to seamlessly switch between the two. It is important to note that Nova is not a hypervisor itself but manages others. At the time of this writing (with Liberty being the current release) the following hypervisors are supported:

KVM

Kernel-based virtual machine is part of Linux since 2.6.20 and serves as an out of the box hypervisors for Linux machines. KVM however requires hardware virtualization extensions such as Intel VT or AMD-V. It is also available on ARM architectures with some Cortex CPUs (such as A7, A15, A17) which come with virtualization extensions. KVM is also known as Qemu KVM as it uses many of the tools used by Qemu and is more or less an extensions of Qemu, not a replacement. OpenStack uses Libvirt to manage KVM.

Qemu

Qemu itself does not need virtualization extensions but is considerably slower than KVM. Due to the fact that most hardware these days comes with these extensions Qemu without KVM is rarely used. OpenStack uses Libvirt to manage Qemu.

Xen

Similar to KVM Xen is part of the Linux kernel. Since Linux 3.0 Xen can be used for Dom0 (hypervisor) and DomU (guest). Xen does not need virtualization extensions and can even run paravirtualized guests (as long as the guest OS supports Xen). Xen itself runs as a micro-kernel before Linux and therefore resources allocated to a DomU are no longer visible or available to the Dom0. OpenStack can use libvirt to manage Xen and Citrix's XenServer can also be used but the features differ from using pure Xen with libvirt.

Hyper-V

Due to the cross platform nature of OpenStack Microsoft's Hyper-V can also be used as hypervisor. It is growing in popularity due to the fact that is now available under a free license. However not all features of OpenStack are fully supported yet.

VMware ESXi

While not on par with KVM in terms of support in OpenStack VMware is widely used as a virtualization solution and has growing support in OpenStack.

3.4.2.2 Networking: Neutron

While not typically visible to the user Amazons Elastic IPs and the cluster and advanced network configurations of the C flavor VMs indicate that there is a similar (while officially unnamed) construct running on Amazons server. Neutron manages all network related actions such as DHCP and DNS for private networks to the VLANs and Bridges used internally for the network. Neutron can be configured using many plugins but under Liberty the documentation recommends using a typical Linux Bridge. Bridges combine one or more network interfaces (physical or otherwise) into a single network interface. Machines running neutron usually have 2 NICs (network interface controllers) in order to separate the public interface (which is connected to the internet) from a private interface (which allow access to the APIs of the components). Neutron can create virtual networks which are not associated with any NIC as well as routers which then route all traffic from a virtual network to another or a network associated with a NIC (such as the public interface). These virtual networks can span any number physical machines using VLANs and similar technologies already part of the Linux kernel.

3.4.2.3 Block Storage: Cinder

While Cinder is comparable to Amazons Elastic Block Storage there are some differences. Mainly Cinder is used to create disks for virtual machines that allow the user to have persistent storage for those machines. This is not mandatory but makes sense for nearly all virtual machines. Unlike Amazons solution a single block device can only be mapped to a single virtual machine at a time. Cinder does allow for snapshots of machines which can then be used to spawn new virtual machines. There is an option to use EC2 commands to control cinder as well which allows user to keep using their existing EC2 scripts for OpenStack.

3.4.2.4 Identity and access management: Keystone

Keystone offers authentication services for OpenStack. It is mandatory for all other components to register as service in Keystone otherwise they cannot work together. Keystone also includes the user and project management as well permission levels (such as user or administrator). Keystone can also integrate with existing identity frameworks such as LDAP (lightweight directory access protocol, which is also compatible with Microsoft's Active Directory).

3.4.2.5 Dashboard: Horizon

Horizon serves as the web-interface for the user and is based on Django (a Python web framework). Once logged in (authentication via Keystone) the user can create virtual machine instances, manage the network (router, subnets, floating IPs) and other features provided by the individual plugins. Administrators can create new projects and users, assign resource pools to projects and create shared machine images (for example a shared version of a cloud optimized Ubuntu available to all users) and more. Horizons features depend heavily on the components installed but usually all components can be managed through horizon.

3.4.2.6 Image service: Glance

Glance is used to store images used for virtual machine instance creation in any number of supported ways. The options for storage back-ends range from simple filesystem, cinder up to object stores such as Amazon S3 or Swift. Images can be stored per project or shared for all users (can be configured per image).

3.4.2.7 Object Storage: Swift

Swift is an example for an optional component. While not mandatory for a basic setup it is comparable to Amazons S3. As with previous components there is a compatibility layer which allows the user to use the S3 API in order to access objects. Swift itself takes care of replication and distribution of stored objects in order to increase reliability and robustness and is therefore useful for high availability systems.

3.4.2.8 System Life-cycle and Management

As OpenStack consists of many components, which are in turn distributed among several machines it is not always a viable option to install and manage it by hand. This is especially true for deployments on large setups. A number of tools can be used to manage and maintain OpenStack such as JuJu, Puppet or Chef. These tools usually have a textual representation which software and configuration each node needs (for example setup with three classes: a swift node, a controller or a compute node). These textual representations serve as template and usually include variables or keywords for the IP address and hostname of the node, so a single configuration can be shared among many machines. JuJu for example comes with its own web-interface and support for Ubuntu's Landscape service (which is a Metal as a Service solution). It can take care of the deployment of the operating system on the bare metal via PXE or live media boot and then either balance the number of nodes among the available systems or allow the user to partition it manually. Without the help of Foreman or similar frameworks Puppet and Chef only allow for deployment on systems with an operating system in place and their respective client installed.

In case of upgrades the configuration file simply needs to be changed to refer to the repository of the new version and an update can be performed in place. The database should be backed up before migration but as the configuration files are maintained by these tools any changes required only need to be changed in a single spot. All tools mentioned

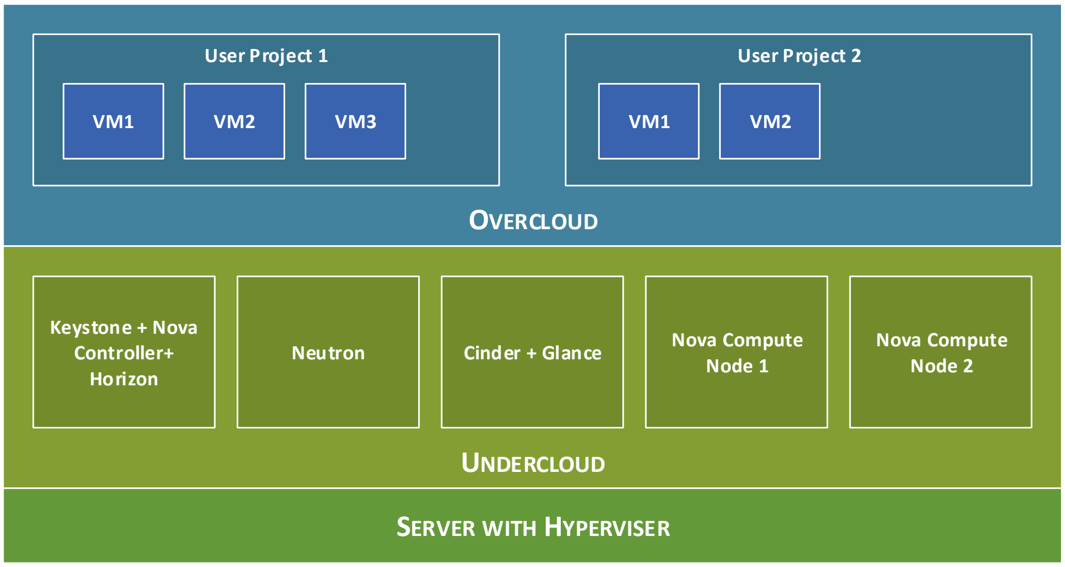


Figure 3.15: Block diagram showing a layered OpenStack installation.

here come with preconfigured solutions for OpenStack so the user does not need to create the templates but only fill in the variables (such as passwords etc.).

3.4.2.9 Cloud in Layers

Sometimes a server is too powerful for a single component and the resources could not be properly. Deploying multiple components on a single machine is possible but in case increasing demands require that component to be moved much of the configuration must be changed. In order to reduce complexity a second layer of virtualization can be added which hosts the OpenStack components. This layer is then called undercloud and the cloud services provided by OpenStack are then called overcloud (the user accessible cloud). This is illustrated in figure 3.15. While underlying hypervisor could also be a minimal OpenStack setup it is not very common. A good example for a combination of the layered approach with a DevOps tool described in the previous section would be Ubuntu's cloud installer (http://openstack.astokes.org/) which uses a combination of JuJu and KVM and Linux Containers (LXC) to deploy a cloud on any number of machines. The All-In-One setup deploys a number of proper virtual machines by using KVM on a single machine (one network node with neutron, one compute node and one controller with Keystone, Horizon, Glance, MySQL, RabbitMQ and the Nova controller) and then isolates the individual components into so called containers (OS level virtualization using the build in Linux Containers) which serve as a thin virtualization layer.

3.5 Data Aggregation and Visualization

Using a large amount of machines not only results in a new linear amount of speedup but also in a linear amount of more data. While it is still possible to evaluate the results of a single simulation locally this gets impractical fast once there are hundreds or thousands of results to analyze. While a location (such as a network share) which is accessible by all machines would be enough to merely store the simulation data it is usually a better idea to store them on a server which also has the ability to evaluate and visualize this data. For this purpose the task of creating testbenches and evaluating the results was spit of from this work into an independent master project [20] by Martin Schachner whose work was integrated in the this workflow.

While currently not supported by the SysCore framework other solutions for visualization of the resulting trace files could be used such as OpenTSDB (a time series database) with Graphana. Unfortunately without modification such a solution only supports finite precision (typically millisecond) time-stamps which might be insufficient for some cases. Other databases such as InfluxDB supports microseconds but at this time Graphana still only supports milliseconds as the minimal resolution.

3.6 Final Design

Using the technologies in the previous sections and initial concept a final design was created. Unlike the initial concept there is now a master who handles the retransmission of messages after outage of workers as well as the overall scheduling which would not have been feasible using only a webservice. Figure 3.16 illustrates the overall flow of information from the creation of the workpackages by the user to the upload of the results by the worker. As in section 3.2 the creation of the UML model is not covered in this diagram. The workers shown in this diagram may be in a public or private cloud, physical machines running the software bare metal or a mix of all three. Figure 3.17 shows an

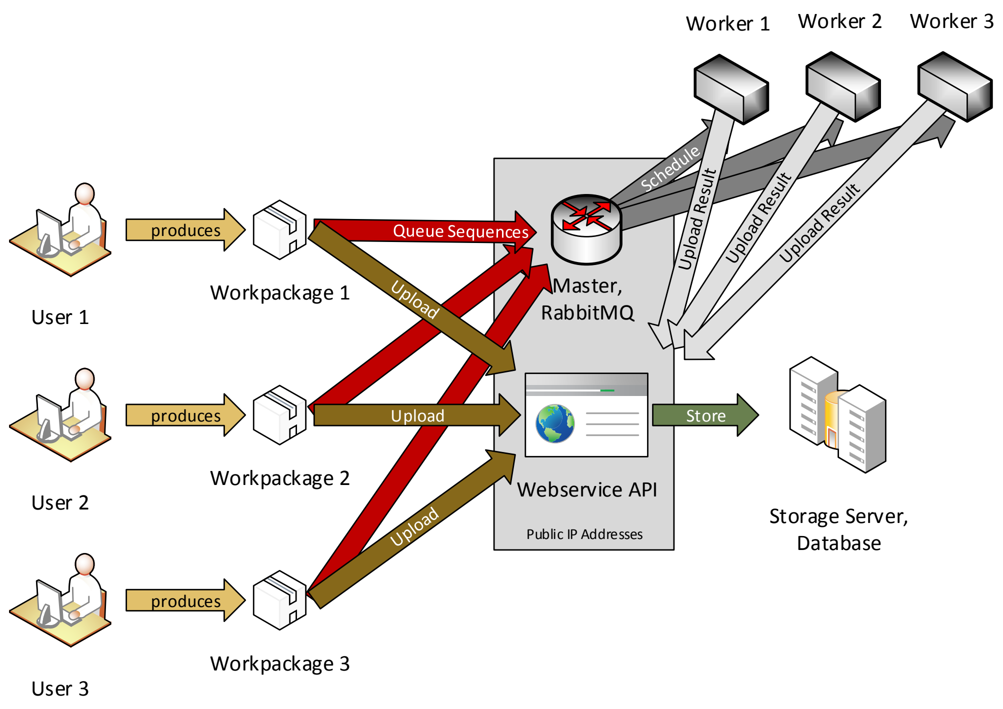


Figure 3.16: Flow diagram of the task creation process.

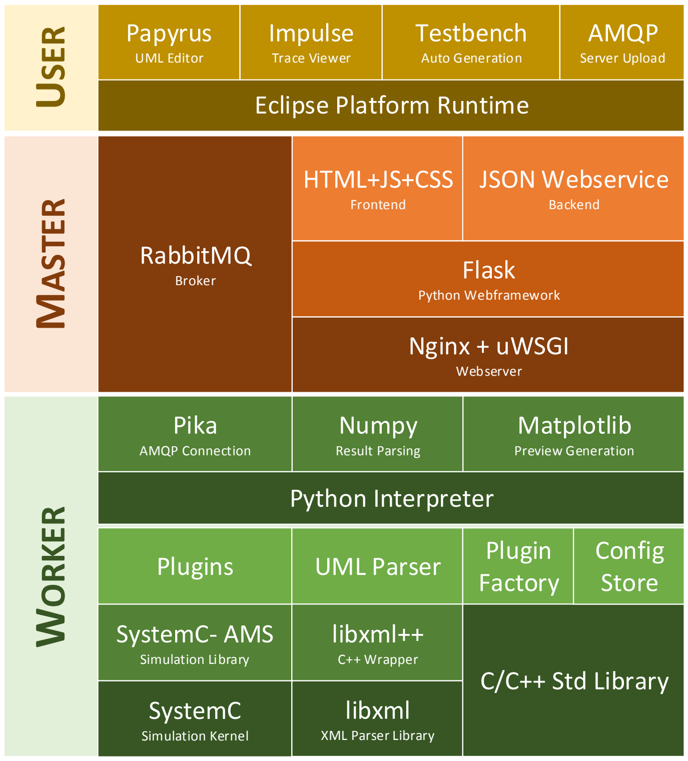


Figure 3.17: Software layers of the SHARC framework.

overview of the individual machines and the software running on it While RabbitMQ and Nginx could run on separate machines for simplicity they are combined in this overview. It can be seen that the most complexity lies in the worker. For this reason fig. 3.18 shows a zoomed in version of the C part of the software. Figure 3.20 shows the dataflow from the UML files to the results. The highlighted entries are files available for download using the web-interface. If for any reason the flow is interrupted (due to hard- and/or software failure) between the AMQP GET and AMQP ACK the task is rescheduled. Should the broker crash the task queue is stored to the filesystem therefore it will resume operation after a reboot. This queue can be mirrored using the build in cluster failover feature of RabbitMQ.

3.7 Class Diagrams

This section contains the class diagrams of the SystemC AMS part of the SHARC framework. The parts written in Python (such as the worker and web-interface) do not necessarily follow the object orient paradigm and therefore no class diagrams were created. The main function creates an instance of the `parser` class which creates all the SystemC AMS instances from a given UML file. These instances are created using the `SimObjectFactory` which loads all plugins from DLL files upon creation. A class diagram for this can be seen in fig. 3.21. The plugins used in the `SimObjectFactory` must expose the functions described by the interface `if_sim_obj_plugin` which inherits from `if_sim_obj`. A class diagram for these can be seen in fig. 3.22. While the first describes the functions required

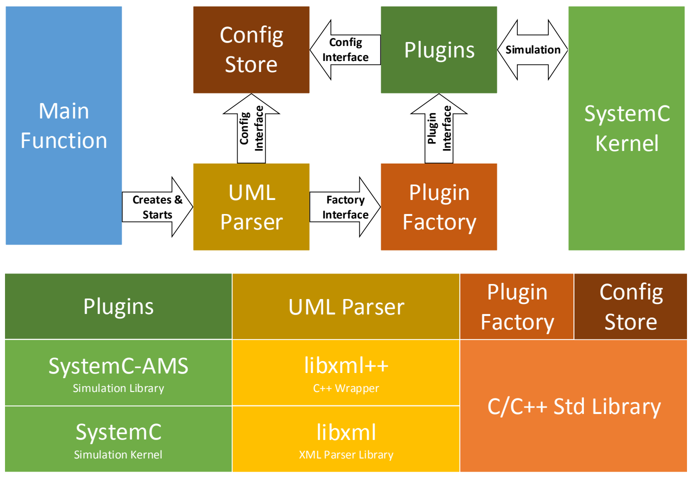


Figure 3.18: Overview of the components (bottom) of SysCore and their interactions (top).

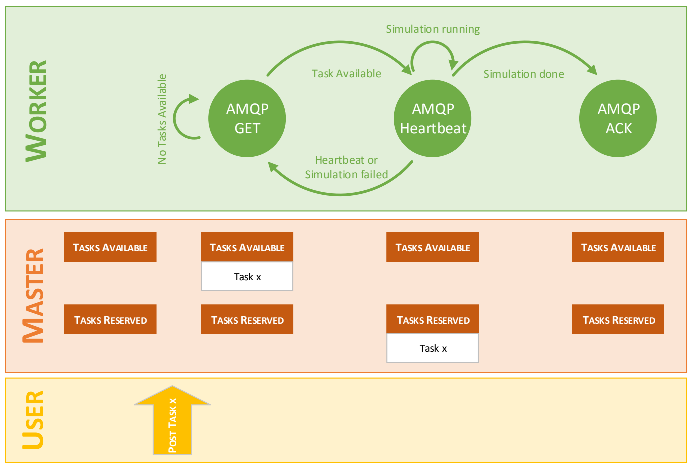


Figure 3.19: Diagram showing the synchronization between master and worker for the SHARC framework (using the colors from fig. 3.17).

in order for the plugin to be used by the factory the second interface is used by the parser to connect the instances created by the factory. In order for plugins to access global or shared information (available to more than one plugin such as the timestep used for simulation) a singleton is used. This means that the `ConfigStore` show in fig. 3.23 has only one instance which can be acquired by calling the `getInstance()` function. This also means that any plugin may store or obtain information from and to any other plugin. As the order in which instances are created cannot be influenced from the UML editor it is recommended to use functions which are called after the creation of the plugins such as SystemC's `before_end_of_elaboration()`.

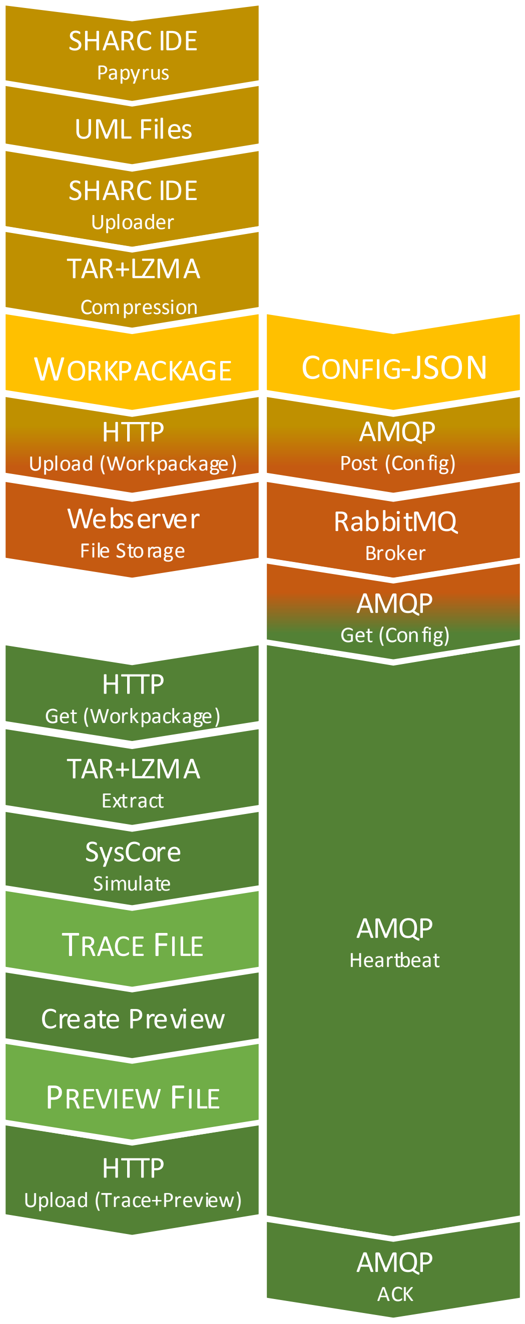
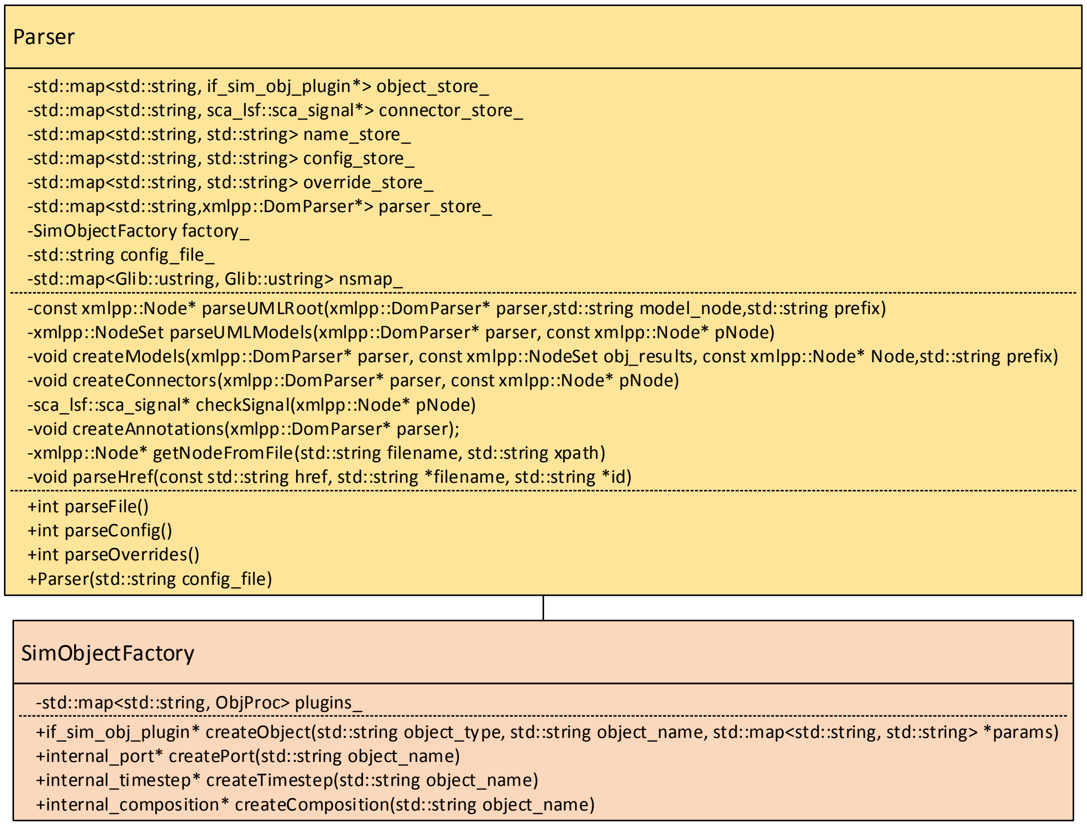


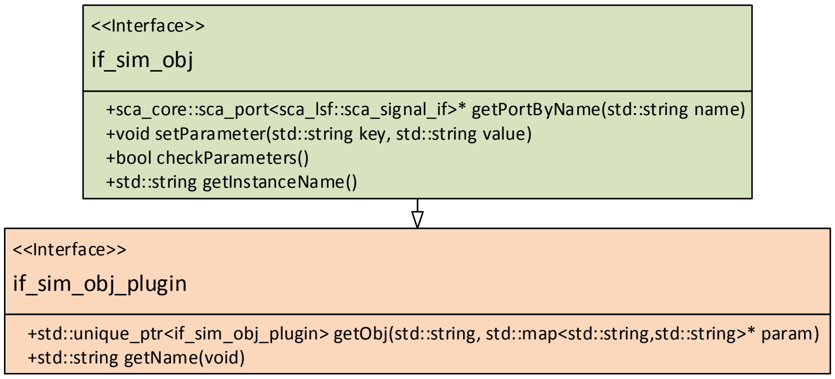
Figure 3.20: Dataflow diagram of the SHARC framework (using the colors from fig. 3.17).



```
Parser
-std::map<std::string, if_sim_obj_plugin*> object_store_
-std::map<std::string, sca_lsf::sca_signal*> connector_store_
-std::map<std::string, std::string> name_store_
-std::map<std::string, std::string> config_store_
-std::map<std::string, std::string> override_store_
-std::map<std::string,xmlpp::DomParser*> parser_store_
-SimObjectFactory factory_
-std::string config_file_
-std::map<Glib::ustring, Glib::ustring> nsmap_
-const xmlpp::Node* parseUMLRoot(xmlpp::DomParser* parser,std::string model_node,std::string prefix)
-xmlpp::NodeSet parseUMLModels(xmlpp::DomParser* parser, const xmlpp::Node* pNode)
-void createModels(xmlpp::DomParser* parser, const xmlpp::NodeSet obj_results, const xmlpp::Node* Node,std::string prefix)
-void createConnectors(xmlpp::DomParser* parser, const xmlpp::Node* pNode)
-sca_lsf::sca_signal* checkSignal(xmlpp::Node* pNode)
-void createAnnotations(xmlpp::DomParser* parser);
-xmlpp::Node* getNodeFromFile(std::string filename, std::string xpath)
-void parseHref(const std::string href, std::string *filename, std::string *id)
+int parseFile()
+int parseConfig()
+int parseOverrides()
+Parser(std::string config_file)

SimObjectFactory
-std::map<std::string, ObjProc> plugins_
+if_sim_obj_plugin* createObject(std::string object_type, std::string object_name, std::map<std::string, std::string> *params)
+internal_port* createPort(std::string object_name)
+internal_timestep* createTimestep(std::string object_name)
+internal_composition* createComposition(std::string object_name)
```

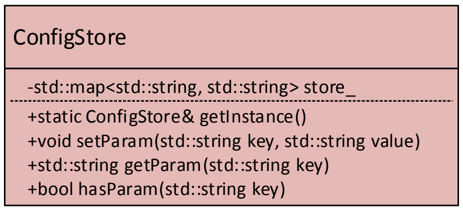
Figure 3.21: Class Diagram of the Core Classes of SysCore (using the colors from fig. 3.18).



```
<<Interface>>
if_sim_obj
+sca_core::sca_port<sca_lsf::sca_signal_if>* getPortByName(std::string name)
+void setParameter(std::string key, std::string value)
+bool checkParameters()
+std::string getInstanceName()

<<Interface>>
if_sim_obj_plugin
+std::unique_ptr<if_sim_obj_plugin> getObj(std::string, std::map<std::string,std::string>* param)
+std::string getName(void)
```

Figure 3.22: Class Diagram of the Interfaces of SysCore (using the colors from fig. 3.18).



```
ConfigStore
-std::map<std::string, std::string> store_
+static ConfigStore& getInstance()
+void setParam(std::string key, std::string value)
+std::string getParam(std::string key)
+bool hasParam(std::string key)
```

Figure 3.23: Class Diagram of the ConfigStore Singleton (using the colors from fig. 3.18).

Chapter 4

Implementation

This chapter gives a brief introduction to the technologies and software projects that make up the foundation of the SHARC framework. The first part covers the principal components as explained in section 1.2 and gives an in dept explanation about their implementation. The second part of this chapter explains the infrastructure chosen for the test deployment as well as options for a private cloud setup explored for this setup. The last part covers the workflow of the SHARC framework with the focus on an automotive example which was used to test the different aspects of the framework during development and testing.

4.1 SysCore

SysCore (short for SystemC Core) was developed as part of a master project [21] at Graz University of Technology with a focus on modularity and simplicity. As the software was developed with the intent to use it in a distributed environment it depends on very few external libraries and has a small footprint in memory (RAM and disk space). It is made up of four mayor components:

- The main.cpp
- The config store
- The Parser
- The Factory

While the behavior of the *main.cpp* is straightforward (parse arguments and write them to the *configstore* (a singleton which stores all the information and is accessible by all plugins and components), call the parser to parse the model and start the simulation) the behavior of the *parser.cpp/h* and *simobjfactory.h* require some explanation.

4.1.1 Factory

This factory takes care of the discovery of all the plugin files in the local (`./plugins`) as well as any global plugins (`/usr/syscore/plugins` and `/usr//local/syscore/plugins`) folders and registers them by their name. Each plugin contains four basic functions that are called by the parser when required.

- *createObject:* looks up the given type among the registered plugins and returns an instance of the class provided by the plugin. All parameters known at the time are passed as a Map of type `<std::string,std::string>`to the constructor of the plugin. If no suitable plugin is found NULL is returned instead.
- *createPort:* creates a standalone port which is a port that is not associated with another part (e.g., a property within the class). These ports are therefore on the border of the class and usually connected internally to ports of properties (e.g., exposing some input or output to a testbench)
- *createTimestep:* creates a dummy object which only servers to propagate a timestep to a cluster for the solver.
- *createComposition:* creates a model which represents the class as a whole but is not visible to the SystemC core. This is usually done for the root or top model of each UML model (e.g., one for the DUT, one for the testbench itself and so on).

4.1.2 Parser

The most complex component of the SysCore environment is the parser. Its purpose is to translate an UML model defined in one or more files to a single SystemC AMS model. This is done at run-time and does not require compilation of the resulting model. When the parser is first created it requires a model name for the top model in the diagram as well as the name of the UML file that contains said model. For a general overview of the steps involved and their transition see fig. 4.1. While most of the process is straight forward there is one potential "loop" if a class has attributes other than ports (usually properties) that node is treated as the new root node and the submodel is created before moving on to the connection creation. Each submodel may contain any number of submodel of its own therefore this step may repeat any number of times. It is important however to know that it is possible in UML to model a class that contains a property of the type of that very same class. This way the parser would not terminate therefore such constructs must be avoided.

In order to better understand the steps within each state the names and functions of each state will be explained here but it is recommended to read the source code to fully grasp the inner workings of the parser if required.

- *initialize Parser and find model root:* The initialization is partially done in the constructor of the parser and the function *parseFile* which then reads the file- and modelname from the configstore and starts searching for the node with the name equal to the modelname in that file. For each file a new `xmlpp::DomParser` is created and configured.
- *parse children of given root:* In this state the node found in the previous state is given as an argument to *parseUMLModels.* This function returns a `xmlpp::NodeSet*` that contains all the children of the specified node. Like all searches this is internally

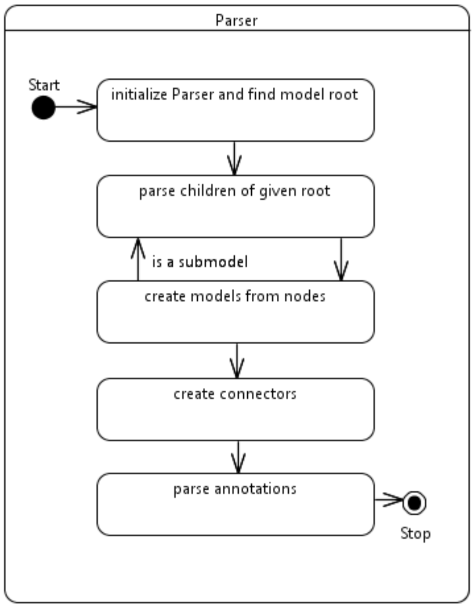


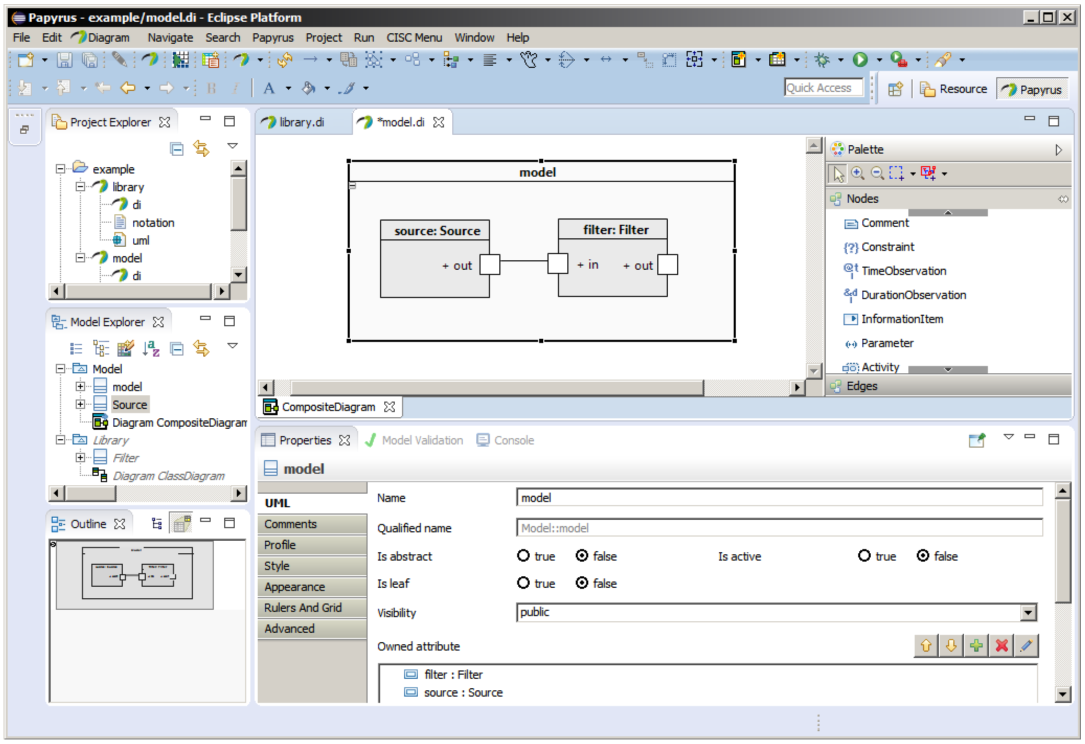
Figure 4.1: Simplified state machine for the Parser used in the SysCore framework.

done via XPATH using the `xmlpp::Node.find(xpath,nsmap)` while the nsmap is populated in the constructor and contains the most common namespaces.

- *create models from nodes:* As stated before this step is somewhat special within the parser as it may move one step "up". For each `xmlpp::Node` in the `xmlpp::NodeSet` a check is done: if it contains attributes that are not ports (e.g., properties) the factory is called to create an composite object and the node is then passed as the new root to *parseUMLModels* detailed in the previous state. Additionally *createConnectors* from the next step is called here to first create the connectors within the submodel. Else information like the type and name as well as any parameters are read from the node and passed to the factory to create a new instance of a *sim_if_obj* (usually containing one or more SystemC AMS models) and is then stored in a `std::map` by name and id. If the node happens to be a `uml:Port` the the factory is called to create a port instead which is then registered with the composite (so getPortByName can be used to obtain it).

- *create connectors:* This step parses all *ownedConnectors* that are children of the (sub-)model root and creates `sca_lsf::sca_signal` objects to represent those connectors. It is important to notice however that even though UML allows multiple 1:1 connections per port SystemC AMS merely allows a port to be bound once but a signal may connect any number of ports (basically 1:n as only one driver is allowed per signal). To translate these differences both ends of each connector (ports) are stored by id and if a signal would be created to connect to a port that is already in use (aka a signal is already connected to it) the old signal is reused instead.

- *parse annotations:* Using CISC annotations it is possible to configure the parameters of any given model. The structure of these annotations represents a key/value stor-



model

Name: model
Qualified name: Model::model
Is abstract: true [✓ false]
Is active: true [✓ false]
Is leaf: true [✓ false]
Visibility: public
Owned attribute:
filter : Filter
source : Source

Figure 4.2: Main window of Eclipse with Papyrus showing the simple diagram from fig. 3.9.

age similar to a `std::map<std::string,std::string>`. For each key/value pair *setParameter* is called on the referenced sim_if_obj. At the end of this state a timestep is created by the factory for each signal in the model and configured with the timestep from the configstore.

4.2 SHARC IDE

The software for the user side of the simulation framework is based on Eclipse and called SHARC. It is built around existing plugins such a papyrus (the UML editor used) and impulse (for visualizing the trace files of the simulation) and a few plugins for online and offline simulation. Papyrus was extended by a number of plugins to only show relevant information by using a custom theme as well as to allow the user easily instantiate predefined library components. As the Eclipse platform runtime is easily extendable via plugins new components can be added later on and managed via the build in software updater.

4.2.1 Papyrus

Papyrus is part of the Eclipse Modeling Tools as well as available as a standalone feature for the Eclipse Platform. It is primarily used to create diagrams for modeling languages such as UML2 and SysML.

Figure 4.2 shows the main elements of Eclipse running Papyrus as a model editor. To the left side there is the Project Explorer which shows an overview of all projects in the

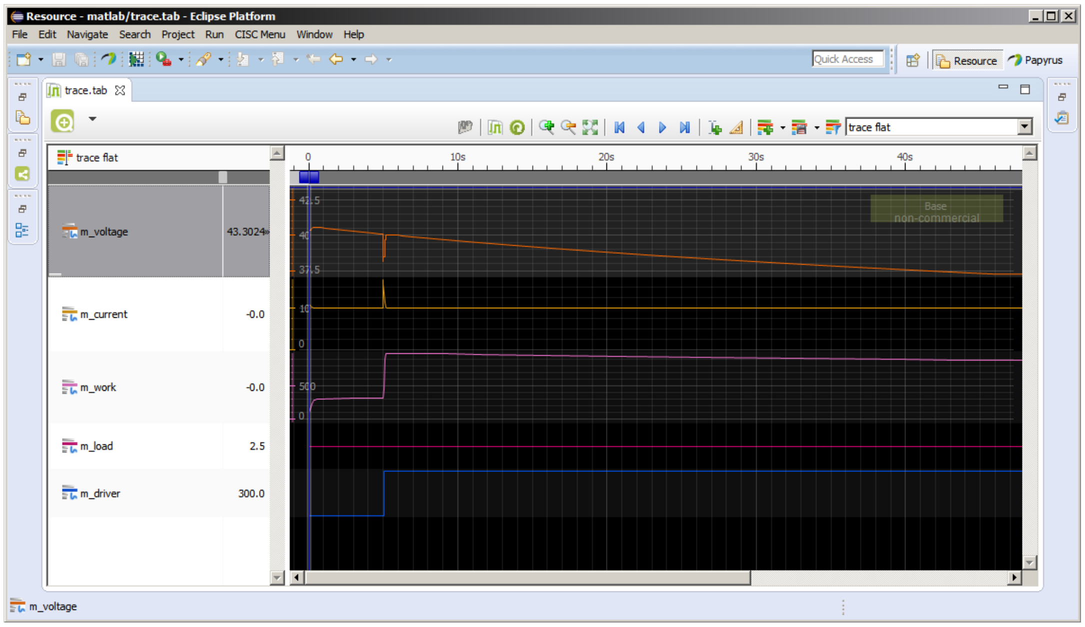


Figure 4.3: Main window of Eclipse with Impulse showing a trace file created by SystemC AMS.

workspace. Papyrus models are shown as a group containing the *.di* (a legacy file which is basically empty these days), the *.notation* (which describes the layout in Papyrus) and the *.uml* (containing the actual UML model). Below is the Model Explorer which contains the current model contained in the file as well as any referenced libraries/models. The Palette on the right contains a set of commonly used elements such as Classes, Properties and various types of Connectors (e.g., Associations). On the Bottom of the screen there is the Overview on the left that shows the entire model and a box highlighting the area now shown in the main window. To the right there is the Properties Editor which allows the user to change any parameter, apply profiles as well as the configuration of the appearance. The main window shows a portion of the model (in this case the entire model) and allows the user to select an element for editing. The appearance of the overall model is defined in a CSS theme which details which portions of the model should be visible to the user as well as their appearance. The SysCore framework already provides a theme which configures Papyrus to only display elements that actually influence the behavior of the simulation as well as enable the synchronization between properties in the model and classes in the library.

4.2.2 Impulse

Impulse is an Eclipse plugin that allows displaying of various data formats in a graphical manner. Both types of trace files available in SystemC AMS (vcd - value change dump and tabular) are supported by impulse by default.

Figure 4.3 shows the main window of Eclipse with impulse running in full screen. The left side shows the available signals and their value at the cursor (0.0 seconds in this example) while the right side shows a plot over time of for the given signals. Impulse allows for a hierarchical representation as well as for a flat one (as in this example). While signals may be plotted in many different ways all signals are configured as analog interpolated by default. For digital signals a display of the values as integers might be more suitable (as is the default for pure SystemC trace files).

4.2.3 SysCore specific plugins

The plugins specifically created for SysCore can be split into two principal sections:

- **Visual Plugins:** These plugins contain mainly papyrus extensions like the theme used for all the screenshots in this work as well as the palette plugin which allows the user to instantiate new components for the library.
- **Simulation Plugins:** These plugins are used to either simulate a created design offline or as either a single simulation or batch online. Additionally the creation of testbenches for an existing design including verification of outputs.

Most of these plugins were created as part of a different work [20] by Martin Schachner and will not be discussed as part of this work.

4.3 Web-Interface

4.3.1 Flask a Python web framework

Due to the nature of this project a user interface that is easily accessible from any PC or notebook is required. As with most applications these days this is achieved by using web technologies such as HTML and Javascript. Writing everything from scratch is very uncommon as maintainability and security would be mayor issues especially for a site which hosts potentially sensible data. For this reason Flask was chosen due to its minimal requirements and good support. To give a better understanding what minimal means here is a fully featured webserver (it does not depend on any external webserver for the hosting such as Apache or Nginx) presenting a minimal, static website:

Listing 4.1: hello.py

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

after importing flask all that is left is to declare a new app, add a route for the title page (the route statement) and run the app (which is a blocking statement). For dynamic applications the decoding of the URL and subsequent function calls are also fairly simple:

Listing 4.2: generic exmple

```
@app.route('/user/<username>')
def show_user_profile(username):
    return 'User %s' % username
```

Listing 4.3: integer exmple

```
@app.route('/post/<int:post_id>')
def show_post(post_id):
    return 'Post %d' % post_id
```

These examples demonstrate how to obtain information from the URL by using the `@app.route`. Python already has excellent support for most SQL databases (such as MySQL and PostgreSQL) using SQL alchemy. It would be fairly easy to craft a simple RESTful API by hand but as with most common applications there are existing solutions for this.

Listing 4.4: REST exmple

```
import flask
import flask.ext.sqlalchemy
import flask.ext.restless

app = flask.Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = flask.ext.sqlalchemy.SQLAlchemy(app)

class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.Unicode, unique=True)
    birth_date = db.Column(db.Date)


class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    isbn = db.Column(db.Unicode, unique=True)
    tible = db.Column(db.Unicode)
    author = db.Column(db.Unicode)
    purchase_time = db.Column(db.DateTime)
    owner_id = db.Column(db.Integer, db.ForeignKey('person.id'))
    owner = db.relationship('Person', backref=db.backref('books',
                                                     lazy='dynamic'))

db.create_all()

manager = flask.ext.restless.APIManager(app, flask_sqlalchemy_db=db)

manager.create_api(Person, methods=['GET', 'POST', 'DELETE'])
manager.create_api(Book, methods=['GET'])

app.run()
```

This example is a fully fledged HTTP server which connects to a SQLite database (a file based SQL solution which requires no separate running SQL server) using Flask-

RESTLess. The classes *Person* and *Book* are converted by SQLAlchemy into tables and stored in the SQLite database. Both tables are then linked using the id and using backref every person has a list with the name books that contains all books. Using a RESTfull API a user may call GET, POST and delete on any *Person* while for *Book* only GET is available. The results are delivered as JSON objects with automatic pagination (splitting large results into several pages to improve performance). Flask-RESTLess also includes the option to filter results by adding a JSON object with the filter parameters in a SQL like syntax.

4.3.2 Twitter Bootstrap

While the functionality of Flask is pretty mighty it the resulting pages are by no means pretty. In order to allow a single website to be properly displayed on devices with different resolutions, aspect ratios and screen sizes Javascript is typically used. As with the backend of the application it is not recommended to create custom solution due to the risk improper use of Javascript poses. Therefore Twitter Bootstrap (or more precisely Flask-Bootstrap) was used in the creation of the web-interface.

4.3.3 Matplotlib and Numpy

In order to generate previews of the simulation trace files (which are stored as tabular files similar to CSV) Numpy was used to parse them to Python matrices which can then be plotted using Matplotlib. Similar to Matlab Matplotlib uses figures and subfigures. Overall the syntax is similar to that of Matlab allowing for legends, dotted or dashed lines and so on.

4.3.4 Nginx and uWSGI

The Web Server Gateway Interface WSGI is a specification used to allow webservers (such as Nginx) to interact with web applications or frameworks written in Python and uWSGI is a small implementation of said specification. The reason for this is that while Flask provides a webserver out of the box it does not scale very well. For load balancing, access control and more advanced features Nginx was chosen due to its versatility and high performance. Nginx is usually also better protected against attacks it is a main development focus while the server that comes with Flask is only meant for development and small deployments. uWSGI consists of an emperor service which spans a few instances of the web applications that are then used for load balancing. The communication between uWSGI and Nginx is done via a standard socket. Using this setup the Flask application can be scaled out to several Nginx servers each communicating with an uWSGI emperor running multiple Flask instances.

4.4 Infrastructure

As Amazon is currently the largest provider of cloud services worldwide it was chosen as the target platform but due to the running cost of hosting instances on EC2 it was decided that a local alternative was required during the development phase as well as an option

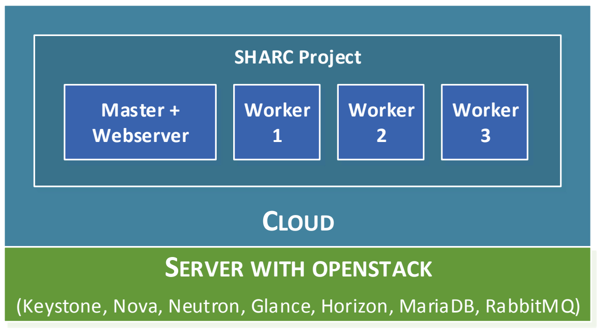


Figure 4.4: Block diagram showing the OpenStack installation used for development.

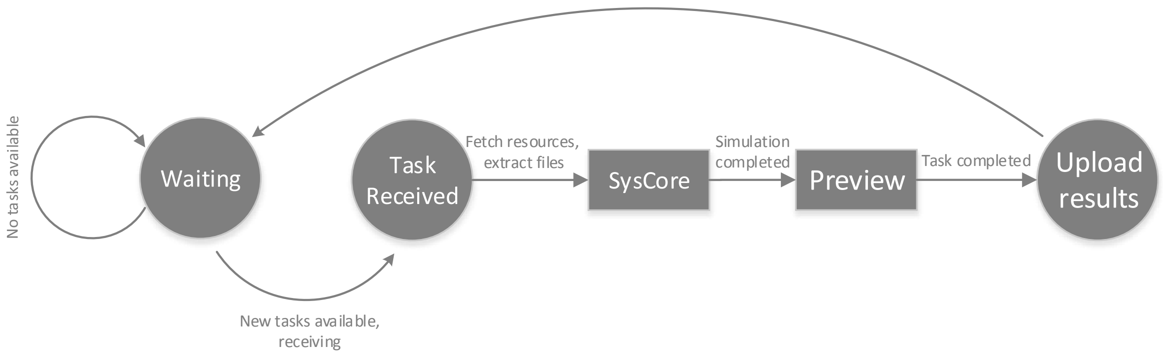


Figure 4.5: State diagram of a worker.

for customers requiring that the data stays in house. OpenStack can be accessed via an API compatible to Amazons and was therefore chosen as development platform for the framework. Currently none of the components actually uses any of these API functions but for the purpose of adding and removing workers under different load scenarios (a feature planned for a later release) this was a required feature. For the purpose of development an All-In-One solution was chosen for the OpenStack deployment and due to the constraints in the network setup the setup was configured by hand. The current setup is illustrated in figure 4.4, the number of workers should not exceed 3 in this scenario due to the number of cores available on the test server (4 Cores). As one core is there required for the overhead of OpenStack (virtual network and storage) as well as the master and webserver.

The bulk of the communication is based on AMQP (with RabbitMQ as its implementation) due to the build in redundancy and robustness. The message throughput is also magnitudes higher than the expected workload of the final system. The web interface is currently hosted by a Nginx due to its small size and high performance although Apache could also be used with uWSGI (as it is currently done by OpenStack Horizon).

4.4.1 Worker

Using the worker queue pattern described in section A.3 worker has been implemented in Python using Pika for AMQP communication. The worker connects to the broker running on the master and uses a heartbeat signal to indicate it has not crashed or is otherwise unavailable. Figure 4.5 shows a state diagram which corresponds to the workers behavior. The simulation is executed in as a child process so the worker can keep sending heartbeat signals during that time. The preview uses Matplotlib and Numpy described in section 4.3.3. As the simulation core is unable to use multiple threads for simulation the ideal setup for a worker is a virtual machine that has only one CPU. Should such a setup be unavailable or physical machines be used instead it is however possible to spawn multiple worker threads on a single machine as well. Each worker requires only a minimal Linux image containing the following software (and their dependencies):

```
libxml++2.6 libgsl0 python-pika python-matplotlib python-numpy
```

While the libraries should not break compatibility the exact versions used were libxml++ 2.36.0 and libgsl 1.16. For the Python packages Ubuntu 14.04 included version 0.9.13 for Pika, 1.3.1 for Matplotlib and 1.8.2 for Numpy. While the software has been tested with different verions minor adjustments may be needed. For the Python interpreter itself version 2.7.6 was used. The SysCore binary already includes the full SystemC and SystemC AMS libraries. Libgsl is currently only used by the provided model for the lithium battery and can be omitted if not needed for this purpose. While it is possible to use a machine running Windows instead of a Linux machine this is generally not recommended. The software has been tested on Ubuntu Linux 14.04 as well as CentOS 6.4. Workers do not require a public IP address or connection to the internet, they need only to be able to connect the master and the webserver (which can run on a single machine). The framework does not impose any limitations on the number of workers, usually there are as many workers as possible at any given time, they can however be easily spawned on demand and destroyed if no longer needed (if necessary even during a running simulation as it will be rescheduled by the master in that case). Workers do not need a persistent state as all information for simulation is gathered from the webserver (UML workpackage) and the master (SysCore configuration).

4.4.2 Master

The master is responsible for accepting and distributing simulations tasks to the workers. As with the workers the master does not store information (other than the persistent queue) and therefore needs only a minimal configuration. Most Linux distributions include a package for RabbitMQ as well as Erlang (the programming language used for RabbitMQ).The SHARC framework was developed using version 3.2.4 on Ubuntu 14.04. It is recommended to install the management web-interface as well by running

```
# rabbitmq-plugins enable rabbitmq_management
```

This will install and start the web-interface for RabbitMQ on port 15672. The login using the guest account is only possible from the machine RabbitMQ is running on therefore an administrative user should be added by running

```
# rabbitmqctl add_user [username] [password]
# rabbitmqctl set_user_tags [username] administrator
```

Note that all commands above require root privileges on a Linux machine. RabbitMQ can also be configured to run in cluster fail-over mode to increase availability but this is out of the scope of this document. While AMQP server could be used only RabbitMQ has been tested so far. The workers also require their own user (e.g., one shared user) in order to connect to the server but no additional tags (permissions) are required. In its current for all workers share a common queue (called task_queue) but this could easily be split into one queue per customer.

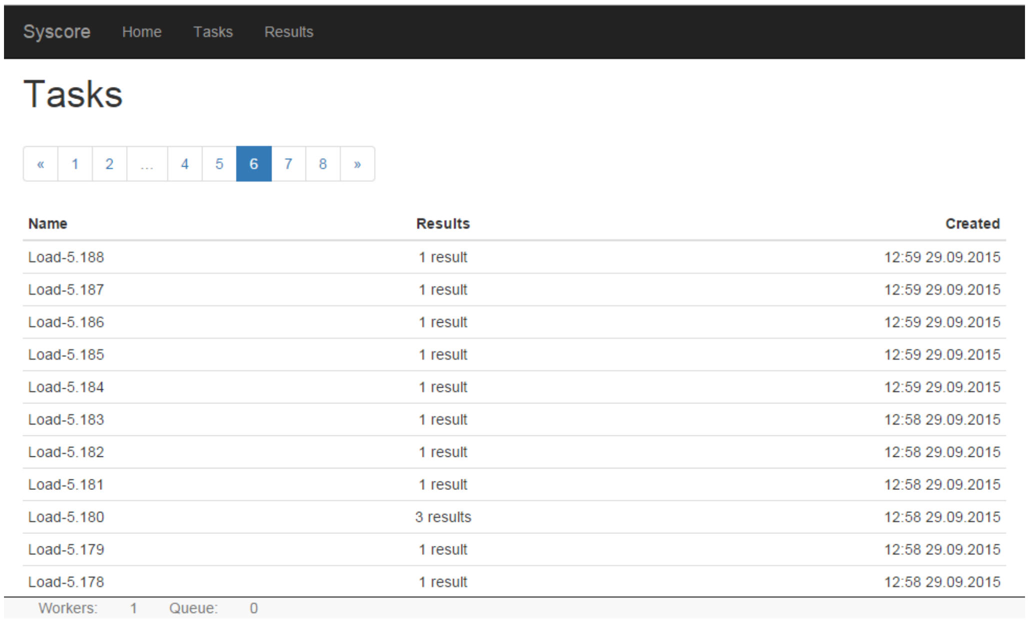
4.4.3 Webserver

The Webserver is currently the only server that actually stores data. It is written completely using Python and Flask. Once again Ubuntu 14.04 with python 2.7.6 was used and Flask 0.10.1 was installed using pip (Pythons own package manager). In order to increase performance Nginx (1.4.6) and uWSGI (1.9.17) are used to balance the load among several Flask instances running in parallel. Standard load balancing setups can be used to increase the amount of webservers but the current setup uses only one such server. The webserver also hosts a SQL database (using MariaDB version 5.5.46, a MySQL fork) which would need to be replicated among all webservers as well or deployed on a separate set of database servers to improve the performance even further. Every simulation task can have a single workpackage (a archive containing all UML files required for the simulation (including any IP libraries required) assigned to it which is the stored on the webserver using a file upload. This feature does not scale as well but could be replaced by an OpenStack Swift object storage server/cluster. The current storage backend however only writes the uploaded file to the filesystem of the webserver. The webserver hosts two types of sites: The JSON API for interaction with other software (e.g., optional Eclipse plugins) as well as a human readable HTML website. The HTML portion uses Twitter Bootstrap for a reactive layout and has been tested on several desktop PCs, notebooks and mobile devices (such as smartphones). For every task the workers create simulation results and store the trace files, the configuration used and the output logs of the simulation core to the server using the file upload feature as well. These files are then presented by the webserver in a human readable fashion:

- The log files are displayed inline in collapsible panels
- The trace files are stored for download by the user
- The config files are stored as well and presented as table inline (values such as timestep and max simulation time as well as overrides of the parameters in the UML)
- The created previews are thumbnailed by the server upon upload and embedded into the site

Figure 4.6 shows a screenshot of the web-interface listing a few sample tasks. The navigaion bar on top can be used to either select tasks or a list of the most recent result (sorted by date in descending order).

Figure 4.7 shows a screenshot of the web-interface showing the results of a single simulation. The Config panel displays the values parsed from the config XML, the Preview



Tasks

Name	Results	Created
Load-5.188	1 result	12:59 29.09.2015
Load-5.187	1 result	12:59 29.09.2015
Load-5.186	1 result	12:59 29.09.2015
Load-5.185	1 result	12:59 29.09.2015
Load-5.184	1 result	12:59 29.09.2015
Load-5.183	1 result	12:58 29.09.2015
Load-5.182	1 result	12:58 29.09.2015
Load-5.181	1 result	12:58 29.09.2015
Load-5.180	3 results	12:58 29.09.2015
Load-5.179	1 result	12:58 29.09.2015
Load-5.178	1 result	12:58 29.09.2015

Workers: 1 Queue: 0

Figure 4.6: A screenshot of the SysCore web-interface listing tasks.

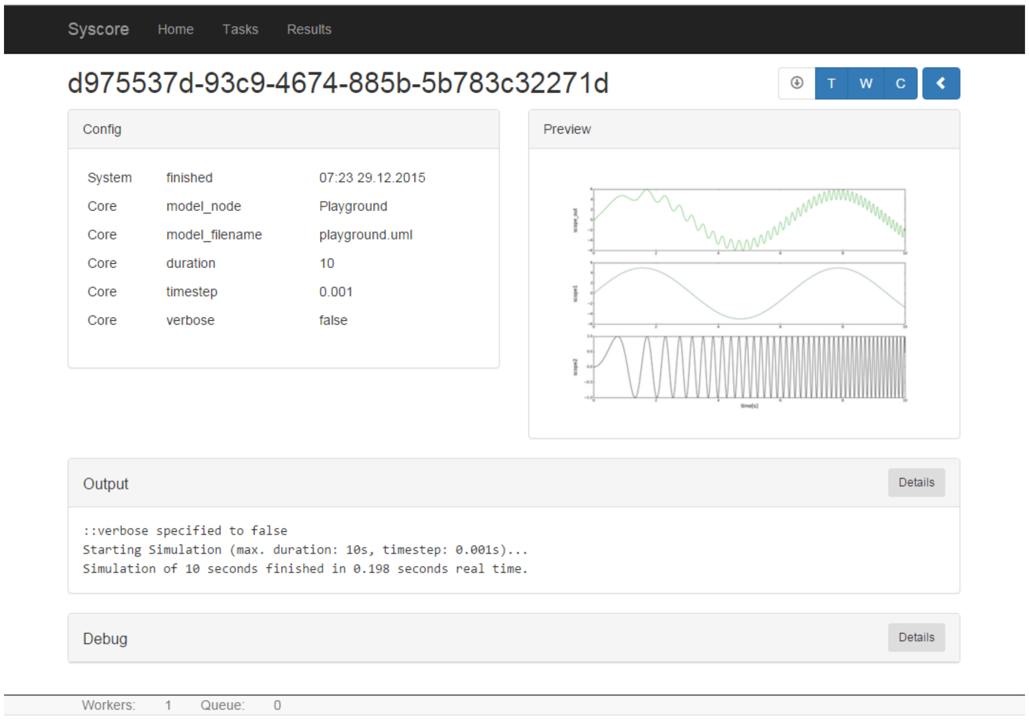
pane shows a thumbnail of the preview created by the worker using Matplotlib and the Output and Debug panels display the output of the simulation core (with debug collapsed in this screenshot).

4.4.4 Hardware Setup

While for development purposes an All-in-One solution is sufficient for a production setup more than one server is recommended. During the course of this work several setups were explored.

4.4.4.1 Commodity Hardware

While the system is designed to run in a cloud environment this is not necessary for typical operation. Using of the shelf hardware is a viable choice in order to keep costs for the deployment minimal. Due to the nature of the SHARC framework it is not even required for all machines to have the same performance. Faster machines complete their tasks in less time and therefore fetch more tasks from the queue, increasing their throughput without any modification to the system. The simulation core does not rely on any architecture specific features and can run on Windows and Linux (while most posix systems should work such as OS X, Solaris and BSD systems). For a cloud approach machines with virtualization extensions are recommended in order to keep the performance at maximum but Intel and AMD offer CPUs with this feature for nearly a decade now. As mentioned in section 3.4.2.1 KVM supports virtualization on non x86 CPUs as well. This allows for using hardware that one would not commonly associate with compute tasks such as the Raspberry PI 2 and the Pine64. PicoCluster LLC offers small cluster setups based on such machines ranging from 3 to 100 nodes (for pre-order at the time of this writing). The



d975537d-93c9-4674-885b-5b783c32271d

Config

System	finished	07:23 29.12.2015
Core	model_node	Playground
Core	model_filename	playground.uml
Core	duration	10
Core	timestep	0.001
Core	verbose	false

Output

```
::verbose specified to false
Starting Simulation (max. duration: 10s, timestep: 0.001s)...
Simulation of 10 seconds finished in 0.198 seconds real time.
```

Debug

Workers: 1 Queue: 0

Figure 4.7: A screenshot of the SysCore web-interface showing the results of a simulation.

price for a 100 node cluster will be approximately $10000 which means a price of $100 per CPU or $25 per core (including case, storage and network switches). While such a price is possible with Intel processors alone a complete setup with storage, case and network infrastructure usually is more expensive (while some Atom based solutions might come close). As these systems support virtualization extensions OpenStack is supported with KVM and while RAM is limited, the simulation core should hardly ever exceed 10MB. Of course the performance of such systems is inferior to even the lowest performing of Intel's offerings.

4.4.4.2 Standard SOHO Servers

The next step would be to use small office/home office servers. These machines usually come in cases which allow for tower or rack usage. These servers typically have one to two CPUs with two (Intel Pentium G4400) to eighteen cores (Intel Xeon E5-2699). Examples for this are systems based on Intel's P4304BTSSFCNR (single CPU) and P4308CP4MHEN (dual CPU). As these systems are designed for 24/7 they usually offer better reliability and replacement parts are available for a longer time. The density of such systems in a rack setup is relatively low as the equivalent height is 4U (a normalized unit of height for rack systems).

4.4.4.3 Standard Rack Servers

Like the SOHO servers these systems offer high reliability and long support but a comparable server can be as small as 1U (offering up to 4 times the density of SOHO servers).

Type	[U]	[CPUs/U]	[Cores/CPU]	[Cores/U]
SOHO	4	2/4	18	9
Standard	1	2/1	18	36
Multi-Node	3	28/3	14 (18)	130 (168)

Table 4.1: Comparison of hardware setups.

Considering

4.4.4.4 Multi node Servers

Offered in a large variety of implementations these servers usually have more than one node (a self-contained server) in one case. Examples for this are IBM's BladeCenters or Supermicro's MicroCloud and MicroBlade systems. While there are many vendors each offering a large selection of configurations Supermicro's MicroBlade offering:

- A standard rack size for server systems is 42U (with 19 inch width).
- The main server comes in 3U (14 Blades) or 6U (28 Blades). This does however not increase the overall density of these systems.
- A single Blade can have multiple configurations from two independent Nodes with a LGA1151 socket (up to 4 cores) to single nodes with dual LGA2011 socket (theoretically up to 18 Core, with up to 14 listed as supported) resulting in a maximum of 28 (36) cores per blade.

This results in up to 196 Xeon dual processor nodes (but usually a few U are used for power distribution and networking) each having up to 14 cores per processor. This means 5488 cores per rack (with a theoretical 7056 cores maximum).

4.4.4.5 Comparison

Table 4.1 shows a comparison between the different server setups. Commodity hardware usually does not fit rack mounts and is therefore omitted from this comparison.

4.5 SHARC Workflow

This section will explain the steps required to execute a simulation and the corresponding actions taken by the framework usually invisible to the user.

4.5.1 Planning and Design

First the user has to create a system to be simulated. For this an Eclipse based tool called SHARC is used. As SHARC uses Papyrus for UML modeling a new papyrus project needs to be created with a composite diagram to model the behavior of the desired system. For this example we assume the created design reflects the structure shown in figure 4.8.

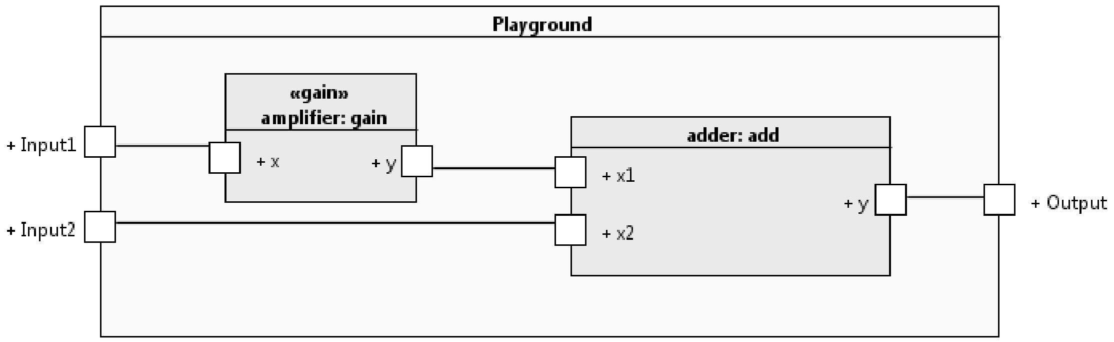


Figure 4.8: Example of an UML diagram.

4.5.2 Testbench creation

As the design does not include any signal generators or monitors a testbench has to be created. This could be done by hand but SHARC already comes with the option to automatically create testbenches for the user. CSV (comma separated value) files are used as input signals (with each row containing the timestamp in milliseconds at which the value will be written and the output value itself, separated by a comma) while the build in monitors are used to write any output to a tabular file (which is similar to the input although separated by a space). Additionally there is the option to create a success validator which compares the actual signal to a reference (CSV) and outputs a high if it deviates by a given margin.

4.5.3 Submission

The finished design (including the testbench) can now be used for simulation. There are two ways this can be done, offline and online. For offline simulation the included SysCore binary is called in a new process with a configuration XML file that does not include any overrides. This is merely a design decision but due to the usually large amount of data resulting from a constrained random or Monte Carlo simulation this option has been omitted from the offline simulation. Using Offline simulation the simulation can be tested and a workpackage (.tar.xz archive containing the UML files) can be created. This file can then be submitted for online simulation. Submission is actually done in two parts:

- **Webserver**: Using the webservice API of the framework a new task is created and using the file upload feature a new workpackage is uploaded for this task.
- **Master**: The individual simulation runs are send to the worker queue in JSON format containing the entire config for each (encoded in base64) and the task it belongs to. While this step could be hidden behind the webservice API of the webserver (effectively hiding the broker from the user) this has not been implemented at this time.

4.5.4 Simulation

The workers now execute the simulations as described in section 4.4.1. They also call the webservice API of the framework to created new results and upload the resulting tracefiles, the given configuration as well as the output (stdout and stderr) to the webserver.

4.5.5 Evaluation

The user can now evaluate the results using the web-interface and download any relevant trace files. This step may be improved in future work by including the results of the success validator in the web-interface. As of impulse version 1.6 it will also has the ability to use Signal Scripts (using Javascript to compare signals) using more than one trace file.

4.6 Case Study: AVTR

This section contains an example workflow based on an existing model taken from [22]. This model was created in Matlab Simulink as part of a prior project and will be used to illustrate the steps required for simulation.

4.6.1 Original Model

Based on the original Simulink shown in fig. 4.9 system a battery powered motor system was to be created as an UML model. In order to do so the principal components had to be analyzed. The following sections will give a short introduction to the four components present in the Matlab Simulink model.

4.6.1.1 Inverter

The inverter shown in fig. 4.10 is itself not a Simulink model but rather described as a piece of Matlab code with *u* being the return value (output) and *VBat* and *uc_out* as arguments (inputs). The function is straight forward as it merely return the minimum over the the arguments.

Listing 4.5: inverter.m

```
function u = PMDCMot_Inverter(Vbat,uc_out)
%#codegen
%This functions implements a simple inverter function for a PM-DC motor
%driving stage . The inputs are the battery voltage (Vbat) and the controller
%output (uc_out).
if (uc_out <= Vbat)
    u = uc_out;
else
    u = Vbat;
end
end
```

4.6.1.2 PI State Space Controller

The internal model of the PI State Space Controller is shown in fig. 4.12 while the inputs and outputs are in the same order as on the model shown in fig. 4.11. Basically the

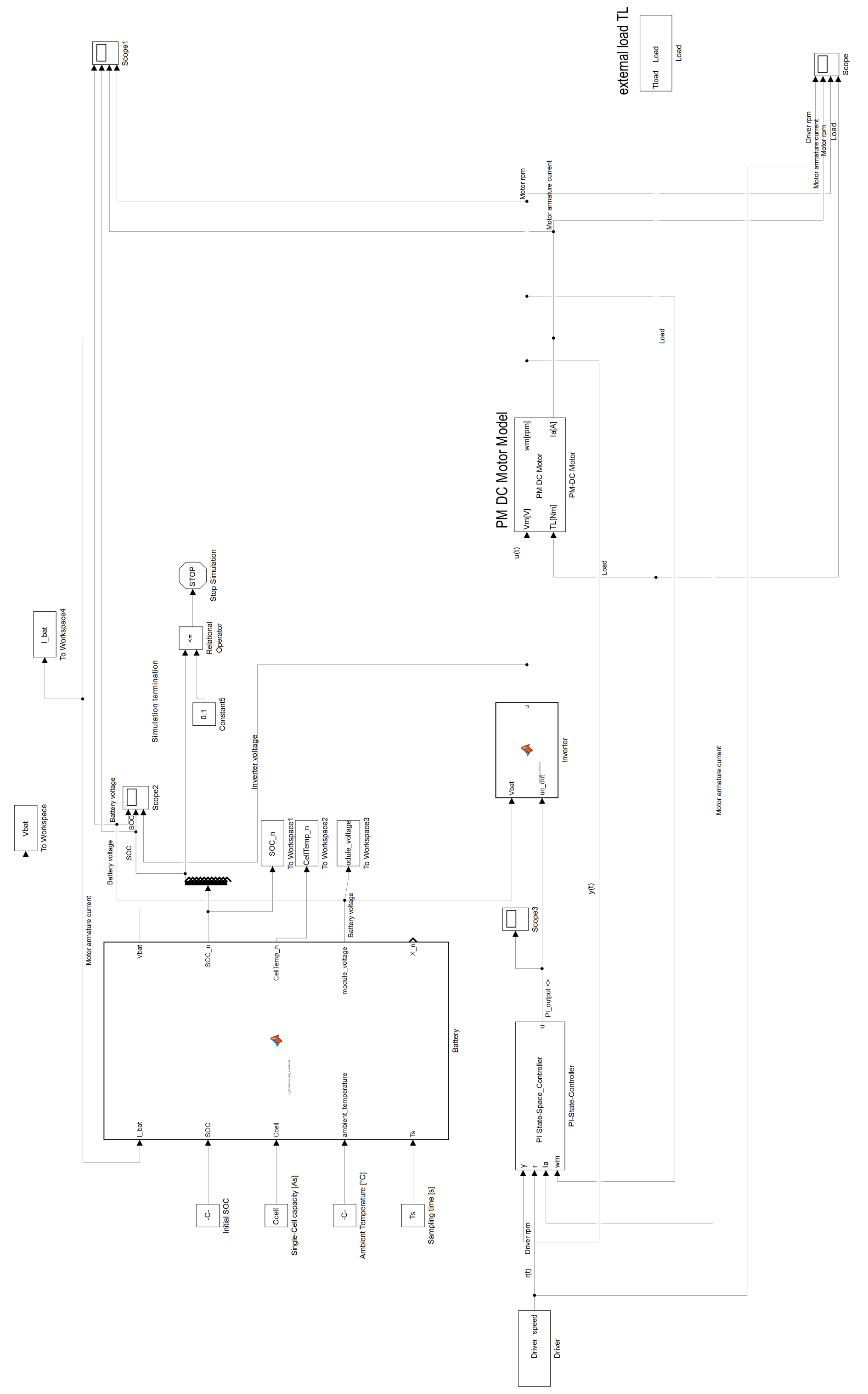


Figure 4.9: A complete overview of the Matlab Simulink Model used for the AVTR case study.

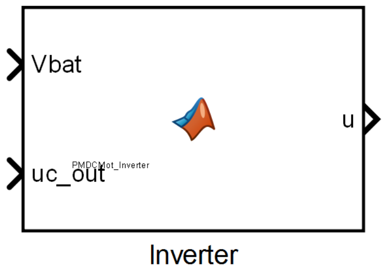


Figure 4.10: The DC inverter model in Matlab Simulink.

controller consists of two separate paths: the upper one (In1=y and In2=r) computes the error signal and integrates over it, while the lower path (In3=Ia and In4=wm) is merely proportional. A gain is later applied to both paths before the sum (with negative signs) is computed. The reasons for this design are given by the original author (Johannes FISCHER) as:

> This controller concept has been used because it has good capabilities in terms of closed-loop stability and set-point tracking. The gain vector K has been calculated by the formula of Ackermann after determination of the poles by the pole-placement method.

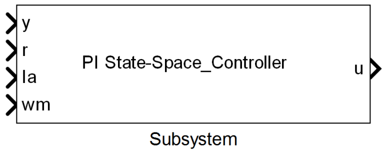


Figure 4.11: The PI State Space Controller model in Matlab Simulink.

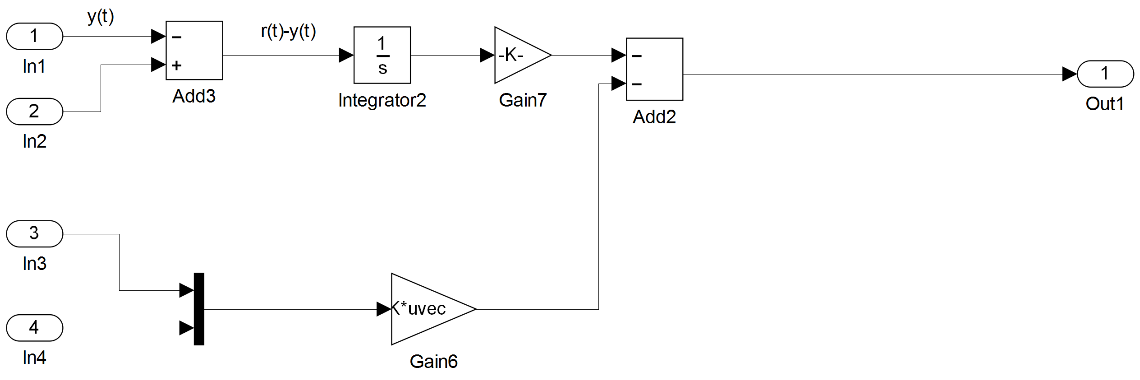


Figure 4.12: The internals of the PI State Space Controller submodel in Matlab Simulink.

4.6.1.3 Motor Model

The motor model shown in fig. 4.13 (and detailed in fig. 4.14) is a simple approximation to the behavior of a DC motor that takes a given voltage Vm and generates a torque wm that is proportional to that voltage. Doing so it consumes a current Ia depending on the load TL.

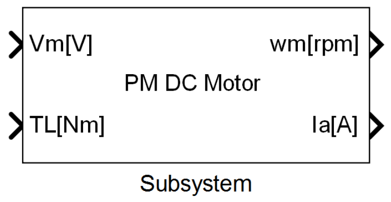


Figure 4.13: The DC Motor model in Matlab Simulink.

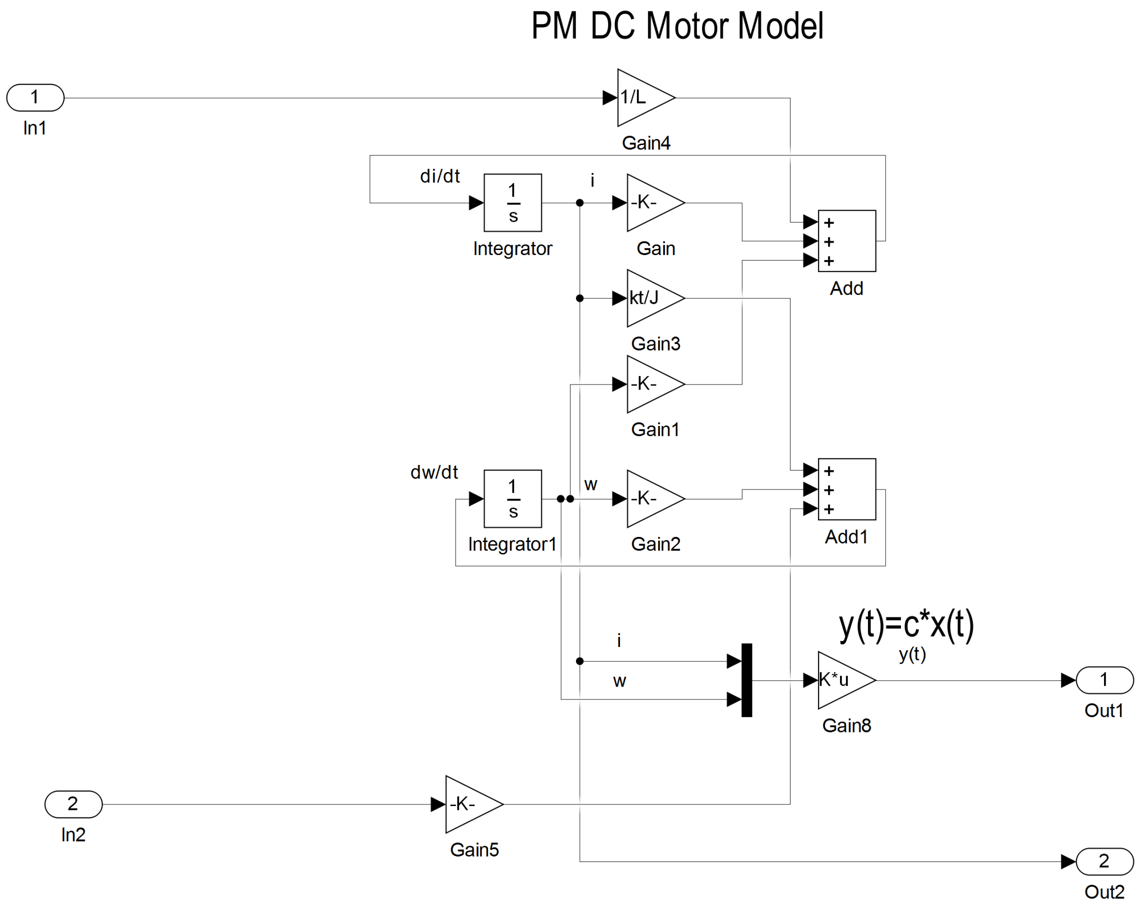


Figure 4.14: The internals of the DC Motor submodel in Matlab Simulink.

4.6.1.4 LiIon Battery

The battery model shown in fig. 4.15 is a detailed model of a lithium-ion battery pack. For this model the UML representation differs greatly as parameters are not modeled as ports but are passed to the factory as a key/value map. Therefore these ports do not exist in the UML model. Likewise the output buses (ending with a $\_n$ as well as Vbat) are not easily reproducible and are therefore omitted in the current SysCore model. Internally the model computes the new state of charge based on the last state of charge and the current drawn from the battery for each timestep as well as the changes in temperature based on this charge/discharge. From the temperature internal parameters such as resistance and capacitance are derived and with the help of the current state of charge the module voltage is computed.

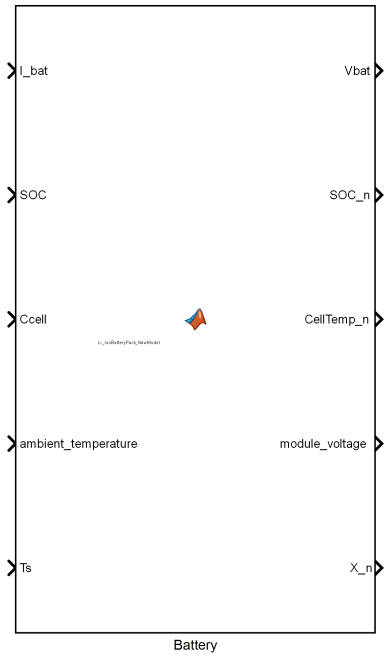


Figure 4.15: The LiIon Battery model in Matlab Simulink.

4.6.2 UML Model

Based on these components a set of equivalent UML model were created using the SHARC IDE. This model shown in fig 4.16 includes all the functionality of the original model excluding the stimuli and scopes will become part of the testbench.

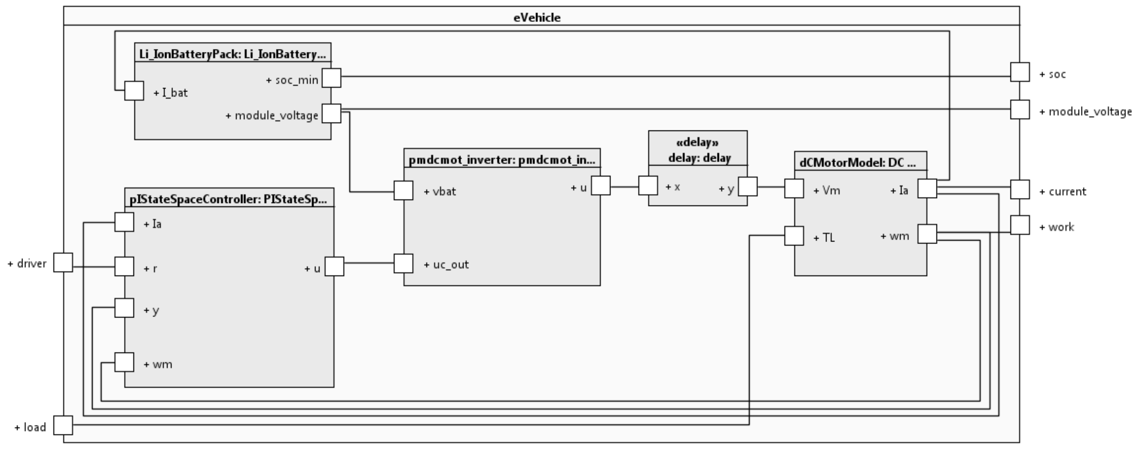


Figure 4.16: UML model of a battery power DC motor system for an e-vehicle.

4.6.3 Testbench generation

Using the SHARC IDE a testbench can be created automatically. The created testbench simply adds a scope to all ports of the DUT (device under test) and a stimuli generator to the inputs. The result can be seen in fig. 4.17.

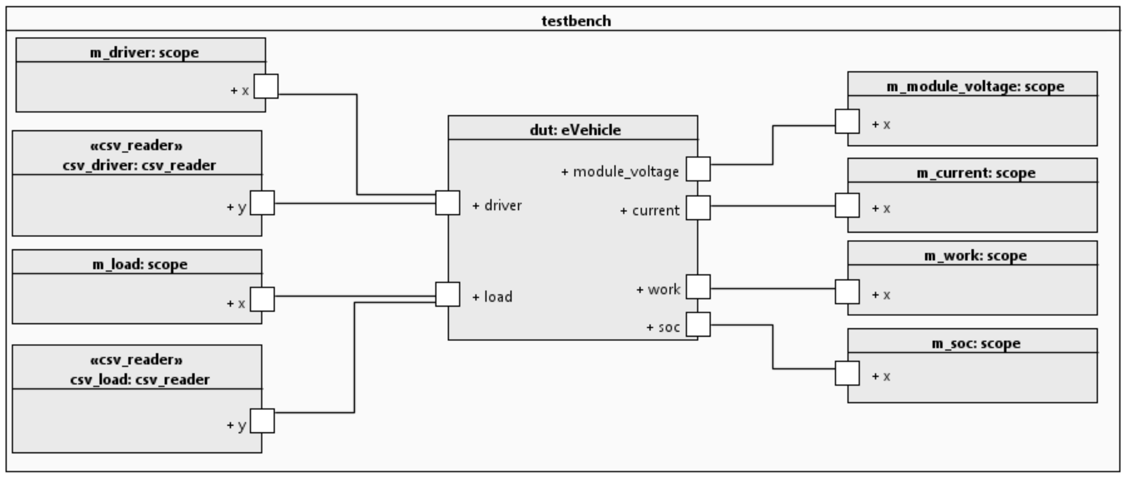


Figure 4.17: A simple testbench for the UML model from fig. 4.16 to reproduce the stimuli from the Simulink model.

4.6.4 Offline simulation

The next step was to verify that the created model behaves the same way as the original one. For this simulation the simulation core included with the SHARC IDE was used and the result is shown in fig. 4.18.

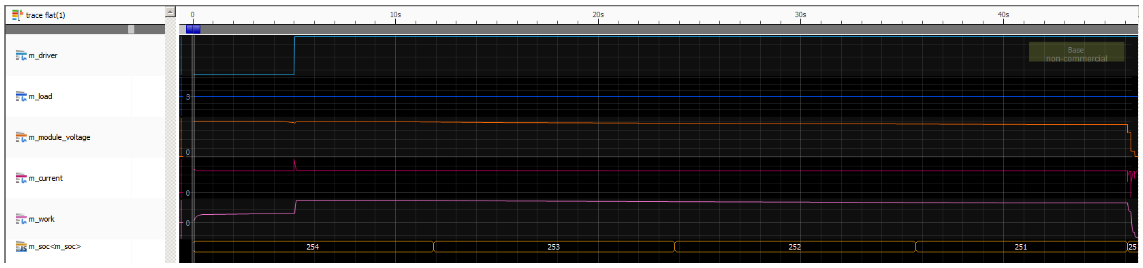


Figure 4.18: Results of the simulation using SHARC offline.

4.6.5 Online simulation

As the result of the offline simulation matches the original model within a reasonable small margin (due to slightly different timing and mathematical precision) the model can now be evaluated for different permutations of parameters on the internal components or external stimuli such as the load or driving maneuvers. As the model is taken out of context and the results would exceed the scope of this work no results will be shown here.

4.6.6 Results

While the results of the simulation are not relevant for this work, the achieved speedup is very relevant. Due to the limited amount of overlapping resources (mainly network and hard-disk) the simulations do not affect each other even when using three workers on the quad core machine used for testing. One core was reserved for the overhead of the virtual machine of the master and webserver as well as the overhead from the OpenStack installation on the machine. This resulted in a (near) linear speedup when using more than one worker as long as the number of simulations is reasonably high. For single simulations there is of course no speedup as the workers cannot share a single run.

Chapter 5

Conclusion and Future Work

In this work we presented a framework which uses a standardized modeling language (UML) in order to simulate systems from any number of domains. Due to the nature of SystemC AMS it can be used for purely digital systems (such as high level chip design) as well as analog and mixed systems such as a model of a vehicle. Due to the lack of a large cloud setup tests were only executed on a commodity hardware server. The speedup perceived for increasing the number of workers from one to three is linear with an impact of disk access on large traces (due to the use of a HDD instead of a SSD). While all technologies allow for near limitless scaling they were not tested in this setup. An extensive test deployment with an evaluation of the impact of network bandwidth, disk latency and throughput as well as the impact of the overhead of scaling will be evaluated in depth in future work. The framework as it is now serves mostly as a proof of concept and is not ready for use in a production environment. Further topics for future work to bring it closer to a production state and add features not implemented in the first version are:

- **Co-Simulation and Model Exchange:** As the frameworks plugin structure is already based on DLL files and a common interface it should be easy to adapt it to FMI. As mentioned this will impact performance when a model is spread over multiple systems therefore the main application would be SMP (symmetric multiprocessing, multiple cores or CPUs in a single machine) or HPC (the conventional cluster setup) where for example one core is used per model (e.g., two instances of SysCore on a single system simulating different domains and/or a conventional Matlab Simulink model exported with an FMI wrapper) and high speed low latency interconnects are used reduce the impact of the overhead.
- **Management:** The framework in its current state is fully ready for a public deployment as features such as user management and authentication are still missing. Once a proper user management is in place (preferably using a standard such as LDAP - the Lightweight Directory Access Protocol) RabbitMQ can use per user queues. This would give users the ability to manage their own tasks (move up in the queue or chancel un-

wanted tasks). Another benefit of this model is that compute time can be billed to individual users if necessary.

- **Remote Access:** While the development of a complete IDE based on web technologies would most outweigh the benefits for such a complex tool there are a number of ways to run the existing IDE in a web browser. Eclipse supports RAP (Remote Application Platform) which allows for a plugin or complete Workbench to run on a JEE (Java Enterprise Edition) server. Another option would be to use gtk3 broadway backent which also allows for any gtk3 based application to run in a web browser (although each user would have an instance of Eclipse running on the server). Either way only a single IDE would need to be maintained while giving users the choice to run it locally or on a remote server.
- **UVM:** While currently an UVM like (due to the lack of an official implementation of the UVM framework for SystemC AMS) approach is supported, once the Accellera published a SystemC AMS implementation this could easily be implemented into the framework.
- **Interactive Web-interface:** While the interface already contains a preview and is based on a responsive framework (scaling to any number of devices from mobile devices to public view screens) it lacks the ability for the user to interact with the data. An example would be a client side viewer for the traces similar to impulse that allows for simple features like zooming and cursors. As the design is based on standardized UML a viewer for the models would also be possible using client side scripting using a framework such as JointJS (which already contains some UML templates).
- **Plugin Management and Market:** The Eclipse platform used for the current IDE supports plugins and already contains a plugin manager. On top of this a ”App Store” could be developed to attract developers and end-users. This would allow users to share their models and components either free or possibly for a price. This could also be used to keep the platform up-to date by updating only components that differ from the currently installed version (reducing the complexity and download size in case of an update).

Appendix A

Examples

A.1 OpenMPI

Hello World Example

The following example is a simple ”hello world” example for a MPI application:

Listing A.1: hello.cpp

```cpp
#include "mpi.h"
#include <iostream>

int main(int argc, char **argv)
{
    int rank, size, len;
    char version[MPI_MAX_LIBRARY_VERSION_STRING];

    MPI::Init();
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "Hello, world! I am " << rank << " of " << size << std::endl;
    MPI::Finalize();

    return 0;
}
```

This demonstrates a few of main concept behind the MPI: The first MPI function that must always be called is `MPI::Init()`. It basically joins the current instance to the cluster. The Rank can be used to distinguish between roles for example rank one is the master which takes the input, spits it and passes it on to any node with a number larger than one. The last command of a MPI program usually is `MPI::Finalize()` which ends the current instance and tells the other MPI instances that this computer has successfully finished his task. Open MPI is a vast library which can itself be extended by plugins and comes with support for many network protocols and media, therefore it is conceivable that one could extend it for a cloud use case (or maybe such a plugin already exists) but it would most certainly not be the intended use case for this library. One can use anything from TCP over Ethernet (e.g., the internet or a local LAN) to Infiniband (a low latency interconnect for clusters). This is done without modification to the program but specified at launch. As for the compiler there is a simplified compile command which takes care of all the includes and linker specific settings:

```
$ mpic++ hello.cpp -o hello -g
$ mpirun --mca btl tcp,self -np 4 hello
```

Would first compile the example and then start the resulting binary called hello using tcp on loopback (the host itself) with 4 instances. The following is the output on a Windows machine using Cygwin and Open MPI with the above commands (the output of the version has been cut of as it contains no relevant information).

```
Hello, world! I am 0 of 4
Hello, world! I am 1 of 4
Hello, world! I am 2 of 4
Hello, world! I am 3 of 4
```

For a distributed approach this would be

```
$ mpirun -np 4 --hostfile my_hostfile hello
```

where a file called my_hostfile contains a list of all the hosts on which the program is to be executed. MPI also has its own datatypes as well as unicast (one to one) and broadcast (one to many) connections.

Communication Example

The following example is taken from the current version of OpenMPI (v2.x branch in git, comments have been removed for simplicity):

Listing A.2: ring.cpp

```cpp
// Copyright (c) 2004-2006 The Trustees of Indiana University and Indiana
//                         University Research and Technology
//                         Corporation.  All rights reserved.
// Copyright (c) 2006      Cisco Systems, Inc.  All rights reserved.

#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
    int rank, size, next, prev, message, tag = 201;

    MPI::Init();
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    next = (rank + 1) % size;
    prev = (rank + size - 1) % size;

    if (0 == rank) {
        message = 10;
        std::cout << "Process 0 sending " << message << " to " << next
                  << ", tag " << tag << " (" << size << " processes in ring)"
                  << std::endl;
        MPI::COMM_WORLD.Send(&message, 1, MPI::INT, next, tag);
        std::cout << "Process 0 sent to " << next << std::endl;
    }

    while (1) {
        MPI::COMM_WORLD.Recv(&message, 1, MPI::INT, prev, tag);
        if (0 == rank) {
            --message;
            std::cout << "Process 0 decremented value: " << message
                      << std::endl;
        }

        MPI::COMM_WORLD.Send(&message, 1, MPI::INT, next, tag);
        if (0 == message) {
            std::cout << "Process " << rank << " exiting" << std::endl;
            break;
        }
    }

    if (0 == rank) {
        MPI::COMM_WORLD.Recv(&message, 1, MPI::INT, prev, tag);
    }

    MPI::Finalize();
    return 0;
}
```

This example uses the communication mechanisms provided by MPI to pass a message from host to host. For simplicity only a single host with 5 processes will be used here (referred to instances from here on). For this the instances are arranged in a ring formation as shown in figure A.1 with the message originating from the instance with rank 0. Each node then listens to any incoming communication from the previous host with rank 0 listening to the instance with the highest rank. As rank 0 starts with sending a message

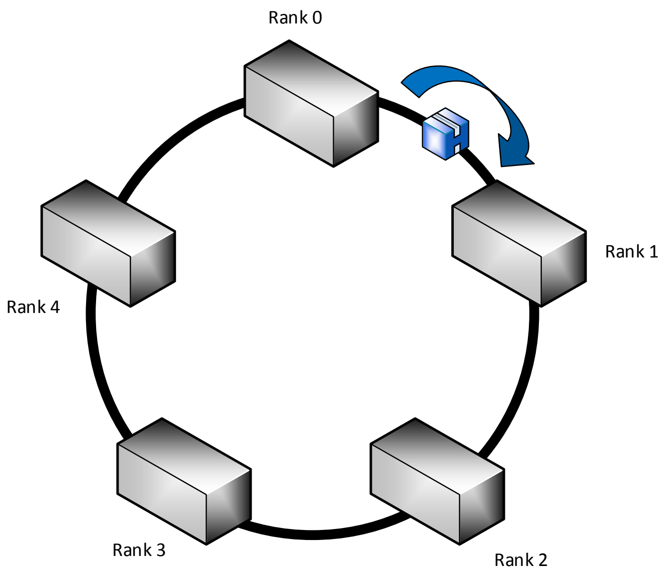


Figure A.1: Block diagram of a ring communication system using MPI.

(an integer with value 10) before entering the receive block rank 1 receives a message and passes it on. This happens up to the instance with the highest rank which then passes it to rank 0 again. Whenever the instance with rank 0 receives a message it decrements it before sending it again. If an instance passes on a message of value 0 it terminates afterwards (except the instance of rank 0 which listens for one more message in order to enable rank 4 to terminate as well). After compilation and execution the following output can be observed:

```
Process 0 sending 10 to 1, tag 201 (5 processes in ring)
Process 0 sent to 1
Process 0 decremented value: 9
Process 0 decremented value: 8
Process 0 decremented value: 7
Process 0 decremented value: 6
Process 0 decremented value: 5
Process 0 decremented value: 4
Process 0 decremented value: 3
Process 0 decremented value: 2
Process 0 decremented value: 1
Process 0 decremented value: 0
Process 0 exiting
Process 1 exiting
Process 2 exiting
Process 3 exiting
Process 4 exiting
```

A.2 FMI

The language used within the FMI specifications is typically C. While there are wrappers for different languages such as Java the concept usually remains unchanged: FMUs come in zip archives containing at least an XML file with the specifications as well as either the source of the model (and the solver) or a number of precompiled binaries (typically one DLL for each platform). The following example (bouncing ball) has been taken from the SDK provided by the FMI development group.

Listing A.3: ball.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="bouncingBall"
  guid="{8c4e810f-3df3-4a00-8276-176fa3c9f003}"
  numberOfEventIndicators="1">

<CoSimulation
  modelIdentifier="bouncingBall"
  canHandleVariableCommunicationStepSize="true"/>

<LogCategories>
  <Category name="logAll"/>
...
</LogCategories>

<ModelVariables>
  <ScalarVariable name="h" valueReference="0" description="height, used as state"
                  causality="local" variability="continuous" initial="exact">
    <Real start="1"/>
  </ScalarVariable>

...
</ModelVariables>

<ModelStructure>
  <Derivatives>
    <Unknown index="2" />
...
  </Derivatives>
  <InitialUnknowns>
    <Unknown index="2"/>
...
  </InitialUnknowns>
</ModelStructure>
</fmiModelDescription>
```

While some values are omitted (usually only showing one per node type) the structure of an XML is remarkably simple. The fmiModelDescription node contains textual information about the model such as the version of the FMI used in its creation as well as its name and an id. The model above is intended for co-simulation therefore the CoSimulation node contains information about the capabilities of the enclosed model. The remaining nodes contain information about internal and external variables and their initial states and derivatives.

Due to the size of the interface no listing will be given in this example but the fmuTemplate using within the FMU-SDK calls an update function which will be shown here due to its significance.

Listing A.4: ball.c (snippet)

```c
void eventUpdate(ModelInstance *comp, fmi2EventInfo *eventInfo, int isTimeEvent) {
    pos(0) = r(h_) > 0;
    if (!pos(0)) {
        r(v_) = - r(e_) * r(v_);
    }
    eventInfo->valuesOfContinuousStatesChanged   = fmi2True;
    eventInfo->nominalsOfContinuousStatesChanged = fmi2False;
    eventInfo->terminateSimulation   = fmi2False;
    eventInfo->nextEventTimeDefined  = fmi2False;
}
```

This function is first called after initialization (which sets initial values and set the next event time to a sane value if necessary) and computes the value of some variables reports to the master that values of variables have changed and it is not yet done with the simulation. No information about a specific next event is given as the equation is continuous. Compared with the values example of the FMU SDK:

Listing A.5: values.c (snippet)

```c
void eventUpdate(ModelInstance *comp, fmi2EventInfo *eventInfo, int isTimeEvent) {
    if (isTimeEvent) {
        eventInfo->nextEventTimeDefined = fmi2True;
        eventInfo->nextEventTime        = 1 + comp->time;
        i(int_out_) += 1;
        b(bool_out_) = !b(bool_out_);
        if (i(int_out_) < 12) copy(string_out_, month[i(int_out_)]);
        else eventInfo->terminateSimulation = fmi2True;
    }
}
```

Here the model tells the master that there is a specific time when the next event will be available. Therefore the master knows that the last value will remain valid until then. While the first example gives an overview of a continuous time model the second example could be found in a discrete event based model. As these models are intended for co-simulation the solver is embedded into these update functions and the master merely distributes a time among the slaves and updates the values at discrete communication points.

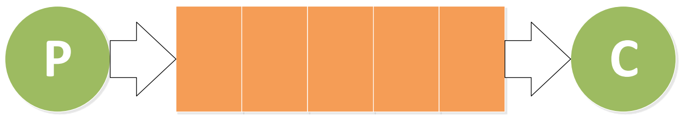


Figure A.2: Block diagram of a simple producer consumer example for AMQP.

A.3 AMQP

Hello World - Consumer Producer

Listing A.6: sender.py

```python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

This example connects to an AMQP broker such as RabbitMQ running on the same machine (localhost) and creates a new queue named hello (if it does not already exist). It then publishes a single message and quits.

Listing A.7: sender.py

```python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

channel.basic_consume(callback, queue='hello', no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

This example also connects to localhost and uses the same queue. It then continually consumes messages published to the queue. The callback function is called for each message received. This example does not use acknowledges nor any special exchange (such as fan out or similar).

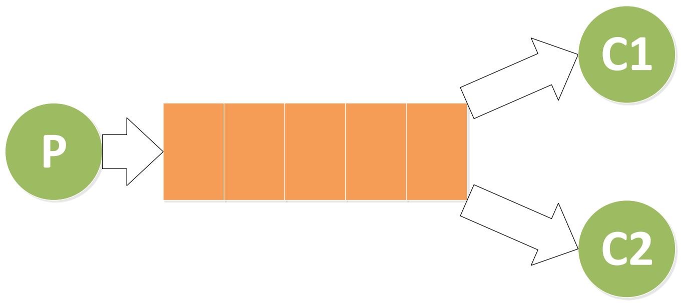


Figure A.3: Block diagram of a work queue example for AMQP.

Work queues

Work queues are a fundamental part of AMQP and already support retransmissions and acknowledges. As each message is consumed by a producer it will be reserved by the broker until it is either acknowledged (delete from the queue) or an error (e.g., a timeout) occurs which causes the broker to put the message back into the queue. Using this simple pattern a worker queue can be implemented.

Listing A.8: task.py

```python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='task_queue', durable=True)

message = ' '.join(sys.argv[1:]) or "Hello World!"
channel.basic_publish(exchange='', routing_key='task_queue', body=message,
      properties=pika.BasicProperties(delivery_mode = 2,))
print(" [x] Sent %r" % message)
connection.close()
```

This example reads the message from the parameters given to the program and published it to the queue task_queue. This time the queue is durable (meaning it persists through restarts and crashes of the broker) and messages are marked as persistent (delivery_mode=2). While this does not fully guarantee that no messages are lost (e.g., the message is accepted by the broker but it crashes during the write operation to the disk). This can be further improved by publisher acknowledges.

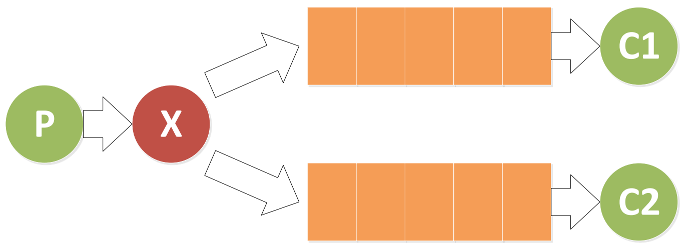


Figure A.4: Block diagram of a publish-subscribe example for AMQP.

Listing A.9: worker.py

```python
import pika
import time

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    time.sleep(body.count(b'.'))
    print(" [x] Done")
    ch.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue='task_queue')

channel.start_consuming()
```

This example is a modified version of the worker from the hello world. The main difference is that is also uses a durable queue and this time sends acknowledges to the broker if a task (message) is successfully consumed (by sleeping proportional to the message length). The second important difference is that using quality of service the prefech count is set to 1, meaning only one message is consumed at a time and none are prefeched from the broker. This ensures that the every worker only consumes one task and only after its completion consumes the next. Otherwise ever n-th worker would get every n-th message (independent of the amount of time it requires to finish a given task).

Publish-Subscribe

When using a task queue all messages are evenly distributed among all workers but every message may only be published to exactly one worker. This time every message is to be delivered to all workers. This pattern is also known as publish-subscribe.

Listing A.10: emit_log.py

```
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='logs', type='fanout')

message = ' '.join(sys.argv[1:]) or "info: Hello World!"
channel.basic_publish(exchange='logs', routing_key='', body=message)
print(" [x] Sent %r" % message)
connection.close()
```

This example is similar to the initial hello world example as it takes the parameters passed and publishes them to the broker. This time however it does not use a routing key but declares an exchange which is configured to fanout, meaning all incoming messages are copied to all attached consumers.

Listing A.11: receive_log.py

```
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='logs', type='fanout')

result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

channel.queue_bind(exchange='logs', queue=queue_name)

print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] %r" % body)

channel.basic_consume(callback, queue=queue_name, no_ack=True)

channel.start_consuming()
```

This example is declaring its own queue and binds it to the same exchange the previous example publishes to. As with the initial hello world example no acknowledges are uses in this example and the queue name is generated by the broker not given in advance.

Routing

The examples here are an extension of the publish-subscribe example with routing keys in order to filter all logs for a given severity. The severity an example listens to or subscribes to is given as parameter at startup. The main

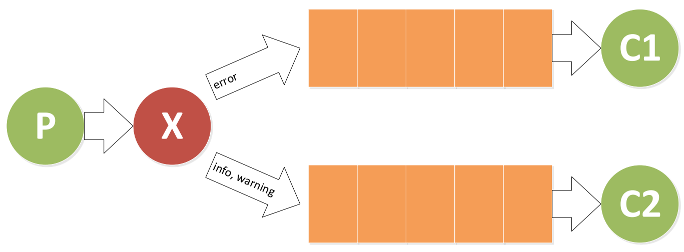


Figure A.5: Block diagram of a publish-subscribe example with routing keys for AMQP.

difference to the publish-subscribe example is that instead of a fanout exchange a direct one is used.

Listing A.12: emit_log.py

```python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='direct_logs', type='direct')

severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
message = ' '.join(sys.argv[2:]) or 'Hello World!'
channel.basic_publish(exchange='direct_logs', routing_key=severity, body=message)
print(" [x] Sent %r:%r" % (severity, message))
connection.close()
```

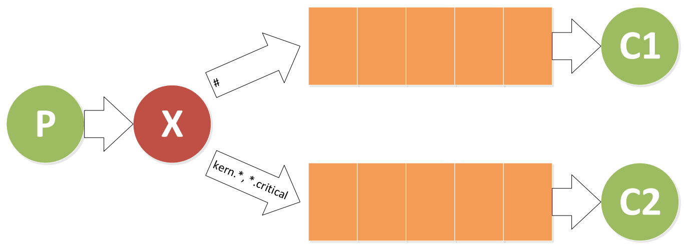


Figure A.6: Block diagram of a publish-subscribe example with topics for AMQP.

Listing A.13: receive_log.py

```python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='direct_logs', type='direct')

result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

severities = sys.argv[1:]
if not severities:
    sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
    sys.exit(1)

for severity in severities:
    channel.queue_bind(exchange='direct_logs',
                       queue=queue_name,
                       routing_key=severity)

print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] %r:%r" % (method.routing_key, body))

channel.basic_consume(callback, queue=queue_name, no_ack=True)

channel.start_consuming()
```

It is important to note that it is possible to bind the same queue and exchange using multiple keys as shown in above example.

Topics

This example takes once again extends on the publish-subscribe example but this time an exchange of the type topic is used. This allows for better filtering compared to the direct exchange used in the last examples. The syntax for this is similar usually the same for all topic based messaging frameworks (e.g., MQTT) with * being a wildcard for any number of characters and # meaning all messages in all topics. Topics support hierarchies such as kern.bus.usb.critical. While # receives all messages *.critical or kern.* would also work in this case.

Listing A.14: emit_log.py

```
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(
        host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='topic_logs',
                         type='topic')

routing_key = sys.argv[1] if len(sys.argv) > 1 else 'anonymous.info'
message = ' '.join(sys.argv[2:]) or 'Hello World!'
channel.basic_publish(exchange='topic_logs',
                      routing_key=routing_key,
                      body=message)
print(" [x] Sent %r:%r" % (routing_key, message))
connection.close()
```

Listing A.15: receive_log.py

```
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='topic_logs', type='topic')

result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue

binding_keys = sys.argv[1:]
if not binding_keys:
    sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
    sys.exit(1)

for binding_key in binding_keys:
    channel.queue_bind(exchange='topic_logs', queue=queue_name, routing_key=
         binding_key)

print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] %r:%r" % (method.routing_key, body))

channel.basic_consume(callback, queue=queue_name, no_ack=True)

channel.start_consuming()
```

A.4 Webservice JSON

This is an example of JSON response from the Flask webservice API. The content is the same as presented by the web-interface shown in fig 4.7.

Listing A.16: JSON API

```
{
  "config": "PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiIHN0YW5kYWx
        vbmU9Im5vIj8+PHN5c2NvcmU+PGNvbmZpZz48cGFyYW0gdHlwZT0ibW9kZWxfbm9k
        ZSIgdmFsdWU9IlBsYXlncm91bmQiLz48cGFyYW0gdHlwZT0ibW9kZWxfZmlsZW5hb
        WUiIHZhbHVlPSJwbGF5Z3JvdW5kLnVtbCIvPjxwYXJhbSB0eXBlPSJkdXJhdGlvbi
        IgdmFsdWU9IjEwIi8+PHBhcmFtIHR5cGU9InRpbWVzdGVwIiB2YWx1ZT0iMC4wMDE
        iLz48cGFyYW0gdHlwZT0idmVyYm9zZSIgdm FsdWU9ImZhbHNlIi8+PC9jb25maWc
        +PG92ZXJyaWRlPg0KCTwvb3ZlcnJpZGU+PC9zeXNjb3JlPg==",
  "created": "2015-12-29T07:23:51",
  "id": 96,
  "name": "d975537d-93c9-4674-885b-5b783c32271d",
  "stderr": "CiAgICAgICAgU3lzdGVtQyAyLjMuMS1BY2NlbGxlcmEgLS0tIERlYyAxNCA
        yMDE1IDE0OjAyOjI1CiAgICAgICAgQ29weXJpZ2h0IChjKSAxOTk2LTIwMTQgYnkg
        YWxsIENvbnRyaWJ1dG9ycywKICAgICAgICBBTEwgUklHSFRTIFJFU0VSVkVECklNlY
        XJjaGluZyBmb2xkZXIgL3Vzci9zaGFyZS9zeXNjb3JlL3BsdWdpbnMgZm9yIHBsdW
        dpbnM6ClNlYXJjaGluZyBmb2xkZXIgL3Vzci9sb2NhbC9zaGFyZS9zeXNjb3JlL3B
        sdWdpbnMgZm9yIHBsdWdpbnM6ClBsdWdpbjpsb2FkIGxvYWRlZCEKUGx1Z2luOmFk
        ZCBsb2FkZWQhClBsdWdpbjpjb25zdGFudCBsb2FkZWQhClBsdWdpbjpkZWxheSBsb
        2FkZWQhClBsdWdpbjpkcml2ZXIgbG9hZGVkIQpQbHVnaW46cG1kY21vdF9pbnZlcn
        RlciBsb2FkZWQhClBsdWdpbjpnYWluIGxvYWRlZCEKUGx1Z2luOmNzdl9yZWFkZXI
        gbG9hZGVkIQpQbHVnaW46UElfU3RhdGVTcGFjZV9Db250cm9sbGVyIGxvYWRlZCEK
        UGx1Z2luOnNjb3BlIGxvYWRlZCEKUGx1Z2luOmRjX21vdG9yX21vZGVsIGxvYWRlZ
        CEKUGx1Z2luOmNvbXBhcmUgbG9hZGVkIQpQbHVnaW46TGlfSW9uQmF0dGVyeeVBhY2
        sgbG9hZGVkIQpQbHVnaW46c3RvcCBsb2FkZWQhClBsdWdpbjppbnRlZ3JhdG9yIGx
        vYWRlZCEKU2VhcmNoaW5nIGZvbGRlciAuL3BsdWdpbnMgZm9yIHBsdWdpbnM6CgoK
        ICAgICBTeXN0ZW1DIEFNUyBleHRlbnNpb25zIDEuMC4xIFZlcnNpb246IDEuMC4xI
        C0tLSBCdWlsZFJldmlzaW9uOiAxNzYyICAyMDE0XzA0XzAxCiAgICAgICAgICBDb3
        B5cmlnaHQgKGMpIDIwMTAtMjAxNCAgYnkgRnJhdW5ob2Zlci1HZXNlbGxzY2hhZnQ
        KICAgICAgICAgICAgICAgICAgICBJbnN0aXR1dCBJbnRlZ3JhdGVkIENpcmN1aXRzIC8g
        RUFTCiAgICAgICAgICAgIExpY2Vuc2VkIHVuZGVyIHRoZSBBcGFjaGUgTGljZW5zZ
        SwgVmVyc2lvbiAyLjAKCgptZXJnaW5nIHNpZ25hbCBpbiBwb3J0OiBfSnplWjRLMz
        dFZVdSU3NGSW50Wi1hZyNfYUVrZ2dHdGxFZVdHV0xaSmRITGFpZwptZXJnaW5nIHN
        pZ25hbCBpbiBwb3J0OiBfdVFfZjBLMzdFZVdSU3NGSW50Wi1hZyNfQkJIZ2dQNUFF
        ZVM1bko5YmlmTEtNQQo=",
  "stdout": "Ojp2ZXJib3NlIHNwZWNpZmllZCB0byBmYWxzZQpTdGFydGluZyBTaW11bGF
        0aW9uIChtYXguIGR1cmF0aW9uOiAxMHMsIHRpbWVzdGVwOiAwLjAwMXMpLi4uClNp
        bXVsYXRpb24gb2YgMTAgc2Vjb25kcyBmaW5pc2hlZCBpbiAwLjE5OCBzZWNvbmRzI
        HJlYWwgdGltZS4K",
  "task_id": 29
}
```

This listing is the result of calling the JSON API using a webbrowser. All strings are encoded using a base64 encoding to ensure that the source encoding (e.g., utf-8) is not altered when storing it into the database.

Bibliography

[1] The Eclipse Foundation., "Eclipse IDE," 2016. [Online]. Available: https://eclipse.org/

[2] UVM Working Group., "UVM (Standard Universal Verification Methodology)," 2014. [Online]. Available: http://www.accellera.org/downloads/standards/uvm

[3] Object Management Group, "The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems," 2013. [Online]. Available: http://www.omgmarte.org/

[4] EAST-ADL Association, "About EAST-ADL," 2014. [Online]. Available: http://www.east-adl.info/Specification.html

[5] Synopsys, "EDA in the Clouds: Myth Busting," Issue 2, 2011. [Online]. Available: https://www.synopsys.com/Company/Publications/SynopsysInsight/Pages/Art6-Clouds-IssQ2-11.aspx?cmp=Insight-I2-2011-Art6

[6] H. Ranjan, "Cloud computing and EDA: Is cloud technology ready for verification (and is verification ready for cloud)?" in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, April 2011, pp. 1–2.

[7] Cadence, "EDA Inches Closer to Cloud Computing," 2013. [Online]. Available: http://community.cadence.com/cadence_blogs_8/b/fullerview/archive/2013/09/02/eda-inches-closer-to-cloud-computing

[8] Cadence, "Software as a Service (SaaS) Lowers the Bar for IC Design," 2013. [Online]. Available: http://community.cadence.com/cadence_blogs_8/b/ii/archive/2013/07/01/software-as-a-service-saas-lowers-the-bar-for-ic-design

[9] D. Paul, M. S. Nakhla, R. Achar, and N. M. Nakhla, "Parallel Circuit Simulation via Binary Link Formulations (PvB)," *IEEE Transactions on Components, Packaging and Manufacturing Technology*, vol. 3, no. 5, pp. 768–782, May 2013.

[10] newelectronics, "As the Software as a Service model develops, will EDA take advantage?" 2015. [Online]. Available: http://www.newelectronics.co.uk/electronics-technology/as-the-software-as-a-service-model-develops-will-eda-take-advantage/89231/

[11] C. Man, Z. Shi, Z. Xu, Y. Zong, K. Pang, and Y. Li, "Cloud-EDA: a PaaS platform architecture and application development for IC design & test," in *Cloud Computing and Internet of Things (CCIOT), 2014 International Conference on*, Dec 2014, pp. 1–4.

[12] V. Kamath, R. Giri, and R. Muralidhar, "Experiences with a Private Enterprise Cloud: Providing Fault Tolerance and High Availability for Interactive EDA Applications," in *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, June 2013, pp. 770–777.

[13] D. Agarwal and S. K. Prasad, "AzureBOT: A Framework for Bag-of-Tasks Applications on the Azure Cloud Platform," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 2139–2146.

[14] S. C. Silva, T. G. Carneiro, J. C. Lima, and R. R. Pereira, "TerraME HPA: Parallel Simulation of Multi-agent Systems over SMPs," in *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM PADS '13. New York, NY, USA: ACM, 2013, pp. 361–366. [Online]. Available: http://doi.acm.org/10.1145/2486092.2486141

[15] A. de Graaf and EEMCS-CAS, "SystemC-AMS Analog & Mixed-Signal System Design," 2014. [Online]. Available: http://ens.ewi.tudelft.nl/Education/courses/et4351/SystemC-AMS-14v1.pdf

[16] G. Christoph, D. Markus, H. Jan, O. Jiong, and Z. Yaseen, "Refinement of embeddedanalog/mixed-signal systems with SystemC-AMS," 2008. [Online]. Available: http://www.systemc-ams.org/documents/date08-tutorial-systemc-ams-1.3.pdf

[17] accellera, "Standard SystemC-AMS® AMS extensions 2.0 language reference manual," 2013. [Online]. Available: http://www.accellera.org/images/downloads/standards/systemc/SystemC_AMS_2_0_LRM.pdf

[18] W3C, "Extensible markup language (XML) 1.0 (fifth edition)," 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[19] Gartner, "2015 Magic Quadrant for Cloud Infrastructure as a Service," 2015. [Online]. Available: https://aws.amazon.com/resources/gartner-2015-mq-learn-more/

[20] M. Schachner, "A novel approach to enhance cloud based verification for SHARC," Institute of Technical Informatics, Graz University of Technology, Tech. Rep., 2016.

[21] M. Schuß, "SHARC - SysCore," Institute of Technical Informatics, Graz University of Technology, Tech. Rep., 2015.

[22] CISC Semiconductors and Fraunhofer IIS, "AVTR Project," 2012. [Online]. Available: http://www.avtr-project.eu/