Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science (60 credits) in Computer Science with Specialization in Software Engineering

TOOL ORCHESTRATION FOR MODELING, VERIFICATION AND ANALYSIS OF COLLABORATING AUTONOMOUS MACHINES

Pavle Mrvaljević
pmc19001@student.mdh.se

Examiner: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Supervisors: Marjan Sirjani and Anas Fattouh
Mälardalen University, Västerås, Sweden

Company supervisor: Stephan Baumgart
Volvo CE, Eskilstuna

May 20, 2020

Abstract

System of systems (SoS) is a collective of multiple system units that have a common purpose. In this thesis, the Volvo Electric Site is investigated as an example case study in which safety and performance properties of collaborating autonomous machines are evaluated and analyzed. Formal methods in software engineering aim to prove the correctness of the system by evaluating its mathematical model. We use an actor-based framework, AdaptiveFlow, for modeling system functionalities and timing features. The aim is to link an abstract model evaluation and a simulation of real-world cases that are deployed in the VCE Simulator. In addition, it is necessary to make sure that AdaptiveFlow provides correct-by-design scenarios. The verification is conducted by developing an orchestration method between the AdaptiveFlow framework and the VCE Simulator. A tool named VMap is developed throughout this thesis for automated mapping of the input models of AdaptiveFlow and the VCE Simulator to make the orchestration possible. Furthermore, AdaptiveFlow is perceived in two different ways, as a design tool, and as an analysis tool. The models created in AdaptiveFlow are directly mapped to the VCE Simulator by using the VMap tool where the VCE Simulator is used as a testbed for checking these models. The outcome of this thesis is reflected in the establishment of a mapping pattern between AdaptiveFlow inputs and VCE simulator by developing the VMap tool for automatic mapping. It was shown that there is a natural mapping between the AdaptiveFlow models and VCE simulator inputs. By using VMap, we can quickly get to the desired scenarios. Through the development of three different cases, the results show that it is possible to design safe and optimal scenarios by orchestrating the AdaptiveFlow and the VCE Simulator using the VMap tool as well as the correlation between results from AdaptiveFlow and VCE Simulator.

Table of Contents

List of Figures

1. Introduction

Collaborative systems, commonly referred to as system-of-systems (SoS), represent a collective of independent systems that cooperate to achieve a common goal [1]. Each unit of the SoS is task-oriented and these tasks are executed in the collaboration with other units. Taking into account the complexity of the SoS we should use a suitable verification method which would ensure that the system matches the desired design specification. In order to design such systems, the techniques for the analysis should be defined. The most common techniques used are testing, simulation, assertion check, lightweight formal verification, and statistical model checking. Considering the complexity of collaborative systems, designing models for different levels of abstraction would prove helpful. As there are many unpredictable factors in this system it is desirable to build adaptive models that would be able to respond to these obstacles. Due to the fact that modern machines have embedded software, it is possible to create a collaborative system that would automatize the workflow. For this thesis, it is envisioned to conduct modeling and analysis of collaborating autonomous heavy machines for the Volvo construction site in order to check their safety and performance properties.

In order for the appropriate model to evolve, it is necessary to consider all the characteristics of the system. The main focus is on the modeling of the fleet control system for the scenario developed for the VCE Simulator. The simulator provides a convenient representation of real-world cases through which the performance and safety would be evaluated.

There are various modeling languages and techniques available with different characteristics and supporting tools. Since the nature of this system is distributed and concurrent we should use appropriate tools for modeling and analysis. The suitable paradigm which would address this is the usage of actor models introduced by Hewitt [2]. This actor model has been later developed as an object-based functional language by Agha [3]. In this case, Rebeca modeling language is used due to the fact that it targets distributed and concurrent systems and its capability to execute assertion checks and deadline misses. Rebeca (Reactive Objects Language) is an open-source actor-based modeling language with a formal foundation that has a purpose to bridge a gap between software engineers and formal method community [4]. It has a syntax similar to Java and it is a platform for developing object-based concurrent systems that are supported by a model checking toolset [1]. Additionally, Rebeca has a feature to model timing constraints by using its real-time extension, Timed Rebeca [5]. Modeling and analyzing this collaborative system is executed with the AdaptiveFlow framework. This framework is based on the Eulerian model and it makes modeling and analyzing track-based flow management systems possible [6]. In order to verify safety and performance properties, the VCE simulator is seen as a testbed.

To conduct this experiment first it is necessary to build TRebeca models with the AdaptiveFlow framework. In this framework, tracks are modeled as actors and moving objects (autonomous machines) as messages [6]. Next, the checking of performance and safety properties needs to be executed. Safety and progress properties include the minimum distance between the machines, relation between the speed of a machine and its distance from others, and deadlock checks. The performance properties comprise the system throughput of several collaborating machines and the least waiting time for a service request to be completed. It is important to adjust mentioned properties with what VCE simulator can provide. The third step is to map specified properties to the VCE simulator and verify if the properties hold. Finally, the mapping between AdaptiveFlow inputs and VCE simulator inputs needs to be automatized.

In this work, we propose a tool orchestration method in which we use both AdaptiveFlow and VCE Simulator as a complete tool stack for verification and evaluation of collaborating autonomous haulers. To retrieve results and to provide a depth of this research, we conducted experiments on three different cases. Each case takes a different approach to tool orchestration to demonstrate the variety of methods of applying these tools.

The main goal is to close the loop in model-driven development in which the usability of AdaptiveFlow is shown. The 2D model abstraction is lowered to a more realistic representation of the system in order to see whether there is a correlation in results. The mappable attributes/elements are recognized by which the mapping procedure has been developed. The main challenges and limitations have also been represented.

1.1. Problem Formulation

In this project, the case study of the electrified site project by Volvo Construction Equipment is considered. This case is based on collaborating autonomous machines, referred to as automated haulers, shown in Figure 1, which purpose is to perform several tasks such as loading, unloading, charging and material transport by operating in the fleet manner [7]. Autonomous machines have a centralized fleet control server that sends information to the machines in a certain time interval.

Figure 1: Automated Hauler

In this case, AdaptiveFlow is perceived as both, design and analysis tool for collaborative systems which generates TRebeca models. On the other hand, VCE Simulator acts as a testing environment for modeled systems. The scope of this thesis includes several objectives. The first step is to analyze the VCE Simulator by developing a suitable TRebeca model of the quarry site scenario similar to the one on Figure 2 with machine operations and conduct safety and performance checks. For this part, it is necessary to design an adequate environment in which the spots for points of interest (PoIs), as well as available segments of the environment, would be defined. This is done with the AdaptiveFlow framework. Furthermore, the configuration of the whole system should be specified. Based on the TRebeca model generated from defined specifications, it is required to perform a verification of that model. If the verification proves to be successful, the next step is to investigate how to perform a mapping between AdaptiveFlow inputs and VCE simulator. Based on this we will run the simulator and see whether the mapping was successful. The final phase is to discover how to automatize this mapping process.

Figure 2: Volvo Electric Site

1.2. Research Questions

The general question is if AdaptiveFlow framework (the design and analysis methods) is effective for model-based development of fleet management systems. This framework operate at an abstract level and to examine its effectiveness we need to see if the analysis results are valid during the simulation when we add more concrete and detailed features. This question can be broken down into the following research questions.

- **RQ1: How usable is the design model of AdaptiveFlow for model-based development of fleet management systems?**

  The difference in the level of abstraction in the two tools introduces limitations and challenges. To examine the usability we need to check how challenging it is to map TRebeca models to more concrete models in the VCE simulator. Considering the implementation within the VCE Simulator and the nature of AdaptiveFlow, it is necessary to come up with an answer to what the limitations are when it comes to mapping. To answer this question, on one hand, we must take into account the complexity of the environment in the VCE Simulator, which is actually a depiction of the real system in 3D. On the other hand, the system in AdaptiveFlow is represented as a 2D abstraction. We have to find a common ground between the two in order to carry out appropriate mapping. Taking this into account, we can concretize the main question into following sub-questions:

  - What is the mapping pattern between AdaptiveFlow inputs and VCE Simulator inputs?
  - What are the major challenges and limitations in building this automated mapping?

- **RQ2: How much the safety and performance properties verified by AdaptiveFlow are valid for cooperative construction machines in the VCE Simulator?**

  Since the AdaptiveFlow framework provides, with the design of TRebeca models, their verification and performance evaluation, it is necessary to find a way to analyze these properties inside the VCE Simulator. To answer this question, it is also necessary to perform an implementation within the VCE simulator that matches the inputs and logic defined in AdaptiveFlow. This question can be divided into sub-questions below.

  - Are the design analysis results of AdaptiveFlow precise enough to be relied on and to be used for model-based development?
  - How effective and precise can the results of analysis performed by AdaptiveFlow be?

1.3. Research Contributions

The contributions of the thesis that are the result of solving the problems described in Section 1.1. are listed below:

- **Patterns for mapping inputs from AdaptiveFlow to VCE Simulator**

  In order to develop a mapping procedure, we extract mappable elements from the AdaptiveFlow inputs to VCE Simulator. These elements were identified through the reverse engineering process in Section 6.1. The main problem is to find out how to transform a concept of the 2D environment of AdaptiveFlow to a complex 3D simulation scenario.

- **The VMap tool developed for automating the mapping procedure**

  One of the aims of this thesis is not only to derive an algorithm that could be implemented for the mapping procedure but to also provide a proof of concept by developing a suitable tool. The major applicative contribution of this thesis work is the VMap tool that represents a link between an abstract representation of a collaborative system from AdaptiveFlow and a simulation that provides a representation of real-world cases. This tool could be viewed as a piece of the puzzle for the verification of collaborative systems in which multiple tools and frameworks are used. It uses AdaptiveFlow inputs as well as complementary information from the VCE Simulator input file (dynamic.content) to generate a 2D representation of VCE elements, such as vertices and edges, and a *dynamic.content* example file.

- **Methods for evaluating collaborative system features through model verification and simulation**

  In this thesis, we have also dealt with the verification and evaluation of systems of the autonomous machines. This is done through orchestration of the AdaptiveFlow and the VCE Simulator. More explicitly, it is conducted by extracting the safety and performance results from the TRebeca models as well as from the simulation. The main problem is to find performance properties that can be measured in AdaptiveFlow as well as in the Simulator. The comparison of identified performance results is made on the basis of which the conclusions are drawn.

2. Background

To be able to understand the research contributions of this work, in this section a few crucial areas have been described. In this thesis, we have touched on topics such as actor-based modeling, system-of-systems, Automated Guided Vehicles (AGVs), formal verification and the tools which are used in order to realize this research. Rebeca modeling language will be explained in depth. In addition to that, we will go through the tools necessary to perform a model verification.

2.1. Model-Driven Development

Model-Driven Development (MDD), also referred to as Model-Driven Engineering (MDE), is a field in software engineering which has an aim to an abstract software system, simplify the development process and lower the cost of the implementation. The *model* in MDD is a representation of the system, including its structure and functionalities. Due to its simplicity, it is a good link in communication between engineers and clients. Thomas et al. [8] have derived the goals of MDD and some of them are listed below.

- Increase of development speed - This is achieved due to the automation of generating code from defined models. Through one or more steps of transformation, this automation is achieved.
- Enhancement of software quality - Through previously mentioned steps of transformation and usage of formal languages, the software quality will be enhanced since the modules from the software architecture will reappear in the implementation.
- Higher level of reusability - As properties such as system architecture, modeling language and transformation have been specified, they could be used as a foundation for other similar projects which increases the level of reusability.
- Manageability of complexity through abstraction - Often, there is an increase in system complexity, so it is necessary to simplify the representation of the system, which MDD does through abstracting system components and structure.

In this work, MDD was performed as we have created models of the concurrent, distributed system of autonomous machines by using the Rebeca modeling language which has been described in the upcoming sections.

2.2. Formal Methods

In software engineering, formal methods represent a set of techniques for development, verification and analysis of software systems. These techniques allow a more reliable design of complex systems by introducing a more mathematically-oriented approach to software development. Due to the discontinuous nature of software systems, minor changes could cause huge and unpredictable errors which highlight the difference in software engineering from other engineering disciplines [9]. This is why formal methods are essential for developing a robust and correctly designed system. When developing autonomous systems, such as the one covered in this thesis, the specification is arguably the biggest bottleneck in performing the verification and validation [10].

Even though formal methods do not guarantee the development of systems that are prone to errors, they do help in developing a greater understanding of the system, by pointing out the inconsistencies in system design. These flaws are of great importance due to the consequences that they could cause [11]. In our case, we are modeling a system that manages autonomous machines. In this field, suitable formal methods are imperative since they would decrease the cost of development of these systems to a great extent.

Formal methods can be categorized in several different ways. In [12] formal methods have been differentiated as *lightweight vs. heavyweight* and the application of formal methodologies has been defined *horizontal* and *vertical*. By the **lightweight** formal methods we refer to partial specification and do not require high expertise, which is in contrast to the **heavyweight** formal methods. The **horizontal** application of formal methods implies a formally designed software, whereas the **vertical** application is pointed to a correct-by-construction approach.

2.2.1. Formal specification

In systems development, formal specification is a set of techniques used for a better understanding and safer construction of systems based on mathematical representation [13]. These techniques are used to lower the cost of system construction and to prevent faults that would be more difficult and expensive to correct in further stages of development. In formal verification, multiple aspects of the system are covered. It can be used for multiple purposes such as better grasp of the system functionalities, analysis of properties of interest and directing the development process [13].

2.2.2. Formal Verification

Formal verification is a method used for proving the system correctness by verifying specific properties of the system or by examining its alignment to the formal specifications. It is performed on the mathematical model of the system that is an abstract representation. Timed automata, finite state machine, Petri nets, vector addition systems are just some of the examples of these mathematical models [14]. The approach used in this thesis is *model checking*. Generally, this approach is applied to finite models. As in Rebeca, model checking goes through the whole state space of the system and explores all of its states and transitions.

The model checking is performed by verification of properties defined as temporal logic. The most common types of properties are linear temporal logic (LTL), Property Specification Language (PSL) and computational tree logic (CTL). LTL represents time as a sequence of states which expand infinitely to the future [14]. To keep the explanation simple, we will make LTL formulations as following. If G represents all future states, F stands for some future state and p is some property of a system, this can be noted as shown in Equations 1 and 2.

$$G(p) \tag{1}$$

$$F(p) \tag{2}$$

If we want to define that whenever property p is true globally in the system, the property q will be finally true, we could formulate it as in Equation 3 if we use notation as in [14].

$$G(p \rightarrow Fq) \tag{3}$$

In this thesis, we will use Rebeca model checking tools for the verification of models generated from AdaptiveFlow.

2.3. Actor model

For the purpose of explaining the **Rebeca** modeling language and the **AdaptiveFlow** framework, it is crucial to comprehend the concept of actor-based modeling and the most important features of this paradigm. Actor-based modeling represents a higher abstraction level of concurrent computation in which actors are presented as universal primitive [2]. In these models, main units are actors that communicate with other actors through asynchronous message transmission. The states and behaviors of actors are encapsulated and they have ability to create new actors, change behavior and redirect communication link through an exchange of actor identities [15]. This model was invented by Carl Hewitt [2], afterward developed as a concurrent object-based language [3] and finally formalized by Agha et al. [16].

2.4. Rebeca Modeling Language

Reactive Objects Language (Rebeca) [15, 17] represents an actor-based modeling language which is used for modeling concurrent and distributed systems, with formal verification support. The main characteristics of the Rebeca model are similar to the actor model, considering that it has asynchronous message passing between reactive objects referred to as **rebecs**. Rebecs have an unbounded buffer, labeled as a message queue, which contains all of the messages related to it. The computation in Rebeca is event-driven and the *rebecs* are taking the messages (events) from

the top of the queue [17]. Execution of functions is performed atomically which means that they are not interleaved with other functions [17].

The structure of the Rebeca Model is shown on Listing 1. When defining reactive class, we need to declare queue size, even though that does not match to principles of actor modeling since queue size in that paradigm is unbounded. Another two declarations which are defined are **knownrebecs** and **statevars**. Knownrebecs represent other reactive objects which communicate with declared rebec. The statevars can be perceived as variables that represent the current state of the rebec. Message servers, labeled as **msgsrv**, are methods used to handle incoming messages from other rebecs. Finally, there are methods that are executed locally inside the message server and method of the same rebec.

```
reactiveclass Rebec1(2)
{
    knownrebecs Rebec2 d;
    statevars{}

    /* the constructor */
    Rebec1()
      { self.msg1();
      }

    /* Handling message 1*/
    msgsrv msg1()
      { d.askForService();
      }

    /* Handling message 2*/
    msgsrv msg2()
      {
      ....
      }
}
```

Listing 1: Class definition in Rebeca

2.4.1. Timed Rebeca

There is an extension of Rebeca which has a feature of defining timing constraints. This extension is called Timed Rebeca and it is used within the AdaptiveFlow framework. The features addressed with this extension are **computation time**, **message delivery time**, **message expiration** and **periods of occurrences of events** [5]. Each rebec has a local time by which new statements are being handled, but also there is a global time used for rebec synchronization. Modeling with Timed Rebeca still includes non-preemptive (atomic) execution of methods, but it also includes modeling of passing time [5]. New primitives which are included in this extension are *delay*, *deadline*, *deadline* and *now*. The *delay* keyword represents a time for the method execution, whereas *deadlines* could be put on messages which means that messages will not be valid after the expiration of the deadline. Local time of the rebec can be retrieved with the statement *now* and the *after* statement assigns that the message cannot be taken from the message bag before determining time [5].

2.4.2. Rebeca model checking tools

In this section, we are going to take a brief overview of the tools used for the verification of Rebeca models. Two tools that are mainly used are Rebeca Model Checker (RMC) and Afra IDE.

Rebeca Model Checker (RMC)

Throughout this project, we will use RMC as a tool for performing a direct model verification. This tool consists of the Modere tool which is used for the linear temporal logic (LTL) model checking. It uses an automata-theoretic method which means that system and its negation are defined with a Buchi automation [18]. The result of these two automata will be empty if the system satisfies the specification. RMC generates C++ classes that represent automata for the system and the

negation of its specification [18]. These files and the engine of Modere must be compiled to deliver the executable file which performs model checking of the Rebeca model and produces a state space file.

Afra

Afra is an IDE used for property specification, model-checking and counter-example visualization of Rebeca models [18]. The user interface of this IDE consists of the project browser, and model and property editor. In Afra, for performing a model verification, we need to specify two files, model and property file. In the model file, we develop a Rebeca model whose properties need to be satisfied. In the property file, we specify LTL or CTL properties which are later used for model verification. Afra is developed as a symbiosis between Sytra, Modere and Symon including the Rebeca and SystemC modeling environments [18].

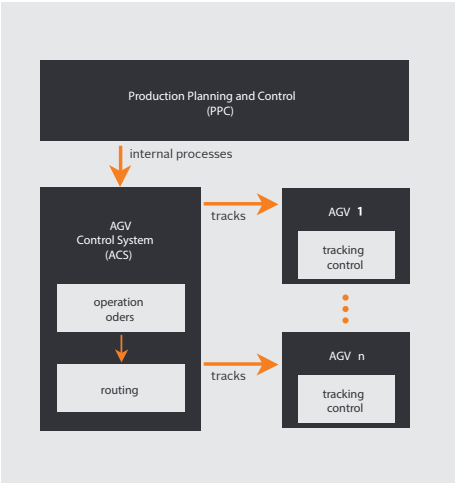
The workflow of Afra is segregated into several steps. First, Afra receives SystemC models and LTL or CTL properties. The model received is translated to Rebeca by using Sytra tool. After that, the Rebeca toolset for model verification is used to verify models and their properties. Additionally, Afra has a feature of displaying a counter-example if the system does not satisfy specified properties.

2.5. Volvo Electric Site

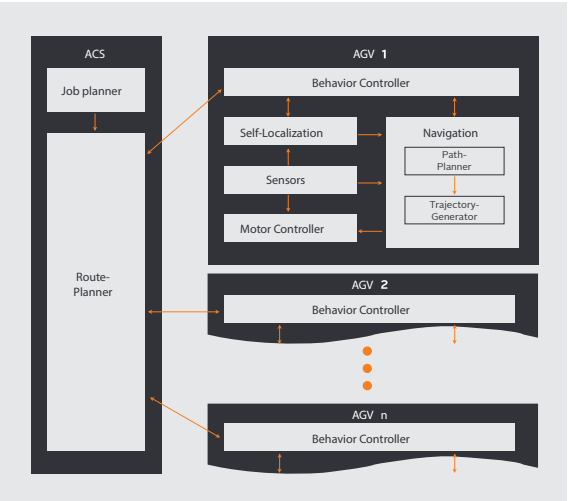
This section provides brief information about Volvo Electric Site since this is the case study covered in this thesis. The goal of the electrified quarry site is improving the efficiency, safety as well as environmental benefits by decreasing the carbon emission to a great extent [19]. The goal of the research is to electrify every process in construction sites, from machine movement to crushing and excavating. The benefits of this research are significant which is proven through testing of this system. The results report over 98% of the decrease in carbon emission, 70% lower energy cost and 40% lower operational cost [19]. In this thesis, we delve into the formal verification of the electrified quarry site by merging the Rebeca actor-based modeling language that addresses concurrent distributed systems and Volvo simulators which provide an accurate representation of real-world cases of electrified quarry sites.

2.6. Automated Guided Vehicles - Distributed Systems

Automated Guided Vehicles (AGVs) represent autonomous machines whose purpose is to perform tasks, most likely transportation. The implementation of AGVs in the industry varies from hospitals (e.g. transportation of the laundry) to harbors [20]. AGVs are equipped with sensors and actuators and in most cases, they operate in a fleet manner. The general representation of the AGV system consists of coordinated autonomous guided vehicles. The main modules if this system is presented in Figure 3a and they are production planning and control (PPC) module, AGV control system (ACS) and the AGVs [20]. Communication between these modules is considered a part of the internal system. In Figure 3b detailed representation of AGV system is presented. ACS is in charge of assigning routes to each AGV and it plans optimal routes for AGVs. AGVs have *behaviour controller* (BC) which is making binary decisions and selects role-plays. BC is in direct communication with ACS and it sends success/failure messages.



(a) Modules of AGV system [20]



(b) AGV system overview [20]

Figure 3: AGV system properties

3. Related Work

To put this work into a broader context, compare it to current research and to expand an understanding of a problem, we need to explore papers which are relevant to the topics covered in this thesis. Researching topics such as system-of-systems, AGVs, actor-based modeling and formal verification was conducted by searching through scientific databases such as IEEE Xplore and Scopus. Additionally, on the *Publications* section of the Rebeca website, there are several works closely related to the topic.

Direct correlation with works which include the Volvo Electric Site case study are [1], [6] and [7]. In [1] and [6], the AdaptiveFlow framework was explained in-depth and some examples for Electric Site case were provided. These papers show the applicability of the AdaptiveFlow through the example of this case. As the main difference from these papers, for this thesis, it is necessary to make a model of a suitable environment for the VCE Simulator scenario and in addition to that, we need to perform the mapping procedure from AdaptiveFlow to VCE Simulator. Jafari et al. [7] have performed formal modeling, using Timed Rebeca, and verification of fleet control system of autonomous haulers. The model proposed in this work uses a different perspective for modeling. More particularly, the machines on the site are perceived as actors, whereas in this project we are using the AdaptiveFlow principle of defining roads as actors and machines as messages sent between actors.

Modeling of self-adaptive systems has been introduced by Bagheri et al. [21]. In this paper, the focus was on Track-Based Traffic Control Systems (TTCS) with centralized control in which traffic flows through pre-specified sub-tracks. They have introduced a coordinated actor model which can be used to design self-adaptive TTCSs. Additionally, authors have proposed several mappings from different TTCSs to the coordinated actor model. However, this paper does not provide model checking solutions.

This work targets distributed systems and it is necessary to find relevant papers that are based on modeling and inspection of such systems. Distributed and asynchronous system modeling and analysis were done by Jafari et al. [22]. Due to the non-deterministic nature of these systems, authors have used Probabilistic Timed Rebeca (PTRebeca) to perform modeling. They have provided SOS rules for PTrebeca, presented a new toolset that generates Markov Automation from PTRebeca models in the form of the input language of the Interactive Markov Chain Analyzer (IMCA) [22]. Khamespanah et al. [23] conduct modeling of real-time Wireless Sensor and Actuator Network. They provide a very accurate view of WSAN application through actor modeling. In this work, authors have also used Timed Rebeca and Afra model checking tool.

Since this work deals with the case of collaborating AGVs, a few works on this topic have been extracted. Saddem et al. [24] targets path-finding algorithms in a 2D grid environment. They

have conducted their model-checking by using UPAAL Timed Automata. A similar problem was covered in [25], where authors inspect the automated planning of optimal paths for multiple robots. They have used weighted transition systems and Linear Temporal Logic (LTL) formula for the goal specifications. In this project, we have to adapt simulator scenarios and the behavior of machines in order to adequately perform the mapping. Authors in [20] have explored the main challenges when it comes to the distribution of knowledge and autonomy of AGVs. This work will provide a better understanding of the AGV system architecture. Transport assignments and collisions in AGV systems have been examined and verified by Weyns et al. [26]. The software architecture has been evaluated and through running the simulation, the test results were obtained. The authors have developed decentralized control architecture by applying Multiagent System (MAS).

In [27] mapping Robot Operating System (ROS) to Rebeca properties has been conducted. Furthermore, this work also examines the Volvo Electric Site case. The contribution of this work is the development of a mapping algorithm between ROS and Rebeca, therefore it proves possible verification of real robotic applications with Rebeca modeling language. The work is highly relevant since it has a similar purpose to this thesis and it is also based on the mapping procedure in which Rebeca is included. Differences reflect in that this thesis focuses on another way of mapping, from Timed Rebeca model to VCE Simulator and different perspectives have been chosen when it comes to modeling. In [27] actors were machines on the site, whereas in this work, due to the nature of the AdaptiveFlow, the roads are perceived as actors that exchange machines as messages.

Webster et. al. [28] introduce a case study of robotic assistants and their formal verification. The experimentation was done through verifying four sample properties of models that were translated from Brahams into the input PROMELA modeling language. The case in this thesis covers the translation of system specifications defined in AdaptiveFlow to both, Rebeca models used for model checking and the Volvo Simulator inputs used for simulating the model. An additional difference is that our focus is on a collaborative system of multiple autonomous machines. In [29] there is a correlation with our project such that authors cover construction machines and their automatization properties. The verification is performed with UPAAL. However, the main difference is that the subject of their research is an autonomous wheel loader and formal verification of its internal properties, whereas in our work the wheel loader is abstracted and is perceived as a part of a collaborative system. Zita et. al. [30] apply the formal verification to a specific module of autonomous vehicles. The lane change module was covered which is a part of the control system inside autonomous vehicles. Our work examines a track-based control system that directs machine movement in the quarry site.

4. Method

The goal of this thesis is to develop a technique for orchestration of several tools in order to evaluate and analyze collaborative systems of heavy machines. As a reference for choosing a suitable method(s), Håkansson [31] has provided a convenient overview of research methods generally used in any kind of research that helped us in identifying the most suitable ones for this project.

The initial point of this research is a hypothesis which states that AdaptiveFlow is useful for the model-driven development of the fleet management systems. Hence, our focus will be on *quantitative* research methods. This hypothesis will be measured through three cases whose *experiments* will be run on the Volvo Simulators and through model verification and evaluation. Through these experiments, the logs from the simulator will be used to extract the performance properties of the system. The models will also provide safety and performance information about the system by analyzing the state-space file generated from the model verification.

Data is collected through the model, as well as the simulation experiments. Besides, we have gone through three case studies in which tools have been used differently and have compared results from model checking and simulations for all three of them. Data collected has been analyzed by using *statistical* methods.

To ensure that this project is replicable and reliable we have provided detailed information about each case and how it was implemented. Each method and methodology has been shown through their applications in multiple activities during this project. Results gathered have been discussed and conclusions have been outlined.

4.1. Performed activities

To answer previously mentioned research questions, defining an appropriate method for conducting this project is imperative. We have split the workflow into four phases which are shown in Figure 4, where each phase is validated.

The first phase of the project is focused on the understanding of the case which is being covered therefore, the background study was conducted. In this phase, it is necessary to understand what problems ought to be solved and how would it be done. In addition to that, the next step is the literature review in which relevant findings and contributions have been extracted.

The second phase consists of model design, mapping, and automatization of the mapping procedure. The first step is to analyze VCE Simulator and its properties. The next step is to generate the Timed Rebeca model, using the model generator from the AdaptiveFlow, by providing 3 input XML files (environment, configuration, and topology). These files are be modified based on what could be derived from the VCE Simulator. The environment.xml file has to be modified based on the terrain of the scenario in the VCE simulator. To assign points of interest (PoIs), such as parking station, charging station, and loading/unloading points, topology.xml file is specified. To define the configuration of the whole system we have to provide configuration.xml file. After that, it is necessary to identify mappable properties between AdaptiveFlow and VCE simulator. Lastly, the mapping procedure is developed and automatized.

The third phase is based on safety verification and performance evaluation. This is done by examining both, models as well as scenario representation of it in the VCE Simulator. Finally, in the fourth phase, the result analysis is performed by comparing results from model verification and the simulation, and finally, the conclusions are derived.

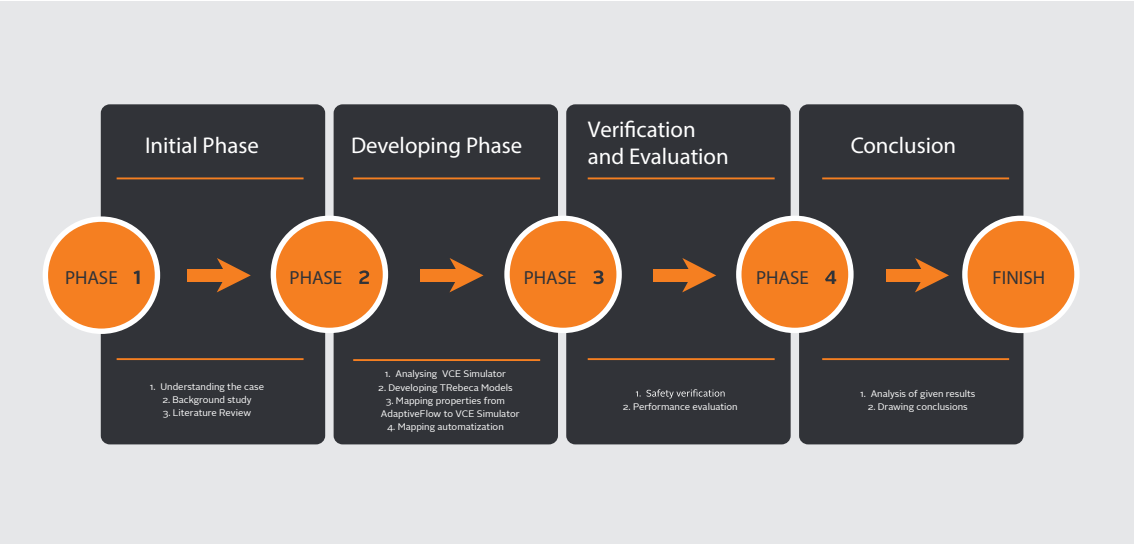


Figure 4: Research Method

5. Mapping AdaptiveFlow Models to VCE Simulator Inputs

This section provides a description of each tool used for the verification and evaluation of the collaborative systems, as well as information on how the mapping was performed. First, we have to go through the AdaptiveFlow framework. Each input file is described in detail because it is necessary to understand all elements and their attributes. The workflow of AdaptiveFlow is provided to make a connection between defined input files, that provide system specifications, and model generating verification, and performance evaluation. Following that, the outline of the VCE Simulator is given. Mainly, the *dynamic.content* input file used for the definition of elements is explained to show a connection between this file and AdaptiveFlow's environment, topology, and configuration files.

During this thesis, we have developed the VMap tool for automatic mapping. It can be seen as just another part of the orchestration which is used for faster development of desired simulation scenarios derived from AdaptiveFlow specifications. Its development, role, as well as the mapping pattern, is thoroughly described in this section.

5.1. AdaptiveFlow framework

AdaptiveFlow [1, 6] is an actor-based Eulerian framework which is used for a track-based flow management system. Usually, when we think of actors in track-based systems, such as traffic systems, we would think of vehicles as actors. By making a parallel with fluid mechanics, this perception would be categorized as the Lagrangian view. In the AdaptiveFlow framework, we use the Eulerian view where the roads would be considered as actors and vehicles as messages sent between the actors. To explain the architecture of the AdaptiveFlow, first, it is necessary to explain in detail three files that are used for the system specification, as it is done did below.

5.1.1. AdaptiveFlow input files

The environment file

The **environment.xml** file is used for defining the base layer of the system, represented as a matrix, in which movement of the vehicles takes place [6]. The layer is split into segments that communicate with each other. Each segment is surrounded by neighboring segments and each neighboring segment is labeled based on the location relative to the current one (i.e. NE-northeast, SW-southwest, E-east, etc.). Segments also have unique identifiers, a capacity that usually assigns the number of vehicles that can be on the segment at the time and free speed which sets the allowed

speed for each segment in m/s. Coordinates of the segments are also defined. The example of the environment.xml file is presented below.

```
<environment>
    <segments>
        <segment id="seg___0_0" N="null" NE="null" E="seg___0_1" ES="seg___1_1"
        S="seg___1_0" SW="null" W="null" WN="null" available="false" x="0" y="0
            " capacity="1" length="200" freespeed="6"/>

        <segment id="seg___0_1" N="null" NE="null" E="seg___0_2" ES="seg___1_2"
             S="seg___1_1" SW="seg___1_0" W="seg___0_0" WN="null" available="
            false" x="0" y="1" capacity="1" length="200" freespeed="6"/>

        <segment id="seg___0_2" N="null" NE="null" E="seg___0_3" ES="seg___1_3"
             S="seg___1_2" SW="seg___1_1" W="seg___0_1" WN="null" available="
            false" x="0" y="2" capacity="1" length="200" freespeed="6"/>
        ...
    </segments>
</environment>
```

Listing 2: Example of the environment.xml file

The topology file

The topology.xml is used for assigning the locations of the points of interest (PoIs). PoIs contain unique identifiers, x and y coordinates, type of the point, and operating time. The PoIs can be perceived as key spots on which tasks are executed. An example of a topology file is shown below.

```
<topology>
    <POIs>
        <poi id="0" x="1" y="3" type="ParkingStation"/>
        <poi id="1" x="3" y="3" type="ChargingStation" chargingTime="0.1"/>
        <poi id="2" x="3" y="1" type="LoadUnloadingPoint" loadTime="0.5"/>
        <poi id="3" x="8" y="1" type="LoadUnloadingPoint" loadTime="0.5"/>
        <poi id="4" x="1" y="7" type="LoadUnloadingPoint" loadTime="0.5"/>
        <poi id="5" x="6" y="7" type="LoadUnloadingPoint" loadTime="0.5"/>
    </POIs>
</topology>
```

Listing 3: Example of the topology.xml file

The configuration file

To specify the system configuration the configuration.xml file (Listing 4) should be defined. It includes information such as the number of machines, resending periods for requests, safe distance between machines, policy for bypassing obstacles, fuel reserve of each machine, and the maximal number of allowed attempts to send a request to the unavailable segment before rerouting. The example part of the configuration.xml file for the system features is presented below.

```
<system>
    <resending_period value="15"/>
    <number_vehicles value="5"/>
    <safe_distance value="20"/>
    <battery_limit value="1000"/>
    <policy value="1"/>
    <obstacle_occurences value="5"/>
    <obstacle_number value="4"/>
    <obstacle_max_time value="1000"/>
    <obstacle_max_duration value="100"/>
    <max_attempts value="2"/>
</system>
```

Listing 4: Example of the *system* element in configuration.xml file

The remaining part of the configuration file is adding machines and their properties. Each machine has tasks that are assigned as a set of IDs of PoIs. The attributes of the machines are unique identifiers, machine type, leaving time from parking station, fuel capacity, fuel consumption, speed, CO2 emission, capacity, and unloading time, as it is shown on Listing 5.

```
<machines>
    <machine id="0" type="hauler" leavingTime="10" fuelCapacity="7000"
        fuelConsumption="1"      speed="6" emission="6" capacity="100"
        unloadTime="0.1">
        <tasks>5,3,2,4,5,3,0</tasks>
    </machine>
    <machine id="1" type="hauler" leavingTime="20" fuelCapacity="7000"
        fuelConsumption="1"      speed="6" emission="6" capacity="100"
        unloadTime="0.1">
        <tasks>3,2,5,4,3,2,0</tasks>
    </machine>
    ...
</machines>
```

Listing 5: Example of the machine specifications

5.1.2. AdaptiveFlow workflow

AdaptiveFlow workflow is split into three phases. The initial phase is the pre-processing phase in which TRebeca models are generated by processing three input files (environment, configuration, and topology) with a Python script that represents the model generator. The next phase consists of formal verification of the generated model by generating the state space [1]. The verification is performed with model checking tools such as Afra or Rebeca Model Checker (RMC) [1]. These tools convert Rebeca models to C++ files which are afterward compiled to an executable file [1]. This file implements the model checking algorithm and it returns results. As mentioned before, model checking tools also generate state space of models used for the final, post-processing phase. In this phase, state-space goes through the Python script that analyses each state. The representation of the whole workflow is presented in Figure 5. Each phase of the workflow is described in-depth in the following sections.

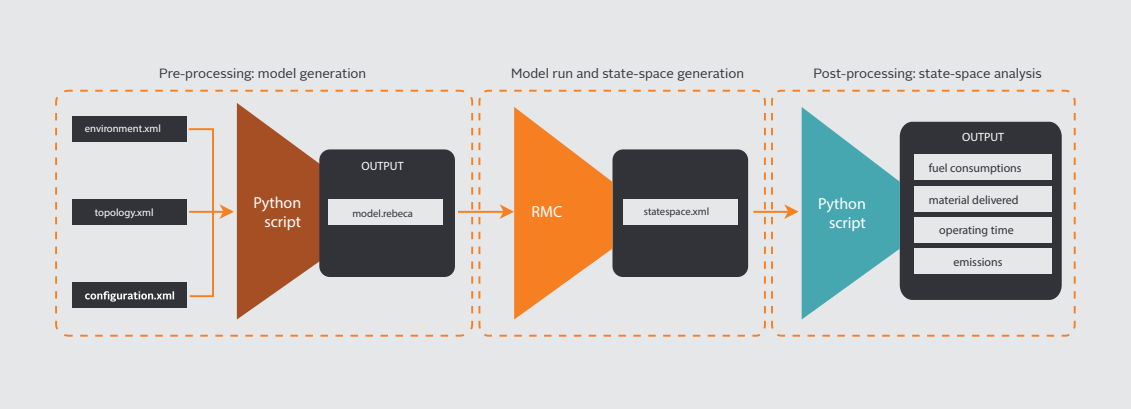


Figure 5: AdaptiveFlow workflow [1]

Phase 1: Pre-processing phase

In the AdaptiveFlow framework 3 input files are used for the system specification and those are *environment.xml*, *topology.xml* and *configuration.xml*.

After defining the elements of three input files, the next step is to run the *model generator* script which will derive the TRebeca model based on those input files. The representation of the model

could be designed as a matrix with PoIs. This representation is an abstraction of the scenario which excludes details of the system. The example of scenario depiction is shown below.

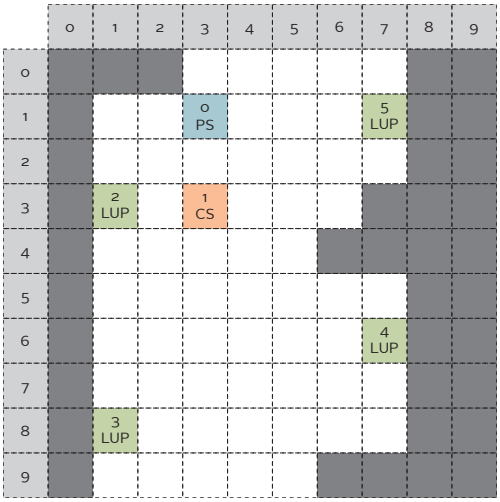


Figure 6: AdaptiveFlow environment [1]

Phase 2: Model run and state-space generating

In this phase, by using a model checking tool (either Afra or RMC) the model analysis is executed by generating C++ files from the input files. Compiling C++ files results in generating an executable file. The execution of this file performs a model checking algorithm on the input model and based on the results, the state space file is generated. Aside from checking regular properties such as deadlock-freedom and safety, AdaptiveFlow also verifies properties like fuel consumption of machines, correct machine movement, absence of machine collision with obstacles, and no-starvation property[6]. The generated state space file is used in the upcoming phase for extracting the performance properties.

Phase 3: Post-processing phase

The last phase of the AdaptiveFlow workflow consists of the evaluation of the performance properties of the system through analysis of resulting, state space, file from the previous phase. The state-space file is analyzed with a Python script that extracts the evaluation of performance properties. The performance evaluation includes total CO2 emissions of machines, the amount of consumed fuel, moved material, and operating time of the collaborative system.

5.1.3. Novel features of AdaptiveFlow

Throughout this thesis, AdaptiveFlow has been improved and upgraded in alignment with conducted project. New features that are implemented are proven as helpful when it comes to the mapping because these changes helped in developing more realistic cases.

With the new features of AdaptiveFlow, this framework now supports unidirectional routes, and users can specify directions between segments in the environment file. This means that even though segments are adjacent they are not necessarily linked. In Listing 6, specification of *direction* attribute of the segment element is presented.

```
<environment>
    <segments>
        <segment id="seg___0_0" N="null" NE="null" E="seg___0_1" ES="seg___1_1"
        S="seg___1_0" SW="null" W="null" WN="null" direction="E,ES" available="
            false" x="0" y="0" capacity="1" length="200" freespeed="6" />
        ...
    </segments>
</environment>
```

Listing 6: Segment's "direction" attribute

An additional feature that was implemented is *policy 4* and the *routes* element in the topology file. Since machines have tasks specified as a set of PoI IDs, the *policy 4* specifies that the machines should follow the path between PoIs that is explicitly specified in *routes* element. The *route* element has attributes *origin*, *destination* and *path*. It is a sequence of segments that the vehicles must follow for reaching a PoI (destination) from the current PoI (origin). The values of *origin* and *destination* correspond to the ID of the PoI itself. The example of the *routes* element is shown in Listing 7.

```
<topology>
    <POIs>
        ...
    </POIs>
    <routes>
        <route origin="0" destination="2" path="seg___8_7, ... ,seg___6_8"/>
        ....
        <route origin="0" destination="3" path="seg___8_7,seg___7_8, ..... ,
            seg___3_7,seg___2_8"/>
    </routes>
</topology>
```

Listing 7: Defined routes in toplogy file

5.2. VCE Simulator

Volvo Simulators provide a real-life imitation of processes in Volvo construction sites. The features range from simulating the process of the individual machine to building up authentic scenarios [32]. VCE Simulator also provides data analysis of the scenarios in which users can evaluate the performance and efficiency of defined simulation models. In this case, we focus on the scenario building feature of the Volvo Simulators in which we build scenarios that are a convenient representation of the models. The simulation is used as an adjunct to the TRebeca models for system design, verification and evaluation.

For the scenario buildup, Volvo Simulators use a **dynamic.content** file as an input in which each object inside the simulation scenario is defined as an element in Extensible Markup Language (XML) format. In this file, we define elements such as *edges*, *vertices*, *loading material*, *machines*, etc. Our goal is to generate *dynamic.content* file according to the specification inside AdaptiveFlow inputs. The following section describes a tool developed to bridge a gap between the level of abstraction of AdaptiveFlow and VCE Simulator.

5.3. VMap: the conversion tool description

In this section, AdaptiveFlow is perceived as a design tool. Section 6.1. shows the case that has been examined in order to perform the analysis of VCE Simulator and AdaptiveFlow. Described steps were necessary to derive mappable elements and based on that the tool for mapping automatization has been developed. The mapping automatization has been developed to bridge a gap between AdaptiveFlow specifications and VCE simulations and to complete model-driven development loop.

VMap is a tool developed in order to perform the mapping automatization from the AdaptiveFlow inputs (environment, configuration and topology) to the dynamic.content file that consists of static features for the simulator. These static features are an XML representation of elements inside the simulator scenarios such as vertices, edges, path points, automated haulers, loading machines, etc. What AdaptiveFlow provides us is a convenient method for design, analysis and verification of collaborative systems. The VMap tool has been developed with a goal to make a transition from an abstraction such as a TRebeca model to a concrete simulation of real-world scenarios. In addition to that, it is necessary to verify whether this transformation from TRebeca models to simulation provides correct-by-design scenarios. To conduct this transition in a valid way and to a full extent, it was essential to implement a logic that is a core of the AdaptiveFlow

inside a Simulator. To illustrate all of the tools involved in orchestration process, including all of the inputs and outputs, we have designed a diagram shown in Figure 7. In upcoming sections, each part of the workflow is be thoroughly described.

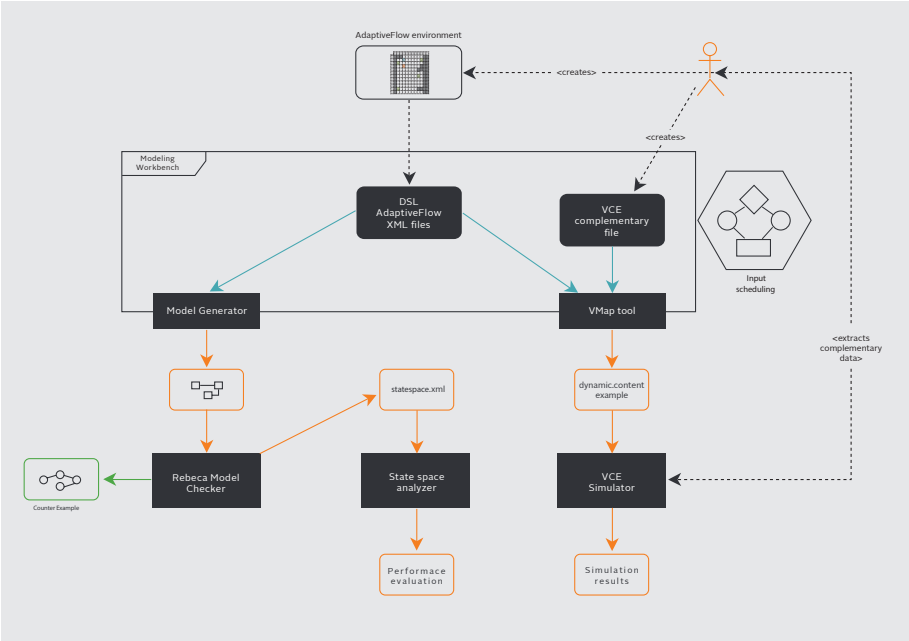


Figure 7: Tools used with their inputs and outputs

5.3.1. Development phases of modeling and conversion

Phase 1

The first phase consists of designing a suitable representation of the system. This includes designing the environment of the system, defining properties of machines and setting up the location of each PoI. When all of this has been set up, the next step is divided into two subbranches. The first branch includes defining AdaptiveFlow input files based on the desired requirements. Specified files will be used as inputs for two tools: *Model Generator*, which is a part of the AdaptiveFlow, and *VMap*. The second branch of the initial phase represents an extraction of complementary files for previously mentioned input files. The complementary file is extracted from *dynamic.content* file of scenarios inside the simulator and it differs based on the scenario. The contents of this file are all elements in the simulation which cannot be mapped directly from the AdaptiveFlow, such as camera elements, a base asset of the scenario, etc. Furthermore, the environment in VCE Simulator is represented in 3D, whereas in AdaptiveFlow we have a 2D representation of the environment. Since our goal here is to perform mapping and its automatization from an abstract 2D TRebeca model to a 3D environment in the simulator it would not be adequate to choose scenarios with the inconsistent and steep environment, therefore the scenarios with the relatively flat surface have been segregated.

Phase 2

The second phase is also split into two different sections. The first section, consists of the model generating, verification and evaluation, whereas the second section consists of the mapping from the defined environment to the VCE Simulator. The first section uses AdaptiveFlow input files for the model generation and goes through the whole process of AdaptiveFlow workflow. A first section is executed before the second one because it is necessary to first verify and evaluate the generated TRebeca model and then map model inputs to the Simulator.

The second section consists of a mapping procedure from AdaptiveFlow inputs to the VCE Simulator. In second part, AdaptiveFlow input files with the VCE complementary file, extracted in a previous phase, are used for generating two outputs. The first output file is an image that is a visual representation of a 2D environment that should be designed inside the Simulator. The second file represents a *dynamic.content* file example that shows how the input file in the Simulator is supposed to look like. It is used to define a suitable simulation scenario for the TRebeca model.

Phase 3

In the final phase, the comparison of the results from two sections of Phase 2 is conducted. Depending on the TRebeca model verification result, that result will be compared to the outcome in the simulator scenario. If the model checking results in dissatisfaction of checked properties, then that counter example will be mapped to the simulator to verify whether that scenario also fails there. Otherwise, if the model satisfies checked properties then the performance evaluation will be conducted and its results will be compared to the simulation results of the mapped model.

5.3.2. Description of files generated by the tool

The VMap tool is developed in Python programming language. It generates two output files that are *dynamic.content* example file and a guideline image which is an abstracted illustration of the desired Simulator scenario represented in 2D. Generated two files are then used to develop a suitable simulation scenario. At the current stage of development, the *dynamic.content* example file is used as a proof of concept. This is the case because there are several attributes in the certain XML representation of objects that could not be assigned externally.

VCE Guideline

Referring back to the previous explanation, the guideline image, generated with the script, is a representation of the AdaptiveFlow specifications in VCE Simulator fashion. This image can be used as an instruction for developing an adequate simulation scenario by converting AdaptiveFlow segments into edges and vertices defined in the script. In addition to that, information from the topology file is extracted to place each PoI to a vertex which is in that PoI location. In shown image, each PoI is labeled relative to its type. The coordinate plane of represented image is calculated based on a scenario. The starting and ending coordinates are extracted from the simulation scene and after that, the coordinate plane is split into a 10x10 grid to equalize it with the AdaptiveFlow segmented environment. After that calculation, each vertex is put on the midpoint of every available segment and every vertex has a directed edge towards those vertices to which that segment is directed to. Identified connections stand for edges in the simulation terms. In Figure 8, for the illustration purpose, we show the transformation from the AdaptiveFlow environment of scenario described in previous sections to the 2D Simulator representation.

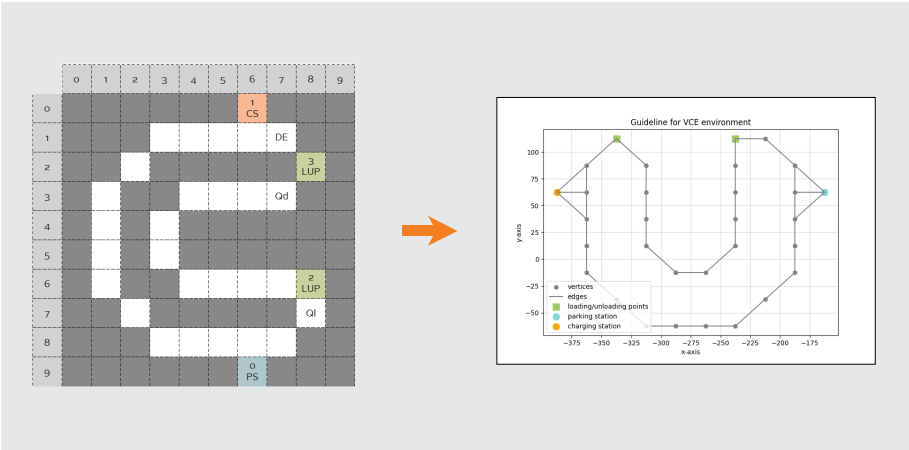


Figure 8: Conversion from AdaptiveFlow inputs to VCE guideline file

The dynamic.content example

The VCE Simulator uses a *dynamic.content* file for each scenario which is an Extensible Markup Language (XML) representation of every element, its attributes and their communication. Based on the information from this file every element is properly set up inside the scenario. The other file that is generated with the VMap tool is *dynamic.content* example file. It used as a proof of concept which tells us that the AdapiveFlow input files could be mapped directly and automatically with the VMap tool. The only obstacle/limitation in running the *dynamic.content* file, generated by the tool, directly inside the simulator is the generating of *persistentID* for each new element created. The VCE Simulator has a specific method for generating this attribute which could not have been accessed. Despite that, the VMap tool can be used with the Rebeca tool stack and VCE Simulator in a combination to perform formal verification of collaborative systems, moreover, it can be observed as a link between the two. For now, this file in a combination with an image file, generated by the VMap tool, is used as instruction files to develop a suitable simulation scenario based on the AdaptiveFlow specifications by using VCE Simulator's Scene Editor. Figure 9 shows an example of how the elements can be generated directly inside the simulator just by strictly following the outputs of the VMap tool.

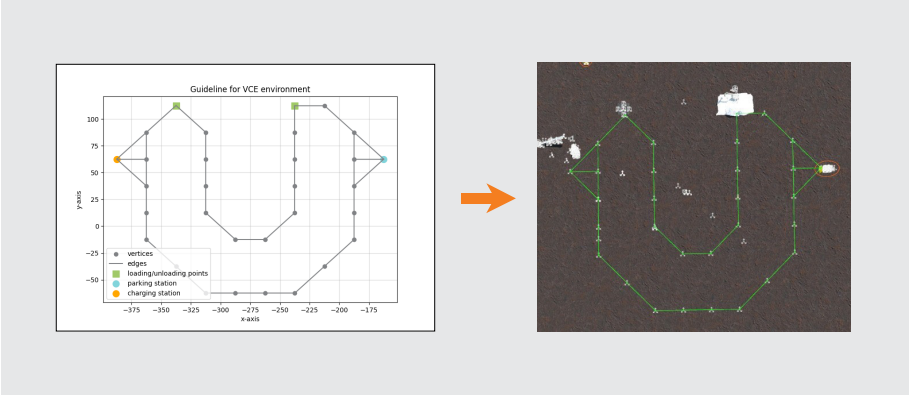


Figure 9: Scenario developed by following the guideline

5.4. Conversion of elements from AdaptiveFlow to VCE Simulator

This section gives an in-depth description of elements that were identified as mappable from AdaptiveFlow to VCE Simulator. The conceptual, as well as applicative description of the mapping pattern, was presented to better grasp how the mapping pattern has been defined and to provide a better understanding of how the VMap tool for mapping automatization, described in section 5.3. has been developed.

The main elements that have been mapped from AdaptiveFlow to the Volvo Simulator are **segments** and **PoIs**. The attributes of elements such as *speed*, *capacity*, *length*, *type*, etc. have been mapped to the *Edge* element of the input file for the Simulator. Newly developed **route** element from AdaptiveFlow has been mapped to the **AhMission** in the Simulator's file in which we define machine paths (missions). As for the **machines**, AdaptiveFlow provides more specific information than the Simulator. Attributes such as *leavingTime*, *unloadTime* and *speed* are related to the same attributes of the Simulator's *Edge* element. However, other information could not be used in the Simulator's input file, since attributes like *fuelConsumption* or *emission* have not yet been implemented. The **system** element in AdaptiveFlow specification represents a very convenient way of specifying system properties of fleet management, but only *safe distance*, *number of vehicles* and *policy* could be mapped to the VCE Simulator. As for the *policies*, policies for rerouting and bypassing obstacles are yet to be implemented in the simulation.

5.4.1. AdaptiveFlow environment

As previously noted, the environment in the AdaptiveFlow is split into segments that communicate with each other, by sending machines as messages. Defined segments represent tracks through which machines travel. Each segment is represented as an actor that includes additional attributes such as allowed speed, segment length, capacity, etc. To derive the mapping pattern, the track representation inside VCE Simulator needs to be identified. The track representation in the simulator consists of multiple components that have been described below.

- Vertices – This element is identified in the initial scenario, but it is not considered as essential for a mapping.
- Edges – The edges are elements which consist of path points and a script that represents logic for the edge control. There are also dynamic properties such as *poiType*, *speed* and *capacity* that were subsequently implemented and defined in examined cases based on attributes of the AdaptiveFlow segments.
- EdgeControl – This component represents a script in which edge is being controlled by certain attributes. This script has been implemented in Section 6.1. of reverse engineering and it is based on the logic of AdaptiveFlow in which all edges are controlled.
- PathPoints - All edges consist of path points that machines are following when they are moving through edges.

Segments from the AdaptiveFlow have been mapped to the several elements described above. All of the segments can be transformed into path points and edges. In the defined mapping pattern, in the middle point of each segment, the path point is put. Adjacent segments, which have defined paths between them by assigned directions in AdaptiveFlow, are connected with edges. For example, if there are segments A and B and if the direction in environment.xml file is defined as from A to B, by using this mapping pattern, the result would be 2 *PathPoints* in the middle of both segments and one *Edge* that represents the connection between these two points. It is important to note that in the mapping procedure, the starting point of the edge is located at the center of that segment and the ending point is located in the middle of the adjacent segment to which the current one is connected to. The new representation could be also defined as a graph. For this case, the environment in AdaptiveFlow is illustrated as a graph G, vertices are *PathPoints* defined as $P_i = (X_i, Y_i)$ by which Edge lengths is derived as a cost matrix $E = (e_{ij})$. The *Edge* element length e_{ij} is Euclidean distance between *PathPoints* which is formulated as in Equation 4.

$$e_{ij} = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2} \tag{4}$$

Segement attributes, such as capacity and allowed speed are defined in the *Edge* component and are controlled in the *EdgeControl* script.

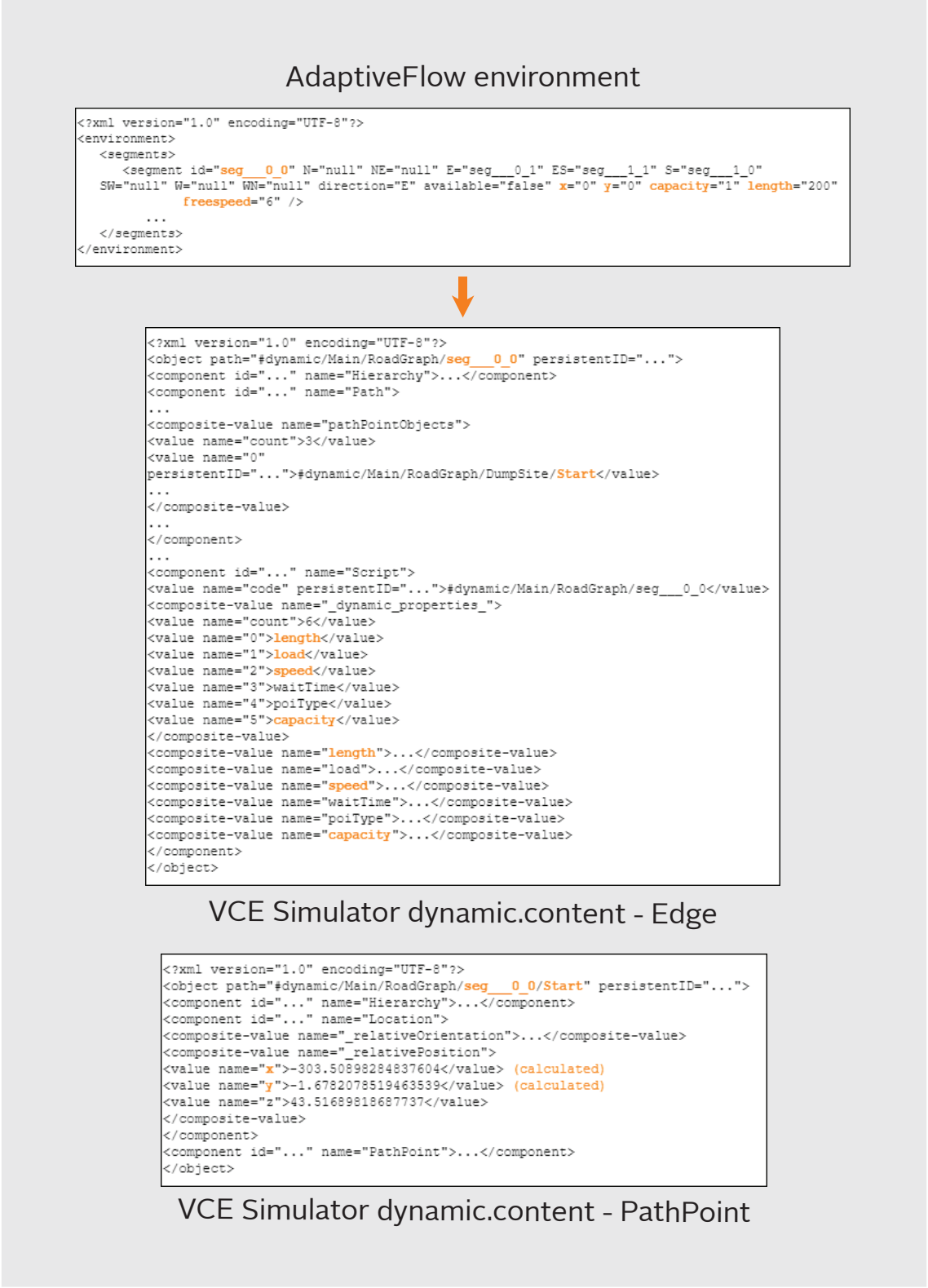

AdaptiveFlow environment
<?xml version="1.0" encoding="UTF-8"?>
<environment>
<segments>
<segment id="seg___0_0" N="null" NE="null" E="seg___0_1" ES="seg___1_1" S="seg___1_0"
SW="null" W="null" WN="null" direction="E" available="false" x="0" y="0" capacity="1" length="200"
freespeed="6" />
...
</segments>
</environment>
<?xml version="1.0" encoding="UTF-8"?>
<object path="#dynamic/Main/RoadGraph/seg___0_0" persistentID="...">
<component id="..." name="Hierarchy">...</component>
<component id="..." name="Path">
...
<composite-value name="pathPointObjects">
<value name="count">3</value>
<value name="0"
persistentID="...">#dynamic/Main/RoadGraph/DumpSite/Start</value>
...
</composite-value>
...
</component>
...
<component id="..." name="Script">
<value name="code" persistentID="...">#dynamic/Main/RoadGraph/seg___0_0</value>
<composite-value name="_dynamic_properties_">
<value name="count">6</value>
<value name="0">length</value>
<value name="1">load</value>
<value name="2">speed</value>
<value name="3">waitTime</value>
<value name="4">poiType</value>
<value name="5">capacity</value>
</composite-value>
<composite-value name="length">...</composite-value>
<composite-value name="load">...</composite-value>
<composite-value name="speed">...</composite-value>
<composite-value name="waitTime">...</composite-value>
<composite-value name="poiType">...</composite-value>
<composite-value name="capacity">...</composite-value>
</component>
</object>
VCE Simulator dynamic.content - Edge
<?xml version="1.0" encoding="UTF-8"?>
<object path="#dynamic/Main/RoadGraph/seg___0_0/Start" persistentID="...">
<component id="..." name="Hierarchy">...</component>
<component id="..." name="Location">
<composite-value name="_relativeOrientation">...</composite-value>
<composite-value name="_relativePosition">
<value name="x">-303.50898284837604</value> (calculated)
<value name="y">-1.6782078519463539</value> (calculated)
<value name="z">43.51689818687737</value>
</composite-value>
</component>
<component id="..." name="PathPoint">...</component>
</object>
VCE Simulator dynamic.content - PathPoint

Figure 10: Mapping segment attributes to simulator elements

5.4.2. AdaptiveFlow topology

Since current implementation inside the simulator did not have defined edge types by which it could be determined whether one edge is loading track, unloading track or parking track, we had to make that implementation to develop a convenient mapping pattern. In the AdaptiveFlow, a topology file contains information about PoIs. The current state of the implementation supports the most essential PoIs which are *Parking Station*, *Charging Station* and *Loading/Unloading point*. Additionally, the information about the x and y coordinate of that PoI in the segmented environment is also defined. Based on that, we can see to which segment that PoI belongs to.

Taking this into consideration, the PoI information was perceived just like additional information to specific segments. New attributes depend on the PoI type, so for the *Loading/Unloading point* there is an additional *loadTime* attribute and for the *Charging Station* there is a *chargingTime*. In the simulator's *dynamic.content* file added attributes were merged to an *Edge* object. Newly defined attributes are *poiType* and *waitTime*. In the Figure below we can see how mentioned attributes are converted.

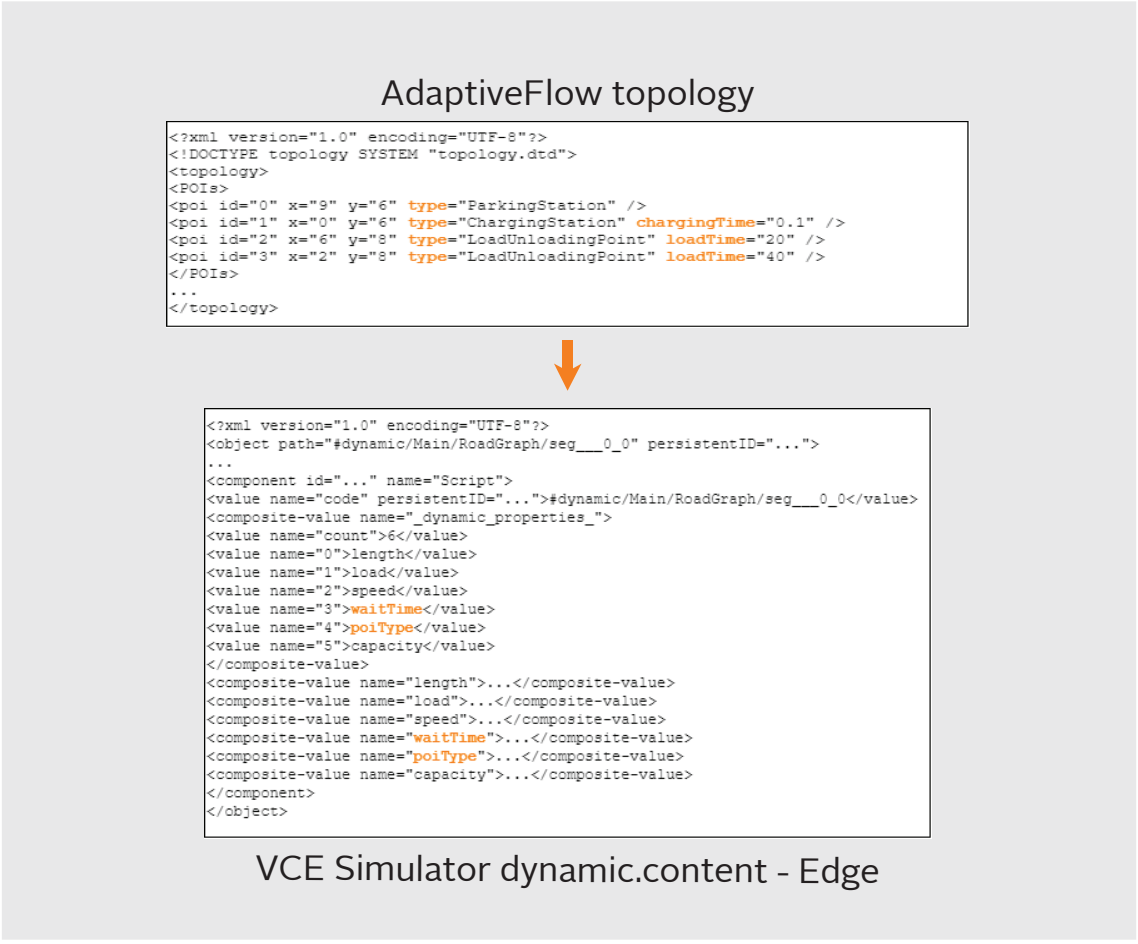


Figure 11: Mapping PoI attributes to simulator Edge attributes

5.4.3. AdaptiveFlow configuration

A final module that is mapped from the AdaptiveFlow is the system configuration. The configuration is divided into two parts: *system* and *machines*. In the system specification, the elements that were recognized as mappable are the number of vehicles and safe distance between machines. The number of machines is not explicitly specified inside the *dynamic.content* file but we can define as many machines as we need. The safe distance between machines is specified in meters and this property is mapped to the *GraphicalField* component inside the machine element in the

dynamic.content file. It represents the range around the machine which could be considered as a safe zone. If the obstacle appears inside the zone the machine will automatically stop.

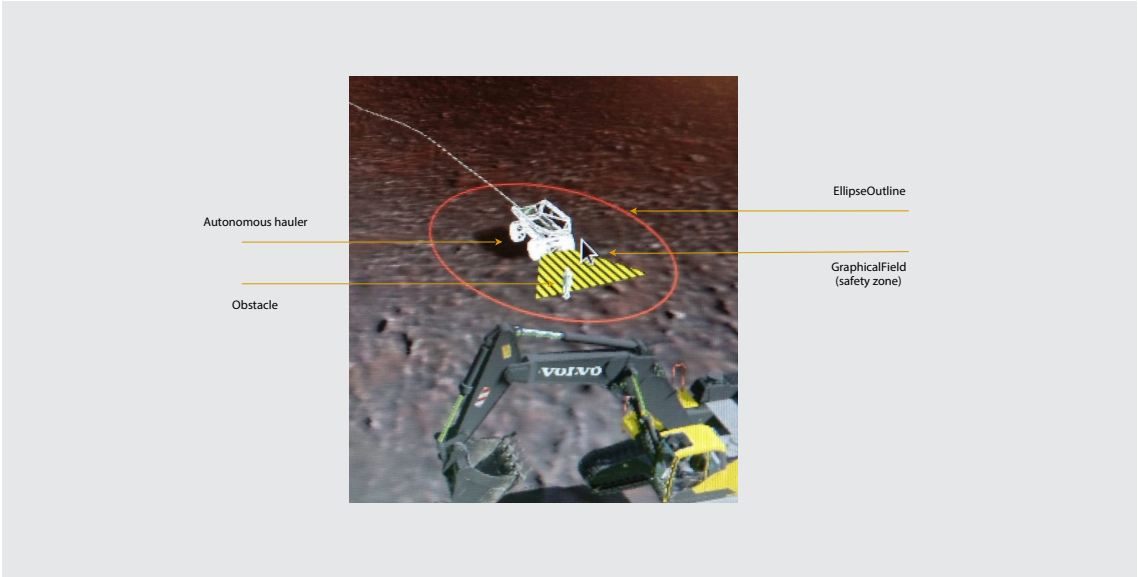


Figure 12: Machine safety zone

The second part is the machine specification. For attributes such as *fuelCapacity*, *fuelConsumption*, *emmision* and *capacity* we do not have equivalent attributes implemented inside machine objects in the VCE Simulator. Regardless, the machine attributes in AdaptiveFlow such as *leavingTime* and *unloadTime* is attached to the edge object in *dynamic.content* file. The final mapped machine property is machine *tasks*. This property is connected to the *routes* from the **topology** file in AdaptiveFlow in which PoIs are also defined. Machine tasks are specified as a set of PoI identifiers and inside the *routes* element we set up paths between PoIs. Those paths represent a set of segments through which machines go in order to reach their destination. The combination of machine tasks and routes results in the mapping to the simulator's *AhMission* object inside the machine. The *AhMission* is a set of edges that defines a full path that the machine has to follow for the purpose of executing its tasks. In Table 1 a mapping pattern of each element and their attribute is presented.

AdaptiveFlow concept	Attribute	Mappable	Directly mappable	VCE Simulator concept	Attribute
Segment	x and y coordinates	Yes	No	PathPoint:Start PathPoint:End	x and y coordinates
	direction	Yes	No		
	freespeed	Yes	Yes	Edge	speed
	capacity	Yes	Yes		capacity
	length	Yes	No		length
PoI	type	Yes	Yes		poiType
	loadTime/chargingTime	Yes	Yes		waitTime
Route	origin	Yes	No	AH15	AhMission
	destination	Yes	No		
	path	Yes	No		
Machine	tasks	Yes	No		
	leavingTime	Yes	Yes	Edge	waitTime
	unloadTime	Yes	Yes		
	speed	Yes	Yes		speed
	fuelConsumption	No	No	/	/
	fuelCapacity	No	No	/	/
	emission	No	No	/	/
System	resending_period	No	No	/	/
	number_vehicles	Yes	No	Derived pattern	/
	safe_distance	Yes	Yes	AH15	GraphicalField
	battery_limit	No	No	/	/
	policy	Yes	No	Derived pattern	/
	obstacle_occurences	No	No	/	/
	obstacle_number	No	No	/	/
	obstacle_max_time	No	No	/	/
	obstacle_max_duration	No	No	/	/
	max_attempts	No	No	/	/

Table 1: Mapping pattern from AdaptiveFlow to VCE Simulator

6. Safe-by-Construction Simulator

For the first case covered in this thesis we have looked at the existing scenario in the Simulator that had some faults in the implementation. The goal was to provide a safe-by-construction alternative to the current scenario by implementing AdaptiveFlow track-based logic as a guide, but yet not to change scenario drastically.

In this case, AdaptiveFlow has been used as an analysis tool. We have set the requirement to develop the closest model representation of this scenario. After the verification and evaluation, we had to implement track-based control logic in the simulation which is also used for the implementation of the other two scenarios from the Section 7. This scenario has helped us in analyzing VCE Simulator and deriving the mapping pattern from which the VMap tool, described from Section 5.3. emerged.

6.1. Initial scenario – Fleet management

In the initial scenario, there were three autonomous machines – automated haulers, and one loading and one unloading point. Each of the machines had a task to transfer material from a loading point to an unloading point and in that execution, it was required to make sure that there would be no collisions nor deadlocks and that each machine would perform tasks undisturbed. To analyze this scenario further, the control system has been examined which is also depicted in Figures 13 and 14.

As can be seen in the figure, several edges are in control of the loading and unloading site. These control systems work with several variables based on which the movement of machines is executed. First, there is loading and dumping time, measured in seconds, assigned to loading site and dumping site control systems, respectively. Next, *Touche Sensor* is a Boolean variable by which it is determined whether the movement of machines from the waiting queue towards the exact spot for loading/unloading has been initialized and detected. Finally, the Loading/Unloading variables are self-explanatory to an extent. They represent the state of task execution.

The main execution of loading/unloading is determined by these variables. It can be noticed in Figure 13 that there are 4 edges in control - *Edge 3*, *Edge 5*, *Edge 7* and *Edge 0*. All of these edges have an initial speed of $0km/h$ until the event from the loading/dumping site control system is triggered. Edges 3 and 7 represent tracks for machine queuing that have a maximum capacity

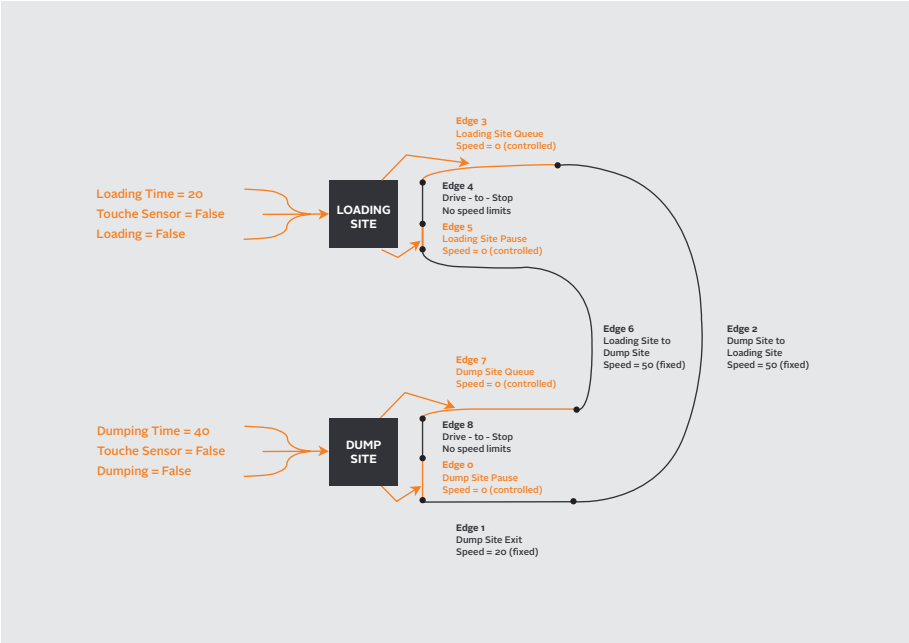


Figure 13: Fleet management scenario - Control System

of 2 machines. When the movement of machines has been triggered by the control system, the allowed speed of this track is set up to $20km/h$ which initializes the movement of machines from the queue. Other two controlled edges, Edge 5 and Edge 0 represent tracks for the main execution of the loading/unloading. The initial speed allowed on these tracks is $0km/h$ and the allowed speed changes to $20km/h$ when the loading/unloading has been finished, so a machine could continue to execute upcoming tasks and the first next machine from the queue could move to that spot. Edges 4 and 8 represent a route of connection between the waiting queue and loading/unloading spot and the speed of these tracks is not in control of the system. Edges 2 and 6 have a fixed speed of $50km/h$, so whenever the machine enters this track, it accelerates to that speed until it reaches the next track. Finally, Edge 1 has allowed speed of $20km/h$ even though it is not a queuing track.

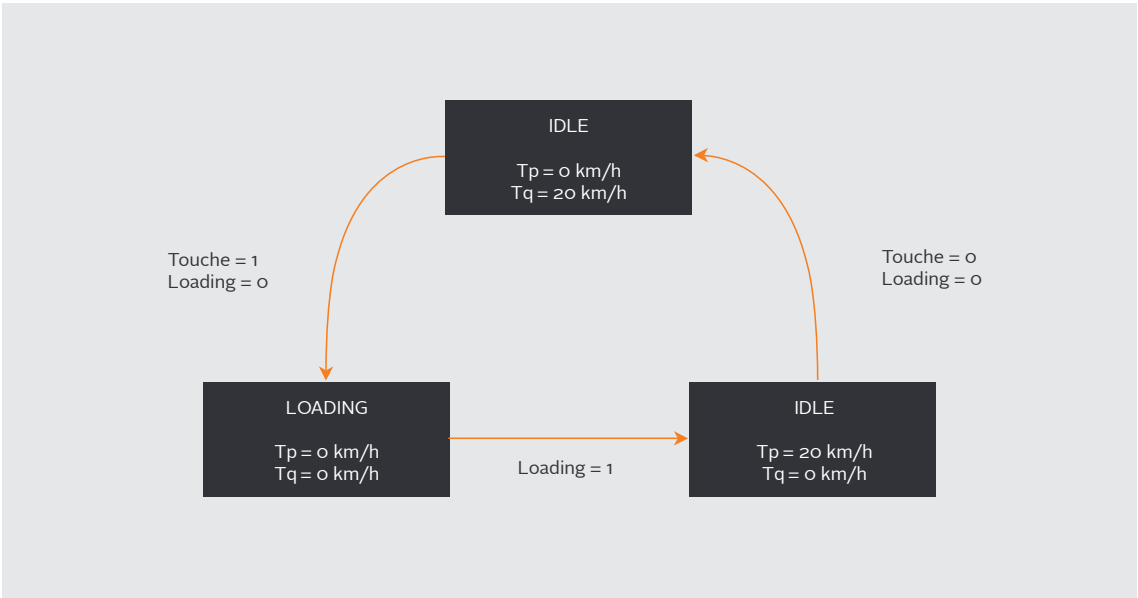


Figure 14: Loading site logic

In Figure 14 the logic behind the loading site has been shown. The same logic is implemented for the dumping site also. The rectangles on the figure represent the state of the control systems which can be either idle or loading. The variables inside the squares represent allowed speed on certain edges. On the one hand, there is a **Tp** variable which is a speed allowed on loading/unloading site pause tracks, which in our case are Edges 5 and 0. On the other hand, there is a **Tq** variable which represents allowed speed on queuing tracks. There are also previously mentioned variables, Touche and Loading, which are Boolean variables, here this domain is represented as $0, 1$.

To explain the execution of the logic it is necessary to choose the starting point for the iteration. We have chosen an idle state, where $Tp = 0km/h$ and $Tq = 20km/h$, as the initial state. In this state, Tq is set to $20km/h$ which means that the event for moving of the machines from the queue to loading/unloading point has been triggered. Here the Touche sensor detects a movement and its value is set to *1 (True)*. After execution of this event, the machine arrives at the loading point where both Tp and Tq is set $0km/h$ until the machine finishes its task. As can be assumed, during the execution of this task the Loading variable is set to *1 (True)*. When this task has been executed, the Tp is being set to $20km/h$ which initializes the machine moves to the next edge and finally, Touche and Loading variables are set to *0 (False)*.

6.1.1. The crash occurrence

In the described scenario, the main issue that has been noticed is a crash that occurs if we place one more machine on the Edge 5 and occupy the loading point for too long while the machines are waiting on the queue. The crash happens when the third machine is arriving at the queuing track while the other two are still waiting. The third machine will not be stopped from entering the queuing track even though the capacity of that track can only handle 2 machines, therefore it will cause a collision.

6.1.2. Design parameters

In order to execute the next step, which is designing a suitable model, the design parameters need to be extracted and defined. Based on the previous analysis of the scenario, the design parameters will be drawn out taking into account what AdaptiveFlow can provide. Some of these design parameters are listed below.

- Number of vehicles
- Path segmentation and allowed speed
- Loading and dumping time

In the upcoming section, the focus will be on the model development, design and the way of inclusion of parameters mentioned above in the model design. Additionally, the collision described in this section is going to be addressed by modifying the logic of the system inside the scenario based on the AdaptiveFlow principles.

6.1.3. Model development with the AdaptiveFlow

For this phase, the extracted design parameters were included in the model design. In Figure 15 the environment representation which is an illustration of the previously described and analyzed scenario is introduced. This interpretation is an abstraction of the respective scenario and all elements have been roughly distributed according to the environment inside the simulator. The whole scenario has been split into segments with specific attributes that communicate and interact in a concurrent manner. For the sake of reminiscing, the PS stands for the parking station, or the starting point from where machines start their execution, LUP stands for loading/unloading point and CS is for the charging station. The charging station has not been implemented inside the simulator, but it has been included in this model because it is mandatory to define it. Segments Ql and Qd represent queuing segments for the loading and dumping. The DE is a segment that represents the "Dump Site Exit" edge from Figure 15. AdaptiveFlow generates models based on the three input files, the definition of those files will be described in more detail.

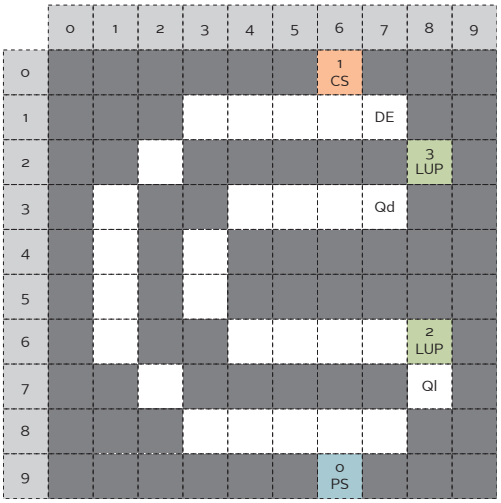


Figure 15: AdaptiveFlow environment of initial scenario

Defining the environment

The attributes of segments that need to be defined are *freespeed*, *capacity*, *length*, *availability*, *neighbours* and *directions*. The *freespeed* property is used for assigning the allowed speed in each segment whose speed unit is m/s. In this case, the speed described in scenario analysis has been converted and assigned for each segment. On the segments Ql, Qd, DE and LUPs the assigned speed has been set to 6m/s, whereas for the other segment was set to 14m/s. Next, the capacity of each segment has been set to 1 except for the queuing segments, which have been set to 2 based on the requirements, and parking spot which has been set to 3 since there are 3 machines in this scenario. The length of each segment was set to 75 meters and neighbors were added based on all adjacent segments. The new feature that has been developed during this work introduced the novel attribute for segments which are directions. This feature allows us to specify the linkage between the segments. In other words, it means that the neighboring segments are not necessarily connected, as it was before this upgrade.

Defining the toplogy

The topology file is used for setting the location of the PoIs and their properties. For this scenario there are 2 loading/unloading points (LUPs) with attribute **loadTime** set to 20 seconds based on the scenario. Even though, one LUP is unloading point this attribute also needs to be set even though it will not be used.

Defining the configuration

The whole system configuration is set in the configuration file. The properties of the control system are set here, including the properties of each machine. Based on the requirements some of these properties have been assigned, other attributes that have not been defined inside the simulator have been set arbitrarily. Properties for the whole system that have been specified are the *numberMachines*, which is set to 3 since there are 3 machines and *policy*, which is set to 1 (waiting policy) because in this scenario machines strictly follow only one path. As for the machines, the *type* has been set to "hauler", *capacity* has been set to 15, *speed* is set to 14m/s as an average speed and *unloadTime* was set to 40 seconds based on the requirements. Additionally, tasks for each machine are defined as $2- > 3- > 2- > 3- > 0$ or in different notation $LP- > UP- > LP- > UP- > PS$, where LP stands for loading point, UP stands for unloading point and PS stands for the parking station.

6.1.4. Model generating

The final step of this phase consists of model generating by using previously defined files as inputs. This TimedRebeca model will be used in the upcoming stage for verification and performance evaluation.

6.1.5. Model verification results and performance evaluation of initial scenario

This phase of AdaptiveFlow can be performed by using one of the Rebeca model-checking tools such as Afra or RMC. For verification of a current model, an RMC has been used and the following results have been generated.

```
<model-checking-report>
    <system-info>
        <total-spent-time>3</total-spent-time>
        <reached-states>846</reached-states>
        <reached-transitions>958</reached-transitions>
        <consumed-mem>388400</consumed-mem>
    </system-info>
    <checked-property>
        <type>Reachability</type>
        <name>Deadlock-Freedom and No Deadline Missed</name>
        <result>satisfied</result>
    <options>
        <option>Export state space</option>
        <option>FTTS</option>
    </options>
    </checked-property>
    <counter-example-trace>
    </counter-example-trace>
</model-checking-report>
```

Listing 8: Model checking results

This listing shows the output of RMC in which it can be noticed that there have been 846 reached states and 958 reached transitions. Received results show that this model does not result in deadlock and that there are no missed deadlines. Another output of RMC is a state space file that shows all of the possible system configurations. Using this file, the *post-processing* phase of AdaptiveFlow is executed. By taking values from the state space, the statespace-analyzer tool derives the evaluation of performance properties. For this model, the performance properties are presented below.

```
Experiments operating time
MODEL: model
    FINAL STATE: 846_0
        VEHICLE: 0, time: 441
        VEHICLE: 1, time: 571
        VEHICLE: 2, time: 506
    TOTAL TRAVELLED TIME: 1518

Experiments material
MODEL: model
    FINAL STATE: 846_0
        VEHICLE: 0, material:  30
        VEHICLE: 1, material:  30
        VEHICLE: 2, material:  30
    TOTAL MATERIAL MOVED: 90

Experiments CO2 emission
MODEL: model
    FINAL STATE: 846_0
        VEHICLE: 0, emission:  16
        VEHICLE: 1, emission:  35
        VEHICLE: 2, emission:  22
    TOTAL CO2 EMISSION: 73
```

Listing 9: Performance evaluation results

By the results of performance evaluation, each machine needs roughly 8 minutes to execute their tasks. The total material moved is 90 tonnes because the machine load capacity has been set to 15 and all machines are loaded and unloaded twice in this scenario. The CO_2 emission calculation is an interesting feature of the AdapiveFlow. Here the total CO_2 emission is presented based on the state space file. Modeling a close scenario to the one defined at the beginning of this section and its verification has been performed. Based on results, this model has passed all assertions and there were no collisions or deadlocks detected. The next step is to implement the logic of AdaptiveFlow, in which all edges are controlled in order to get simulation results.

6.1.6. Changes inside the Simulator scenario based on the AdaptiveFlow

After running the model verification and its performance evaluation, which was passed successfully, the last step of the reverse engineering procedure is to make crucial, but not too radical changes to detach it from the initial scenario. This is mandatory so that the scenario would work in a predictive way, without any flaws.

In the initial version of the scenario, the control was established only over the 4 edges, whereas the other edges remained uncontrolled. This was the main cause of the crash occurred in that scenario. Following the AdaptiveFlow logic in which all tracks communicate between each other, the sensor on each track has been set up. By doing this, the control over every edge in the scenario has been established. To define the control system over every edge the desired logic needed to be implemented. To each *Edge* object inside the simulator, the *EdgeControl* object was added to it as a sub-element that would determine the behavior of that Edge. To initiate the *EdgeControl* we needed to define some additional attributes to each *Edge*. This was done dynamically through the code and these new edge attributes are presented and described below.

- **poiType** – This attribute was derived from the AdaptiveFlow topology file in which essential PoIs have been defined. The values of this variable can be: *track*, *loading*, *unloading*, *parking* and *charging*.
- **waitTime** – This value is used for charging, loading and unloading PoIs. It represents the value of time needed to execute a certain task on that PoI.
- **capacity** – Similarly to the segment capacity, edge capacity represents the number of machines that are allowed to be on it at the same time.
- **load** – The number of machines that are currently on the edge is controlled with this variable.
- **length** – This is a variable calculated from the path points of the edge and it represents the edge length.
- **speed** – This variable represents the allowed speed on each edge. This is equivalent to he *freespeed* variable from AdaptiveFlow environment file.

The *EdgeControl* logic was implemented with the Javascript programming language. This script is set up in the sensor and each sensor is located at the endpoint of each edge. By using the attributes mentioned above, the script for edge control has been developed. The script is perceived as an internal part of the edge. There is also another part of the implementation that is confidential and that is the implementation in the internal code of the automated hauler. For the explanation purposes, it is only important to know that each automated hauler has an *AhMission* property defined that represents a set of edges. This set is the path that haulers need to go through in order to execute their tasks.

6.1.7. Edge control implementation

Before defining the function for the edge control the interacting objects and adequate variables have to be assigned. In the Listing 10 initialization of the variables has been shown. First, the *defineProperty* function is called. This function defines the property *in_touchingBodies* inside the edge object, which will refer to the machine touching the sensor. Next, the *currentAh*, *currentPath* and *nextPath* variables are initialized. The *currentAh* refers to the automated hauler that has

touched the sensor, the *currentPath* refers to the current edge which is used to change the load, that is the current number of machines on the edge, and the *nextPath* path is used to see whether the next edge is free and if it is to allow the current machine to enter that edge and to increase its load. Finally, the *currentPathIndex* is used as an index used to retrieve the current edge from *AhMission* and assign that object to the *currentPath* variable. The last variable initialized is the *nextPathAvailable* which is more or less self-explanatory. This variable is used to check whether the next edge is free for the current machine to enter.

```
defineProperty("in_touchingBodies", new ObjectRef());


var currentAh = new ObjectRef();     // to send event "playNext"
var currentPath = new ObjectRef();   // to change its load
var nextPath = new ObjectRef();      // to check its capacity and change its load
var currentPathIndex = 0;
var nextPathAvailable = false;
```

Listing 10: Initialized variables inside the script

After everything has been initialized, the next step is the event handling of sensor touching. This event is triggered when a machine touches the edge sensor and this function retrieves the automated hauler object which is handled with its values. To assign the machine to the *currentAh* variable, this function takes the object defined in the "in_touchingBodies" property of the *Edge* object. After the machine object is received, the values to other variables are set up.

The next condition that is checked is whether the current edge is the last one in the AhMission. If it is not, then the *nextPath* value receives an edge that comes up next. Otherwise, we need to see whether the AhMission is defined as a loop. If it is, then the AhMission will be repeated and if not, the machine will stop.

In the following step, we have to check whether the next path in the AhMission exists. If it does, we need to check if that track is available by checking its load property. After that, for every PoI, except for the parking spot, the function *sendEventRecursively()* is called. This function sends the automated hauler to the next track. In addition to that, this function is being delayed by the *waitTime* of the PoI if the current edge's *poiType* attribute is either **loading**, **unloading** or **charging**. The *waitTime* represents the necessary time for the execution of the task on the current edge.

Lastly, the load attribute of *currentPath* and *nextPath* are updated. For the current path, the load is decreased by 1 because the machine is leaving the current edge and for the next path, the load is increased since the machine has entered it.

```
handleProperty("in_sensorTouches", function () {
    delay(1.0, function () {
        // execute when in_sensorTouches change from 0 to 1 (and not from 1 to 0)
        if (property.in_sensorTouches) {
            currentAh = property.in_touchingBodies[0].scope;
            currentPathIndex = currentAh.AhMission.currentPath;
            currentPath = currentAh.AhMission.plannedMission[currentPathIndex];
            // If it is not the last path
            if (!(currentPathIndex == currentAh.AhMission.plannedMission.length -
                1)) {
                nextPath = currentAh.AhMission.plannedMission[currentPathIndex +
                    1];
            }
            else if (currentAh.AhMission.loop) { // Loop
                nextPath = currentAh.AhMission.plannedMission[0];
                console.log("Loop Loop Loop");
            }
            if (nextPath.isValid())
                // check availability randomly for concurrency
                delay(Math.round(Math.random() * (5 - 1) + 1), function () {
                    if (nextPath.Script.load < nextPath.Script.capacity)
                        nextPathAvailable = true;
                    if (nextPathAvailable)
                        if (currentPath.Script.poiType == "track") {      // check
                            the track type
                            sendEventRecursively(currentAh, "playNext", nextPath.
                                Script.speed);
```

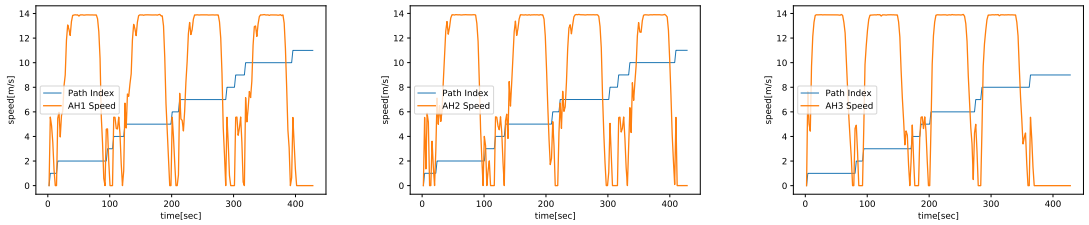
```
24                              // send other events
25                          }
26                          else if (currentPath.Script.poiType == "loading") {
27                              // send other events
28                              delay(currentPath.Script.waitTime, function () {
29                                  sendEventRecursively(currentAh, "playNext",
                                        nextPath.Script.speed);
30                              })
31                              // send other events
32                          }
33                          else if (currentPath.Script.poiType == "unloading") {
34                              // send other events
35                              delay(currentPath.Script.waitTime, function () {
36                                  sendEventRecursively(currentAh, "playNext",
                                        nextPath.Script.speed);
37                              })
38                              // send other events
39                          }
40                          else if (currentPath.Script.poiType == "parking") {
41                              // send other events
42                          }
43                          else if (currentPath.Script.poiType == "charging") {
44                              // send other events
45                              delay(currentPath.Script.waitTime, function () {
46                                  sendEventRecursively(currentAh, "playNext",
                                        nextPath.Script.speed);
47                              })
48                              // send other events
49                          }
50                          else {
51                              // send other events
52                          }
53
54
55                      // If automated hauler started to play the next path, update
                            the paths parameters
56                      delay(2.0, function () {
57                          if (currentAh.AhMission.currentPath == currentPathIndex +
                                1) {
58                              currentPath.Script.load--;
59                              nextPath.Script.load++;
60                          }
61                          else {
62                              // send other events (after 2 seconds, automated hauler
                                    is not playing the next path)
63                          }
64                      })
65                  })
66          }
67      })
68  });
```

Listing 11: Function for handling sensor touching event

6.1.8. Simulation results

After everything has been implemented correctly, we had to explore what performance properties can we measure with the Simulator. In Figure 16 we show the speed of each hauler during their task execution. Due to the bigger edge length, we can see periodic consistency at the peak of allowed speed.

In Figure 17 we present the relation between the total traveled distance of each hauler and time. Because of the randomized delay in the edge control, implemented for the concurrency purposes the third autonomous hauler finishes execution of its tasks first. Although AdaptiveFlow does not measure the traveled distance of each machine, we will use this data as complementary information to the evaluation of models.



(a) AH1 (b) AH2 (c) AH3

Figure 16: Fleet Management: Path index and autonomous hauler speed

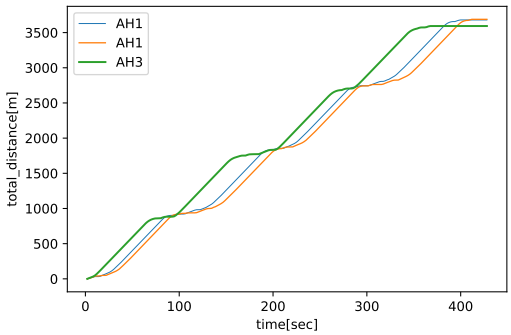


Figure 17: Fleet Management: Traveled distance of each autonomous hauler

6.1.9. Comparing the simulation results and model performance evaluation

The final phase of this process is to compare evaluations from the model and simulation. Identified property that can be compared between the two is the *operational time*. It is important to note that segment length in the models is probably not completely accurate to the represented simulation due to the difference in the level of abstraction. This is why the other two scenarios have been developed. It is necessary to obtain and compare results when these tools are orchestrated differently.

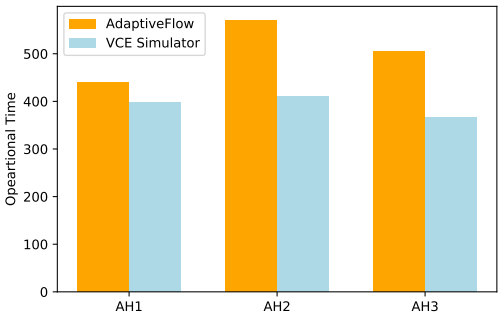


Figure 18: Fleet Management: Operating time in AdaptiveFlow and VCE Simulator

Here we can see that the model designed with AdaptiveFlow results in a higher operational time than its simulation equivalent. This can be attributed to the design of AdaptiveFlow that is exempt from details in the environment. To see whether this case stands we need to develop models that will be mapped to the Simulator in which the segment length of the model and track length in the simulation will be equal.

7. The Analysis Results of Case Studies

In this section, we have followed through the development of two new scenarios. The whole cycle of design, verification, mapping and evaluation of these three examples has been described. Each scenario focuses on a different perspective of tool orchestration. At the end of each subsection, we have compared results received from the Timed Rebeca model and simulation scenario.

In the Volvo Electric Site scenario, the aim was to develop a correct by design scenario from scratch, first in AdaptiveFlow and then following to that in the VCE Simulator. It was necessary to make sure that the newly designed model will be prone to errors and that it will not result in deadlock or an assertion check failure.

The counter-example was built in order to see how would we fix the falsely designed scenario by using the simulator. First, the designed model went through the model verification which reported an assertion failure. Next, that model was mapped to the simulator and we verified that the model is not correctly designed in the simulator also. Then, we have applied the fix in the simulation by modifying routes of machines and in that way, we ensured that there would be no deadlock or any other fallacies with the scenario. Finally, changes were then applied to the previously built model and result from the simulation and model evaluation were compared.

7.1. Volvo Electric Site example scenario

The second scenario which is covered was inspired by the schema from [6]. This schema is shown in Figure 19. For our case, we have chosen to design a scenario with three autonomous machines. In this case, there are two loading sections and one, common, unloading section. We need to develop a suitable AdaptiveFlow model which will go through model verification and performance evaluation. If the results were successful the next step is to map this model by using the VMap tool and we need to verify whether the properties hold.

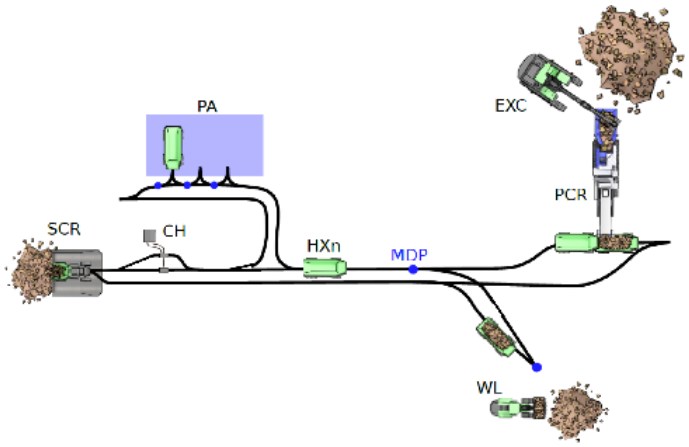


Figure 19: Volvo Electric Site Schema

7.1.1. Model verification and performance evaluation

The model developed consisted of 3 automated haulers (AH), 2 loading points and 1 unloading point. Depending on whether the model goes through the verification successfully, we need to see how much material was transferred and how long did it take in the performance evaluation of the model. Figure 20 represents a design of the model for the scenario shown in the previous section. The task for each hauler is presented below in Table 2 as the set of PoI IDs. Each machine moves in speed of 15m/s, loading time on both loading points is set to 20s and machine unloading takes 40s.

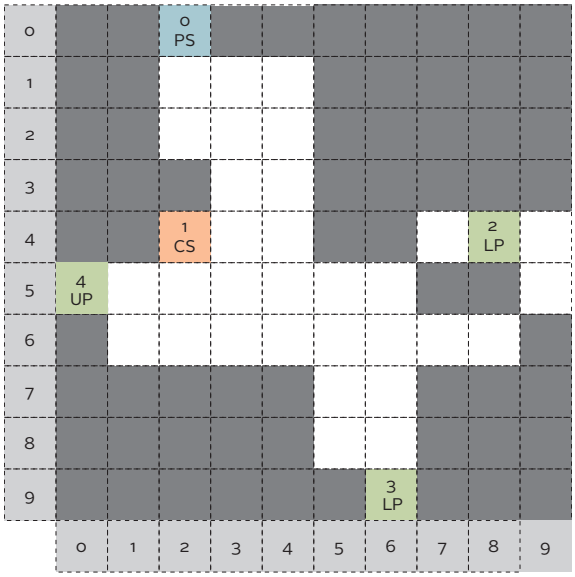


Figure 20: Volvo Electric Site: Model design

Machine	Tasks	Speed (m/s)	Capacity
AH 1	2, 4, 2, 4, 0	15	15
AH 2	3, 4, 3, 4, 0	15	15
AH 3	2, 4, 2, 4, 0	15	15

Table 2: Volvo Electric Site: Machine attributes and tasks

Model-checking results

After the designed model was generated, we had to run the model verification with the RMC tool. The resulting state space file was then inspected with the state-space-analyzer tool by which the performance results for this model were extracted. We were interested in the operational time and transferred material since those two properties can be evaluated in the simulator. Listings 12 and 13 shows the results of the model verification and performance evaluation.

```
<model-checking-report>
    <system-info>
        <total-spent-time>7</total-spent-time>
        <reached-states>1248</reached-states>
        <reached-transitions>1377</reached-transitions>
        <consumed-mem>499200</consumed-mem>
    </system-info>
    <checked-property>
        <type>Reachability</type>
        <name>Deadlock-Freedom and No Deadline Missed</name>
        <result>satisfied</result>
    <options>
        <option>Export state space</option>
        <option>FTTS</option>
    </options>
    </checked-property>
    <counter-example-trace>
    </counter-example-trace>
</model-checking-report>
```

Listing 12: Volvo Electric Site: Model checking results

```
Experiments operating time
MODEL: vce_model
    FINAL STATE: 1211_0
        VEHICLE: 0, time: 258
```

```
        VEHICLE: 1, time: 304
        VEHICLE: 2, time: 359
    TOTAL TRAVELLED TIME: 921

Experiments material
MODEL: vce_model
    FINAL STATE: 1211_0
        VEHICLE: 0, material: 30
        VEHICLE: 1, material: 30
        VEHICLE: 2, material: 30
    TOTAL MATERIAL MOVED: 90
```

Listing 13: Volvo Electric Site: performance evaluation results

The results show that the model of this system does not lead to a deadlock nor any deadline is missed. The performance evaluation results display that each machine needs about 5 minutes to execute their tasks and each machine has moved 30 tonnes of material.

7.1.2. Mapping the model from AdaptiveFlow to the VCE Simulator

The third step of this process consists of mapping the inputs of this model to the VCE simulator. This was done by using the VMap tool. By generating the *dynamic.content* example file the information on how each edge in the simulation is supposed to be defined. The attribute of each hauler, *AhMission*, is set according to the set of edges which lead to PoIs assigned in tasks. The image is file generated as a result of the conversion of AdaptiveFlow inputs to suitable 2D representation for the simulator. By combining this and *dynamic.content* example file, the suitable scenario was designed in the Simulator. In Figure 21 we can see how the environment was set up to conduct the experiment by following the guidelines.

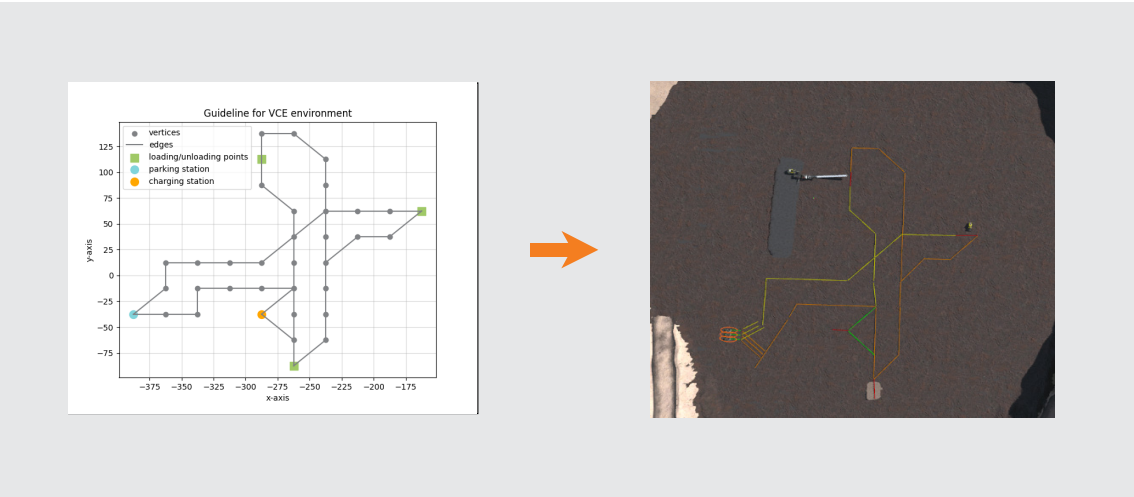
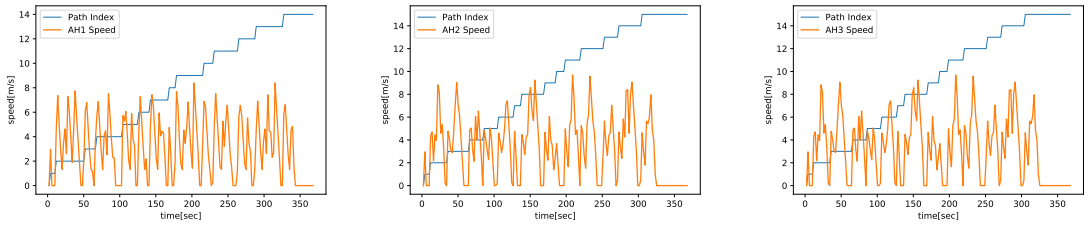


Figure 21: Volvo Electric Site: Transformation from guideline to the Simulator environment

7.1.3. Simulation results

In relation to the previous case, this scenario has a more dynamic change of hauler speed. We can see frequent oscillations in speed due to the shorter length of edges. This is the case because of the arrangement of edge sensors in the environment. Each sensor causes machines to stop because the edge control is triggered. Edge control after that redirects the machine to its next path. In Figure 22 we can see that machines do not reach the peak of allowed speed because of the edge shortness and because of the acceleration and deceleration. These details are not covered in the model verification.

Figure 23 represents the distance traveled within a specified time for each machine. Inconsistency in speed shows the difference in the smoothness of the line representing the traveled distance in relation to the previous case.



(a) AH1 (b) AH2 (c) AH3

Figure 22: Volvo Electric Site: Path index and autonomous hauler speed

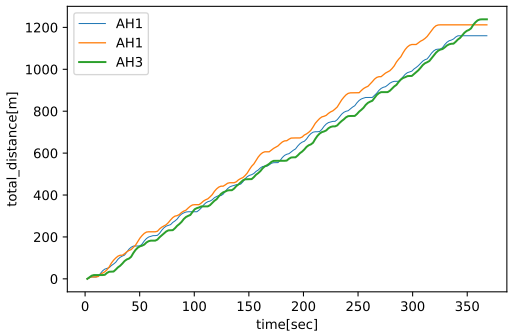


Figure 23: Volvo Electric Site: Traveled distance of each autonomous hauler

7.1.4. Comparing the simulation results and model performance evaluation

In this case, we can see a shift when it comes to operational time measurement with the tools used. This is caused mostly because of the more accurate representation of the Rebeca model inside the simulation. In Figure 24 we can see that there is no much difference in the operational time between the model and Simulator scenario, except for the first machine.

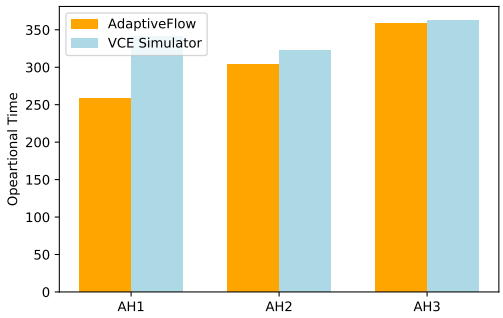


Figure 24: Volvo Electric Site: Operating time in AdaptiveFlow and VCE Simulator

7.2. Counter-example scenario

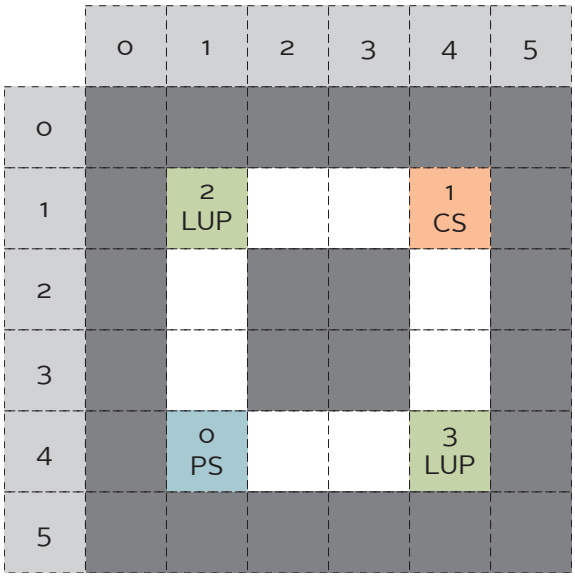


Figure 25: Counter-example: Model design

To make this story complete, we have built a third scenario which represents a counter-example. We had to make sure that the model was designed in such a way that it causes an assertion failure and/or deadlock. Figure 25 shows the design of this model. The main features of the scenario are listed below.

- One loading site with loading time 2s.
- One dump site with dumping time 4s.
- One charging station.
- Three autonomous haulers (AHs).

In Table 3 machine attributes and tasks are presented.

Machine	**Tasks**	**Speed (m/s)**
AH 1	2, 3, 2, 3, 0	14
AH 2	3, 2, 3, 2, 0	14
AH 3	2, 3, 2, 3, 0	14

Table 3: Machine attributes and tasks for a counter-example scenario

The scenario specification also included battery consumption, even though the battery consumption is not implemented in the simulator. This was done to retrieve results from the model checking, otherwise, the model verification would be executed indefinitely.

7.2.1. Model-checking results

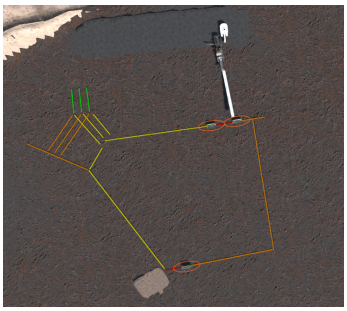
From the specification above it is more or less trivial why would it come to the failure of the system. Listing 14 shows model verification results with this configuration. In this case, two vehicles need to recharge their batteries and try to access the charging station from two different segments (from West and South). However, at the same time a third vehicle, that has been just refilled, wants to leave the charging station. Since the segment's capacities are all equal to one, they will attempt to access their next segment until they run out of battery.

```
<model-checking-report>
<system-info>
    <total-spent-time>0</total-spent-time>
    <reached-states>197</reached-states>
    <reached-transitions>214</reached-transitions>
    <consumed-mem>28368</consumed-mem>
</system-info>
<checked-property>
    <type>Reachability</type>
    <name>Deadlock-Freedom and No Deadline Missed</name>
    <result>assertion failed</result>
    <message>The battery is out of charge!</message>
<options>
    <option>Export state space</option>
    <option>FTTS</option>
</options>
</checked-property>
```

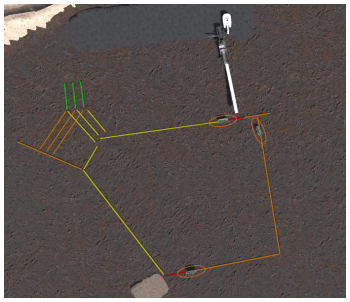
Listing 14: Model checking results

7.2.2. Implementation of the counter-example in the simulator

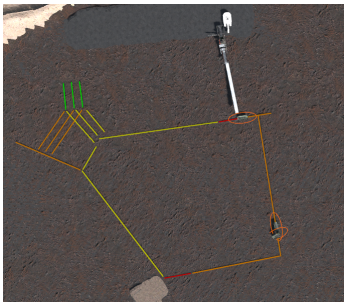
Given that the implementation of the battery consumption is not implemented in the simulation, we have assigned machine tasks so they could move in the same direction as they move in the model. It turned out that this case would cause the deadlock as the machines would stop after a certain period because they would try to go in the opposite direction. To demonstrate the malfunction of this model, we have disabled machine sensors and executed the scenario. Figure 26 shows snapshots of the scenario execution.



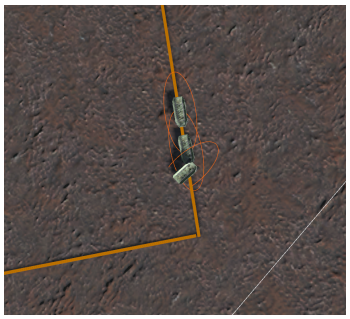
(a) Snapshot 1



(b) Snapshot 2



(c) Snapshot 3



(d) Snapshot 4

Figure 26: Counter-example: Scenario execution

The next step was to modify the scenario to provide a correct execution of machine tasks. The main change reflected in the changing of *AhMission* for each machine. Figure 27 shows the setup of the scenario in the simulator.

The modified scenario consisted of 1 loading site and 1 unloading site and 7 tracks. Each track has a specific feature such as allowed speed, type and waiting time. In Table 4 the attributes of each path are presented.

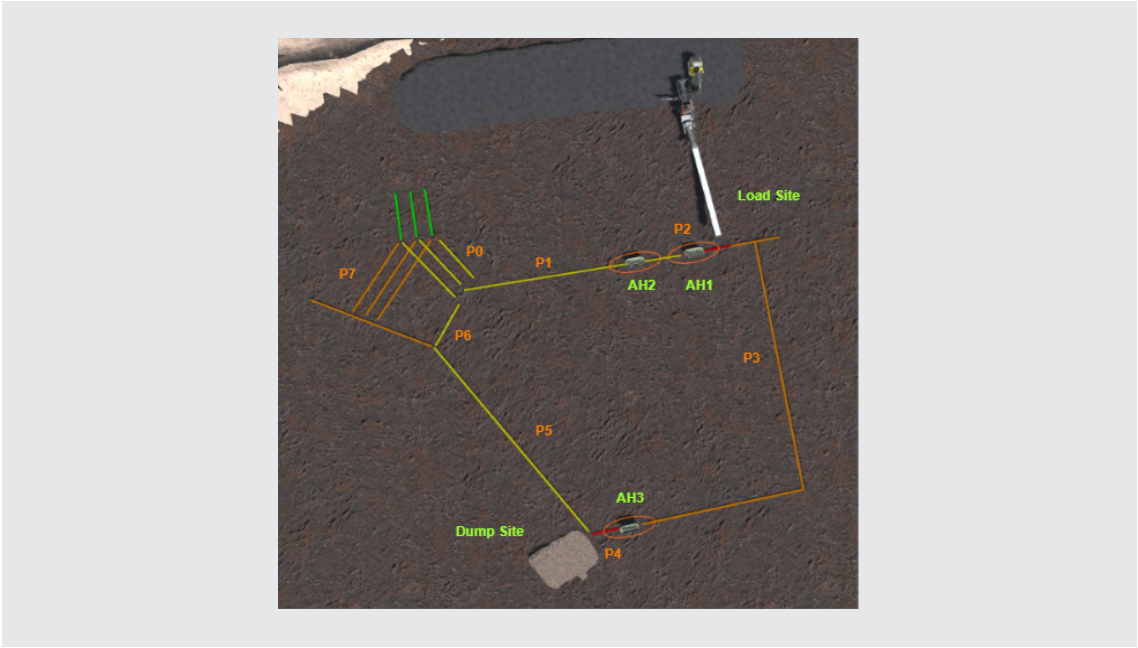


Figure 27: Counter-example: Simulator environment

Path number	Path type	Speed (km/s)	Waiting time (s)
P0	Track	20.0	0.0
P1	Track	50.0	0.0
P2	Loading	20.0	2.0
P3	Track	50.0	0.0
P4	Unloading	50.0	4.0
P5	Track	20.0	0.0
P6	Track	20.0	0.0
P7	Parking	20.0	10.0

Table 4: Paths in the counter-example scenario

To run this scenario successfully we had to make some changes relative to the initial setup. It was necessary to ensure that machines would go through the same tasks as in the model. Hence, the main solution was to modify machine routes to each task to avoid deadlock. After going through changes and testing the final routes for each machine are derived. In Table 5 the final *AhMission* for each autonomous hauler is displayed.

AH\Path	1	2	3	4	5	6	7	8	9	10	11	12	13	14
AH1	P2	P3	P4	P5	P6	P1	P2	P3	P4	P5	P7			
AH2	P4	P5	P6	P1	P2	P3	P4	P5	P6	P1	P2	P3	P4	P5
AH3	P1	P2	P3	P4	P5	P6	P1	P2	P3	P4	P5	P7		

Table 5: Mission of each automated hauler in the counter-example scenario

7.2.3. Simulation results

After the configuration has been set, the scenario was executed. Figures 28, 29 and 30 present results extracted from the simulation. Figure 28 shows the route of each automated hauler in the 2D space representation. Next, Figure 29 is a representation of a change of speed during a certain amount of time. Besides, the change of the path inside *AhMission* is also shown to see a relation

between the speed and the path index. Finally, the traveled distance has been compared between automated haulers in Figure 30.

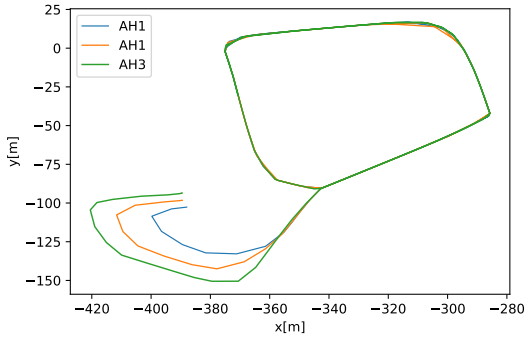
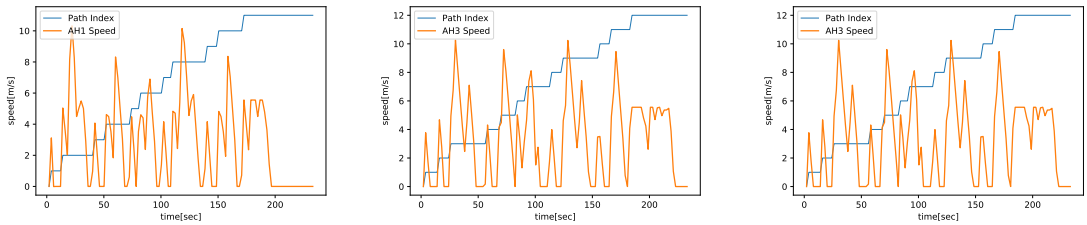


Figure 28: Counter-example: The path played by each autonomous hauler



(a) AH1 (b) AH2 (c) AH3

Figure 29: Counter-example: Path index and autonomous hauler speed

In Figure 29 we can notice the oscillations of the hauler speed when performing tasks. These oscillations are caused by the implementation inside the simulator where the machine after reaching the end sensor of each edge waits to be directed to the next path.

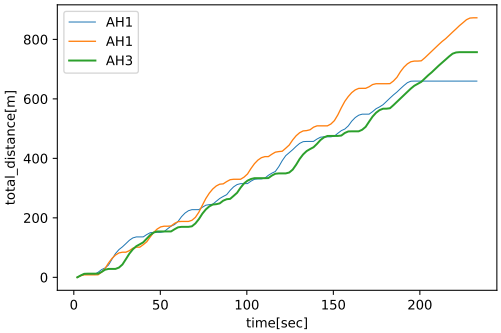


Figure 30: Counter-example: Traveled distance of each autonomous hauler

As we have successfully executed the scenario, the next step is to make identical changes within the model. If the verification proves to be successful, the final step is to compare the results.

7.2.4. Comparing the simulation results and model performance evaluation

To correct the model we have used a *policy 4* in AdaptiveFlow which allows us to define strict routes between the PoIs that machines should follow. The changes were made in *topology* file and

the path between all PoIs have been assigned based on Table 5. With these changes, the model has successfully passed the verification, and the results of the performance analysis are extracted. The main property which we are interested in is the *operating time* because we can compare it to the simulation. The *operating time* results for this model is presented on Listing 15.

```
Experiments operating time
MODEL: model4
    FINAL STATE: 1143_0
        VEHICLE: 0, time: 113
        VEHICLE: 1, time: 122
        VEHICLE: 2, time: 127
    TOTAL TRAVELLED TIME: 362
```

Listing 15: Performance evaluation results of modified model

The last phase of analysis is the comparison between results from the simulation and a model. A bar chart presented in Figure 31 shows the difference in operational time between the AdaptiveFlow model and the VCE Simulator scenario.

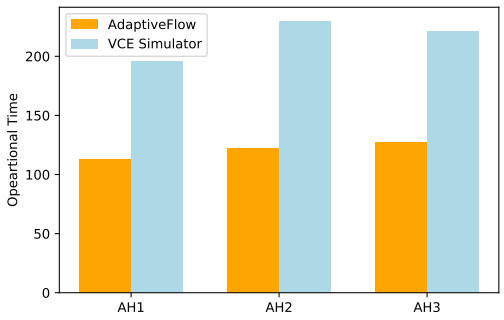


Figure 31: Counter-example: Operating time in AdaptiveFlow and VCE Simulator

The discrepancy between the results of the operating time in the model evaluation and simulation is noticeable. Each machine takes longer to execute their tasks in the simulation environment mostly due to the delay in the simulation where each machine has to wait a few seconds before it is sent to another edge. This causes speed oscillations that do not happen in Rebeca models, where speed is more or less consistent.

8. Discussion

Throughout this work, we have covered the Volvo Electric Site case as we have performed formal verification and evaluation of collaborative systems. The main goal was to discover how could abstract Timed Rebeca models, generated by AdaptiveFlow, be transformed into a complex 3D simulation in Volvo Simulator scenarios. This has been done by developing a VMap tool which is used as a link between abstract models and simulation. VMap was envisioned to provide an input file that can be directly used in the Simulator. Due to the inaccessibility to the most internal parts of the Simulator, VMap provides an example of the input file that could be used as a directive for creating an adequate scenario. This tool transforms the segmented 2D environment from AdaptiveFlow to 3D simulated complex environment.

The results of the orchestration have been extracted by examining three case studies covered in this thesis. Each case covers a unique way of tool orchestration. The results indicate that design, verification and evaluation of collaborative systems could be performed with AdaptiveFlow and VCE Simulator as complementary tools. The analysis confirms that AdaptiveFlow can be used as both, design and analysis tool. By conducting case studies we have derived three different methods of performing safety and performance evaluation of collaborating heavy machines. In the "Fleet management" scenario, evaluation of the system was performed by an in-depth analysis of the existing scenario, developing a suitable model with AdaptiveFlow and finally implementing the track-based control logic to the simulation which was used for other cases also. In the Volvo Electric Site example, we have developed a model from scratch. This model had to go through safety verification successfully to ensure the correct design of the system. Through the medium of the VMap tool, the simulation of this model has been developed. The final scenario represented a counter-example which was developed to verify whether the faulty models developed with AdaptiveFlow would cause faults in the simulation. Additionally, the VCE Simulator has been used as a correction tool where we modified scenarios to ensure proper execution and those changes have been reversely mapped to the AdaptiveFlow to verify whether a model is correctly designed.

The results show a correlation in safety verification between AdaptiveFlow and VCE Simulator. The common property in performance evaluation between these tools was the operational time. Results differ depending on the approach to the tool orchestration. If AdaptiveFlow is perceived as an analysis tool there can be the inaccuracy of some details of the system due to lifting the complex simulation to a simplified model. Comparing the results from the first case to the other two examples we can see a shift in results. In the first case, where AdaptiveFlow is seen as an analysis tool, the performance evaluation shows that operational time is higher in Rebeca models. However, this could go the other way around, depending on the method of deriving adequate AdaptiveFlow specifications. On the other hand, two other cases provide a more suitable representation of the model in the simulation and results here show that AdaptiveFlow provides models with lower operational time in relation to the scenario in the Simulator. If we see AdaptiveFlow as a design tool, this would provide more reliable data for its comparison with VCE Simulator. From retrieved results, we can conclude that Rebeca models exclude some of the real-world constraints, such as machine acceleration. The reliability of retrieved data is impacted by the implementation of the edge control inside the simulator since its execution causes a delay in machine movement. Other performance properties have been used as complementary data for the system evaluation.

This work provides the applicability of AdaptiveFlow in the specific domain by using tool orchestration. This could be seen as an extension to [1] and [6] where the models generated with AdaptiveFlow have been shifted to an adequate simulated environment. AdaptiveFlow, VMap and VCE Simulator could be used as a tool stack for the design and analysis of collaborative systems. As a difference to papers such as [28], [29] and [30], we provide a different perspective when it comes to formal verification of autonomous systems.

Further research is required to establish the adaptability in the simulation by implementing policy such as rerouting taken from the AdaptiveFlow. Besides, more cases with varying complexities should be examined in order to provide more reliable generalization.

9. Conclusions

The formal verification of collaborative systems represents a complex process in which multiple tools are involved. This thesis provides a proposal of the tool orchestration methods for verification of these types of systems for the electrified quarry site. The verification was done through developing and evaluating models with the AdaptiveFlow framework that were mapped with the VMap tool to the Volvo Simulator. The mapping procedure was required to remove a gap between abstract Timed Rebeca models and simulation scenarios. All of these tools are a part of tool stack adequate for addressing the complexity of safety verification and performance evaluation of collaborative autonomous machines.

Through the development of three case studies, these methods are proven as adequate for developing correct-by-design systems. Features of AdaptiveFlow and VCE Simulator present different ways of evaluation of these systems. Safety verification is shown as a common ground between these tools. The difference in the evaluated performance properties is viewed as complementary information that provides a broader picture of the system itself. Regardless, the operational time is recognized as a common performance property that can be evaluated. A combination and a mapping between AdaptiveFlow and VCE Simulator impart a more reliable verification that allows a more secure and efficient development of collaborative systems.

The extension of this work includes the development of more cases with different specifications. Additionally, the VMap tool should be upgraded as the understanding of specific modules of the VCE Simulator increases. Following that, the AdaptiveFlow framework is also a work in progress that is getting more sophisticated as its usage expands. Also, we see an open field for research when it comes to the automatization of reverse mapping in which complex environments would be translated into abstract models. Finally, the domain of this analysis should broaden to other use cases as these tools continue to evolve.

References

[1] M. Sirjani, G. Forcina, A. Jafari, S. Baumgart, E. Khamespanah, and A. Sedaghatbaf, "An actor-based design platform for system of systems," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 579–587.

[2] C. Hewitt, "Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot," Massachusetts Institute of Technology Cambridge Artificial Intelligence Lab, Tech. Rep., 1972.

[3] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[4] M. Sirjani, A. Jafari, E. Khamespahanh, H. Hojjat, and Z. S. Kaviani, "Rebeca user manual," 2016.

[5] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingolfsdottir, and S. H. Sigurdarson, "Modelling and simulation of asynchronous real-time systems using timed rebeca," *Science of Computer Programming*, vol. 89, pp. 41 – 68, 2014, special issue on the 10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642314000239

[6] G. Forcina, E. Khamespanah, A. Jafari, A. Sedaghatbaf, S. Baumgart, and M. Sirjani, "Adaptiveflow : An actor-based eulerian framework for track-based flow management," Technical Report, 2019.

[7] A. Jafari, J. J. S. Nair, S. Baumgart, and M. Sirjani, "Safe and efficient fleet operation for autonomous machines: An actor-based approach," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 423–426. [Online]. Available: https://doi.org/10.1145/3167132.3167382

[8] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: John Wiley Sons, Inc., 2006.

[9] C. M. Holloway, "Why engineers should consider formal methods," in *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings*, vol. 1, 1997, pp. 1.3–16.

[10] K. Y. Rozier, "Specification: The biggest bottleneck in formal methods and autonomy," in *Verified Software. Theories, Tools, and Experiments*, S. Blazy and M. Chechik, Eds. Cham: Springer International Publishing, 2016, pp. 8–26.

[11] E. Clarke and J. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, 12 1996.

[12] J. S. P. S. M. d. S. a. José Bacelar Almeida, Maria João Frade, *Rigorous Software Development: An Introduction to Program Verification*, 1st ed., ser. Undergraduate Topics in Computer Science. Springer-Verlag London, 2011. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=3373A1C9FA04CAF5742589CA18F81459

[13] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, Feb. 2009. [Online]. Available: https://doi.org/10.1145/1459352.1459354

[14] "Introduction to formal verification." [Online]. Available: https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html

[15] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 102–126.

[16] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "Towards a theory of actor computation," in *CONCUR '92*, W. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 565–579.

[17] M. Sirjani and M. M. Jaghoori, *Ten Years of Analyzing Actors: Rebeca Experience*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 20–56. [Online]. Available: https://doi.org/10.1007/978-3-642-24933-4_3

[18] "Rebeca modeling language." [Online]. Available: http://rebeca-lang.org/

[19] "Electric site research project." [Online]. Available: https://www.volvoce.com/global/en/this-is-volvo-ce/what-we-believe-in/innovation/electric-site/

[20] A. Böckenkamp, F. Weichert, J. Stenzel, and D. Lünsch, "Towards autonomously navigating and cooperating vehicles in cyber-physical production systems," in *Machine Learning for Cyber Physical Systems*, O. Niggemann and J. Beyerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 111–121.

[21] M. Bagheri, M. Sirjani, E. Khamespanah, N. Khakpour, I. Akkaya, A. Movaghar, and E. A. Lee, "Coordinated actor model of self-adaptive track-based traffic control systems," *Journal of Systems and Software*, vol. 143, pp. 116 – 139, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218301092

[22] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, and M. Cimini, "Ptrebeca: Modeling and analysis of distributed and asynchronous systems," *Science of Computer Programming*, vol. 128, pp. 22 – 50, 2016, special issue on Automated Verification of Critical Systems (AVoCS'14). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642316000800

[23] E. Khamespanah, K. Mechitov, M. Sirjani, and G. Agha, "Schedulability analysis of distributed real-time sensor network applications using actor-based model checking," in *Model Checking Software - 23rd International Symposium*, vol. 9641, April 2016. [Online]. Available: http://www.es.mdh.se/publications/4612-

[24] R. Saddem, O. Naud, K. G. Dejean], and D. Crestani, "Decomposing the model-checking of mobile robotics actions on a grid," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 11 156 – 11 162, 2017, 20th IFAC World Congress. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896317317445

[25] A. Ulusoy, S. L. Smith, and C. Belta, "Optimal multi-robot path planning with ltl constraints: Guaranteeing correctness through synchronization," in *Distributed Autonomous Robotic Systems*, M. Ani Hsieh and G. Chirikjian, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 337–351.

[26] D. Weyns and T. Holvoet, "Architectural design of a situated multiagent system for controlling automatic guided vehicles," *Int. J. Agent-Oriented Softw. Eng.*, vol. 2, no. 1, p. 90–128, Jan. 2008. [Online]. Available: https://doi.org/10.1504/IJAOSE.2008.016801

[27] J. Jayanthi Surendran Nair, "Modelling and analysing collaborating heavy machines," Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2017.

[28] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons, "Toward reliable autonomous robotic assistants through formal verification: A case study," *IEEE Transactions on Human-Machine Systems*, vol. 46, no. 2, pp. 186–196, 2016.

[29] R. Gu, R. Marinescu, C. Seceleanu, and K. Lundqvist, "Formal verification of an autonomous wheel loader by model checking," in *2018 IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2018, pp. 74–83.

[30] A. Zita, S. Mohajerani, and M. Fabian, "Application of formal verification to the lane change module of an autonomous vehicle," in *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, 2017, pp. 932–937.

[31] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," 2013.

[32] "Volvo simulators." [Online]. Available: https://www.volvoce.com/europe/en/services/volvo-services/productivity-services/volvo-simulators/