KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — 3001 Leuven

Simulation of Distributed Control Applications in Dynamic Environments

Promotoren :
Prof. Dr. T. HOLVOET
Prof. Dr. ir. Y. BERBERS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Alexander HELLEBOOGH

Mei 2007

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — 3001 Leuven

Simulation of Distributed Control Applications in Dynamic Environments

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Alexander HELLEBOOGH

U.D.C. 681.3∗I6, 681.3∗C24, 681.3∗D2

Mei 2007

D/2007/7515/69
ISBN 978-90-5682-835-6

Abstract

Distributed control applications are software systems designed to coordinate and control the operation of several distributed devices. An example of a distributed control application is a software system that controls production machines in a manufacturing environment. The environment in which a distributed control application operates is typically dynamic. In a dynamic environment the operating conditions of the control application are continuously changing. For example, in a manufacturing environment new materials and product orders may arrive; other machines, vehicles and/or humans are operating, etc. It is essential that a distributed control application takes into account the dynamic environment in which it operates.

Simulation is imperative for the development of distributed control applications. Simulation offers a safe and cost-effective way for studying, evaluating and configuring the behavior of a distributed control application in a simulated environment before it is deployed in the real world. In this dissertation, we focus on *software-in-the-loop simulation* of distributed control applications in dynamic environments. Software-in-the-loop simulation means that the software of the real distributed control application is embedded in the simulation, i.e. the control software itself is part of the simulation loop. Existing approaches to support this family of simulations either rely on (1) general-purpose modeling constructs that are formally specified, but offer no support specifically targeted at this family of simulations, or on (2) informal abstractions that offer support specifically targeted at this family of simulations, but of which the meaning is implicit and coupled to the implementation of a particular simulation platform.

We put forward a formally founded modeling framework for software-in-the-loop simulations of distributed control applications in dynamic environments. The constructs of the modeling framework offer support that is specifically aimed at this family of simulations. Moreover, the modeling constructs are formally specified, which is crucial to decouple the simulation model from the simulation platform to execute the model.

The modeling framework captures core characteristics of this family of simulations in a first-class manner. The modeling framework comprises an environment part and a control application part. The *environment part* offers special-purpose modeling constructs for dynamic environments. These modeling constructs capture (1) the structure of the environment, (2) dynamism in the environment, (3) the way dynamism is affected by the sources of dynamism and (4) the way dynamism can interact. The *control application part* offers special-purpose modeling constructs for integrating the software of a real distributed control application in the simulation model. These modeling constructs capture (1) the execution time of the control software and (2) the interface of the control software for interacting with the environment.

To validate the modeling constructs, we developed a simulation platform that supports the constructs in an executable simulation, and we used the constructs to underpin a simulator for an industrial case, i.e. a distributed control application controlling unmanned vehicles in a warehouse environment. The simulator comprises a simulation model that is decoupled from the simulation platform to execute it. This enables customizing the simulation model, which is paramount to support the study and evaluation of different functionalities of the distributed control application.

Voorwoord

Ik heb me vroeger wel eens afgevraagd hoe ik me zou voelen, zo op het einde van mijn doctoraatsonderzoek. Zou ik vooral tevreden zijn met het werk op zich? Of eerder opgelucht dat het er op zit? Of wat meer volwassen, een beetje toch? Nu sta ik, na dik 5 jaar onderzoek, op dat moment waarnaar ik vroeger zo vol verwachting uitkeek. En het gevoel dat ik bij het terugblikken heb, zou ik misschien nog het best kunnen omschrijven als een gevoel van "verwondering", in verschillende opzichten.

In een eerste opzicht is er de verwondering voor het onderzoek zelf. Achteraf bekeken is het fascinerend hoe ideeën tot stand zijn komen en geëvolueerd zijn. De ene keer was het een idee dat je plots te binnen schiet, bijvoorbeeld midden in de nacht, en waarvoor je dan nog even opstaat om het neer te schrijven – al was ik achteraf bezien soms beter blijven liggen. De andere keer was een opmerking over een paper, feedback op een presentatie of een losse discussie tijdens de koffiepauze de aanzet tot nieuwe of andere inzichten. Het heeft mij enorm gefascineerd hoe al die dingen die vaak op een grillige manier zijn ontstaan, uiteindelijk evolueren tot puzzelstukjes die een logische plaats krijgen in de grotere puzzel van een doctoraat.

Verwondering voel ik eveneens voor de uitgebreide ondersteuning die ik in mijn onderzoekswerk heb gekregen. Ik dank mijn promotor, Prof. Tom Holvoet, voor de kansen, de steun en het vertrouwen die ik de afgelopen jaren onafgebroken heb gekregen. Mijn co-promotor, Prof. Yolande Berbers, ben ik dankbaar voor de nodige impulsen voor mijn onderzoek. De andere leden van mijn begeleidingscommissie, Prof. Frank Piessens en Prof. Danny De Schreye, dank ik voor hun kritische feedback op de tekst van dit proefschrift. Tot slot bedank ik ook Prof. Adelinde Uhrmacher en Prof. Wouter Joosen voor hun bereidwilligheid om in mijn jury te zetelen. Last but not least dank ik de leden van AgentWise, en Danny Weyns in het bijzonder, voor de interessante discussies en de vele suggesties voor mijn onderzoek.

De grootste verwondering zit echter vervat in de ontelbare mooie momenten die ik tijdens de jaren van mijn doctoraatsonderzoek heb mogen beleven. Ervaringen samen met vrienden, (oud-)collega's en familie, die op mij een onuitwisbare indruk hebben nagelaten. En waaraan ik met een glimlach terugdenk.

Alexander Helleboogh, Mei 2007

Aan Ellen
Voor jouw onvoorwaardelijke liefde en steun

Contents

List of Figures

Chapter 1

Introduction

The goal of the work described in this dissertation is to underpin the development of simulations for testing distributed control applications by means of (1) new modeling constructs for simulating the environment of the control application, and (2) a plug-and-play manner to embed the control software in a simulation.

In this chapter, we clarify and elaborate on this statement. Section 1.1 outlines the context of our work. In section 1.2, we pinpoint the main problems our research is focussed at. We put forward the contributions of our research in Section 1.3. Finally, we give an outline of the text in Section 1.4.

1.1 Context

In this dissertation, we are concerned with software-in-the-loop simulations for distributed control applications. We first elaborate on distributed control applications and afterwards on simulations for such applications.

1.1.1 Distributed Control Applications

We employ the term *distributed control applications* to refer to a family of applications which share a number of characteristics.

A *control application* is a software system connected to an underlying physical or software environment [HJJ03]. The environment is the part of the external world with which the control application interacts, and in which the effects of the control software will be observed. The task of the control application is to ensure that particular functionalities are achieved in the environment. The interaction between the control application and the environment happens through sensors and actuators. An example of a control application is a car's cruise control system. The environment of the control application consists of the car and the road it drives on. The control application interacts with this environment through a sensor that can be used to observe the car's speed and an actuator that can be used to adjust the car's throttle. The task of the control application is to ensure that the car drives at a constant speed across the road.

A *distributed control application* is a control application of which the software is a distributed application [WSHL05]. A distributed application is a software system that consists of a number of components that are deployed on distinct computers connected by a network. An example of a distributed control application is an application to control a team of RoboCup Soccer robots [NRSSV05]. The task of the control application is to score goals and to prevent the opponent team from scoring, by using the robots of the team. The control application is distributed, as components are deployed on each of the robots. Each of the components has access to the sensors and actuators of a particular robot, and coordinates by communicating with other robots of the team in order to achieve the desired overall behavior.

Typically, the environment of a distributed control application is *highly dynamic*. A dynamic environment is an environment that is under constant change [RN95]. In a dynamic environment, the operating conditions of a distributed control application are continuously changing. Dynamism in the environment can originate from various sources. For example, in a RoboCup Soccer environment, dynamism encompasses the ball that is rolling, the movements of other robots of the team, the movements of the members of the opponent team and even disturbances such as the limited accuracy of passing the ball in a particular direction or a sudden breakdown of a particular robot.

A dynamic environment can have a significant impact on the actions of the distributed control application [FM96, Woo01]. In a dynamic environment, the actions of a control application do not always proceed as expected, but can be affected by the environment. For example, the component of a particular RoboCup robot triggers the actuators to move the robot forward, in order to kick the ball in front of it to a team member. In a dynamic environment, this action could be affected in different ways. For example, the robot could collide with other robots that prevent it from kicking the ball, resulting in damage. Or the robot's movement could be affected by jitter in the hardware, causing the robot to hit the ball under a different angle. Or even if the robot succeeds in kicking the ball in the right direction, another robot could intercept the pass afterwards by moving into the path of the rolling ball.

We give other examples of dynamic environments that affect distributed control applications. Consider a physical manufacturing environment where a distributed control application controls and coordinates several production machines. The task of the control application is ensuring that the machines manufacture all products in time. Examples of dynamism in a manufacturing environment include the arrival of new product orders, the operation of machines that are manufacturing products, the movement of machines that transport products. Examples of the impact of the environment on actions of the control application include the obstruction of machines by other machines, people or obstacles, message loss for machines moving out of communication range, breakdown of machines. As another example, consider a distributed software environment that consists of digital content providers connected by a peer-to-peer network. The task of a distributed control application is to enable content sharing across the content providers. Dynamism in such an environment includes the entrance and exit of content providers, the exchange of content, changes of the available communication bandwidth. The environment affects the actions of the control application for example when content transmissions are delayed due to congestion or interrupted because the content receiver exits.

1.1.2 Simulation

It is clear that a distributed control application needs to take into account dynamism in the environment and its potential impact on actions. Before deployment of a distributed control application, it is crucial to experiment and test the behavior of the application in scenarios typically occurring in a dynamic environment.

Simulation can be defined as “the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system” [Sha98]. Two important phases of a simulation study are the *model formulation phase*, i.e. building a simulation model of the real system, and the *model translation phase*, i.e. translating the simulation model to an executable simulation.

Simulation is crucial to study and test the behavior of the distributed control application in scenarios typically occurring in dynamic environments [UK00, HRU03, RR03]. Simulation enables (1) safe experimentation and testing in high-risk scenarios, (2) executing experiments faster than real-time, and (3) setting up and monitoring experiments in a less costly way. For example, consider an experiment that involves a distributed control application to steer robots in a manufacturing environment. The goal of the experiment is to test the ability of the robots to avoid collisions in a scenario where communication services are temporarily unavailable. Performing such a test on physical robots is unfeasible because of (1) the high risk of damaging the robots, (2) the amount of time it takes to test long-term scenarios or slow robots, and (3) the cost associated with setting up and monitoring a large-scale experiment that involves many robots over an extended time period.

In simulation research and development, support has been developed for the model formulation and model translation phase:

1. *Modeling constructs support model formulation.* Various simulation paradigms exist that offer established modeling constructs to support formulating a simulation model. For example, discrete-event simulation offers constructs such as *state* and *events* to express a simulation model. A model's state is a list of values that are sufficient to define the state of the system at any point in time [Car03]. An event represents a change in the state of the simulation that occurs instantaneously at a well-defined point in simulation time [SB99]. In a discrete-event model, the state remains constant over intervals of time and changes value only at the time an event occurs. Using state and events, a developer can model a system and its evolution over time as an initial state and an ordered sequence of atomic state changes specified by time-stamped events. Other examples are continuous simulation [HW91] where models are expressed in terms of *state variables* and *equations*, and hybrid simulation [Mos99] where models are expressed in terms of *state variables*, *equations*, *time events* and *state events*.

2. *Simulation platforms support model translation.* Simulation platforms encapsulate the functionality that is needed to support the modeling constructs in an executable simulation. The encapsulated functionality can be reused for every model expressed in terms of the supported constructs, accelerating the development of an executable simulation. For example, simulation platforms for discrete-event simulation encapsulate mechanisms for guaranteeing that events are always applied in increasing time stamp order [Fuj98], even in the presence of arbitrary delays in the underlying execution platform. Encapsulating such functionality in a simulation platform prevents that they have to be developed from scratch for each simulation study.

In this dissertation, we focus on *software-in-the-loop simulations of distributed control applications in dynamic environments.* Such simulations are used to test or configure the software of a distributed control application in a simulated environment before the software is deployed in its real environment [CK99]. Software-in-the-loop simulations of distributed control applications in dynamic environments have the following characteristics:

- The environment to-be-simulated is dynamic. A dynamic environment may contain sources of dynamism external to the distributed control application. These sources of dynamism can have a significant impact on the distributed control application, as they change the operation conditions of the application.

- The software of the real distributed control application is embedded in the simulation. The distributed control application is not substituted by a model, but the control software itself is part of the simulation loop, which is denoted by the term *software-in-the-loop* simulation.

1.2 Problem Statement

Developing software-in-the-loop simulations of distributed control applications in dynamic environments is complex. The system-to-be-simulated comprises two parts: a dynamic environment on the one hand and a distributed control application embedded in that environment on the other hand. We illustrate a number of challenges when building simulations for such systems:

- *Simulating dynamic environments is complex.* For example, in a dynamic environment the outcome of actions of a control application cannot be determined a priori [FM96, HHB05]. Other activities that are happening in the environment can have a significant impact on the outcome of actions. Consider a robot that was instructed to start driving north. In a dynamic environment, the action of the robot can be affected in different ways. For example, another machine could move into the path of the first robot, blocking it or pushing it aside. Or the robot's path could deviate from the intended path due to jitter in the hardware. Or the robot could run out of energy, causing its movement to stop prematurely. Even a combination of these phenomena could occur. When simulating dynamic environments, it is non-trivial to reproduce the variety of possibly cascading interactions that may occur and the precise way these interactions have an impact on the actions.

- *Integrating the software of a real distributed control application in a simulation is complicated.* For example, the devices on which the distributed control application is deployed in the real world determine how fast that application can execute and consequently how much time it takes the application to react to changes in the environment. However, the characteristics of the computer platform on which the simulation is executed, can differ significantly from the devices on which the control application is deployed in the real world. Moreover, a simulation can be executed faster or slower than real time. It is non-trivial to reproduce the real-world timing characteristics of a distributed control application in a simulation.

To support the development of software-in-the-loop simulations of distributed control applications in dynamic environments, a developer can rely on general-purpose simulation platforms or on special-purpose simulation platforms.

- *General-purpose simulation platforms* support the execution of simulation models that are described in terms of general-purpose modeling constructs. For example, *JAMES* [HRU03] (recently updated to JAMES II) is a simulation platform that supports discrete event models described in terms of the constructs of DEVS (Discrete EVent System specification) [ZP00]. DEVS is a modeling framework that supports atomic and coupled discrete event models. Atomic models are described in terms of modeling constructs such as a state set, input and output ports, internal and external transition functions, etc.

The meaning of general-purpose modeling constructs is formally specified. This is crucial to decouple the simulation model from the simulation platform to execute the model. As such, a modeler can use the modeling constructs for formulating a simulation model, without taking into account the simulation platform that will be used to execute the model.

However, general-purpose modeling constructs offer no support specifically targeted at software-in-the-loop simulation of distributed control applications in dynamic environments. General-purpose modeling constructs support a broad range of simulations, and their use is not limited to simulating distributed control applications. Consequently, general-purpose simulation platforms do not support the developer to tackle specific challenges associated with modeling distributed control applications in particular.

- *Special-purpose simulation platforms* are specifically aimed at simulating distributed control applications. For example, XRaptor [BMP$^+$] is a simulation platform to study the behavior of a large number of agents in two- or three-dimensional continuous virtual worlds. XRaptor describes an agent as either a point, a circular area or a spherical volume that contains a sensor unit for observing the world, an actuator unit for performing actions and a control kernel for action selection. Ordinary differential equations are used for modeling movements.

  Special-purpose simulation platforms incorporate support specifically aimed at software-in-the-loop simulation of distributed control applications in dynamic environments. For example, in XRaptor there is a world in which agents controlled by a control kernel can sense and act, etc. Compared to general-purpose simulation platforms, special-purpose simulation platforms support the developer to tackle specific challenges associated with simulating distributed control applications in particular.

  However, current special-purpose simulation platforms only provide informal abstractions of which the precise meaning is implicit and coupled to the design and implementation of a particular simulation platform. For example, as the XRaptor abstractions “control kernel”, “actuator unit” and “circular area” are not formally specified, their precise meaning and relations are vague. It is unclear how the control kernel triggers movements in the environment, or to what extent the timing characteristics of the control kernel are supported. Due to the lack of a formal specification, building a simulation supported by a special-purpose simulation platform requires detailed knowledge of its design and implementation, and results in a simulation which is tightly coupled with the simulation platform that is used to execute it.

To sum up, existing approaches to support the target family of simulations either rely on (1) general-purpose modeling constructs that are formally specified, but offer no support specifically targeted at this family of simulations, or on (2) informal abstractions that offer support specifically targeted at this family of simulations, but of which the meaning is implicit and coupled to the design and implementation of a particular simulation platform.

We conclude that there is a lack of formally specified modeling constructs that provide support specifically aimed at software-in-the-loop simulation of distributed control applications in dynamic environments.

1.3 Contributions

The main contribution of the research described in this dissertation is the introduction of a formally founded modeling framework for software-in-the-loop simulations of distributed control applications in dynamic environments. The modeling framework offers constructs for formulating a simulation model for this family of simulations and captures core characteristics of these simulations in a first-class manner. Moreover, the modeling constructs are formally specified to unambiguously specify their meaning and relations. This is crucial to decouple the simulation model from the simulation platform to execute the model. As such, the modeling framework enables formulating a simulation model without taking into account the design and implementation of the simulation platform to execute the model.

Concrete contributions are the following.

The introduction of a modeling framework for software-in-the-loop simulations of distributed control applications in dynamic environments [HHB05, HVUM07, HHW04a, HHW04b]. The modeling framework presents a coherent set of modeling constructs that capture core characteristics of this family of simulations in a first-class manner. The modeling framework comprises two parts: an environment part and a control application part. The *environment part* [HHB05, HVUM07] offers special-purpose modeling constructs for capturing dynamic environments in a simulation model. The *control application part* [HHW04a, HHW04b] offers special-purpose modeling constructs for integrating the software of a real distributed control application in the simulation model.

The presentation of a formal specification of the modeling framework [HVUM07, HHWB05a]. We present a formal specification to underpin the modeling framework. The advantage of the formal specification is twofold.

On the one hand, the formal specification is crucial to decouple the modeling constructs from their implementation in a particular simulation platform. This enables using the modeling constructs for formulating a simulation model while making abstraction of the simulation platform to execute the model. The formal specification unambiguously specifies the meaning of the modeling constructs, and describes the way the constructs are related to each other.

On the other hand, the formal specification enables a developer to consider several design alternatives for translating a simulation model into an executable simulation. The formal description specifies the functionality that is needed to support the constructs in an executable simulation, without commitment to particular design decisions. As such, the formal specification guides the development of an executable simulation and prevents reinventing its functionality from scratch.

The development of a simulation platform that supports the modeling constructs [HHW04b, WHH05]. We developed a simulation platform to demonstrate that the modeling framework is feasible for developing executable simulations. The simulation platform encapsulates the functionality to support the modeling constructs in an executable simulation. The simulation platform supports simulations (1) of which the simulation model is described in terms of the proposed modeling constructs, and (2) in which the software of a real distributed control application can be embedded.

We advocate that state-of-the-art software engineering principles, such as an elegant software architecture and advanced separation of concerns, are required to manage the complexity inherent to a simulation platform in a structured and evolvable way.

A validation in an industrial case [HHB06, HHWB05b]. To validate the usability of the modeling framework, we applied the modeling constructs and the simulation platform for simulating an industrial distributed control application, i.e. a warehouse transportation system where a distributed control application instructs automated guided vehicles (AGVs) to move and transport loads through a manufacturing environment. For this application, it is important to have a means to test new or altered features of the control application in a safe manner, i.e. without the risk of damaging the AGVs, and under a variety of scenarios, including those scenarios in which defects and potential conflicts such as collisions occur. We demonstrated the usability of the proposed constructs for the development of a simulation in which (1) the model of the manufacturing environment supports such scenarios, and (2) the control application can be embedded in a plug-and-play manner.

1.4 Outline

This dissertation is structured as follows.

Chapter 2, Background and Scope, gives the necessary background on simulation in the context of distributed control applications, and delineates the scope of our research.

Chapter 3, Modeling Dynamic Environments, introduces the environment part of the modeling framework, together with its formal specification. The focus is on the specification of the modeling constructs to capture dynamic environments in a simulation model. Design and implementation issues to support the constructs are tackled in Chapter 5.

Chapter 4, Modeling the Integration of the Control Software, introduces the control application part of the modeling framework, together with its formal specification. The focus is on the specification of modeling constructs to support the integration of the control software in the simulation model. Design and implementation issues to support the constructs are tackled in Chapter 5.

Chapter 5, Architecture of the Simulation Platform, shows the feasibility of the modeling framework described in Chapters 3 and 4. We describe the software architecture of a simulation platform that supports the modeling constructs in an executable simulation.

Chapter 6, Simulation of AGV Control Applications in Dynamic Warehouse Environments, demonstrates the usability of the modeling constructs and the simulation platform by applying them on a real-world case: software-in-the-loop simulation of distributed control applications for automated guided vehicles in dynamic warehouse environments.

Chapter 7, Conclusions, rounds up this dissertation by giving a high-level overview of our approach and the way it addresses the problems. We pinpoint the main contributions and suggest possible tracks for future work. We end with a closing reflection on the work presented in this dissertation.

Chapter 2

Background and Scope

In this chapter, we present the necessary background for the following chapters, and delineate the scope of our research.

Section 2.1 presents basic concepts of simulation in general. In Section 2.2, we discuss the characteristics of distributed control applications. Section 2.3 elaborates on different ways simulations can support the development of distributed control applications. In Section 2.4, we give an overview of existing simulation platforms for simulating distributed control applications. Finally, in Section 2.5 we delineate the scope of the research described in this dissertation.

2.1 Basic Concepts of Simulation

Simulation can be defined as "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system" [Sha98].

A *simulation model* is a representation of a real-world system that incorporates time and evolution, i.e. the changes that occur over time [Car03]. One should always design the simulation model around the questions to be answered about the system rather than try to imitate the real system exactly [Sha98]. A simulation model represents a subset of characteristics of the real system that is sufficient to serve the objective of the simulation study. A model should neither oversimplify the system, nor carry so much detail that is becomes expensive to build and run.

We elaborate different ways to incorporate time and evolution in a simulation model.

2.1.1 Different Concepts of Time

There are three different concepts of time that are important in the context of simulation [Fuj98].

- **Physical time** is the time in the system to-be-modeled. For example, a robot system starts operating at physical time 13:00:00 on June 2nd, till physical time 14:00:00 on June 2nd. In the robot system, a particular robot could start driving forward at physical time 13:01:34 and stops driving at physical time 13:01:55. So that particular movement takes the robot 21 physical seconds to complete.
- **Simulation time** (also called logical or virtual time) is the software representation of time used in a simulation. For example, in a simulation of a robot system, simulation time could be an integer number in the interval $[0, 3600]$, where a unit of simulation time corresponds to one second of physical time.
- **Wallclock time** is the time for running the simulation as measured on a physical clock. For example, a simulation run of the robot system could be started at 8:02:14 on July 20th and ends at 8:03:11 on July 20th. So it takes the simulation platform 57 seconds of wallclock time to advance simulation time from time 0 till time 3600.

According to the way simulation time advances in relation to wallclock time, a number of execution modes for a simulation can be distinguished [Fuj98]. In a *real-time simulations*, simulation time advances in pace with wallclock time. For example, running the simulation over one second of wallclock time advances simulation time by one second. In *as-fast-as-possible simulations*, simulation time is advanced as quickly as possible, without direct relationship to wallclock time. For example, running a simulation for 5 minutes of wallclock time may advance simulation time by 200 seconds, whereas running the simulation for another 5 minutes of wallclock time may only advance simulation time by 25 seconds.

2.1.2 Modeling Evolution

The evolution of a system over time can be modeled in a discrete, continuous or hybrid way. A vast amount of research exists on discrete, continuous and hybrid models. Rather than going into details, we limit the discussion to an explanation of the basic rationale for each group of models, give an example and indicate how the execution of such models can be supported in a simulation platform.

2.1.2.1 Discrete Event Models

In a discrete event model [CL99], the evolution of a system is modeled as discontinuous changes happening at discrete points in simulation time. A discrete

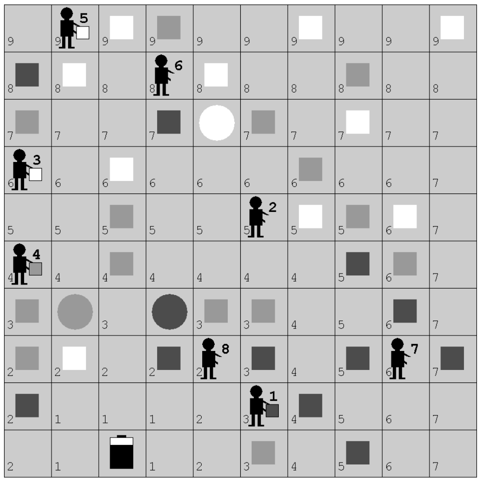


Figure 2.1: A screenshot of the Packet-World: a discrete model of robots and packets in a grid world.

event model is based on the concepts *state* and *events*. A model's state is a list of values that are sufficient to define the state of the system at any point in simulation time [Car03]. An event represents a change in the state of the simulation that occurs instantaneously at a well-defined point in simulation time [SB99]. In a discrete-event model, the state remains constant over intervals of simulation time and changes value only at the time instant an event occurs. Using state and events, a developer can model a system and its evolution over time as an initial state and an ordered sequence of atomic state changes specified by time-stamped events.

As an illustration of a discrete event model, consider the Packet-World simulation depicted in Figure 2.1. The state of the model comprises the position of black robots, square packets and circular destinations on a grid. The actions of the robots are modeled as events that change the position of robots and packets on the grid in a discrete manner.

We highlight two main approaches to support the execution of discrete event models in a simulation platform [FT94]:

- **Time-driven execution**. In this approach, simulation time is discretized in a number of intervals of equal size, e.g. 1 second intervals. The size of this interval is called the *step size*. The model is evolved forward in time by repeatedly adding the step size to the simulation clock, regardless of whether events are scheduled at that time or not. A simulation platform using time-driven execution is MASS [VHL01].

- **Event-driven execution**. In this approach, events are sorted according to their time stamp [Lam78, Mis86]. During execution, the next event to be processed is the one with the smallest time stamp. In contrast to the time-driven approach, event-driven execution is able to skip periods of inactivity. A simulation platform using event-driven execution is JAMES [SU01].

2.1.2.2 Continuous Models

In a continuous model, the evolution of a system is modeled as a continuous change of the state of the model over time. A continuous model is typically expressed as a *differential equation* or set of differential equations [S.S99]. The *order* or *dimension* of a continuous model is the number of state variables in the equations. The *state space* of a continuous model is the vector space in which the state of the model takes values.

An example of a one-dimensional continuous model is a differential equation that models the vertical movement of an object in gravitational free fall:
$x''(t) = -g$
$x'(0) = 0$
$x(0) = a$
$x(t)$ is the position of the object, $x'(t)$ the first derivative, i.e. its velocity, $x''(t)$ the second derivative, i.e. its acceleration, a is the initial height of the object and $-g$ is the gravitational acceleration.

We highlight two main approaches to support continuous models for conducting experiments:

- *Analytical solution.* Solving a continuous model analytically means calculating a mathematical expression for its *trajectory*, i.e. an explicit function that specifies the state of the system for each point in simulation time. For our example, the trajectory can be expressed as $x(t) = a - \frac{g*t^2}{2}$. An analytical solution for the continuous model specifies the evolution of the model in a compact and elegant way.

- *Numerical approximation.* In this case the trajectory is not formulated as a mathematical expression, but approximated using numerical algorithms for computing integrals, e.g. Euler or Runge-Kutta integration [FMM77]. Such algorithms divide simulation time in intervals of equal size, and approximate the trajectory over each interval.

2.1.2.3 Hybrid Models

In hybrid models [vdSS98], the evolution of the system is both continuous and discrete. In a hybrid model, the system evolves in continuous phases, alternated by discrete events. In a continuous phase, time advances, and the values of the state variables evolve continuously over time as determined by differential equations.

When a discrete event occurs, the equations and the state variables are altered in a discontinuous manner. Events can be of two kinds:

- *Time events.* Time events are scheduled at a predetermined time.
- *State events.* State events are scheduled at the occurrence of a particular condition, i.e. when the continuous phase exceeds certain thresholds. As such, it is not known a priori at what time a state event occurs.

As an example, consider a hybrid model of a bouncing ball. The continuous phase of the model is a differential equation of motion. As soon as the position of the ball in the continuous phase reaches the ground, a bounce occurs. A bounce is represented by a state event that changes the equation of motion and the state variables in a discontinuous manner, as the speed of the ball changes direction and has a smaller value as a fraction of the energy is lost.

The main challenge when performing simulation runs with a hybrid model, is managing state events [Mos99]. In contrast to time events, the simulation time at which a state event occurs, is not know a priori. Two main approaches exist for detecting state events and determining the time of their occurrence:

- *Retroactive detection* [Kam93], i.e. checking after each integrated time step whether a state event occurred.
- *Conservative advancement* [HBZ90], i.e. advancing the simulation conservatively by choosing the time step so that no state event will occur during it.

2.2 Characteristics of Distributed Control Applications

A distributed control application is a distributed software application that continuously and autonomously acts in, and reacts to, an underlying environment. Examples of distributed control applications include manufacturing control systems [VGVVB06, Bru00], collective robotic systems [GH04, PVR04, BJNT06], traffic control systems [Wan05, Roo99, DS05] and sensor networks [SSS$^+$03, DYP06]. Figure 2.2 gives a schematic overview of a distributed control application in an environment.

A distributed control application consists of several controllers. Controllers are active software components that are distributed in the environment and that cooperate to solve a particular problem in the environment. In Figure 2.2, three controllers are depicted: `Controller 1`, `Controller 2` and `Controller 3`.

The controllers of a distributed control application are deployed on particular devices in the environment. A device consists of a software and a hardware part.

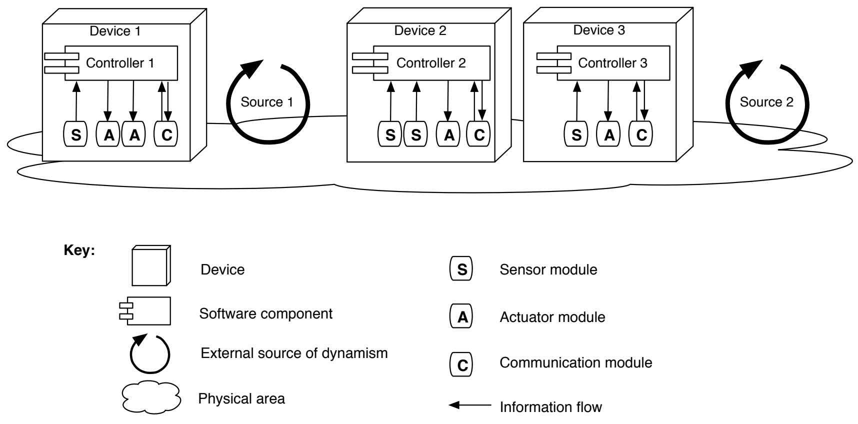


Figure 2.2: Schematic view of a distributed control application in an environment

The software part is one of the controllers that constitutes the distributed control application, whereas the hardware part comprises sensor, actuator and communication modules. A controller can use the sensor, actuator and communication modules of its device to interact with the environment. Figure 2.2 depicts three devices in the environment: `Device 1`, `Device 2` and `Device 3`.

The environment of a distributed control application is the part of the external world in which the problem resides and in which the effects of the control application, once installed and set in operation, will be observed [HJJ03]. Typically, a distributed control application operates in a *dynamic* environment, i.e. an environment where other sources of dynamism are present, e.g. other systems, processes or even humans. These sources of dynamism are external to the distributed control application. Figure 2.2 depicts two external sources of dynamism present in the environment: `Source 1` and `Source 2`. The operation of external sources of dynamism can have a significant impact on a distributed control application.

Designing and testing a distributed control application is complex as it requires an integrated approach that takes into account the environment in which the application is situated [HZ05a]. A distributed control application should take into account dynamism originating from other systems, processes or humans in the environment and react appropriately to their presence.

2.3 Simulation Modes for Distributed Control Application Development

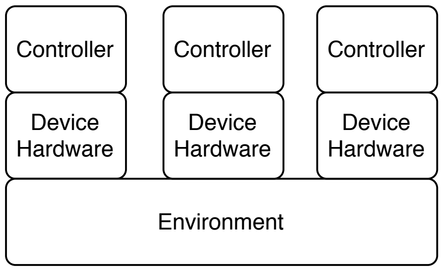
Modeling and simulation approaches are frequently used to support the development of distributed control applications. Simulations can help to analyze, design, configure and test a distributed control application [SPLK01]. The scope of a simulation for distributed control applications typically comprises the distributed control application situated in a particular environment. As such, the simulation incorporates external sources of dynamism in the environment that affect the operation of a distributed control application. In a simulation, the operation of the control application can be observed within a simulated problem setting in a dynamic environment.

We make a distinction between different simulation modes: model-in-the-loop simulation mode, software-in-the-loop simulation mode and hardware-in-the-loop simulation mode. The simulation modes differ in the way they integrate a control application "in the simulation loop". Model-in-the-loop simulation mode relies on a model to substitute the control application in the simulation. Software-in-the-loop simulation mode integrates the software of the real distributed control application in the simulation loop. Hardware-in-the-loop simulation mode integrates the hardware on which the real distributed control application is deployed in the simulation loop. Figure 6.2 gives a schematic overview of the simulation modes. We elaborate on each simulation mode and indicate how it can support the development of distributed control applications.

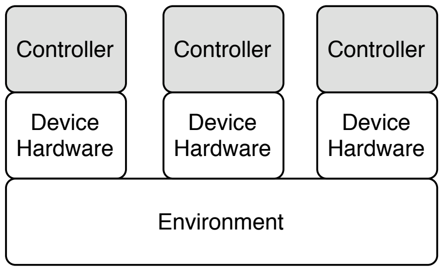
2.3.1 Model-in-the-Loop Simulation Mode

Model-in-the-loop simulation mode denotes simulations in which the distributed control application is substituted by a *model*. Model-in-the-loop simulation mode is depicted in Figure 2.3(a). The simulation incorporates a model counterpart for each part of the real system: the controllers that constitute the real distributed control application are substituted by *controller models*; the real environment with real devices is substituted by a *simulated environment* with *simulated devices*. In model-in-the-loop simulation mode, controller models are deployed on simulated devices in a simulated environment in which simulated sources of dynamism reside.

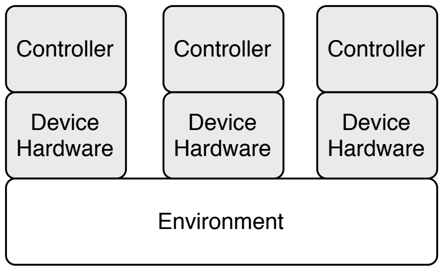
Model-in-the-loop simulation is typically used during the early stages of application development. The controller models represent the behavior (or a part thereof) of the distributed control application that is to be built later on [Neu04]. Controller models are often used for rapid prototyping, and play a crucial role for checking important properties of control applications at an early stage of development. For example, controller models are used for checking safety guarantees [LTS99]. This is particularly important in case of safety critical systems such as air traffic control [TPS98, GL04, LGLM05] and automated high-



(a) Model-in-the-loop simulation mode.



(b) Software-in-the-loop simulation mode.



(c) Hardware-in-the-loop simulation mode.

Figure 2.3: Simulation modes for distributed control applications. White blocks are simulated parts. Grey blocks are parts of the real system that are integrated in the simulation loop.

way systems [LGS98, HCd05]. The safety specifications for the distributed control application are translated into restrictions on the model's reachable set of states. Model-in-the-loop simulation is used to design controller models whose trajectory is guaranteed to remain within the safe subset of the set of reachable states [TLSS00]. Besides checking properties such as safety, the controller models of a distributed control application can also serve as a system specification that provides a guidance during detailed design and implementation, and enables deriving test cases [Rut06].

2.3.2 Software-in-the-Loop Simulation Mode

Software-in-the-loop simulation mode denotes simulations in which the software of the real controllers of the distributed control application is embedded in the simulation loop. Software-in-the-loop simulation mode is depicted in Figure 2.3(b). The simulation contains real parts of the system, i.e. the controller software, together with simulated parts, i.e. the device hardware and the environment. The executable code of the real controllers is directly embedded in the simulation. In software-in-the-loop simulation mode, the software of the real controllers is deployed on simulated devices that reside within a simulated environment with simulated sources of dynamism.

Software-in-the-loop simulation is typically used during the late stages of application development, i.e. after the software of the distributed control application (or parts thereof) has been implemented. Software-in-the-loop simulation enables experimenting with the controllers of a distributed control application on simulated devices before deployment on real devices. Software-in-the-loop simulation is extensively used for the development of control applications for robots. For example, software-in-the-loop simulations enable testing the robustness and fault-tolerance of control applications for robots [BKW06, FBT$^+$03] or can facilitate parameter estimation of a control application for robots [SA06].

2.3.3 Hardware-in-the-Loop Simulation Mode

Hardware-in-the-loop simulation mode comprises simulations that embed not only the software of the real controllers, but also real device hardware on which the controllers are deployed. Hardware-in-the-loop simulation mode is depicted in Figure 2.3(c). The software of the real controllers of the distributed control application is deployed on real devices. The real devices are connected to a simulated environment where additionally simulated devices or simulated sources of dynamism can be present. The real controllers use real sensors and actuators on a real devices that are connected to a simulated environment.

Hardware-in-the-loop simulations are typically used during the late stages of application development, i.e. after the controllers have been implemented and have been deployed on the devices. Hardware-in-the-loop simulation enables experimenting with real embedded devices in a simulated environment before these devices are installed in the real environment [Gom01]. For example, hardware-in-the-loop simulation is used for testing embedded control applications for traffic lights without disturbing real traffic [BKB$^+$05], or testing controllers embedded in intelligent vehicles [GPDV06], without exposing the vehicle to real traffic.

2.3.4 Simulation-Based Design

We describe a number of simulation-based design approaches that rely on simulation modes discussed above. Typically, a control application's implementation is not derived in any direct way from the controller models that were constructed during early design. However, simulation-based design approaches aim at bridging the gap between modeling and implementation artifacts. Examples include:

- *MIDAS* [BS91] is an approach that supports the design of distributed systems via iterative refinement of a partially implemented design where some components exist as simulation models and others as operational subsystems.

- *Model Continuity Methodology* [HZ05b] is a methodology that supports designing and testing of DEVSJAVA-based controller models by simulation, and deploying the same controller models on the real target system for execution.

- *Giotto* [HKSP03] is a tool-supported design methodology for embedded control applications that supports a phased evolution to derive executable code from a high-level controller model.

2.4 Support for Simulating Distributed Control Applications

There are various ways to support simulations of distributed control applications. We studied several simulation platforms for simulating distributed control applications. Based on our study, we make a distinction between three groups of simulation platforms for simulating distributed control applications: case-specific simulation platforms, domain-specific simulation platforms and general-purpose simulation platforms. Case-specific simulation platforms provide support for one specific simulation case. Domain-specific simulation platforms support a family of simulations. General-purpose simulation platforms provide support for a broad range of simulations.

We elaborate on each group by delineating the scope and providing a number of examples.

2.4.1 Case-Specific Simulation Platforms

Case-specific simulation platforms are simulation platforms that offer elaborate support for one specific case of simulating distributed control applications, i.e. support for one specific simulation model. The supported simulation model not only prescribes the kind of system (i.e. the kind of control application, devices and environment), but also at what level of abstraction the system is represented. Case-specific simulation platforms have a limited scope of applicability. Case-specific simulation platforms can only be used in case (1) the system-to-be-simulated matches the kind of system described in the model, and (2) the abstraction level of the model is suitable, i.e. not too high nor to detailed for the purpose of the simulation.

Examples of case-specific simulation platforms for simulating distributed control applications include:

- *Webots* [Mic04] is a robot simulation platform that offers support for mobile robots, including Khepera robots, Fujitsu HOAP-2 humanoid robots and Sony Aibo ERS-210 robots. The supported model is very fine-grained, and includes representations of individual servo engines, sensors and physical volumes, elementary forces, friction between the wheels and the surface.

- *Übersim* [BT03] is a multi-robot simulation engine for simulating games of robot soccer. Übersim captures at a reasonable level of resolution the dynamics and physical interactions between the robots, field and ball. Übersim provides a set of predefined robot models.

- *NS-2* [USC] is a discrete event simulator targeted at simulating computer networks. The NS-2 model represents routing, network protocols over wired and wireless networks and network dynamics such as traffic pattern changes, node movement and node failure.

- *Green Light District* [WVvVK04] is a java simulation platform for testing controllers of traffic lights. The model supports in various maps for representing road networks and various traffic densities.

2.4.2 Domain-Specific Simulation Platforms

Domain-specific simulation platforms are simulation platforms that support a family of simulations, e.g. simulations of distributed control applications. The applicability of domain-specific simulation platforms is broader than case-specific simulation platforms, but narrower than general-purpose simulation platforms.

We give examples of domain-specific simulation platforms for simulating distributed control applications:

- XRaptor [BMP$^+$] is a simulation platform to study the behavior of a large number of agents in two- or three-dimensional continuous virtual worlds. For XRaptor, an agent is either a point, a circular area or a spherical volume that contains a sensor unit for observing the world, an actuator unit for performing actions and a control kernel for action selection. Ordinary differential equations are used for modeling movements. Simulations based on XRaptor include the movement of Braitenberg vehicles, food foraging behavior of ants and predator-prey scenarios.

- *SPARK* [OR04] is a simulation platform for physical multi-agent systems in three dimensional environments. SPARK supports a flexible agent representation with different sensors, actuators and morphologies.

- *SPADES/MPADES* [RR03, Ril03] is a simulation platform not tied to a particular simulation. SPADES/MPADES tracks on a controller's execution time between sense and act events.

2.4.3 General-Purpose Simulation Platforms

General-purpose simulation platforms support all simulation models insofar these models are expressed in terms of particular general-purpose modeling constructs. General-purpose simulation platforms are not limited to simulating distributed control applications, but support a much broader range of simulations.

We give examples of general-purpose simulation platforms:

- *JAMES* [HRU03] (recently updated to JAMES II) is a simulation platform that supports discrete event models described in terms of the constructs of DEVS (Discrete EVent System specification) [ZP00] as well as several extensions of this formalism (e.g. DynDEVS [Uhr01] for dynamic structures). It supports the construction of composed simulation models based on these constructs and concurrent, distributed execution of such models.

- *JiST* [BHvR05] is a Java-based simulation system that supports discrete event models by embedding event semantics directly into the Java execution model.

2.4.4 Discussion

In case no case-specific simulation platform suits the needs of the simulation study, a new simulation must be developed. The development of simulations of distributed control applications can be supported by means of general-purpose simulation platforms or domain-specific simulation platforms for distributed control applications. We elaborate on the support offered by general-purpose simulation platforms and domain-specific simulation platforms.

2.4.4.1 Support Offered by General-Purpose Simulation Platforms

General-purpose simulation platforms support the developer by means of general-purpose modeling constructs. General-purpose modeling constructs offer no support that is specifically targeted at software-in-the-loop simulation of distributed control applications in dynamic environments. General-purpose modeling constructs support a broad range of simulations, and their use is not limited to simulating distributed control applications. Due to their broad scope, general-purpose simulation platforms do not offer special-purpose support to tackle the challenges associated with modeling distributed control applications in particular. For example, *JAMES* [HRU03] supports discrete event models described in terms of the constructs of DEVS (Discrete EVent System specification) [ZP00]. DEVS is a modeling framework that supports atomic and coupled discrete event models. Atomic models are described in terms of modeling constructs such as a state set, input and output ports, internal and external transition functions, etc. These constructs are not specifically targeted at simulating distributed control applications in dynamic environments.

General-purpose modeling constructs are formally founded. For example, the meaning and execution semantics of DEVS modeling constructs is formally specified. Due to the formal description of their modeling constructs, general-purpose simulation platforms offer well-defined building blocks for describing a simulation model. As such, a modeler can use the modeling constructs for formulating a simulation model, without taking into account the simulation platform that will be used to execute the model. The formal description of the modeling constructs decouples the simulation model from the simulation platform that is used to execute the model.

2.4.4.2 Support Offered by Domain-Specific Simulation Platforms

Domain-specific simulation platforms for distributed control applications incorporate support specifically aimed at simulating distributed control applications. For example, *XRaptor* [BMP+] supports simulations consisting of a world in which “agents” can act. An “agent” is described as a point, a circular area or a spherical volume that contains a “sensor unit” for observing the world, an “actuator unit” for performing actions and a “control kernel” for action selection. Compared to general-purpose simulation platforms, domain-specific simulation platforms support the developer to tackle specific challenges associated with simulating distributed control applications in particular.

However, the modeling constructs of current domain-specific simulation platforms are not formally founded. Current domain-specific simulation platforms only provide informal abstractions of which the precise meaning is implicit and coupled to the design and implementation of a particular simulation platform. For example, the *XRaptor* abstractions “agent”, “actuator unit”, “sensor unit”, “control kernel”, etc. are not formally specified. As a result, the precise meaning and relation of the building blocks for describing a simulation model is rather vague. In the case of *XRaptor*, it is unclear how the control kernel triggers movements in the environment, or to what extent the timing characteristics of the control kernel are supported. Due to the lack of a formal specification, building a simulation model supported by a domain-specific simulation platform requires detailed knowledge of its design and implementation, and the resulting simulation model is tightly coupled with the simulation platform that is used to execute it.

2.5 Scope

In this dissertation, we focus on *domain-specific support* for software-in-the-loop simulations of distributed control applications in dynamic environments.

To sum up, existing approaches to support this family of simulations either rely on (1) general-purpose modeling constructs that are formally specified, but offer no support specifically targeted at this family of simulations, or on (2) informal abstractions that offer support specifically targeted at this family of simulations, but of which the meaning is implicit and coupled to the design and implementation of a particular simulation platform.

Therefore, we put forward a formally founded modeling framework for software-in-the-loop simulation of distributed control applications in dynamic environments. The modeling framework comprises modeling constructs that are specifically aimed at capturing core characteristics of this family of simulations in a first-class manner. Moreover, the modeling constructs are formally specified to unambiguously specify their meaning and relations. This is crucial to decouple the simulation model from the simulation platform to execute the model. As such, the modeling framework enables formulating a simulation model without taking into account the design and implementation of the simulation platform to execute the model.

The modeling framework is the result of our experience with simulating distributed control applications in dynamic environments and is underpinned by current state-of-the-art research on modeling such applications. The modeling framework comprises two parts: an environment part and a control application part.

- The *environment part* offers special-purpose modeling constructs for dynamic environments. These modeling constructs capture (1) the structure of the environment, (2) dynamism in the environment, (3) the way dynamism is affected by the sources of dynamism in the environment and (4) the way dynamism can interact in the environment. The modeling constructs of the environment part are described in detail in Chapter 3.

- The *control application part* offers special-purpose modeling constructs for integrating the software of a real distributed control application in the simulation model. These modeling constructs capture (1) the execution time of the control software and (2) the interface of the control software for interacting with the environment. The modeling constructs of the control application part are described in detail in Chapter 4.

To validate the modeling constructs, we developed a simulation platform that supports the constructs in an executable simulation, and we used the constructs to underpin a simulator for an industrial case, i.e. a distributed control application controlling unmanned vehicles in a warehouse environment. The simulator comprises a simulation model that is decoupled from the simulation platform to execute it. This enables customizing the simulation model, which is paramount to support the study and evaluation of different functionalities of the distributed control application. The simulation platform is described in Chapter 5, whereas the case is described in Chapter 6.

Chapter 3

Modeling Dynamic Environments

In this chapter, we focus on the environment part of the modeling framework. We introduce modeling constructs that support the modeling of dynamic environments of distributed control applications. The focus is on the specification of the modeling constructs. Design and implementation issues to support the constructs are tackled in Chapter 5.

3.1 Introduction

We put forward modeling constructs for modeling dynamic environments of distributed control applications. The proposed modeling constructs capture the characteristics of a dynamic environment in an explicit manner.

The modeling constructs are described in an explicit modeling framework. The modeling framework specifies the meaning, relations and execution semantics of all modeling constructs in a formal way. The formal description of the modeling framework decouples the modeling constructs from a particular implementation in a simulation platform. This enables a developer to formulate a simulation model without taking into account the simulation platform that will be used to execute the model. As such, the formal description of the modeling framework is crucial to obtain a clean distinction between the model formulation phase, i.e. describing a simulation model, and the model translation phase, i.e. translating the simulation model into an executable simulation on a particular simulation platform.

The foundation for the constructs of the modeling framework is twofold. On the one hand, the modeling framework results from our own experience of building simulations for distributed control applications in dynamic environments. Examples include simulations of the Packet-World [WHH05], Lego Mindstorms robots [Bor06] and Automated Guided Vehicles [HHB06]. On the other hand, the modeling constructs are underpinned by existing practice on modeling dynamic environments of distributed control applications. We will relate the modeling constructs to current state-of-the-art research on simulating dynamic environments of distributed control applications.

To introduce the modeling constructs of the modeling framework in an intuitive manner, we use the modeling constructs for modeling a RoboCup Soccer environment. RoboCup Soccer [NRSSV05] is a game of soccer that is played by a team of autonomous robots that competes against an opponent team of human or robotic players. RoboCup Soccer offers a complex, yet easy to understand problem domain that is frequently used to illustrate research on distributed control applications, simulation and collective robotics.

We employ set theory to formally specify the modeling constructs and their execution semantics. The notation of set theory is compact and easy to understand. Moreover, set theory is frequently used for describing modeling constructs of general-purpose modeling frameworks. For example, the discrete event modeling framework DEVS (Discrete EVent System specification) [ZP00] and its derivatives such as Parallel DEVS rely on set theory to formally specify their modeling constructs and execution semantics.

This chapter is structured as follows. We start with a brief overview of the modeling framework in Section 3.2. In the next sections, we elaborate on each of the modeling constructs in detail: each construct is explained informally, illustrated in the context of RoboCup Soccer and complemented with a formal description. In Section 3.3 we introduce constructs to model the constituting parts of the environment, which serve as starting point to describe dynamism in the next sections. Section 3.4 describes the constructs to reify *dynamism* as first-class modeling construct in the simulated environment. Section 3.5 focusses on constructs to describe the various sources of dynamism and the way they affect dynamism in the environment. Section 3.6 introduces modeling constructs to describe how dynamism in the environment interacts. Section 3.7 gives a formal description of the way a simulation model based on the constructs of the modeling framework can be executed. Section 3.8 discusses the added value of the modeling framework. Section 3.9 relates the modeling constructs to state-of-the-art research on simulating dynamic environment of distributed control applications. Finally, we draw conclusions in Section 3.10.

3.2 Overview of the Modeling Framework

We start with an overview of all constructs of the modeling framework for dynamic environments, before elaborating on each construct in detail in the next sections.

Figure 3.1 gives a graphical overview of the modeling framework for dynamic environments, depicting all modeling constructs and the relations between the

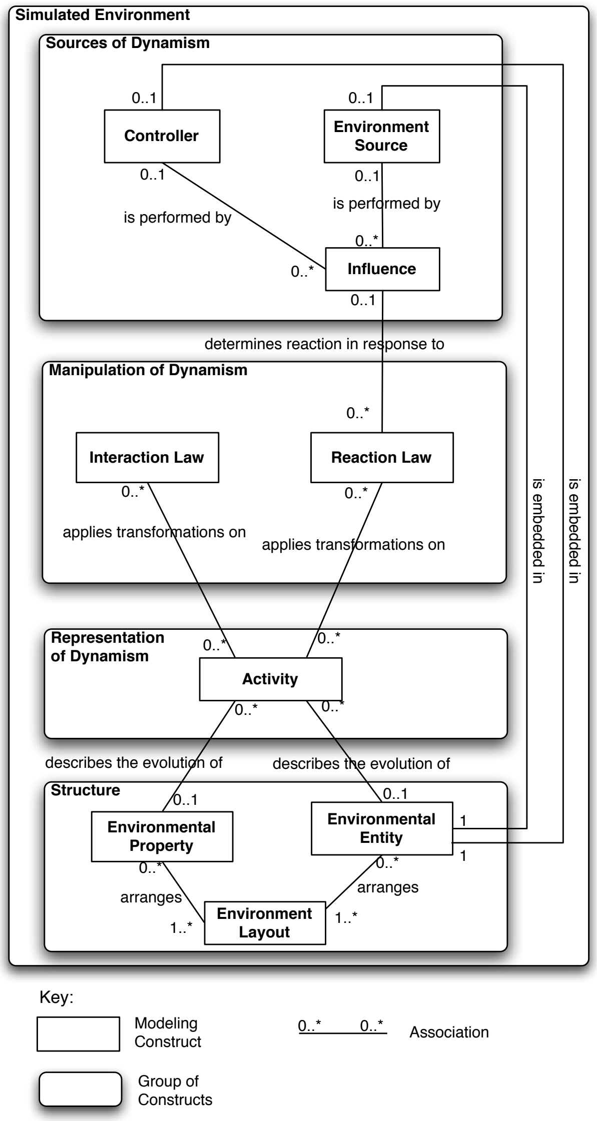


Figure 3.1: Overview of the constructs in the modeling framework and their associations

constructs. The modeling constructs are organized in four groups:

1. Constructs to represent the *structure of the environment* in the simulation model.
2. Constructs to represent *dynamism in the environment* in the simulation model.
3. Constructs to represent the *manipulation of dynamism in the environment* in the simulation model.
4. Constructs to represent the *sources of dynamism in the environment* in the simulation model.

We give an overview of the modeling constructs in each group.

Structure of the environment. A first group of modeling constructs captures the structure of the environment. To capture the constituting parts of the environment in a simulation model, we put forward the modeling constructs `Environmental Entity` and `Environmental Property`. Examples of environmental entities are all sorts of objects in the environment, such as the robots on which a distributed control application is deployed. An example of an environmental property is the temperature in the environment. To represent a physical or logical structure that arranges the different environmental entities and environmental properties with respect to each other, we put forward the modeling construct `Environment Layout`. An example of an environment layout is a two-dimensional geometrical arrangement of the entities.

Dynamism in the Environment. A second group of modeling constructs captures dynamism in the environment in an explicit manner. To represent dynamism explicitly in the simulation model, we put forward an `Activity` as a modeling construct. The association between `Activity` and `Environmental Entity` and the association between `Activity` and `Environmental Property` expresses that an activity describes the evolution of a particular environmental entity or property over time. Examples of activities are the movement of a robot or the rolling of a ball.

Manipulation of Dynamism. A third group of modeling constructs captures the way dynamism in the environment can alter, i.e. the way activities arise, interact and terminate. We put forward the modeling constructs `Reaction Law` and `Interaction Law` to capture the way activities in the environment are manipulated.

A `Reaction Law` is a modeling construct that specifies what happens in the environment in reaction to a particular trigger of a source of dynamism. An example is a reaction law that specifies what happens in the environment in reaction to the trigger of a controller to start the engines of a robot. The reaction law specifies what kind of activity is created, e.g. a movement of that robot characterized by a particular velocity in a particular direction.

An `Interaction Law` is a modeling construct to specify the way dynamism can interact in the environment. For example, an interaction law can specify what happens in case a robot involved in a movement activity hits a wall or another robot.

The associations between `Reaction Law` and `Activity` on the one hand, and between `Interaction Law` and `Activity` on the other hand, express that reaction laws and interaction laws alter the activities present in the environment.

Sources of dynamism. A fourth group of modeling constructs captures the sources of dynamism in the environment. We put forward the modeling constructs `Controller` and `Environment Source` to represent the behavior of the various sources of dynamism present in the environment.

A `Controller` is a source of dynamism that is part of the distributed control application. An example of a controller is the software program that controls a particular robot. An `Environment Source` is a source of dynamism that resides in the environment and that is external to the distributed control application. An example of an environment source is the behavior of a machine in the environment that is controlled by a human. Controllers and environment sources are embedded in some of the environmental entities. For example, a robot contains a source of dynamism, i.e. its controller, whereas a ball is passive and does not contain a source of dynamism.

Controllers and environment sources can initiate, terminate or alter dynamism in the environment. We put forward an `Influence` as a modeling construct to capture the *attempt* of a controller or environment source to affect the environment. An example of an influence is the attempt of a controller to start or stop the movement of a robot. The association between `Environment Source` and `Influence` and between `Controller` and `Influence` represents that dynamism can only be manipulated indirectly, i.e. by means of performing influences in the environment. Reaction laws determine the actual reaction of the environment in response to influences. This is represented by the association between `Reaction Law` and `Influence`.

3.3 Structure of the Simulated Environment

In this section, we introduce modeling constructs to represent the constituting parts of the environment. We deliberately kept this part of the modeling framework simple, as this suffices to discuss dynamism in the next sections.

The modeling constructs, together with their formal description, are introduced in Section 3.3.1. Subsequently, in Section 3.3.2 we define the state of the simulated

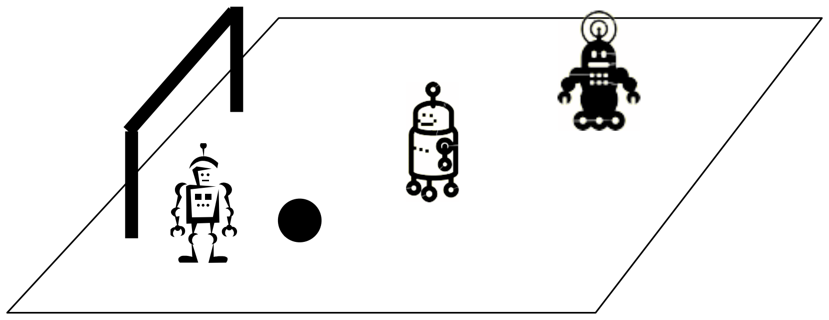


Figure 3.2: A representation of a RoboCup Soccer environment

environment.

3.3.1 Environmental Entities, Properties and Layout

We represent the parts that constitute the simulated environment by means of two constructs: environmental entities and environmental properties. We do not address methodological issues on how to apply these constructs in practice, as this is highly dependent upon the objective of the simulation study [Sha98].

- *Environmental entities.* Environmental entities represent entities characterized by their own, distinct existence in the real environment. The real environment typically contains numerous entities of different kinds that can be incorporated as environmental entities in the simulated environment. We define:

  $E = \{e_1, e_2, \ldots, e_n\}$ | the set of environmental entities.

  Environmental entities can be partitioned into a set of disjoint subsets, with each subset grouping entities of the same kind along the relation $\simeq$. Formally:

  $Part_{E,\simeq} = \{E_1, E_2, \ldots, E_k\}$ | a partition of environmental entities with:
  $E_i \subseteq E$
  $E = \bigcup_{i=1\ldots k} E_i$
  $E_i \cap E_j = \phi, \forall i \neq j$
  $\forall e_j, e_k \in E_i : e_j \simeq e_k$

  For example, consider Figure 3.2, depicting a part of a RoboCup Soccer environment [NRSSV05] where a robots play soccer. The environmental entities we distinguish are three robots, a ball, a field and a goal:

$E = \{robot_1, robot_2, robot_3, ball, field, goal\}$	the set of entities.
$Part_{E,\simeq} = \{Robot, Ball, Field, Goal\}$	a partition into four kinds of entities.
$Robot = \{robot_1, robot_2, robot_3\}$	the set of robots.
$Ball = \{ball\}$	the set of balls.
$Field = \{field\}$	the set of fields.
$Goal = \{goal\}$	the set of goals.

- *Environmental properties.* An environmental property is a distributed quantity that represents a measurable, system-wide characteristic of the real environment. Environmental properties can be directly represented in the simulated environment if needed. We define:

$P = \{p_1, p_2, \ldots, p_m\}$	the set of environmental properties.
$Part_{P,\simeq} = \{P_1, P_2, \ldots, P_l\}$	a partition of properties in along the relation "is the same kind", denoted by $\simeq$.

  Examples of environmental properties are gravitation and magnetic fields in the environment. In the RoboCup Soccer environment, the environmental properties we distinguish are temperature and humidity:

$P = \{temp, hum\}$	the set of environmental properties.
$Part_P = \{Heat, Humidity\}$	a partition in two kinds of properties.
$Heat = \{temp\}$	the set of temperature properties.
$Humidity = \{hum\}$	the set of humidity properties.

The set of all constituents is defined as the union of the set of environmental entities and properties: $C = E \cup P$. Constituents can be partitioned according to their kind, respecting the partitions of entities and properties: $Part_{C,\simeq} = Part_{E,\simeq} \cup Part_{P,\simeq}$. After relabeling: $Part_{C,\simeq} = \{C_1, C_2, \ldots, C_{k+l}\}$.

Environmental entities and/or properties are typically arranged according to a particular *environment layout.* An environment layout is a physical or logical structure that arranges the constituents with respect to each other. The canonical example of an environment layout is a topology. A topology expresses the spatial positioning of various entities with respect to each other, for example grid-based [BDD03], graph-based [KB04] or continuous topologies [HSKM97]. In Figure 3.2, the environment layout is a two dimensional topology in which the field, the robots, the ball and the goal are arranged with respect to each other.

3.3.2 The State of the Simulated Environment

The state of the simulated environment is defined by the state of all its constituents, i.e. the state of all environmental entities and properties. We describe the state of a constituent of the simulated environment as a list of values that are sufficient to define the status of the constituent. The state typically comprises time-dependent characteristics of a constituent. We introduce the following definitions to describe the state of constituents:

S_{C_i}	the set of all possible states of constituents of kind C_i.
$s_c = \langle v_1, v_2, \ldots, v_r \rangle \in S_{C_i}$	the state of a particular constituent $c \in C_i$, represented as a tuple of values $v_j \in V_j$, with V_j a value domain.
$S = \bigcup_{C_i \in Part_C} S_{C_i}$	the set of all possible states of constituents of any kind.

To specify the initial state of each constituent, we define a function $Init$ which maps a constituent on its initial state:

$$Init : C \to S$$
$$Init(c) = s_c$$

For a given constituent c, the function $Init$ specifies the initial state $s_c \in S_{C_i}$.

For the RoboCup Soccer environment, the state of a robot and a ball is a coordinate in a two dimensional spatial layout:

$S_{Robot} = \mathbb{R}^2$	the set of all possible states for a robot.
$s_{robot_1} = \langle \overrightarrow{pos} \rangle \in S_{Robot}$	the state of a robot $robot_1$, with $\overrightarrow{pos} \in \mathbb{R}^2$ a coordinate in the two dimensional layout that indicates the position of $robot_1$.
$S_{Ball} = \mathbb{R}^2$	the set of all possible states for a ball.
$s_{ball} = \langle \overrightarrow{pos} \rangle \in S_{Ball}$	the state of a ball $ball$, with $\overrightarrow{pos} \in \mathbb{R}^2$ a coordinate in the two dimensional space that indicates the position of $ball$.

We use the shorthand notations $s_r|_{pos}$ to select the position of a robot $r \in Robot$ and $s_b|_{pos}$ to select the position of a ball $b \in Ball$.

3.4 Dynamism in the Simulated Environment

Starting from the basic model of the structure of the environment, we now elaborate on dynamism. To support the modeler in describing dynamism, we provide a first-class representation of dynamism in the simulated environment.

We describe the modeling constructs for capturing dynamism in the simulated environment in Section 3.4.1, then explain how they can be used to describe scenarios in Section 3.4.2 and how they specify the state Section 3.4.3. Here, we make abstraction from the way dynamism is initiated or how it interacts, as these will be the topics of the next two sections.

3.4.1 Activities

Dynamism in the environment comprises the evolution of environmental entities and properties over time. We introduce *activities* as a construct for representing dynamism in the simulated environment in an explicit manner. An activity describes a well-specified evolution of a particular constituent of the simulated environment, that is active over a specific time interval. An example of an activity in the RoboCup Soccer environment is a robot that is driving during a particular time interval. An activity comprises the following: the constituent involved, the time interval of occurrence and the evolution strategy.

- *The constituent involved.* Dynamism has an impact on particular parts of the simulated environment. Each activity is associated with the environmental entity or property it describes the evolution of. For example, in Figure 3.3, the activity a_3 represents the rolling of the ball. The activities a_1 and a_2 represent the driving of one of the robots.

- *The time interval of occurrence.* Dynamism occurs over time. Consequently, each activity is characterized by a specific time interval. The time interval of an activity specifies the point in time the activity starts and the time its evolution completes[1]. In case the activity never ends, the time interval is infinitely long. For example, in Figure 3.3, activity a_1 representing the driving of a RoboCup robot $robot_1$, starts at time $t = 3$ after the start of the game, until $t = 5$. Activity a_2 starts at time $t = 6$ after the start of the game, until $t = 10$. Activity a_3 represents the rolling of the ball, starting at time $t = 7$, until the moment it stops rolling, at time $t = 9$.

- *The evolution strategy.* Dynamism evolves in a particular way. Consequently, activities are characterized by an evolution strategy that describes the specific way the status of the involved constituent changes over the time interval

[1]For now, we make abstraction of the fact that the time interval may not be know at the start of the activity, as this is discussed in the next section

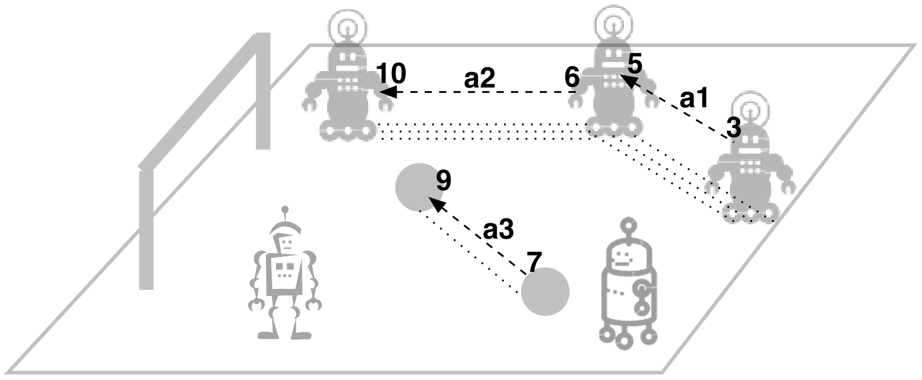


Figure 3.3: Activities a_1, a_2 and a_3 in a RoboCup Soccer environment.

of occurrence. For example, for activity a_2 in Figure 3.3, this could correspond to a change of the position of the robot according to a constant velocity vector.

Before giving a formal description of activities, we first introduce a number of general definitions:

$t \in T$	a particular time instant, with T the time domain.
$\Delta t \in \Delta T : t + \Delta t = t'$	a particular duration, with ΔT the set of all possible durations, including ∞.
$\Delta s_c \in \Delta S_{C_k}$	a state change for a constituent c of kind C_k, with ΔS_{C_k} the set of all possible state changes for constituents of kind C_k.
$\Delta S = \bigcup_{C_i \in Part_C} \Delta S_{C_i}$	the set of all possible state changes of constituents of any kind.
$\oplus : S \times \Delta S \rightarrow S$ $s \oplus \Delta s = s'$	the state-composition operator $\oplus$ defines a new state from a given state and a state change. Note that the operator is overloaded for each kind of constituent.

An activity a is defined as as tuple containing the constituent of the activity, its starting time, its duration until completion and an evolution strategy. For each activity, the evolution strategy is defined by a tuple of custom parameters and a function that is “instantiated” with these custom parameters[2]:

$a = \langle c, t, \Delta t, par, F \rangle$	an activity with the following characteristics: $c \in C$: the constituent involved. $t \in T$: the starting time. $\Delta t \in \Delta T$: the duration until completion. $par = \langle v_1, \ldots, v_r \rangle \in V_1 \times \ldots \times V_r$: the parameters of the evolution strategy. $F : V_1 \times \ldots \times V_r \times \Delta T \rightarrow \Delta S$: the state change function, returning a state change $\Delta s \in \Delta S$ relative to the start of the activity, given a tuple of parameters and any duration not greater than the duration Δt of the activity.

We use the following shorthand notations: $a|_c$ denotes the constituent, $a|_t$ and $a|_{\Delta t}$ denote the begin time and duration of activity a, respectively. Furthermore, $a|_{par}$ denotes the parameters and $a|_F$ denotes the state change function of activity a.

For the RoboCup Soccer scenario in Figure 3.3, consider the following activity as an example:

$$a_1 = \langle robot_1, 3, 2, \langle \vec{v_1} \rangle, Driving \rangle$$

Activity a_1 represents that constituent $c = robot_1$ starts driving at time $t = 3$ for a duration of $\Delta t = 2$. The state change is defined by the velocity vector $\vec{v_1}$ and the function *Driving*.

The function *Driving* is defined as:

$$Driving : \mathbb{R}^2 \times \Delta T \rightarrow \Delta S_{Robot}$$
$$Driving(\vec{v}, \Delta t) = \langle \vec{v} * \Delta t \rangle$$

Driving returns a state change $\Delta s \in \Delta S_{Robot}$ for robots that drive with a given velocity vector $\vec{v} \in \mathbb{R}^2$ during a given duration $\Delta t \in \Delta T$. As the state of a robot is a tuple $\langle \overrightarrow{pos} \rangle$ (see Section 3.3.2), the state change returned by *Driving* is the change of the position $\overrightarrow{pos}$. The change of the position $\overrightarrow{pos}$ is expressed as the function $\vec{v} * \Delta t$ with Δt the duration since the robot started driving with velocity $\vec{v}$.

As another example, consider the following activity from the RoboCup Soccer scenario in Figure 3.3:

$$a_3 = \langle ball, 7, 2, \langle \vec{v_3}, \vec{d_3} \rangle, Rolling \rangle$$

[2]This notation might seem awkward at first sight, but is more expressive when being used in examples later on.

Activity a_3 represents that constituent $c = \mathit{ball}$ starts rolling at time $t = 7$ for a duration of $\Delta t = 2$. The state change is defined by velocity vector $\overrightarrow{v_3}$ and deceleration vector $\overrightarrow{d_3}$, and the function *Rolling*.

The function *Rolling* is defined as:

$$\begin{aligned}&\mathit{Rolling} : \mathbb{R}^4 \times \Delta T \rightarrow \Delta S_{\mathit{Ball}}\\&\mathit{Rolling}(\vec{v}, \vec{d}, \Delta t) = \langle \vec{v} * \Delta t - \tfrac{\vec{d}}{2} * \Delta t^2 \rangle\end{aligned}$$

The function *Rolling* returns a state change for a ball that rolls with a initial velocity vector $\vec{v} \in \mathbb{R}^2$ and deceleration $\vec{d} \in \mathbb{R}^2$ during a given duration $\Delta t \in \Delta T$. The state of a ball is a tuple $\langle \overrightarrow{pos} \rangle$ (see Section 3.3.2). The change of the position returned by *Rolling* is expressed as the function $\vec{v} * \Delta t - \frac{\vec{d}}{2} * \Delta t^2$ with Δt the duration since the ball started rolling with velocity $\vec{v}$ and deceleration $\vec{d}$.

3.4.2 Scenarios

We now focus on how scenarios can be described. Scenarios describe a particular evolution of the environment and can be expressed in terms of activities for the various constituents. We define:

$a \in A^\Omega$	A^Ω is the set of all possible activities.
2^{A^Ω}	the powerset of A^Ω, i.e. the set of all possible subsets of A^Ω
$A = \{a_1, a_2, \ldots, a_r\}$	a scenario described by a set of activities, with $A \in 2^{A^\Omega}$

The scenario for the RoboCup Soccer environment depicted in Figure 3.3, can be expressed as:

$$\begin{aligned}&A = \{a_1, a_2, a_3\}, \text{ with:}\\&\quad a_1 = \langle \mathit{robot}_1, 3, 2, \langle \overrightarrow{v_1} \rangle, \mathit{Driving} \rangle\\&\quad a_2 = \langle \mathit{robot}_1, 6, 4, \langle \overrightarrow{v_2} \rangle, \mathit{Driving} \rangle\\&\quad a_3 = \langle \mathit{ball}, 7, 2, \langle \overrightarrow{v_3}, \overrightarrow{d_3} \rangle, \mathit{Rolling} \rangle\end{aligned}$$

3.4.3 Scenarios and State

So far, we introduced activities and illustrated how activities can be used to describe scenarios. We now elaborate on how a scenario specifies the state of each constituent at any point in time.

We first define a function *Active* that returns for a given scenario, a given constituent and a given time instant, the subset of activities that is active for the given constituent at the given time instant:

$$\begin{aligned}
&Active : 2^{A^\Omega} \times C \times T \rightarrow 2^{A^\Omega}\\
&Active(A, c, t) = \{a \in A \mid (a|_c = c) \wedge (a|_t < t \leq (a|_t + a|_{\Delta t})\}
\end{aligned}$$

For a given scenario A, a given constituent c and a given time instant t, the *Active* function returns the subset of activities for which the given constituent c is the constituent involved in the activity, and for which that the given time instant t is situated between the begin and end of the activity.

For a given initial state and a given scenario, we define a function *State* that specifies in a recursive manner the state of any constituent at any particular point in time.

$$\begin{aligned}
&State : 2^{A^\Omega} \times (C \rightarrow S) \times C \times T \rightarrow S\\
&State(A, Init, c, t) =\\
&\begin{cases}
State(A, Init, c, a|_t) \oplus a|_F(a|_{par}, t - a|_t) \\
\qquad \text{if } a \in Active(A, c, t); \\
\\
State(A, Init, c, t_x) \text{ with:} \\
\quad t_x = max\{t_k \in T | (t_k < t) \wedge (Active(A, c, t_k) \neq \phi)\} \\
\qquad \text{if (} Active(A, c, t) = \phi \text{) } \wedge \\
\qquad\quad \text{(} \exists t_k \in T : t_k < t \wedge (Active(A, c, t_k) \neq \phi) \text{);} \\
\\
Init(c, t) \\
\qquad \text{otherwise;}
\end{cases}
\end{aligned}$$

For a particular scenario A and initial state *Init*, the state of a constituent c at time t can be derived in the following way. Note that we assume for now that at a specific time instant, a constituent may be involved in at most one activity. We explain the three cases in the domain of the *State* function:

1. In the first case, an activity a is active for the given constituent c and time t. In this case, the state is recursively defined as the state $State(A, Init, c, a|_t)$ at the start $a|_t$ of the activity, composed with the state change specified by the activity. This state change is obtained by applying the function $a|_F$ with the following arguments: on the one hand the parameters $a|_{par}$ specified by the activity, and on the other hand the duration $t - a|_t$, i.e. the time elapsed from the time $a|_t$ the activity started, until time t.

2. In the second case, there is no activity that is active for constituent c and time t; however, there exists an earlier time instant t_k at which there is an activity that is active for c. In this case, the state of the constituent c at time t is the same as the state of the constituent c at time t_x, where t_x is the latest time instant before t at which an activity was active for constituent c.

3. Otherwise, i.e. when there is no activity active for constituent c and time t, and there are no earlier activities that describe the evolution of c, then the state of c at time t is the initial state of c.

We illustrate the *State* function by means of the scenario in the RoboCup Soccer environment depicted in Figure 3.3. We expand the recursion for:

$$\begin{aligned}
&State(A, Init, robot_1, 9)\\
&\quad = State(A, Init, robot_1, 6) \oplus Driving(\overrightarrow{v_2}, 3)\\
&\quad = (\ State(A, Init, robot_1, 5)\ ) \oplus Driving(\overrightarrow{v_2}, 3)\\
&\quad = (\ State(A, Init, robot_1, 3) \oplus Driving(\overrightarrow{v_1}, 2)\ ) \oplus Driving(\overrightarrow{v_2}, 3)\\
&\quad = (\ (\ Init(robot_1)\ ) \oplus Driving(\overrightarrow{v_1}, 2)\ ) \oplus Driving(\overrightarrow{v_2}, 3)
\end{aligned}$$

We explain the four expansions:

1. In the first expansion, $State(A, Init, robot_1, 9)$ is expressed in terms of $State(A, Init, robot_1, 6)$. At time $t = 9$, activity a_2 is active, indicating that $robot_1$ is driving. Consequently, the expansion is obtained by applying the first case of the *State* function.

2. In the second expansion, $State(A, Init, robot_1, 6)$ is expressed in terms of $State(A, Init, robot_1, 5)$. At time $t = 6$, activity a_2 is not active yet, i.e. $robot_1$ is not driving. However, there exists an earlier time at which another activity, i.e. activity a_1, is active. As the robot is not driving between $t = 5$, i.e. the end of a_1, and $t = 6$, i.e. the beginning of a_2, $State(A, Init, robot_1, 6)$ is the same as $State(A, Init, robot_1, 5)$, according to the second case of the *State* function.

3. For $State(A, Init, robot_1, 5)$, the first case of the *State* function applies, and it is expanded in terms of $State(A, Init, robot_1, 3)$.

4. For $State(A, Init, robot_1, 3)$, only the third case of the *State* function applies, which specifies that $State(A, Init, robot_1, 3)$ is equal to the initial state $Init(robot_1)$.

3.5 Manipulation of Dynamism

We now focus on the way scenarios arise by relating activities to sources of dynamism. In Section 3.5.1, we focus on modeling constructs for representing the sources of dynamism. In Section 3.5.2 and Section 3.5.3, we elaborate on modeling constructs to capture the way these sources can manipulate activities.

3.5.1 Sources of Dynamism

Sources of dynamism can initiate, alter or terminate dynamism in the environment. *Embedment* is central in describing the relation between a source of dynamism and the environment. Sources of dynamism are not external to the simulated environment [Cla96]; they are *embedded* in environmental entities. The environmental entity represents the tangible part, e.g. hardware, by means of which a particular source has access to the environment. We make a distinction between two kinds of sources:

- *Controllers.* Controllers are the software components that constitute a distributed control application. For example, in a RoboCup Soccer game a distributed control application consists of cooperating controllers embedded in each of the robots of a team. It is clear that a controller must have a means to manipulate the entity it is embedded in, as the controllers can make their robot start driving around, stop driving, etc. We define:

$co_i \in Co^\Omega$	a controller co_i, with Co^Ω the set of all possible controllers.
$Co = \{co_1, co_2, \ldots, co_p\} \in 2^{Co^\Omega}$	the set of controllers of a distributed control application. 2^{Co^Ω} is the power set, i.e. the set of all subsets of all possible controllers.

- *Environment sources.* Environment sources are sources of dynamism in the environment, whose operation is external to the distributed control application. For example, in a RoboCup Soccer game the opponent robot team is remote controlled by humans. The behavior of the opponent team members is an environment source of dynamism. It is clear that environment sources can also manipulate entities in the environment, for example the movement of members of the opponent team. The functioning of environment sources is external to the distributed control application. We define:

$es_i \in Es^\Omega$	an environment source es_i, with Es^Ω the set of all possible environment sources.
$Es = \{es_1, es_2, \ldots, es_q\} \in 2^{Es^\Omega}$	a set of environment sources. 2^{Es^Ω} is the power set, i.e. the set of all subsets, of all possible environment sources.

The set of sources of dynamism is defined as the union of the set of controllers and the set of environment sources:

$so_i \in So^{\Omega}$	a source of dynamism so_i, with So^{Ω} the set of all possible sources of dynamism.
$So = (Co \cup Es) \in 2^{So^{\Omega}}$	the set of sources of dynamism. $2^{So^{\Omega}}$ is the powerset, i.e. the set of all subsets, of all possible sources of dynamism.
$So = \{so_1, so_2, \ldots, so_{p+q}\}$	So after relabeling.

To specify in which environmental entity each source of dynamism is embedded, we define the *Embed* function that maps a source to the environmental entity it is embedded in:

$$Embed : So \rightarrow E$$
$$Embed(so) = e$$

3.5.1.1 The State of Sources of Dynamism

Here, we put forward an abstract description of the state of sources of dynamism. This suffices for the remainder of this chapter. A detailed discussion on what comprises the state of various sources is described in Chapter 4.

The state of the sources of dynamism is the internal status of the controllers and environment sources. Note that the state of a source of dynamism is distinguished from the state of the constituent in which that source is embedded. We define:

$s_{so_i} \in S_{so}^{\Omega}$	the state of a particular source so_i. S_{so}^{Ω} is the set of all possible states of a source of dynamism.
$S_{so} = \{s_{so_1}, \ldots, s_{so_r}\} \in 2^{S_{so}^{\Omega}}$	the state of all sources of dynamism in set So. $2^{S_{so}^{\Omega}}$ is the powerset, i.e. the set of all subsets of S_{so}^{Ω}.

We use the shorthand notation $S_{so}|_{so_i}$ to select the state of source $so_i \in So$. Hence $S_{so}|_{so_i} = s_{so_i}$.

3.5.1.2 RoboCup Soccer Environment: Sources of Dynamism

For the RoboCup Soccer environment, we consider the following sources of dynamism. First, we consider a distributed control application consisting of two controllers *deepblue* and *deepred*, embedded in $robot_1$ and $robot_2$ respectively. Second, we consider an environment source, more specifically a behavior *droid* that is embedded in $robot_3$ and that is not part of the distributed control application. *droid* belongs to the opponent team that is situated in the environment.

$Co = \{deepblue, deepred\}$
$Es = \{droid\}$
$Embed(deepblue) = robot_1$
$Embed(deepred) = robot_2$
$Embed(droid) = robot_3$

The state of the sources of dynamism:

$S_{so} = \langle s_{db}, s_{dr}, s_d \rangle$
$S_{so}|_{deepblue} = s_{db}$
$S_{so}|_{deepred} = s_{dr}$
$S_{so}|_{droid} = s_d$

3.5.2 Influences

An environmental entity in which a source of dynamism is embedded, mediates that source's access to the environment. The embedment of a source determines the way the source can affect the environment and vice versa. A source of dynamism cannot bring about the desired effects directly in the environment. The desired effects must be brought about indirectly, through the causal properties of the environment [HJJ03].

The influence-reaction model [FM96] introduces influences and reactions to model a source's mediated access. A source of dynamism can only perform influences. An influence represents the *attempt* of the source to manipulate the environment. The reaction models what actually happens in the environment in response to the attempts. In contrast to the influence-reaction model, we express the reaction of the environment in terms of manipulation of activities instead of manipulation of state. Influences initiate and terminate activities in the environment.

A source of dynamism autonomously decides at what time to perform an influence. The amount of time it takes a controller to decide upon what to do, results in a cost, i.e. a delay for all its subsequent influences. To determine the time instant an influence occurs, an explicit mapping between the computation process of a source and simulation time is necessary so as to determine how long a source has been computing or waiting (see Chapter 4).

3.5.2.1 Formal Description of Influences

We define an influence as a tuple:

$f \in Inf^{\Omega} = \langle so, t, name, \langle v_1, \ldots, v_r \rangle \rangle$	an influence with following characteristics: $so \in So$: the source that performed the influence. $t \in T$: the time at which the influence occurs. $name \in InfNames$: the name of the influence. $InfNames$ is the set of all possible names for influences. $\langle v_1, \ldots, v_r \rangle \in V_1 \times \ldots \times V_r$: the parameters of the influence The set of all possible influences is Inf^{Ω}.

We use shorthand notations $f|_{so}$, $f|_t$, $f|_n$, $f|_{par}$ and $f|_{v_i}$ to refer to the source, time, name, the parameters and a specific parameter of the influence, respectively.

For the RoboCup Soccer environment, consider the following influences as an example:

$$f_1 \in Inf^{\Omega} = \langle deepblue, t_a, startDriving, \langle \overrightarrow{v_0} \rangle \rangle$$
$$f_2 \in Inf^{\Omega} = \langle deepblue, t_b, stopDriving, \langle \rangle \rangle$$

The influence f_1 represents controller *deepblue* attempting to start driving with velocity vector $\overrightarrow{v_0}$ at time t_a. The influence f_2 represents controller *deepblue* attempting to stop driving at time t_b.

3.5.2.2 Sources Performing Influences

Here, we put forward an abstract description of the way influences are performed. This suffices for the remainder of this chapter. A detailed discussion on the way influences are performed is described in Chapter 4.

From the point of view of the simulated environment, influences are inputs generated by sources of dynamism. To specify the next set of influences performed by the sources of dynamism, we define a function $NextInfs$ that maps the sources of dynamism and their current state on a set of influences they perform next.

$$\begin{aligned} &NextInfs : 2^{So^{\Omega}} \times 2^{S_{so}^{\Omega}} \rightarrow 2^{Inf^{\Omega}} \\ &NextInfs(So, S_{so}) = \{ \quad \langle so_1, t_a, name_0, \langle v_1, \ldots, v_r \rangle \rangle, \\ &\qquad \langle so_2, t_a, name_1, \langle v_1, \ldots, v_u \rangle \rangle, \\ &\qquad \ldots, \\ &\qquad \langle so_k, t_a, name_q, \langle v_1, \ldots, v_w \rangle \rangle \quad \} \end{aligned}$$

The function $NextInfs$ determines the set of influences with the earliest time of occurrence t_a of all influences performed by a set of sources of dynamism So in a given state S_{so}. The details of the function $NextInfs$ are specified in Section 4.7.

Sources of dynamism perform several influences over time. We define a function *SoCycle* that returns a new state for the sources of dynamism, based on their current state. The new state is the result of evolving the current state of the sources of dynamism until the next time instant one or several of the sources perform a new influence:

$$SoCycle : 2^{So^{\Omega}} \times 2^{S_{so}^{\Omega}} \rightarrow 2^{S_{so}^{\Omega}}$$
$$SoCycle(So, S_{so}) = S'_{so}$$

For a set of sources of dynamism So with current state S_{So} of the sources of dynamism, the *SoCycle* function returns a new state S'_{so} for the sources of dynamism. The *SoCycle* function is described in detail in Chapter 4.

3.5.3 Reaction Laws

Because a source's access to the environment is mediated, an influence can lead to a different result than the one intended by the source. For example, consider a controller embedded in a robot, that performs an influence at a particular point in time to start moving with a particular velocity in a particular direction. The activity that is initiated in response to this influence, represents the robot moving forward. However, the precise characteristics of the activity are determined by the characteristics of the robot in which the controller is embedded. For example, due to jitter in the hardware, the direction and velocity of the activity can slightly differ from the ones specified in the influence. Moreover, a robot is not able to travel at a higher velocity than the one it is physically able to achieve, even if a controller attempts to travel faster by performing an influence specifying a higher velocity.

To capture the sources' mediated access in a model, we introduce *reaction laws.* A reaction law is a rule that specifies the reaction of the environment in response to an influence. To determine a reaction, a reaction law takes into account (1) the characteristics of the influence the source performed and (2) the characteristics of the environmental entity in which the source is embedded. The reaction specified by a reaction law manipulates the activities that involve the environmental entity in which the source is embedded. Note that a reaction law does not take into account other constituents, nor does it manipulate activities of other constituents. Dealing with interaction between several constituents is the topic of Section 3.6.

3.5.3.1 Formal Description of Reaction Laws

To formalize reaction laws, we first define a transformation of activities:

$trans \in Trans^{\Omega} = \langle t, Rem, Add \rangle$	an activity transformation at time $t \in T$, which removes the activities of set $Rem \in 2^{A^{\Omega}}$ and adds the activities of set $Add \in 2^{A^{\Omega}}$. The set of all possible transformations is $Trans^{\Omega}$.

We use the shorthand notation $trans|_t$ to select the time of an activity transformation $trans$.

We define a function $ApplyTrans$ which returns a set of activities representing the result of applying a given transformation on a given set of activities:

$$ApplyTrans : 2^{A^{\Omega}} \times Trans^{\Omega} \rightarrow 2^{A^{\Omega}}$$
$$ApplyTrans(A, \langle t, Rem, Add \rangle) = (A \setminus Rem) \cup Add$$

Reaction laws determine the way the environment reacts to the influences performed by the source. We define:

$$Rlaw : 2^{A^{\Omega}} \times (C \rightarrow S) \times Inf^{\Omega} \rightarrow Inf^{\Omega} \times Trans^{\Omega}$$
$$Rlaw(A, Init, f) = \langle f, trans \rangle$$

A reaction law $Rlaw$ is represented as a function that, for a given scenario A, a given $Init$ function and a given influence f, returns a tuple containing the influence and the transformation in response to the influence.

In addition, we define:

$Rlaw \in Rlaws^{\Omega}$	$Rlaws^{\Omega}$ is the set of all possible reaction laws.
$Rlaws = \{Rlaw_1, Rlaw_2, \ldots, Rlaw_r\}$	a set of reaction laws, $Rlaws \in 2^{Rlaws^{\Omega}}$

We define a function $ApplyRlaws$ that applies a set of reaction laws to a given set of influences to determine a set of reactions to the influences:

$$ApplyRlaws : 2^{A^{\Omega}} \times (C \rightarrow S) \times 2^{Rlaws^{\Omega}} \times 2^{Inf^{\Omega}} \rightarrow 2^{Inf^{\Omega} \times Trans^{\Omega}}$$
$$\begin{aligned} ApplyRlaws(A, Init, Rlaws, \{f_1, \ldots, f_r\}) = \; & \{\langle f_1, trans_1 \rangle, \\ & \ldots, \\ & \langle f_k, trans_k \rangle\} \end{aligned}$$

3.5.3.2 RoboCup Soccer Environment: Reaction Law Examples

In the RoboCup Soccer environment, we consider two reaction laws as example: $StartDriveLaw$ and $StopDriveLaw$.

Example 1: StartDriveLaw. The reaction law *StartDriveLaw* is responsible for initiating, in response to an influence, an activity that represents a robot driving in the environment.

$$
\begin{array}{l}
\mathit{StartDriveLaw} : 2^{A^{\Omega}} \times (C \rightarrow S) \times \mathit{Inf}^{\Omega} \rightarrow \mathit{Inf}^{\Omega} \times \mathit{Trans}^{\Omega} \\
\mathit{StartDriveLaw}(A, \mathit{Init}, f) = \\
\qquad \begin{cases}
\langle f, \langle f|_t, \phi, \{a\} \rangle \rangle \\
\quad\quad a = \langle \mathit{Embed}(f|_{so}), f|_t, \infty, \langle \mathit{Jitter}(f|_{\overrightarrow{v_0}}) \rangle, \mathit{Driving} \rangle \\
\quad \text{if } (\ f|_n = \mathit{startDriving}\ ) \wedge \\
\quad\quad (\ \mathit{Embed}(f|_{so}) \in \mathit{Robot}\ ) \wedge \\
\quad\quad (\ \mathit{Active}(A, \mathit{Embed}(f|_{so}), f|_t) = \phi\ ) \wedge \\
\quad\quad (\ \| f|_{\overrightarrow{v_0}} \| \leq v_{max}\ ); \\
\\
\mathit{undefined} \\
\quad \text{otherwise;}
\end{cases}
\end{array}
$$

The law *StartDriveLaw* can be understood as follows. The outcome of the law is undefined, unless the condition, described after if, is valid. This condition is a conjunction of four subconditions:

1. The first subcondition expresses that the name $f|_n$ of the influence f must be equal to *startDriving*.

2. The second subcondition expresses that the source $f|_{so}$ that performed the influence, must be embedded in a robot.

3. The third subcondition states that the robot in which the source is embedded, i.e. $\mathit{Embed}(f|_{so})$ may not yet be driving at the time $f|_t$ the influence is performed. The condition states that no activity is active for the robot at the time the influence is performed.

4. The fourth subcondition states that the length (expressed by the vector norm $\|.\|$) of the velocity vector $f|_{\overrightarrow{v_0}}$ of the influence must be smaller than or equal to the maximum velocity v_{max} the robot can achieve. Otherwise the robot will not start driving.

If all these conditions are valid, the robot starts driving. This is expressed in the outcome of the reaction law *StartDriveLaw*. The outcome is a tuple consisting of the influence f, and an activity transformation $\langle f|_t, \phi, \{a\} \rangle$ in response to f. The activity transformation occurs at the time $f|_t$ of the influence, and results in an extra activity a that is initiated. This activity describes the driving of the robot in which the source of dynamism is embedded. The activity starts at the time $f|_t$, i.e. the same time as the influence, and never ends (its duration is ∞), as the robot will continue driving until instructed otherwise. The velocity of the robot while driving corresponds to the velocity $f|_{\overrightarrow{v_0}}$ described in the influence, after applying a stochastic perturbation, as described by the *Jitter*-function:

$$\begin{aligned}&\mathit{Jitter} : \mathbb{R}^2 \rightarrow \mathbb{R}^2\\&\mathit{Jitter}(\vec{v}) = \vec{v}'\end{aligned}$$

Example 2: StopDriveLaw. The reaction law *StopDriveLaw* is responsible for stopping a driving robot in response to an influence:

$$\begin{aligned}&\mathit{StopDriveLaw} : 2^{A^\Omega} \times (C \rightarrow S) \times \mathit{Inf}^\Omega \rightarrow \mathit{Inf}^\Omega \times \mathit{Trans}^\Omega\\&\mathit{StopDriveLaw}(A, \mathit{Init}, f) =\\&\qquad\begin{cases}\langle f, \langle f|_t, \{a_1\}, \{a_2\}\rangle\rangle\\ \quad \text{with:}\quad a_2 = \langle a_1|_c, a_1|_t, f|_t - a_1|_t, a_1|_{par}, a_1|_F\rangle\\ \quad \text{if } (\ f|_n = \mathit{stopDriving}\ ) \wedge\\ \qquad (\ \mathit{Embed}(f|_{so}) \in \mathit{Robot}\ ) \wedge\\ \qquad (\ \exists a_1 \in \mathit{Active}(A, \mathit{Embed}(f|_{so}), f|_t) : a_1|_F = \mathit{Driving}\ )\\ \\ \mathit{undefined}\\ \quad \text{otherwise;}\end{cases}\end{aligned}$$

The law *StopDriveLaw* can be understood as follows. The outcome of the law is undefined, unless the condition described after *if* is valid. This condition is a conjunction of three subconditions:

1. The first subcondition expresses that the name $f|_n$ of the influence f must be equal to *stopDriving*.
2. The second subcondition expresses that the source $f|_{so}$ that performed the influence, must be embedded in a robot.
3. The third subcondition states that the robot in which the source is embedded, i.e. $\mathit{Embed}(f|_{so})$ is driving at the time $f|_t$ the influence is performed. The condition states that there exists an activity a_1 such that a_1 is active for the robot at that time, and such that a_1 is a *Driving* activity.

If all these conditions are valid, the law *StopDriveLaw* stops the robot from driving any further: the original driving activity a_1 is removed, and a new one a_2 is inserted that ends at the time the influence with name *stopDriving* is performed. Consequently, this outcome of the law is a tuple consisting of the influence f, and an activity transformation $\langle f|_t, \{a_1\}, \{a_2\}\rangle$ in response to f. This transformation removes the activity a_1 that represents the driving of the robot. An extra activity a_2 is added. a_2 is the same activity as a_1, i.e. the elements in its tuple correspond to the elements of a_1, except for its duration. The duration of a_2 is such that the robot stops driving at time $f|_t$, i.e. the time the influence made it stop.

3.5.3.3 RoboCup Soccer Environment: Example Scenario

As an example of both laws, we illustrate how a controller can generate the scenario of the moving robot in Figure 3.3. We start from the initial situation where both robots are standing still on their initial position, as defined by $Init$: we have $A = \phi$, i.e. no robot is moving yet. Based on their current state S_{so}, the sources of dynamism perform the following set of influences:

$$\begin{aligned}&NextInfs(\{deepblue, deepred, droid\}, S_{so}) = \{\ f_1\ \}\\&\qquad f_1 = \langle deepblue, 3, startDriving, \langle \overrightarrow{v_1'} \rangle\rangle\end{aligned}$$

By means of this influence, performed at time $t = 3$, *deepblue* indicates that it wants to start driving. *ApplyRlaws* determines and applies all reaction laws that are applicable. In response to this influence, *StopDriveLaw* is undefined, but *StartDriveLaw* is applicable, and returns:

$$\begin{aligned}&StartDriveLaw(\phi, Init, f_1) = \langle f_1, \langle 3, \phi, \{a_0\}\rangle\rangle\\&\qquad a_0 = \langle robot_1, 3, \infty, \langle \overrightarrow{v_1} \rangle, Driving\rangle\end{aligned}$$

We apply the proposed transformation on $A = \phi$:

$$ApplyTrans(\phi, \langle 3, \phi, \{a_0\}\rangle) = \{a_0\}$$

This represents the situation in Figure 3.4, where $robot_1$ is driving infinitely long.

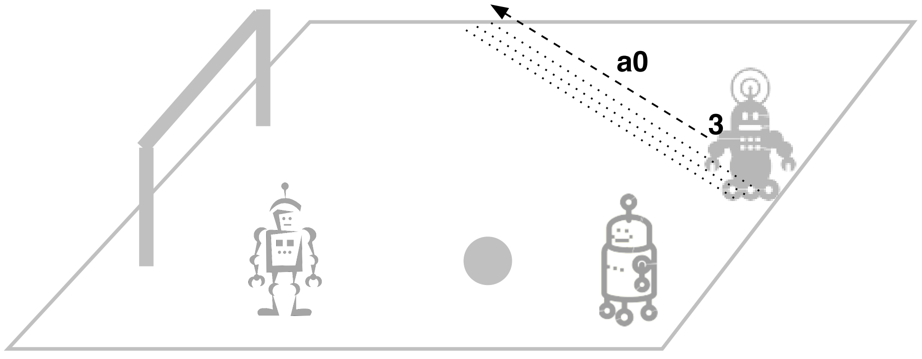


Figure 3.4: Reaction of the environment: activity a_0 initiated in response to influence f_1

The state of the sources of dynamism is evolved until one or several sources perform a new influence:

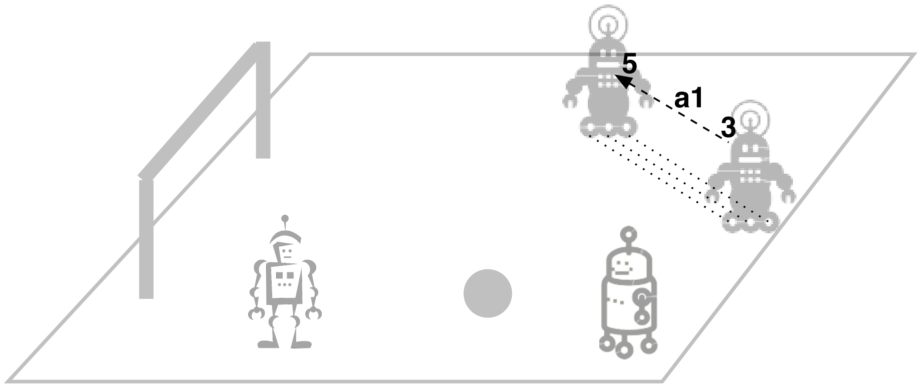


Figure 3.5: Reaction of the environment: activity a_0 replaced by activity a_1 in response to influence f_2

$$SoCycle(\{deepblue, deepred, droid\}, S_{so}) = S'_{so}$$

Based on the new state S'_{so}, we determine the next set of influences:

$$NextInfs(\{deepblue, deepred, droid\}, S'_{so}) = \{\ f_2\ \}$$
$$f_2 = \langle deepblue, 5, stopDriving, \langle\rangle\rangle$$

By means of this influence, *deepblue* indicates it wants to stop driving. *ApplyRlaws* determines and applies all reaction laws that are applicable. To this influence, *StartDriveLaw* is undefined, but *StopDriveLaw* is applicable, and returns:

$$StopDriveLaw(\{a_0\}, Init, f_2) = \langle f_2, \langle 5, \{a_0\}, \{a_1\}\rangle\rangle$$
$$a_1 = \langle robot_1, 3, 2, \langle \overrightarrow{v_1} \rangle, Driving\rangle$$

We apply the proposed transformation on $A = \{a_0\}$:

$$ApplyTrans(\{a_0\}, \langle 5, \{a_0\}, \{a_1\}\rangle) = \{a_1\}$$

The scenario now corresponds to Figure 3.5, where the robot stops driving at time $t = 5$.

3.6 Interaction of Dynamism

Not only sources of dynamism can initiate activities, as entities in which no source of dynamism is embedded can also be involved in activities. For example, it is obvious that a ball can roll, although there is no source embedded in a ball. Activities related to entities in which no source is embedded are typically initiated indirectly. For example, the rolling of a ball is initiated by a robot that kicks the ball. Such indirect initiation of activities is possible because of *interaction*. Dynamism in the environment can interact.

Interaction of dynamism typically implies a discontinuity in its evolution [WGW90]. Dynamism can interact in complex ways. Interaction of dynamism can be unwanted by a controller, typically because it causes a controller's actions not to yield the intended result [FM96, Woo01]. For example, a robot that has started to drive forward in a straight line, hits an obstacle or is pushed aside by another robot. The result is that the robot reaches a position different from the one it intended to reach. However, interaction of dynamism can also be desired by a controller. For example, a robot moves forward into the path of a rolling ball, because it wants to deviate the ball in the direction of the goal.

Interaction of dynamism plays a crucial role, particularly in scenarios involving several sources of dynamism, i.e. a number of controllers as well as several environment sources. Supporting the modeler to cope with interaction of dynamism, requires modeling constructs to represent interaction in the simulation model.

In Section 3.6.1, we describe an example scenario of interaction that will be used throughout the whole section. The modeling constructs to capture interaction in the simulation model, are introduced in Section 3.6.2.

3.6.1 RoboCup Soccer Environment: Example Interaction Scenario

We illustrate interaction of dynamism in the context of the RoboCup Soccer scenario in Figure 3.6. The scenario describes two robots that cooperate to score a goal, but ultimately the goal is prevented by the third robot. We elaborate on the interaction of dynamism in this scenario. $robot_2$ drives according to activity a_3 and interacts with the ball. A robot and a ball are not allowed to penetrate: the robot kicks the ball when both come into contact, which happens at time $t = 5$. As a result of this contact, the ball starts rolling, denoted by activity a_4. At time $t = 8$, the rolling ball makes contact with $robot_1$, which is at that time driving according to activity a_2. As a result of this interaction, the ball is deviated towards the goal, as indicated by activity a_5. However, $robot_3$ has positioned itself in front of the goal according to activity a_6 and blocks the ball to prevent $robot_1$ from scoring.

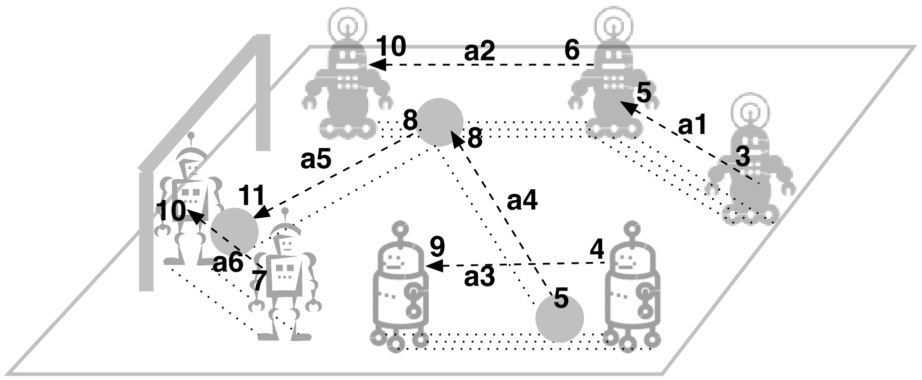


Figure 3.6: An scenario with interaction between robots and a ball.

3.6.2 Interaction Laws

Whereas a reaction law determines a transformation of activities related to one environmental entity, without taking into account any others, an interaction law determines the way several environmental entities can interact.

An *interaction law* represents a domain-specific rule that specifies a way entities can interact. As such, interaction laws can be used to constrain dynamism in the simulated environment according to what is possible in the real environment. The description of an interaction law comprises the following:

- *Interaction conditions.* An interaction condition specifies the circumstances for interaction to occur. Interaction conditions are used to check whether an activity interacts with other entities. An example of an interaction condition is the penetration of a moving robot and the ball.

- *Activity transformations.* In case interaction is detected at a particular time in a given scenario, the interaction law specifies how the interaction affects the evolution described by the activities involved. An *activity transformation* specifies a particular transformation to be applied on the scenario.

Interaction laws are a modeling construct that enables a modeler to define the interactions that are possible in the simulated environment. An interaction law encapsulates interaction conditions and activity transformations to describe when an interaction occurs and what its outcome is.

3.6.2.1 Formal Description of Interaction Laws

We define an interaction law as a function that returns an activity transformation in case its interaction conditions are met:

$$ILaw : 2^{A^\Omega} \times (C \rightarrow S) \rightarrow Trans^\Omega$$
$$Ilaw(A, Init) = trans$$

In addition, we define:

$Ilaw \in Ilaws^\Omega$	$Ilaws^\Omega$ is the set of all possible interaction laws.
$Ilaws = \{Ilaw_1, Ilaw_2, \ldots, Ilaw_r\}$	a set of interaction laws, $Ilaws \in 2^{Ilaws^\Omega}$

We define a function *ApplyIlaws* that applies a set of interaction laws to a given scenario to determine a set of activity transformations:

$$ApplyIlaws : 2^{A^\Omega} \times (C \rightarrow S) \times 2^{Ilaws^\Omega} \rightarrow 2^{Trans^\Omega}$$
$$ApplyIlaws(A, Init, Ilaws) = \{trans_1, \ldots, trans_k\}$$

3.6.3 RoboCup Soccer Environment: Interaction Law Examples

For the RoboCup Soccer Environment, we give three examples of an interaction law: *KickBallLaw*, *DeviateBallLaw* and *StopBallLaw*.

Example 1: KickBallLaw. The interaction law *KickBallLaw* only applies to the case in which a stationary ball is hit by a moving robot. *KickBallLaw* enforces that the ball starts rolling by returning an activity transformation that initates a new activity.

$$
\begin{array}{l}
\mathit{KickBallLaw} : 2^{A^{\Omega}} \times (C \to S) \to \mathit{Trans}^{\Omega} \\
\mathit{KickBallLaw}(A, \mathit{Init}) = \\
\left\{
\begin{array}{l}
\langle t, \phi, \{a\} \rangle \\
\qquad a = \langle b, t, \Delta t, \langle \vec{v}, \vec{d} \rangle, \mathit{Rolling} \rangle \\
\quad \text{if } \exists b \in \mathit{Ball}; \exists t \in T; \exists \Delta t \in \Delta T; \exists r \in \mathit{Robot}; \exists a_1 \in A; \exists \vec{v}, \vec{d} \in \mathbb{R}^2 : \\
\qquad a_1 \in \mathit{Active}(A, r, t) \wedge \\
\qquad \mathit{Active}(A, b, t) = \phi \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t)|_{pos}\| = \epsilon \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t + dt)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t + dt)|_{pos}\| < \epsilon \wedge \\
\qquad \vec{v} = Y_1(A, \mathit{Init}) \wedge \\
\qquad \vec{d} = Y_2(A, \mathit{Init}) \wedge \\
\qquad \Delta t = Y_3(\vec{v}, \vec{d}, \mathit{Rolling}) \\
\\
\mathit{undefined} \\
\quad \text{otherwise;}
\end{array}
\right.
\end{array}
$$

The expression of the law can be understood as follows. The condition states that the law is applicable in case there is a ball b, a time instant t, a duration Δt, a robot r, an activity a_1 and velocity and deceleration vectors $\vec{v}$ and $\vec{d}$ such that:

1. The robot r is driving at time t, expressed by activity a_1.
2. The ball b is not rolling at time t, i.e. none of the activities is active for the ball at time t.
3. The distance between the position of the ball and the robot at time t is ϵ. ϵ represents the distance at which the bodies of the ball and the robot make contact. For example, in case the bodies of ball and robot are circles with radiuses ρ_b and ρ_r, then $\epsilon = \rho_b + \rho_r$
4. The distance at time $t + dt$, with dt an infinitesimal small amount of time, between robot r and ball b is smaller than ϵ. This means that the robot would actually penetrate the ball after time t. This condition is not satisfied in situations where the robot stops driving right when it touches the ball.
5. The vectors $\vec{v}$ and $\vec{d}$ are determined by functions Y_1 and Y_2. We make abstraction on how their precise values are determined.
6. The duration Δt is determined by function Y_3, and corresponds to the time that elapses until the ball's speed reaches zero.

If all these conditions are satisfied, *KickBallLaw* proposes an activity transformation at time t. The transformation represents the addition of an activity a to the scenario. a describes the ball b rolling after time t. Note that according to *KickBallLaw*, the robot is not affected by hitting the ball.

Example 2: DeviateBallLaw. We define the interaction law *DeviateBallLaw* to determine the outcome in case a rolling ball is hit by a robot that is driving:

$$
\begin{array}{l}
\mathit{DeviateBallLaw} : 2^{A^{\Omega}} \times (C \rightarrow S) \rightarrow \mathit{Trans}^{\Omega} \\
\mathit{DeviateBallLaw}(A, \mathit{Init}) = \\
\begin{cases}
\begin{array}{l}
\langle t, \{a_1\}, \{a_2, a_3\}\rangle \\
\qquad a_2 = \langle b, a_1|_t, t - a_1|_t, \langle a_1|_{\vec{v}}, a_1|_{\vec{d}}\rangle, a_1|_F\rangle \\
\qquad a_3 = \langle b, t, \Delta t, \langle \vec{v}, \vec{d}\rangle, \mathit{Rolling}\rangle \\
\quad \text{if } \exists b \in \mathit{Ball}; \exists t \in T; \exists \Delta t \in \Delta T; \exists r \in \mathit{Robot}; \exists a_1, a_4 \in A; \exists \vec{v}, \vec{d} \in \mathbb{R}^2 : \\
\qquad a_1 \in \mathit{Active}(A, b, t) \wedge \\
\qquad a_4 \in \mathit{Active}(A, r, t) \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t)|_{pos}\| = \epsilon \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t + dt)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t + dt)|_{pos}\| < \epsilon \wedge \\
\qquad \vec{v} = Y_1(A, \mathit{Init}) \wedge \\
\qquad \vec{d} = Y_2(A, \mathit{Init}) \wedge \\
\qquad \Delta t = Y_3(\vec{v}, \vec{d}, \mathit{Rolling}) \\
\\
\mathit{undefined} \\
\quad \text{otherwise;}
\end{array}
\end{cases}
\end{array}
$$

The expression of the law can be understood as follows. The condition states that the law is applicable in case there is a ball b, a time instant t, a duration Δt a robot r, an activity a_1 and a_4 and velocity deceleration vectors $\vec{v}$ and $\vec{d}$ such that:

1. The ball b is rolling at time t, expressed by activity a_1.
2. The robot r is driving at time t, expressed by activity a_4.
3. The distance between the position of the ball and the robot at time t is ϵ. ϵ represents the distance at which the bodies of the ball and the robot make contact.
4. The distance at time $t + dt$, with dt an infinitesimal small amount of time, between robot r and ball b is smaller than ϵ. This means that the bodies of the robot and the ball would actually penetrate after time t.
5. The vectors $\vec{v}$ and $\vec{d}$ are determined by functions Y_1 and Y_2. We make abstraction on how their precise values are determined.
6. The duration Δt is determined by functions Y_3, and corresponds to the time that elapses until the ball's speed reaches zero.

If all these conditions are satisfied, *DeviateBallLaw* proposes an activity transformation at time t. The transformation performs two things. On the one hand, it removes the original activity a_1 of the ball, and replaces it with an activity a_2 which is identical to a_1, except that a_2 now stops at the moment ball and robot make contact. Consequently, ball and robot no longer penetrate. On the other hand, the transformation adds a new activity a_3 that represents the new, deviated trajectory of the ball after the time t it hits the robot. Note that according to *DeviateBallLaw*, the robot is not affected by hitting the rolling ball.

Example 3: StopBallLaw. We define the interaction law *StopBallLaw* to determine the outcome in case a rolling ball hits a stationary robot:

$$
\begin{array}{l}
\mathit{DeviateBallLaw} : 2^{A^{\Omega}} \times (C \to S) \to \mathit{Trans}^{\Omega} \\
\mathit{DeviateBallLaw}(A, \mathit{Init}) = \\
\left\{
\begin{array}{l}
\langle t, \{a_1\}, \{a_2\}\rangle \\
\qquad a_2 = \langle b, a_1|_t, t - a_1|_t, \langle a_1|_{\vec{v}}, a_1|_{\vec{d}}\rangle, a_1|_F\rangle \\
\quad \text{if } \exists b \in \mathit{Ball}; \exists t \in T; \exists r \in \mathit{Robot}; \exists a_1 \in A : \\
\qquad a_1 \in \mathit{Active}(A, b, t) \wedge \\
\qquad \mathit{Active}(A, r, t) = \phi \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t)|_{pos}\| = \epsilon \wedge \\
\qquad \|\mathit{State}(A, \mathit{Init}, b, t + dt)|_{pos} - \mathit{State}(A, \mathit{Init}, r, t + dt)|_{pos}\| < \epsilon \wedge \\
\\
\mathit{undefined} \\
\quad \text{otherwise;}
\end{array}
\right.
\end{array}
$$

The expression of the law can be understood as follows. The condition states that the law is applicable in case there is a ball b, a time instant t, a robot r and an activity a_1 such that:

1. The ball b is rolling at time t, expressed by activity a_1.
2. The robot r is not driving at time t, i.e. none of the activities is active for the robot at time t.
3. The distance between the position of the ball and the robot at time t is ϵ. ϵ represents the distance at which the bodies of the ball and the robot make contact.
4. The distance at time $t + dt$, with dt an infinitesimal small amount of time, between robot r and ball b is smaller than ϵ. This means that the bodies of the robot and the ball would actually penetrate after time t.

If all these conditions are satisfied, *StopBallLaw* proposes an activity transformation at time t. The transformation removes the original activity a_1 of the ball, and replaces it with an activity a_2 which is identical to a_1, except that a_2 now stops at the moment ball and robot make contact. Consequently, ball and robot no longer penetrate and the ball stops rolling as soon as it hits the robot. Note that according to *StopBallLaw*, the robot is not affected by hitting the rolling ball.

From the definitions of *KickBallLaw*, *DeviateBallLaw* and *StopBallLaw*, it is clear that the interaction scenario of Figure 3.6 can be supported. In analogy to the *KickBallLaw*, *DeviateBallLaw* and *StopBallLaw*, an interaction laws can be defined to determine what happens in case a robot collides with another robot (that is either stationary or moving), in case a robot or a ball hits the goalpost, etc.

3.7 The Evolution of the Model

We formally specify the evolution of the model as a whole. This specifies how simulation models that are described in terms of the modeling constructs can be executed.

The evolution of the model is defined formally by means of the *Evol* function:

$$\begin{aligned}
&Evol : 2^{So^{\Omega}} \times 2^{A^{\Omega}} \times 2^{S_{so}^{\Omega}} \times (C \rightarrow S) \times 2^{Rlaws^{\Omega}} \times 2^{Ilaws^{\Omega}} \rightarrow 2^{A^{\Omega}} \times 2^{S_{so}^{\Omega}} \\
&Evol(So, A, S_{so}, Init, Rlaws, Ilaws) = \langle A', S'_{so} \rangle \text{ with:} \\
&\qquad A' = Cycle(NextInfs(So, S_{so}), Rlaws, Ilaws, A, Init) \\
&\qquad S'_{so} = SoCycle(So, S_{so})
\end{aligned}$$

The *Evol* function returns an updated scenario A' and an updated state of the sources S'_{so} that comprise the following:

- The updated scenario A' is the result of applying the $NextInfs$ function that determines the next set of influences performed by the sources of dynamism (see Section 3.5.2.2). Subsequently, the *Cycle* function returns an updated scenario in reaction to this set of influences, by applying all transformations specified by the reaction laws and interaction laws in the correct temporal order.

- The updated state of the sources S'_{so} is the result of applying the *SoCycle* function on the current state of the sources (see Section 3.5.2.2).

The *Cycle* function is defined as follows:

$$
\begin{array}{l}
Cycle : 2^{Inf^{\Omega}} \times 2^{Rlaws^{\Omega}} \times 2^{Ilaws^{\Omega}} \times 2^{A^{\Omega}} \times (C \to S) \to 2^{A^{\Omega}} \\
Cycle(Inf, Rlaws, Ilaws, A, Init) = \\
\begin{cases}
Cycle(Inf \setminus \{f\}, Rlaws, Ilaws, ApplyTrans(A, trans), Init) \\
\quad \text{if } \exists f \in Inf; \exists trans \in Trans^{\Omega} : \\
\qquad (\ \langle f, trans \rangle \in ApplyRlaws(A, Init, Rlaws, Inf)\ ) \wedge \\
\qquad (\ \nexists \langle f_i, trans_i \rangle \in ApplyRlaws(A, Init, Rlaws, Inf) : \\
\qquad\quad trans_i|_t < trans|_t\ ) \wedge \\
\qquad (\ \forall trans_j \in ApplyIlaws(A, Init, Ilaws) : \\
\qquad\quad trans|_t < trans_j|_t\ ) \\
\\
Cycle(Inf, Rlaws, Ilaws, ApplyTrans(A, trans), Init) \\
\quad \text{if } \exists trans \in Trans^{\Omega} : \\
\qquad (\ trans \in ApplyIlaws(A, Init, Ilaws)\ ) \wedge \\
\qquad (\ \nexists trans_i \in ApplyIlaws(A, Init, Ilaws) : \\
\qquad\quad trans_i|_t < trans|_t\ ) \wedge \\
\qquad (\ \forall \langle f_j, trans_j \rangle \in ApplyRlaws(A, Init, Rlaws, Inf) : \\
\qquad\quad trans|_t \leq trans_j|_t\ ) \\
\\
A \\
\quad \text{otherwise;}
\end{cases}
\end{array}
$$

The *Cycle* function takes as input a set of newly performed influences, the set of reaction laws and the set of interaction laws, the actual scenario and a function that returns the initial state. The *Cycle* function then determines the updated scenario by applying laws in a recursive manner. There are three possible cases:

1. The first law that is applicable, is a reaction law. This is the case if there exists an influence f and an activity transformation $trans$ for which the following three conditions are satisfied:

   (a) One of the reaction laws proposes the given transformation $trans$ in reaction to the influence f of the set of influences. In other words, the tuple $\langle f, trans \rangle$ is an element of the set that is produced by the *ApplyRlaws* function.

   (b) There does *not* exist another transformation $trans_i$ that is also proposed by the *ApplyRlaws* function and that occurs earlier in time than $trans$.

   (c) All transformations $trans_j$ that are proposed by the interaction laws occur later than the transformation $trans$.

   If all these conditions hold, the transformation $trans$ in response to f is applicable before any other transformation. Consequently, the function *Cycle* is called recursively. The parameters reflect the outcome of the reaction law that proposed $trans$: (1) in the set of influences for which the reactions have yet to be determined the influence f is removed, and (2) the scenario is now the one in which the transformation *trans* is already applied.

2. The first law that is applicable, is an interaction law. This is the case if there exists an activity transformation *trans* for which the following three conditions are satisfied:

   (a) One of the interaction laws proposes the given transformation *trans* for the given scenario A. In other words, *trans* is an element of the set of transformations that is produced by the *ApplyIlaws* function.

   (b) There does *not* exist another transformation $trans_i$ that is also proposed by the *ApplyIlaws* function and that occurs earlier in time than *trans*.

   (c) All transformations $trans_j$ that are proposed by the reaction laws occur at the same time or later than the transformation *trans*.

   If all these conditions hold, the transformation *trans* is applicable to A before any other transformation. Consequently, the function *Cycle* is called recursively. The parameters reflect the outcome of the interaction law that proposed *trans*, i.e. the scenario is now the one in which the transformation *trans* is already applied.

3. No laws are applicable. The current scenario A is returned, as it already reflects all changes.

Note that all activity transformations are applied in increasing temporal order, which is required to guarantee correct causal relations. With respect to activity transformations that occur at the same time, the *Cycle* function specifies the following conventions. In case an activity transformation proposed by a reaction law occurs simultaneously with an activity transformation proposed by an interaction law, the transformation of the interaction law is applied first. For activity transformations proposed by reaction laws that occur simultaneously, the order is not specified. Finally, for activity transformations proposed by interaction laws that occur simultaneously, the order is not specified. It is the responsibility of the developer to specify laws that are consistent with each other, such that the simulation results cannot be affected by the order in which simultaneous laws are applied.

3.8 Discussion

We elaborate on the added value of the modeling framework. In Section 3.8.1, we elaborate on the way the modeling framework supports *model formulation.* In Section 3.8.2, we point out how the modeling framework supports *model translation.* Finally, we reflect upon factors that affect the computational cost in Section 3.8.3.

3.8.1 Support for Model Formulation

The modeling framework describes the meaning, relations and execution semantics of all modeling constructs in a formal way, independent of their implementation in a particular simulation platform. This enables a developer to focus on formulating a simulation model, without taking into account the simulation platform that will be used to execute the model. This raises the abstraction level for the modeler, making it easier to reason about dynamic environments.

The modeling framework provides explicit modeling constructs for modeling dynamic environments of distributed control applications. The constructs capture the characteristics of a dynamic environment in a clear and comprehensive way. The modeling framework provides modeling constructs that represent in an explicit manner (1) the structure of the environment, (2) dynamism in the environment, (3) manipulation, i.e. interaction and reaction, of dynamism in the environment, and (4) sources of dynamism in the environment.

3.8.2 Support for Model Translation

Decoupling the modeling framework from its implementation in a particular simulation platform is also beneficial for model translation. We explain that the modeling framework opens different options for translating a simulation model described in terms of the modeling constructs into an executable simulation, rather than being constrained to one particular simulation platform.

The formal description of the modeling framework *specifies* the functionality that is necessary to support the modeling constructs in an executable simulation. For example:

- The formal description unambiguously specifies how each of the constructs can be expressed.

- The formal description specifies how the state of any constituent of the environment can be derived at any time in a given scenario (see Section 3.4.3).

- The formal description specifies the evolution of models that are expressed in terms of its constructs. In particular it specifies the synchronization between (1) the execution of the controllers, (2) the enforcement of reaction laws and (3) the enforcement interaction laws in the simulated environment (see Section 3.7).

Based on the formal specification, different options can be explored for translating a simulation model described in terms of the modeling constructs into an executable simulation. We indicate three possible tracks.

Development of a Case-Specific Simulation Platform. A first track is to develop a case-specific simulation platform that supports one particular simulation model that is described in terms of the constructs. An example is a simulation platform that supports the RoboCup Soccer simulation model that was described throughout this chapter. The formal specification supports for the development of such a simulation platform by specifying the functionality that is needed to support such a simulation model in an executable simulation.

Development of a Domain-Specific Simulation Platform. A second track is developing a *domain-specific simulation platform* of which the functionality can be reused for any simulation model described in terms of the constructs of the modeling framework. Such a simulation platform encapsulates the functionality to support the modeling constructs in an executable simulation. This track is discussed in detail Chapter 5.

Mapping onto General-Purpose Modeling Constructs. A third track is mapping the constructs of the modeling framework into general-purpose modeling constructs. Such a mapping would enable transforming a simulation model described in terms of the modeling constructs for dynamic environments into a simulation model described in terms of general purpose modeling constructs. As such, general-purpose simulation platforms can be reused to support an executable simulation.

As a guideline for future research, we give a first indication on possible ways to relate the modeling framework to modeling constructs of discrete event simulation and hybrid simulation.

- *Relation to discrete event simulation.* In discrete event simulation [BCNN00], models are described in terms of *state* and *events*. The state is a list of values that are sufficient to describe the state of the system at any point in time. An event is defined as a change of the state that occurs instantaneously at a specific point in time [SB99].

  To support the modeling framework in discrete event simulation, we need to devise a way for mapping the constructs of the modeling framework onto *state* and *events*. There are various design choices. We briefly indicate two possible alternatives.

  In a first alternative, the constituents are considered to form a state. Activities can be represented as a combination of state and events: the parameters of an activity are described as a state, attributed to a particular constituent, whereas the start and the end of each activity are translated into two successive events.

  Supporting activity transformations that result from reaction and interaction laws, is less trivial, as this requires additional infrastructure for inspecting and manipulating the event queue.

In a second alternative, the set of activities is considered to form a state, and the activity transformations proposed by the laws are represented as events. Consequently, the state of the constituents is defined implicitly, and the *State* function specified in Section 3.4.3 has to be implemented to derive the state.

- *Relation to hybrid simulation.* In hybrid simulation [Mos99, EK04], the evolution of the system is continuous and discrete. Continuous phases are alternated by discrete events. In a continuous phase, time advances, and the values of the state variables are determined by equations as a function of time. When a discrete event occurs, the equations and the state variables are altered in a discontinuous manner.

  Events can be of two kinds:

  - *Time events.* Time events are scheduled at a predetermined time.
  - *State events.* State events are scheduled at the occurrence of a particular condition, i.e. when the continuous phase exceeds certain thresholds. As such, it is not known a priori at what time a state event occurs. Consequently, the simulation engine has to detect whether state events occur, and at what time their occurrence takes place.

  To map the modeling framework on the concepts of hybrid simulation, we need to devise a way for translating the constructs of the modeling framework onto *state variables*, *equations*, *time events* and *state events*. We give some further indications.

  The state of the constituents could be translated into state variables, and activities represented by equations. As interaction laws represent conditions on the continuous phase, they can be represented as state events. Supporting influences and reaction laws requires further investigation. Reaction laws are triggered by influences. Influences are performed by sources of dynamism that autonomously decide when and which influences to perform. As such, the occurrence of influences is not known beforehand whereas time events are predetermined. Moreover, state events are expressed as conditions on the continuous phase, whereas influences result from discrete computation performed by the sources of dynamism.

3.8.3 Computational Cost

The formal description serves as a specification, not committed to specific algorithmic solutions to underpin it. As such, the formal description enables the use of custom algorithmic solutions that are optimized for a specific simulation study. The computational cost of executing a model that is based on the modeling framework, is dependent upon the following:

- *Complexity of the model.* The inherent complexity of the model has a significant impact on the computational cost to execute it. The complexity of a model is determined by the granularity of the activities, the quantity and complexity of the laws, etc.
- *Characteristics of the experiments.* For a particular model, the experimental setting can have an impact on the computational cost. For example, the initial position and the density of robots can have an impact on the frequency at which collisions occur.
- *Algorithmic solutions.* For a particular model and experiment, the chosen algorithmic solutions can have an impact on computational cost. Examples are fast collision detection algorithms, efficient mechanisms for matching reaction laws with influences, excluding activities that can never interact from checks by the interaction laws, etc.

A measurement of the computational cost, in the context of a specific simulation study is described in Chapter 6.

3.9 Related work

Various simulations exist that incorporate characteristics of dynamic environments. We first elaborate on approaches that rely on general-purpose modeling constructs to represent dynamic environments. Afterwards, we elaborate on the support offered by domain-specific simulation platforms.

3.9.1 Modeling Dynamic Environments with General-Purpose Modeling Constructs

We give examples of existing approaches that use general-purpose modeling constructs to incorporate the characteristics of dynamic environments in a simulation model.

- [SU01] uses the constructs of discrete event simulation to represent the Tile-World [PR90], a grid environment where robots can transport tiles. The requests from the agents are decoupled from the reaction of the environment. Events are used to capture the request (i.e. influence) of an agent to move, as well as for representing the actual movement (i.e. activity) that results from it. Protocols (i.e. reaction laws) are used to specify the movement events that correspond to request events.

- [EKP01] uses the constructs of hybrid simulation [Mos99, EK04] to represent an environment of automated guided vehicles. The movements (i.e. activities) of the vehicles are modeled by means of *differential equations* that define the position and velocity of each vehicle as a function of time. *State events* are used to model interaction laws that happen at the occurrence of a particular condition, i.e. when the values of the differential equations exceed certain thresholds.

- Rigid body simulation [Mir00] applies the concepts of hybrid simulation to simulate fine-grained interactions of non-deformable bodies: fast collision detection algorithms [GLM96, Mir98, Hub96], impact models and methods to enforce general motion constraints – especially the non-penetration constraints [Bar94] – have been developed.

The examples illustrate that general-purpose modeling constructs allow the characteristics of dynamic environments to be incorporated in a simulation model. The meaning of general purpose modeling constructs is formally documented, which facilitates their use. However, their meaning does not specifically refer to dynamic environments of distributed control applications. As such, general-purpose modeling constructs do not support dynamic environment in an explicit manner.

3.9.2 Modeling Dynamic Environments in Domain-Specific Simulation Platforms

We give examples of simulation platforms for simulating distributed control applications in dynamic environments, and investigate the modeling constructs they offer to incorporate the characteristics of dynamic environments in a simulation model.

- XRaptor [BMP$^+$] is a simulation platform that supports two- or three-dimensional continuous environments to study the behavior of a large number of agents. XRaptor offers a number of abstractions to support simulations of mobile devices in an environment: an *agent* is either a point, a circular area or a spherical volume that contains a *sensor unit* for observing the world, an *actuator unit* for performing actions and a *control kernel* for action selection. Ordinary differential equations are used for modeling movements.

  It is clear that supported abstractions are not general-purpose modeling constructs, but are targeted towards simulations of mobile devices in an environment. However, the precise meaning and responsibilities of the abstractions is not documented in a formal way. Consequently, using the constructs for a specific simulation study requires investigating the design and implementation of the simulation platform, e.g. to understand the precise meaning of an agent, the relation of an agent to the sensor and actuator units, the responsibilities of sensor and actuator units, the relation between agents and points, circular areas and volumes the environment, etc.

- *SPARK* [OR04] is a simulation platform for physical multi-agent systems in three dimensional environments. SPARK supports a flexible *agent* representation with different *sensors*, *actuators* and *morphologies.* These abstractions are described as follows [OR04]:

  "Agent programs *are external processes for the simulator. The representation of the agent properties inside the simulator is almost equal to the representation of all other* objects *in the simulation. There are* bodies *(i.e. mass and a mass distribution) for the physical simulation. Additionally, agents possess perceptors and effectors.* Perceptors *provide sensory input to the agent program associated with the representation of the agent in the simulator, and the agent program uses the* effectors *to act in its environment. Other objects in the simulation and the physics of the system can affect the situation of agents; this is reflected in the respective aspects by changing the positions or velocities."*

  It is clear that the supported abstractions are targeted at simulations of mobile devices in a dynamic environment. However, the precise meaning and responsibilities of all abstractions is not documented in a formal way and requires detailed knowledge of the design and implementation of the simulation platform.

The examples above illustrate that simulation platforms for simulating distributed control applications in dynamic environments offer constructs that are targeted at capturing dynamic environments of distributed control applications. However, the meaning of these modeling constructs is only described in an informal manner. Consequently, formulating a simulation model in terms of such constructs requires detailed knowledge of the design and implementation of the simulation platform.

3.10 Conclusions

In this chapter, we presented a modeling framework that contains formally specified modeling constructs that are specifically aimed at modeling dynamic environments of distributed control applications. The modeling framework supports in an explicit manner a number of characteristics that are pertinent for modeling dynamic environments of distributed control applications:

- A dynamic environment has a particular *scope.* Environmental entities, environmental properties and environment layout are modeling constructs to represent all constituting parts in the environment and the way they are arranged with respect to each.

- A dynamic environment *encapsulates* its own dynamism. Activities capture the evolution of all entities in the environment in an explicit manner.
- A dynamic environment *regulates* its own dynamism. By means of reaction laws, the environment governs how dynamism is affected in response to influences. By means of interaction laws, interactions of dynamism in the environment can be supported.
- A dynamic environment *embeds sources* of dynamism. These sources can be controllers of the distributed control application or environment sources external to the distributed control application. Sources of dynamism are inherently part of the environment, as they are embedded in environmental entities. Nevertheless, sources of dynamism are restricted in their ability to affect dynamism in the environment. Sources can only affect dynamism in an indirect manner, using influences.

We illustrated the modeling framework by applying it for modeling a dynamic RoboCup Soccer environment.

For software-in-the-loop simulations of distributed control applications in dynamic environments, the contribution of the modeling framework is the twofold. On the one hand, the modeling framework provides explicit support for *model formulation* by offering formally specified constructs for modeling dynamic environments. The formal specification enables a modeler to formulate a simulation model without taking into account a particular simulation platform. On the other hand, the modeling framework support *model translation* by specifying the functionality that is needed to support the constructs in an executable simulation.

Chapter 4

Modeling the Integration of the Control Software

In this chapter, we focus on the control application part of the modeling framework. We introduce modeling constructs aimed at modeling the integration of the software of a real distributed control application in a simulation. The focus is on the specification of the modeling constructs. Design and implementation issues to support the constructs are tackled in Chapter 5.

4.1 Introduction

In software-in-the-loop simulations, the software of the real controllers of the distributed control application is embedded in a simulated environment. We put forward modeling constructs that enable a modeler to specify the way the control software of a distributed control application is integrated in the simulation model.

The modeling constructs are part of the modeling framework. The modeling framework specifies the meaning, relations and execution semantics of the modeling constructs. The formal description decouples the modeling constructs from their implementation in a particular simulation platform.

To introduce the modeling constructs in an intuitive manner, we use the modeling constructs for modeling the integration in a simulation of a simplified distributed control application for steering RoboCup Soccer robots.

This chapter is structured as follows. Section 4.2 gives an overview of all modeling constructs of the control application part of the modeling framework. In Section 4.3, we describe a simplified distributed control application for RoboCup Soccer robots that will be used as an example throughout this chapter. In the next sections, we elaborate on each of the modeling constructs in detail: each construct is described informally, illustrated in the context of the example distributed control application and complemented with a formal specification. In Section 4.4, we introduce modeling constructs to capture the execution time of each controller. Section 4.5 focusses on modeling constructs for capturing the way the software of a controller affects the environment. We focus on the way executing the control software generates influences in Section 4.6. In Section 4.7, the evolution of the model is revisited to include the evolution of the control software and the environment sources of dynamism. Finally, we discuss related work in Section 4.8 and draw conclusions in Section 4.9.

4.2 Overview of the Modeling Constructs

We start with an overview of all constructs of the modeling framework for distributed control applications, before elaborating on each construct in detail in the next sections.

Figure 4.1 is a detailed view on the group of concepts to represent the sources of dynamism in Figure 3.1, with additional modeling constructs for the controller. We give an overview of the modeling constructs in the controller. The modeling constructs focus on representing the following characteristics of the control software explicitly in the simulation model:

- *Representing the real-world execution time of the software in the simulation model.* The real-world execution time of a controller is the amount of wall-clock time that elapses until that controller triggers its next action. The execution time of a controller determines the timing of its actions. In a dynamic environment, the timing of actions is crucial as opportunities come and go.

  To capture the real-world execution time of a controller in the simulation model, we put forward the modeling constructs `Duration Primitive` and `Duration Mapping`. A duration primitive represents a code segment that takes an amount of execution time in the real world that is pertinent for the simulation. An example of a duration primitive is a particular java method *foo()* in the software of a particular controller. A duration mapping is a modeling construct that specifies the execution time for invocations of duration primitives of a controller. For example, a duration mapping can specify that invoking the method *foo()* takes 0.338 seconds.

- *Capturing the interaction of the software with the environment.* The software of a controller interacts with its environment. Consequently, the execution of the software of a controller will trigger particular things to happen in the environment. When integrating the software of the controllers with the simulated environment, it is crucial to identify the set of software instructions

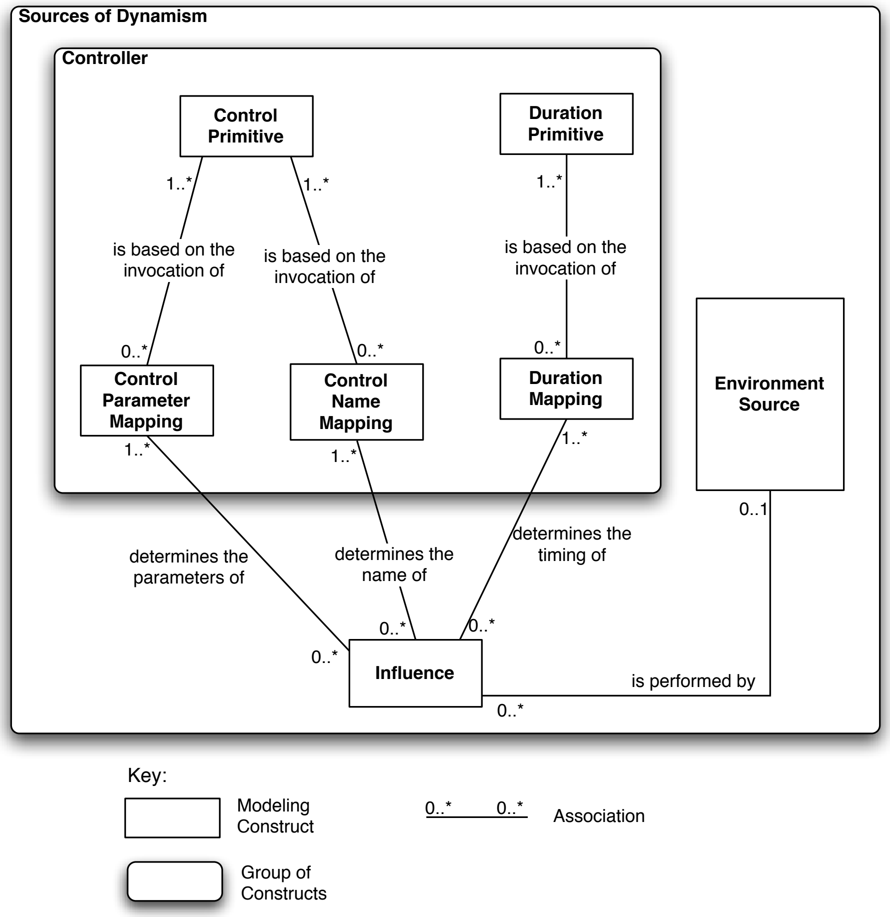


Figure 4.1: Overview of the modeling constructs for the control software and their associations

that are used by the controller to interact with the environment, and to specify the precise consequences in the environment that result from triggering these instructions.

To capture the interaction of the software with the environment in a simulation model, we put forward the modeling constructs `Control Primitive`, `Control Name Mapping` and `Control Parameter Mapping`. A control primitive represents a particular software instruction that can be used by the

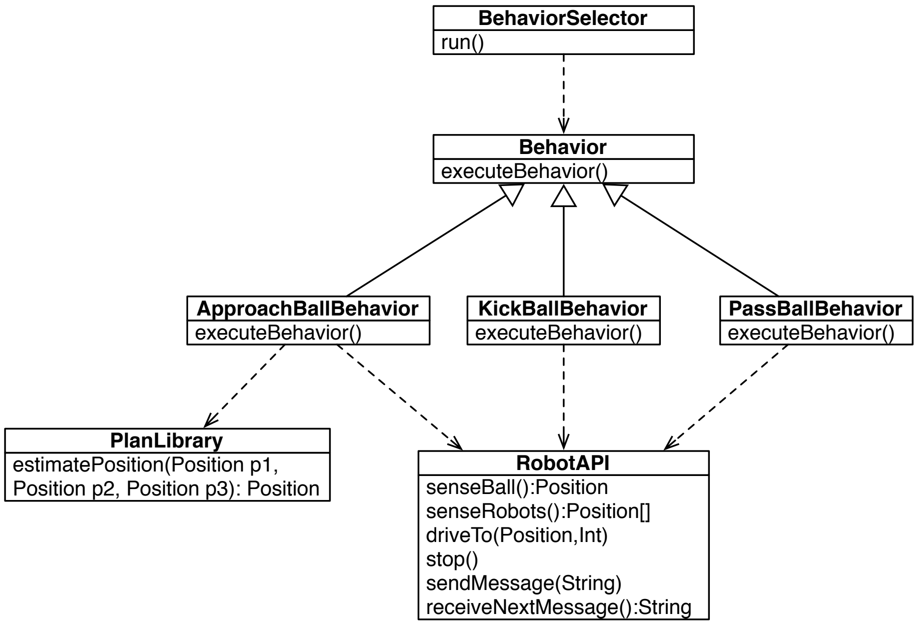


Figure 4.2: Class diagram of a simplified RoboCup Soccer controller

control software to interact with its environment. An example of a control primitive is a java method *bar()* that triggers the engine of a robot to start running at full power. The modeling constructs `Control Name Mapping` and `Control Parameter Mapping` specify the name and the parameters of the influence that result from invoking a control primitive. A control name mapping and control parameter mapping decouple the signature of control primitives from the specific representation of influences that is used in the simulated environment. For example, a control name mapping specifies that an invocation of *bar()* corresponds to an influence with name *startDriving*, whereas a control parameter mapping specifies that the invocation of *bar()* results in the value 10 to be associated with the parameter of the *startDriving* influence to indicate the speed.

4.3 An Example Distributed Control Application for RoboCup Soccer Robots

We describe a simplified distributed control application that will be used as an example throughout this chapter. The distributed control application is targeted at the RoboCup Soccer robots presented in Chapter 3: the distributed control application comprises two controllers *deepblue* and *deepred*, embedded in two different RoboCup Soccer robots, i.e. $robot_1$ and $robot_2$ respectively (see Section 3.5.1.2). We describe a simplified design for the RoboCup Soccer controllers.

The class diagram of a single controller is depicted in Figure 4.2. The controller design comprises of the following classes:

- `BehaviorSelector` is responsible for selecting the active behavior for the robot. `BehaviorSelector` can switch the active behavior depending on the circumstances.

- `Behavior` encapsulates a particular behavior of the robot. A behavior instructs the robot to achieve a particular goal. The behaviors include `ApproachBallBehavior` to move towards the ball, `PassBallBehavior` to pass the ball towards a team member, and `KickBallBehavior` to kick the ball towards the goal.

- `RobotAPI` is a class that represents the interface to the hardware of the robot. `RobotAPI` provides a number of methods to instruct the sensors and actuators of the robot: `senseBall` and `senseRobots` use the robot's sensors to return the current position of the ball and the positions of other robots respectively; `driveTo` uses the robot's actuators to start driving towards a given position with a given velocity; `stop` causes the robot to stop driving. Finally, `sendMessage` and `receiveMessage` use a robot's communication module to send and receive messages to and from other robots.

- `PlanLibrary` is a class that encapsulates logic to support the `ApproachBallBehavior`. `PlanLibrary` provides a method `estimatePosition` that uses an extrapolation algorithm to return an estimated future position based on three position samples.

We focus on the `ApproachBallBehavior`. This behavior continuously moves its robot towards the ball, based on a rough estimate of where the ball will be positioned in the future. The implementation of the `executeBehavior` method of `ApproachBallBehavior` is shown in Figure 4.3. `executeBehavior` runs in a loop as long as `ApproachBallBehavior` is active (line 16). The loop comprises the following:

- Sampling the position of the ball (lines 19 to 23): three samples (i.e. `ballpos1`, `ballpos1` and `ballpos1`) of the position of the ball are taken

```
public class ApproachBallBehavior extends Behavior{

    /**
     *Override Behavior.executeBehavior()
     */
    public void executeBehavior() {

        //variables representing position samples of the ball
        Position ballpos1;
        Position ballpos2;
        Position ballpos3;
        //variable representing the target position of the robot
        Position targetpos;

        //enter the control loop as long as this behavior is active
        while (getBehaviorSelector().isActive(this)) {

            //sample ball position three times with a 400ms interval
            ballpos1 = getRobotAPI().senseBall();
            Thread.sleep(400);
            ballpos2 = getRobotAPI().senseBall();
            Thread.sleep(400);
            ballpos3 = getRobotAPI().senseBall();

            //log position samples
            Logger.log("Sensed ball positions: "+ballpos1+ballpos2+ballpos3);

            //estimate future position of ball
            targetpos = getPlanLibrary().estimatePosition(ballpos1,ballpos2,ballpos3);

            //move towards estimated position with velocity 10
            getRobotAPI().driveTo(targetpos,10);

            //log target position
            Logger.log("Started driving towards "+targetpos);

            //wait 1 second while the robot is moving
            Thread.sleep(1000);
        }
        //stop driving to end this behavior
        getRobotAPI().stop();
    }

//other methods omitted
}
```

Figure 4.3: Implementation of the executeBehavior() method of ApproachBallBehavior

by invoking `senseBall` on `RobotAPI`. In between, `DecisionTaker` invokes a `Thread.sleep(400)` to suspend its execution during 400 milliseconds.

- Logging the sampled positions (line 26).
- Estimating the future position of the ball (line 29): based on the three samples, the future position `targetpos` of the ball is estimated by invoking `estimatePosition` on the `PlanLibrary`.
- Driving to the estimated position of the ball (line 32): the robot is instructed to start driving to the estimated position `targetpos` of the ball by invoking `driveTo` on `RobotAPI`.
- Logging the target position of the robot (line 35).
- Waiting one second (line 38): while driving, `DecisionTaker` invokes a `Thread.sleep(1000)` to suspend its execution during 1 second before proceeding with the next cycle of the loop.

When exiting the loop, the robot is instructed to stop driving (line 41) by invoking `stop` on `RobotAPI` before the `executeBehavior` method ends.

4.4 Execution Time of a Controller

We put forward modeling constructs that enable a developer to specify the execution time of a control application in an explicit manner. We advocate that an explicit model of the execution time of a distributed control application is imperative:

- *An explicit model of execution time is platform independent.* The execution time of a controller is dependent upon the devices on which the controllers are deployed in the real world. However, the computer platform on which a distributed control application is deployed for simulation purposes can differ significantly from the characteristics of the devices on which the controllers are deployed in the real world. Consequently, during a simulation run on a particular platform, the execution time of the controllers typically differs from their execution time in the real world. An explicit model specifies the execution time in the real world, independent of the computer platform that is used to execute the simulation.
- *An explicit model of execution time is selective.* To support application development, auxiliary software is often inserted in controllers of a distributed control application. Auxiliary software can comprise code for debugging, logging to file, configuration, interfacing with the user, implementation stubs, etc. Typically this auxiliary software is removed from the distributed control

application before it is deployed in the real environment and as such it does not affect the execution time of a controller in the real environment. An explicit model can selectively specify which parts of the controller software have a relevant execution time.

- *An explicit model of execution time enables repeatable simulations.* The execution time of a controller is typically non-deterministic, i.e. small random variations are possible with respect to the execution time. In simulation, non-determinism must always be supported in a controlled manner, i.e. in a simulation all non-determinism should be based on random numbers originating from a random number generator with a known seed. Using the same seed for the random number generator then guarantees the same trace of random numbers during a simulation run, which is a prerequisite to obtain simulation results that can be repeated over and over again. By specifying all random variations on the execution time explicitly, an explicit model of execution time supports non-determinism in a controlled manner.

To model the execution time of a distributed control application, we introduce modeling constructs (1) for identifying all computations of which the execution time is relevant according to the modeler (see Section 4.4.1), and (2) for specifying the real-world execution time for each of these computations in an explicit model (see Section 4.4.2).

4.4.1 Duration Primitives and Duration Primitive Invocations

We introduce *duration primitives* and *duration primitive invocations* as modeling constructs to identify computations of a controller with an execution time that is pertinent for the simulation.

- A *duration primitive* identifies a code segment that takes an amount of execution time in the real world that is pertinent for the simulation. A modeler can identify duration primitives at a desired level of granularity. An example of employing duration primitives at a high level of granularity, is the case where duration primitives are used to pinpoint time-consuming methods, functions or procedures in the controller's code. An example of employing duration primitives at a lower level of granularity, is the case where duration primitives are used to pinpoint language primitives of the programming language in which the controller's code is written.

  We define:

$dp_i \in DP^{\Omega}$	the set of all possible duration primitives.
$DP_{co_i} = \{dp_1, dp_2, \ldots, dp_n\} \subseteq DP^{\Omega}$	the set of duration primitives of controller $co_i \in Co$.

- A *duration primitive invocation* represents an invocation of a particular duration primitive of a particular controller with particular parameters. We define:

$dpi = \langle co_i, dp_j, par \rangle$	a duration primitive invocation with the following characteristics: $co_i \in Co^{\Omega}$: the controller of which a duration primitive is invoked. $dp \in DP^{\Omega}$: the duration primitive invoked. $par = \langle v_1, \ldots, v_r \rangle \in V_1 \times \ldots \times V_r$: a tuple containing the parameters of the invocation, with V_i the value domain of parameter v_i.
$dpi \in DPI^{\Omega}$	the set of all possible duration primitive invocations.
$2^{DPI^{\Omega}}$	the powerset, i.e. the set of all subsets, of duration primitive invocations.

We use the following shorthand notations: $dpi|_{co}$ to select the controller, $dpi|_{dp}$ to select the duration primitive, $dpi|_{par}$ to select the tuple of parameters, and $dpi|_{v_i}$ to select a specific parameter v_i.

An Example. We apply the constructs on the `ApproachBallBehavior` described in Section 4.3. We consider two methods, i.e. `Thread.sleep` and `PlanLibrary.estimatePosition`, that take a relevant amount of execution time. Consequently, we identify the following duration primitives:

$$DP_{deepblue} = \{Thread.sleep, PlanLibrary.estimatePosition\}$$
$$DP_{deepred} = \{Thread.sleep, PlanLibrary.estimatePosition\}$$

As examples of duration primitive invocations, we consider the following invocations of `Thread.sleep` and `PlanLibrary.estimatePosition` of *deepblue* and *deepred*:

$$dpi_1 = \langle deepblue, Thread.sleep, \langle 400 \rangle \rangle$$
$$dpi_2 = \langle deepblue, PlanLibrary.estimatePosition, \langle pos_1, pos_2, pos_3 \rangle \ \rangle$$
$$dpi_3 = \langle deepred, Thread.sleep, \langle 1000 \rangle \rangle$$
$$dpi_4 = \langle deepred, PlanLibrary.estimatePosition, \langle pos_4, pos_5, pos_6 \rangle \ \rangle$$

The arguments of each method call are represented in the parameter tuple of the respective duration primitive invocation. pos_i is a position. Note that these duration primitives exclude auxiliary code such as logging.

4.4.2 Duration Mapping

We introduce a *duration mapping* as a modeling construct that specifies a duration for all duration primitive invocations that can happen in a distributed control application. The specified duration represents the real-world execution time of a particular duration primitive invocation.

We define a duration mapping as a function $DurMap$:

$$\begin{aligned}&DurMap : DPI^{\Omega} \rightarrow \Delta T\\&DurMap(dpi) = \Delta t\end{aligned}$$

$DurMap$ is a function that specifies a duration Δt for a given duration primitive invocation dpi.

An Example. We describe an example duration mapping for the distributed control application described in Section 4.3 that consists of controllers *deepblue* and *deepred*:

$$DurMap(\langle co_i, dp_j, par\rangle) = \begin{cases} par|_{v_1}/1000 \\ \quad \text{if } dp_j = Thread.sleep \\ \\ Gauss(0.120, 0.010) \\ \quad \text{if } dp_j = PlanLibrary.estimatePosition \wedge \\ \qquad co_i = deepblue \\ \\ Gauss(0.200, 0.015) \\ \quad \text{if } dp_j = PlanLibrary.estimatePosition \wedge \\ \qquad co_i = deepred \end{cases}$$

The duration mapping $DurMap$ can be understood as follows. The expression contains three cases:

1. The first case applies if the duration primitive is $Thread.sleep$. In this case, the duration of the duration primitive invocation is the value of its parameter, i.e. the argument specifying the number of milliseconds the thread should sleep, divided by 1000 (to rescale the duration expressed in milliseconds to seconds).
2. The second case applies if the duration primitive is $PlanLibrary.estimatePosition$ and the controller is *deepblue*. In this case, the duration of the duration primitive invocation is returned by the $Gauss$ function, which generates a Gaussian random number with given

mean and given standard deviation. Note that *Gauss* is an example of supporting non-determinism in a controlled manner, which allows repeatable simulation results. In this case, *Gauss* returns a Gaussian random number with mean 0.120 and standard deviation 0.010.

3. The third case applies if the duration primitive is *PlanLibrary.estimatePosition* and the controller is *deepred*. In this case, the duration of the duration primitive invocation is a Gaussian random number with mean 0.200 and standard deviation 0.015.

We apply the example duration mapping on the example duration primitive invocations of Section 4.4.1:

$$\begin{aligned}&DurMap(\langle deepblue, Thread.sleep, \langle 400\rangle\rangle) = 0.400\\&DurMap(\langle deepblue, PlanLibrary.estimatePosition, \langle pos_1, pos_2, pos_3\rangle\ \rangle) = 0.123\\&DurMap(\langle deepred, Thread.sleep, \langle 1000\rangle\rangle) = 1.000\\&DurMap(\langle deepred, PlanLibrary.estimatePosition, \langle pos_4, pos_5, pos_6\rangle\ \rangle) = 0.194\end{aligned}$$

4.4.3 Determining Execution Time

Based on a duration map, determining the execution time of a controller happens by counting the durations of all duration primitives invocations that happen during the execution of that controller.

We define a function *ExTime* that determines the execution time of a set of duration primitive invocations of a controller according to a specified duration map:

$$\begin{aligned}&ExTime : 2^{DPI^{\Omega}} \times (DPI^{\Omega} \to \Delta T) \to \Delta T\\&ExTime(DPI, DurMap) =\\&\qquad\begin{cases} DurMap(dpi) + ExTime(DPI \setminus \{dpi\}) \\ \quad \text{if } dpi \in DPI \\ \\ 0 \\ \quad \text{if } DPI = \phi \end{cases}\end{aligned}$$

Based on a particular duration map *DurMap*, the *ExTime* function specifies in a recursive manner the execution time for performing a set of duration primitive invocations $DPI \in 2^{DPI^{\Omega}}$. We assume that all duration primitive invocations in DPI are of the same controller: $\forall dpi_i, dpi_j \in DPI : dpi_i|_{co} = dpi_j|_{co}$. We explain the two cases in the domain of the *ExTime* function:

1. The first case applies when the set of duration primitive invocations DPI is non-empty, i.e. DPI contains at least one duration primitive invocation dpi. In this case, *ExTime* is recursively specified as the sum of (1) the duration $DurMap(dpi)$, i.e. the duration of duration primitive invocation $dpi \in DPI$

according to duration map $DurMap$, and (2) the duration $ExTime(DPI \setminus \{dpi\})$, i.e. the execution time of the duration primitive invocations of the set DPI in which duration primitive invocation dpi has been removed.

2. The second case applies when the set of duration primitive invocations DPI equals the empty set ϕ. In this case, the outcome of $ExTime$ is zero.

An Example. We illustrate the $ExTime$ function on the following set of duration primitive invocations:

$DPI = \{dpi_1, dpi_2, dpi_3\}$ with:
$dpi_1 = \langle deepblue, Thread.sleep, \langle 400 \rangle \rangle$
$dpi_2 = \langle deepblue, PlanLibrary.estimatePosition, \langle pos_1, pos_2, pos_3 \rangle \ \rangle$
$dpi_3 = \langle deepblue, Thread.sleep, \langle 1000 \rangle \rangle$

We expand the recursion for applying $ExTime$ for the set DPI and the example duration mapping $DurMap$ described in Section 4.4.2:

$$\begin{aligned}
&ExTime(\{dpi_1, dpi_2, dpi_3\}, DurMap)\\
&= DurMap(dpi_1) + ExTime(\{dpi_2, dpi_3\}, DurMap)\\
&= DurMap(dpi_1) + (DurMap(dpi_2) + ExTime(\{dpi_3\}, DurMap))\\
&= DurMap(dpi_1) + (DurMap(dpi_2) + (DurMap(dpi_3)+\\
&\quad ExTime(\phi, DurMap)))\\
&= 0.400 + (0.123 + (1.000 + 0))\\
&= 1.523
\end{aligned}$$

The total execution time for the three duration primitive invocations of controller *deepblue* is 1.523 seconds.

4.5 Capturing the Control Interface

Each controller uses a particular *control interface* to interact with the entity in which it is embedded. The control interface delineates the possible ways of a controller to access the environment. We introduce modeling constructs to capture the control interface of each particular controller in Section 4.5.1, and modeling constructs to specify the influences that result from triggering this interface in Section 4.5.2.

4.5.1 Control Primitives and Control Primitive Invocations

We introduce *control primitives* and *control primitive invocations* as modeling constructs to capture the control interface and the way it is triggered.

We define:

$cp_j \in CP^{\Omega}$	CP^{Ω} is the set of all possible control primitives.
$CP_{co_i} = \{cp_1, \ldots, cp_n\}$	the control primitives cp_j that constitute the control interface CP_{co_i} of controller co_i.

An Example. For the example control application described in Section 4.3, the control primitives of the control interface of *deepblue* and *deepred* are:

$$\begin{aligned} CP_{deepblue} = CP_{deepred} = \{&RobotAPI.senseBall, \\ &RobotAPI.senseRobots, \\ &RobotAPI.driveTo, \\ &RobotAPI.stop, \\ &RobotAPI.sendMessage, \\ &RobotAPI.receiveNextMessage\} \end{aligned}$$

In analogy with duration primitive invocations, a control primitive invocation is the invocation of a particular control primitive. We define:

$cpi = \langle co_i, cp_j, par \rangle$	a control primitive invocation with the following characteristics: $co_i \in Co^{\Omega}$: the controller of which a control primitive is invoked. $cp \in CP^{\Omega}$: the control primitive invoked. $par = \langle v_1, \ldots, v_r \rangle \in V_1 \times \ldots \times V_r$: a tuple containing the parameters of the invocation, with V_i the value domain of parameter v_i.
$cpi \in CPI^{\Omega}$	the set of all possible control primitive invocations.
$2^{CPI^{\Omega}}$	the powerset, i.e. the set of all subsets, of control primitive invocations.

We use the following shorthand notations: $cpi|_{co}$ to select the controller, $cpi|_{cp}$ to select the control primitive, $cpi|_{par}$ to select the tuple of parameters, and $cpi|_{v_i}$ to select a specific parameter v_i.

An Example. As examples of control primitive invocations, we consider the following invocations of `RobotAPI.stop` and `RobotAPI.driveTo` of *deepblue* and *deepred*:

$$cpi_1 = \langle deepblue, RobotAPI.stop, \langle\rangle \ \rangle$$
$$cpi_2 = \langle deepred, RobotAPI.driveTo, \langle pos, 10\rangle \ \rangle$$

4.5.2 Mapping the Control Interface

In the real environment, a controller employs its control interface to bring about desired effects in the environment. In the simulated environment, influences are used to reify the attempts to manipulate the environment. Consequently, when embedding controllers in the simulated environment, all invocations on a control interface should be mapped to the corresponding influences in the simulated environment.

We introduce modeling constructs to capture this mapping explicitly in the simulation model. An explicit mapping decouples the definition of a control interface from the representation of influences in the simulated environment, such that both control interface and influence representation can evolve independently.

In Section 4.5.2.1, we describe an explicit mapping that defines the *name* of the influence that results from a particular control primitive invocation. In Section 4.5.2.2, we describe an explicit mapping that defines the *parameters* of the influence that results from a particular control primitive invocation.

4.5.2.1 Control Name Mapping

We introduce a *control name mapping* as a modeling construct that specifies the name of the influence that results from a particular control primitive invocation. We define a control name mapping as a function $NameMap$:

$$NameMap : 2^{CPI^{\Omega}} \rightarrow InfNames$$
$$NameMap(cpi) = infname$$

$NameMap$ is a function that specifies a name $infname$ for a given control primitive invocation *cpi*.

An Example. We describe a control name mapping for control primitive invocations of two control primitives of the `ApproachBallBehavior` described in Section 4.3:

$$NameMap(cpi) = \begin{cases} startDriving \\ \quad \text{if } cpi|_{cp} = RobotAPI.driveTo \\ \\ stopdriving \\ \quad \text{if } cpi|_{cp} = RobotAPI.stop \end{cases}$$

The control name mapping $NameMap$ can be understood as follows. The expression contains two cases:

- The first case applies if the control primitive is $RobotAPI.driveTo$. The name of the corresponding influence is $startDriving$ (see Section 3.5.2).

- The second case applies if the control primitive is *RobotAPI.stop*. The name of the corresponding influence is *stopdriving*.

4.5.2.2 Control Parameter Mapping

We introduce a *control parameter mapping* as a modeling construct that specifies the parameters of the influence that results from a given control primitive invocation. We define a control parameter mapping as a function $ParMap$:

$$ParMap : 2^{CPI^{\Omega}} \rightarrow V_1 \times \ldots \times V_n$$
$$ParMap(cpi) = \langle v_1, \ldots, v_n \rangle$$

An Example. We describe a control parameter mapping for control primitive invocations of two control primitives of the `ApproachBallBehavior` of the example distributed control application described in Section 4.3:

$$ParMap(cpi) = \begin{cases} \langle VelocityVector(cpi|_{par}) \rangle \\ \quad \text{if } cpi|_{cp} = RobotAPI.driveTo \\ \\ \langle \rangle \\ \quad \text{if } cpi|_{cp} = RobotAPI.stop \end{cases}$$

The control parameter mapping $ParMap$ can be understood as follows. The expression contains two cases:

- The first case applies if the control primitive is *RobotAPI.driveTo*. The parameters of the control primitive invocation are a target position and an integer denoting the speed. However, the parameter of the corresponding *startDriving* influence is the velocity vector of the movement (see Section 3.5.2). Consequently, the parameters of the control primitive invocation need to be translated into the parameters of a *startDriving* influence. The function $VelocityVector$ returns a velocity vector for a given tuple containing a target position and speed. We make abstraction on the way the value of this vector is determined.

- The second case applies if the control primitive is *RobotAPI.stop*. No parameters are necessary for the corresponding influence.

4.6 Generating Influences

In this section, we elaborate on the way influences are generated by the various sources of dynamism.

In Section 4.6.1, we elaborate on influences that originate from the controllers of a distributed control application. In Section 4.6.2, we elaborate on influences that originate from the environment sources of dynamism.

4.6.1 Influences of Controllers

We describe the way influences originate from the controllers of a distributed control application. We first elaborate on the state of controllers in Section 4.6.1.1. Afterwards, we describe the way controllers generate influences in Section 4.6.1.2. Finally, we elaborate on the execution cycle of controllers in Section 4.6.1.3.

4.6.1.1 State of a Controller

A controller is an active software component. The state of a controller is a snapshot of the controller at a particular time during its execution. We consider the state of a controller each time it performs a control primitive invocation. We assume that a controller has a single thread of control.

The state of a controller comprises two parts. The first part describes all duration primitive invocations a controller executed so far. The second part comprises the current control primitive invocation. We define:

$s_{co_i} = \langle\{dpi_1, \ldots, dpi_r\}, cpi\rangle \in S_{co}^{\Omega}$	the state of a particular controller $co_i \in Co$, represented as a 2-tuple with the following characteristics: $\{dpi_1, \ldots, dpi_r\} \in 2^{DPI^{\Omega}}$ the set of duration primitive invocations already performed by controller co. $cpi \in CPI^{\Omega}$ the current control primitive invocation of controller co. S_{co}^{Ω} is the set of all possible states of the controllers.

We use the following shorthand notations: $s_{co_i}|_{DPI}$ to select the set of duration primitive invocations performed by controller co_i, $s_{co_i}|_{cpi}$ to select the current control primitive invocation of controller co_i.

An Example. As an example, consider the following state of controller *deepblue*:

$$\begin{aligned}
s_{deepblue} = \langle\{dpi_1, dpi_2, dpi_3\}, cpi_1\rangle \text{ with:}&\\
dpi_1 &= \langle deepblue, Thread.sleep, \langle 400\rangle\rangle\\
dpi_2 &= \langle deepblue, Thread.sleep, \langle 400\rangle\rangle\\
dpi_3 &= \langle deepblue, PlanLibrary.estimatePosition, \langle pos_1, pos_2, pos_3\rangle\ \rangle\\
cpi_1 &= \langle deepred, RobotAPI.driveTo, \langle pos, 10\rangle\ \rangle
\end{aligned}$$

4.6.1.2 Generating Influences

We describe the way influences are derived from the state of a controller. To specify the current influence of a controller, we define a function $CurCoInf$ that returns the current influence of a given controller in a given state:

$$\begin{aligned}
&CurCoInf : Co \times S_{co}^{\Omega} \rightarrow Inf^{\Omega}\\
&CurCoInf(co_i, s_{co_i}) = \langle co_i, t, name, par\rangle \text{ with:}\\
&\qquad t = 0 + ExTime(s_{co_i}|_{DPI})\\
&\qquad name = NameMap(s_{co_i}|_{cpi})\\
&\qquad par = ParMap(s_{co_i}|_{cpi})
\end{aligned}$$

The function $CurCoInf$ specifies the current influence given a particular controller co_i with state s_{co_i}. The resulting influence has the following characteristics:

- The source that performs the influence, is the given controller co_i.
- The time of occurrence of the influence is determined by the execution time of all duration primitives $s_{co_i}|_{DPI}$ that the controller performed so far.
- The name of the influence is determined by the control name mapping applied the current control primitive invocation $s_{co_i}|_{cpi}$.
- The parameters of the influence are determined by the control parameter mapping applied the current control primitive invocation $s_{co_i}|_{cpi}$.

An Example. We illustrate the $CurCoInf$ function on the example state of controller *deepblue* described in Section 4.6.1.1:

$$\begin{aligned}
&CurCoInf(deepblue, \langle\{dpi_1, dpi_2, dpi_3\}, cpi_1\rangle) =\\
&\qquad \langle deepblue, 0.923, startDriving, \vec{v}\rangle \text{ with:}\\
&\qquad\qquad 0.923 = 0 + ExTime(\{dpi_1, dpi_2, dpi_3\})\\
&\qquad\qquad\qquad = 0 + 400 + 400 + 123\\
&\qquad\qquad startDriving = NameMap(cpi_1)\\
&\qquad\qquad \vec{v} = ParMap(cpi_1)\\
&\qquad\qquad\quad = VelocityVector(cpi_1|_{par})
\end{aligned}$$

The current influence of controller *deepblue* is an influence with name $startDriving$ with parameter $\vec{v}$ that occurs at time $T = 0.923$.

4.6.1.3 Execution Cycle of a Controller

We describe the way controllers execute over time. An execution cycle of a controller comprises the execution of that controller from its current control primitive invocation until the next control primitive invocation.

We define a function *ExecCo* that determines the next state of a controller that is the result of executing that controller from its current state until its next control primitive invocation:

$$\begin{aligned}&ExecCo : Co^{\Omega} \times S^{\Omega}_{co} \rightarrow S^{\Omega}_{co}\\ &ExecCo(co_i, \langle DPI, cpi\rangle) = \langle\{dpi_1, \ldots, dpi_r\} \cup DPI, cpi'\rangle\end{aligned}$$

The function *ExecCo* returns a new state with the following characteristics:

- The set of duration primitive invocations of the new state is the union of the duration primitive invocations DPI of the old state and a set of newly performed duration primitive invocations $\{dpi_1, \ldots, dpi_r\}$.
- The current control primitive invocation of the new state is cpi' instead of cpi.

An Example. We illustrate the *ExecCo* function on the example state of controller *deepblue* described in Section 4.6.1.1:

$$\begin{aligned}&ExecCo : Co^{\Omega} \times S^{\Omega}_{co} \rightarrow S^{\Omega}_{co}\\ &ExecCo(deepblue, \langle\{dpi_1, dpi_2, dpi_3\}, cpi_1\rangle) =\\ &\quad\langle\{dpi_1, dpi_2, dpi_3, dpi_4, dpi_5, dpi_6, dpi_7\}, cpi_2\rangle \text{ with:}\\ &\qquad dpi_4 = \langle deepblue, Thread.sleep, \langle 1000\rangle\rangle\\ &\qquad dpi_5 = \langle deepblue, Thread.sleep, \langle 400\rangle\rangle\\ &\qquad dpi_6 = \langle deepblue, Thread.sleep, \langle 400\rangle\rangle\\ &\qquad dpi_7 = \langle deepblue, PlanLibrary.estimatePosition, \langle pos'_1, pos'_2, pos'_3\rangle\ \rangle\\ &\qquad cpi_2 = \langle deepred, RobotAPI.driveTo, \langle pos', 10\rangle\ \rangle\end{aligned}$$

The four new duration primitive invocations dpi_4, dpi_5, dpi_6 and dpi_7 and the new control primitive invocation cpi_2 are the result of executing one loop of controller *deepbleu* in Figure 4.3: the execution cycle starts from cpi_1 (i.e. the invocation of line 32), over dpi_4 (i.e. the invocation of line 38), dpi_5 (i.e. the invocation of line 20), dpi_6 (i.e. the invocation of line 22), dpi_7 (i.e. the invocation of line 29) until cpi_2 (i.e. the next time line 32 is invoked).

4.6.2 Influences of Environment Sources

We elaborate on influences that originate from environment sources. From the point of view of the modeling framework, each environment source is a black-box source of influences. Modeling the internal behavior of an environment source is outside the scope of this dissertation. The internal behavior of an environment is highly dependent upon the simulation study, and can range from a simple predetermined schedule of influences towards a complex cognitive model or even the code of another distributed control application that is embedded in the simulation.

We focus on capturing the influences that are generated by environment sources. We make abstraction of the internal behavior of an environment source that determines when to perform which influence. We start from a basic model of the state of an environment source in Section 4.6.2.1. Afterwards, we describe a mapping of the state of environment sources to influences in Section 4.6.2.2 and describe the evolution of an environment source in Section 4.6.2.3

4.6.2.1 State of an Environment Source

We limit ourselves to an abstract representation of the state of an environment source. In analogy with the state of a controller, the state of an environment source is a snapshot of the environment source at the moment it performs an influence. The state of an environment source is defined as:

$s_{es_i} \in S^{\Omega}_{es}$	the state of a particular environment source $es_i \in Es^{\Omega}$ S^{Ω}_{es} is the set of all possible states of environment sources.

4.6.2.2 Generating Influences

We describe a mapping between the state of environment sources and influences. To specify the current influence associated with an environment source that is in a given state, we define a function $CurEsInf$ that returns the current influence of a given environment source in a given state:

$$CurEsInf : Es \times S^{\Omega}_{es} \rightarrow Inf^{\Omega}$$
$$CurEsInf(es_i, s_{es_i}) = f$$

The function $CurEsInf$ specifies the current influence f of a particular environment source es_i with state s_{es_i}.

4.6.2.3 Evolution Cycle of an Environment Source

An evolution cycle of an environment source specifies a state snapshot of that environment source at the moment it generates its next influence. We define the $ExecEs$ function to advance an environment souce until it performs its next influence:

$$ExecEs : Es^{\Omega} \times S^{\Omega}_{es} \rightarrow S^{\Omega}_{es}$$
$$ExecEs(es_i, s_{es_i}) = s'_{es_i}$$

The function $ExecEs$ determines the next state s'_{es} for an environment source es_i with given state s_{es_i}.

4.7 The Evolution of the Model Revisited

We revisit the evolution of the model that was described in Section 3.7. The evolution of the model relies on the functions $NextInfs$ and $SoCycle$ that were defined only in an abstract manner, see Section 3.5.2.2.

Here, we formally specify these functions. This integrates the evolution of the model with the way influences are generated by executing the controller software and by the environment sources.

Recall from Section 3.5.2.2 that the function $NextInfs$ should determine the set of influences with the earliest time of occurrence of all influences performed by a set of sources of dynamism in a given state. The function $SoCycle$ should specify a new state for the sources of dynamism, based on their current state. The new state is the result of evolving the current state of the sources of dynamism until the earliest time instant one or several of the sources perform a new influence.

To define $NextInfs$ and $SoCycle$, we integrate the way controllers (Section 4.6.1) and environment sources (Section 4.6.2) generate influences and evolve their state.

In Section 3.5.1, we defined:

$So = Co \cup Es = \{so_1, \ldots, so_r\}$	the set of sources of dynamism in the environment.

4.7.1 State of Sources of Dynamism

We generalize the state of controllers (Section 4.6.1.1) and the state of environment sources (Section 4.6.2.1) into the state of sources of dynamism in general. The state of a source of dynamism is a snapshot of that source at the moment it performs an influence. We define:

$s_{so_i} \in S^{\Omega}_{so} = S^{\Omega}_{co} \cup S^{\Omega}_{es}$	the state of a particular source so_i. S^{Ω}_{so} is the set of all possible states of all sources of dynamism.
$S_{so} = \{s_{so_1}, \ldots, s_{so_r}\} \in 2^{S^{\Omega}_{so}}$	the state of a set of sources of dynamism. $2^{S^{\Omega}_{so}}$ is the powerset, i.e. the set of all subsets of S^{Ω}_{so}.

We use the shorthand notation $S_{so}|_{so_i}$ to select the state of source so_i. Hence $S_{so}|_{so_i} = s_{so_i}$.

Note that the state description is identical to the one given in Section 3.5.1.1.

4.7.2 Generating Influences: the $NextInfs$ Function Revisited

To determine the current influence of a source of dynamism with given state, we define a function $CurInf$ that integrates the functions $CurCoInf$ for controllers (Section 4.6.1.2) and $CurEsInf$ for environment sources (Section 4.6.2.2):

$$CurInf : So^{\Omega} \times S_{so}^{\Omega} \to Inf^{\Omega}$$
$$CurInf(so_i, s_{so_i}) = \begin{cases} CurCoInf(so_i, s_{so_i}) \\ \quad \text{if } so_i \in Co^{\Omega} \\ \\ CurEsInf(so_i, s_{so_i}) \\ \quad \text{if } so_i \in Es^{\Omega} \end{cases}$$

In case the source so_i is a controller, the function $CurInf$ returns the current influence as defined by $CurCoInf$. In case the source so_i is an environment source, the function $CurInf$ returns the current influence as defined by $CurEsInf$.

We define a function $AllInfs$ that determines the set of all current influences of a given set of sources of dynamism and a given set of states of these sources:

$$AllInfs : 2^{So^{\Omega}} \times 2^{S_{so}^{\Omega}} \to 2^{Inf^{\Omega}}$$
$$AllInfs(So, S_{so}) =$$
$$\{f \in Inf^{\Omega} \mid \exists so_i \in So : f = CurInf(so_i, S_{so}|_{so_i})\}$$

We now define the function $NextInfs$ of Section 3.5.2.2. The function $NextInfs$ determines the set of influences with the earliest time of occurrence of all influences performed by a set of sources of dynamism in a given state. We define:

$$NextInfs : 2^{So^{\Omega}} \times 2^{S_{so}^{\Omega}} \to 2^{Inf^{\Omega}}$$
$$NextInfs(So, S_{so}) =$$
$$\{f \in AllInfs(So, S_{so}) \mid \forall f_i \in AllInfs(So, S_{so}) : f|_t \leqslant f_i|_t)\ \}$$

For a given set of sources of dynamism So and a given set of states S_{so} of these sources, $NextInfs$ returns a set that contains all influences f of the set of all current influences $AllInfs(So, S_{so})$ for which the following holds: the time of occurrence $f|_t$ of the influences f must be smaller than or equal to the time of occurrence $f_i|_t$ of each influence f_i within the set of all current influences.

4.7.3 Evolution Cycle of Sources of Dynamism: the $SoCycle$ Function Revisited

To determine a new state for a source of dynamism, we define a function $ExecSo$ that integrates the functions $ExecCo$ for controllers (Section 4.6.1.3) and $ExecEs$

for environment sources (Section 4.6.2.3). We define the *ExecSo* function to execute a source until it performs its next influence:

$$ExecSo : So^{\Omega} \times S_{so}^{\Omega} \rightarrow S_{so}^{\Omega}$$

$$ExecSo(so_i, s_{so_i}) = \begin{cases} ExecCo(so_i, s_{so_i}) \\ \quad \text{if } so_i \in Co^{\Omega} \\ \\ ExecEs(so_i, s_{so_i}) \\ \quad \text{if } so_i \in Es^{\Omega} \end{cases}$$

In case the source so_i is a controller, the function *ExecSo* returns the next state as defined by *ExecCo*. In case the source so_i is an environment source, the function *ExecSo* returns the next state as defined by *ExecEs*.

We now define the function *SoCycle* of Section 3.5.2.2. The function *SoCycle* returns a set of new states for the sources of dynamism, based on their current states. In this set of new states, only the state of those sources is updated whose current influence is in the set of earliest occurring influences of all sources:

$$SoCycle : 2^{So^{\Omega}} \times 2^{S_{so}^{\Omega}} \rightarrow 2^{S_{so}^{\Omega}}$$
$$SoCycle(So, S_{so}) =$$
$$\{s' \in S_{so}^{\Omega} \mid \exists so_i \in So : s' = S_{so}|_{so_i} \wedge CurInf(so_i, S_{so}|_{so_i}) \notin NextInfs(So, S_{so}))$$
$$\vee$$
$$\exists so_i \in So : (s' = ExecSo(so_i, S_{so}|_{so_i})) \wedge$$
$$(CurInf(so_i, S_{so}|_{so_i}) \in NextInfs(So, S_{so}))\ \}$$

The *SoCycle* function can be understood as follows. Given a set of sources of dynamism *So* and a set S_{so} of current states for these sources, the *SoCycle* function determines a set of new states for these sources. The new states s' in the result set are defined as follows (we explain the two cases separated by the logical $\vee$):

- The first case states that the new state s' of a source is equal to the old state of that source, i.e. $s' = S_{so}|_{so_i}$. This is the case if the current influence of that source is not part of the set $NextInfs(So, S_{so})$, i.e. the set of influences with the earliest time of occurrence of all influences of the set of sources *So* in states S_{so}.

- The second case states that the new state s' of a source is updated to the outcome of *ExecSo* on the current state of that source, i.e. $s' = ExecSo(so_i, S_{so}|_{so_i})$. This is the case if the current influence of that source is part of the set $NextInfs(So, S_{so})$, i.e. the set of influences with the earliest time of occurrence of all influences of the set of sources *So* in states S_{so}.

4.8 Related Work

Simulation platforms use various approaches to integrate a control application. We focus on the way the simulation platforms support the execution time of a control application. We make a distinction between approaches that incorporate execution time based on direct measurement and approaches that rely on a specification of execution time.

4.8.1 Measurement of Execution Time

A first group of approaches rely on a direct measurement of the execution time during a simulation run. Examples include:

- Player/Stage [GVH03] supports software-in-the-loop simulations in which the execution time in taken into account implicitly. The controllers of the distributed control application run on remote hosts and interact with the simulated environment over a network connection. The simulation proceeds in real-time. The execution time is taken into account implicitly: the timing of the actions of the controllers is determined by their arrival time at the host that manages simulated environment. This means that the execution time of a controller is influenced by the performance of the remote host on which the controller is deployed, but also by the latency of the computer network.

- DGensim [And00] supports software-in-the-loop simulations in which wall clock time stamps are used to measure the execution time. Each controller runs on a remote host and interacts with the simulated environment over a network connection. At fixed time intervals, perceptions are given to the controllers, and a controller has a fixed window of time to react to the perception. Before transmitting the actions to the simulated environment, the remote host attaches a time-stamp with the wall clock time of each action. At the host of the simulated environment, all actions within a time window are arranged according to their time stamp in wall clock time. The use of wall clock time stamps reduces the effect of network latencies on the ordering of actions. However, problems arise in case network latencies cause actions do not reach the simulated environment within the time window.

- SPADES [RR03] supports software-in-the-loop simulations with a direct measurement of execution time. Each controller of the distributed control application runs on a dedicated host together with a SPADES communication server, which sends the actions of that controller to the simulated environment. The SPADES communication server supports low-level performance monitoring by means of *perfctr*, a linux kernel driver that offers low-level performance monitoring with per-process CPU-cycle counters. The controller operates in a sense-think-act cycle, and notifies the SPADES communication server of the start and end of each cycle. The simulation time of the actions corresponds to applying an linear scale factor to the performance measurement of the *perfctr* driver.

Measurement is an easy and intuitive way to incorporate the execution time of controllers of a distributed control application in a simulation. Nevertheless, compared to an explicit model of the execution time, measuring the execution time during a simulation has a number of drawbacks [And97]:

- *Measurements are platform dependent.* The computer platform on which a distributed control application is deployed for simulation purposes typically differs from the (heterogeneous) devices on which the controllers are deployed in the real world. Consequently, the measurement of the execution time of a controller is not necessarily a decent estimate of the execution time of that controller in the real world.

- *Measurements are not selective.* A measurement takes into account auxiliary code for debugging, logging to file, configuration, interfacing with the user, although this auxiliary code is removed from the distributed control application before it is deployed in the real environment. Auxiliary code can significantly affect the execution time that is measured of a particular controller.

- *Measurements jeopardize repeatable simulation runs.* The measurements that are employed are non-deterministic, i.e. small random variations are possible when measuring the execution time. In simulation, non-determinism must always be supported in a controlled manner, i.e. in a simulation all non-determinism should be based on random numbers originating from a random number generator with a known seed. Using the same seed for the random number generator then guarantees the same trace of random numbers during a simulation run, which is a prerequisite to obtain simulation results that can be repeated over and over again. However, measuring the execution time of a controller during a simulation is an example of supporting non-determinism in an uncontrolled manner. As the trace of measurements of the execution time during a simulation run cannot be controlled, it can be extremely difficult or even impossible to reproduce the same simulation result twice.

4.8.2 Specification of Execution Time

A second group of approaches specify the execution time of a distributed control application instead of using measurements. Examples include:

- MESS [AC96] supports software-in-the-loop simulation of controllers written in the Common Lisp programming language. To model the execution time of a controller, individual language instructs of Common Lisp are associated with a particular duration. MESS relies on TCL (Timed Common Lisp) to derive the execution time of a controller. TCL is an extended version of Common Lisp that advances a clock upon execution of each Common Lisp primitive. The duration for each primitive can be specified by the modeler. Auxiliary code can be annotated such that its duration is not taken into account.

- EyeSim [BKW06] supports software-in-the-loop simulations of controllers for robotic systems based on the RoBIOS, a list of library functions for motor control, sensor feedback and multi-tasking. To incorporate the execution time of a controller, EyeSim employs a duration for each of the RoBIOS system calls. The duration of all code besides the function calls to the RoBIOS library is disregarded.

- The Packet-World [WHH05] employs a very coarse-grained model to specify the execution time of a controller. Each controller has a fixed, constant execution time between consecutive actions, irrespective of the amount of computation it needs to determine its next action. This is a suitable model in case the execution time of a controller in the real world does not vary a lot, or in case only a rough estimate is sufficient.

The examples illustrate that the constructs of the modeling framework introduced in this chapter are underpinned by existing approaches. Existing simulation platforms incorporate the execution time based on different approaches at different levels of abstraction.

The added value of the modeling framework is that it puts forward explicit modeling constructs for capturing the execution time. The formal specification of the modeling framework decouples the modeling constructs from their implementation in a particular simulation platform. As such, the execution time can be specified in a simulation model without taking into account a specific simulation platform.

4.9 Conclusions

In this chapter, we presented a modeling framework that contains formally specified modeling constructs that are specifically aimed at integrating the software of a distributed control application in a simulation.

The modeling framework presents and supports in an explicit manner a number of challenges that are pertinent when embedding the software of real controllers of a distributed control application in a simulation:

- The real-world execution time of the control software should be represented explicitly in the simulation model. *Duration primitives* specify segments of code of which the execution time is relevant for the simulation. A *duration mapping* specifies a duration for each invocation of a duration primitive.

- The control interface of the control software should be represented explicitly in the simulation model. *Control primitives* specify the control interface that each controller uses to access the environment. A *control name mapping* and *control parameter mapping* specify the influences in the environment that result from invoking a control primitive.

We illustrated the modeling framework for describing the execution time and the control interface of a distributed control application for controlling RoboCup Soccer robots.

For software-in-the-loop simulations of distributed control applications in dynamic environments, the contribution of the modeling framework is twofold. On the one hand, the modeling framework provides explicit support for *model formulation* by offering formally specified constructs for representing the way the software of a real distributed control application is integrated in a simulation. The formal specification enables a modeler to specify how the software is integrated in a simulation, irrespective of the design and implementation of a particular simulation platform. On the other hand, the modeling framework provides explicit support for *model translation* by specifying the functionality that is needed to support the constructs in an executable simulation.

Chapter 5

Architecture of the Simulation Platform

In this chapter, we describe the architecture of a simulation platform that supports the modeling constructs of the modeling framework described in Chapters 3 and 4. The simulation platform can be used to execute simulation models that are described in terms of these modeling constructs. The simulation platform shows the feasibility of the modeling framework for simulating distributed control applications in a dynamic environment.

5.1 Introduction

We motivate the simulation platform from both a research perspective and a simulation developer's perspective. From a research perspective, the simulation platform illustrates the feasibility of the modeling framework to support simulations of distributed control applications in dynamic environments. Furthermore, the simulation platform incorporates state-of-the-art software engineering technology to address a number of difficult engineering challenges that are paramount when building simulations for distributed control applications. An example is the use of aspect-oriented programming technology to modularize simulation concerns that crosscut with the control application. From a simulation developer's perspective, the simulation platform facilitates building executable simulations. The simulation platform encapsulates the functionality to support the constructs of the modeling framework in an executable simulation. The simulation platform can be reused for different simulation models insofar as they are formulated in terms of the constructs of the modeling framework. The simulation platform prevents developers from reinventing the functionality to support the modeling framework from scratch for each simulation study. The simulation platform raises the abstraction level for building executable simulations.

We describe the architecture of a simulation platform that supports the modeling framework. The software architecture of a system is defined as “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” [BCK03]. The software architecture of a system realizes the functionality of the system in a way that its quality requirements are satisfied.

We use several architectural views to document the architecture of the simulation platform. A view is a representation of a coherent set of architectural elements and the relations among them [BCK03]. Each view presents a particular perspective on the architecture or a part thereof. The documentation of the architecture of the simulation platform includes a module decomposition view and several component and connector views. For each view, we start with a general explanation of the goal of the view and the software elements and relations between elements that are considered in that view. Afterwards, we document each view for the simulation platform using a graphical notation and we explain how important quality requirements are realized.

This chapter is structured as follows. In Section 5.2, we put forward the functional and quality requirements of the simulation platform. In the following sections, we elaborate on the different architectural views. We start with a top-level module decomposition view in Section 5.3. We describe a component and connector view of the functionality to support dynamic environments in Section 5.4. We elaborate on a component and connector to explain the simulation engine that synchronizes all parts of the simulation in Section 5.5. In Section 5.6, we put forward an aspect-oriented approach to integrate the control software in the simulation. We describe a component and connector view of the functionality that keeps track of the execution time of a controller in Section 5.7. Finally, we summarize and draw conclusions in Section 5.8.

5.2 Requirements

The goal of the simulation platform is to provide run-time support for software-in-the-loop simulations of distributed control applications in dynamic environments, of which the simulation model is described in terms of the modeling constructs proposed in Chapters 3 and 4. We discuss the functional and quality requirements that are the main drivers for the architecture of the simulation platform.

The main functional requirements for the simulation platform are the following:

- *Support the modeling constructs for dynamic environments.* The simulation platform should encapsulate the functionality to support the modeling constructs for dynamic environments described in Chapter 3. This functionality includes (1) managing the sources of dynamism and the influences that result from their execution (2) applying the appropriate reaction laws to determine the reaction of the environment to the various influences (3) handling all activities in the environment during a simulation run, and (4) applying the interaction laws to enforce interactions between activities.

- *Support the modeling constructs for the control software.* The simulation platform should encapsulate the functionality to support the modeling constructs for embedding the software of real controllers, described in Chapter 4. This functionality includes (1) keeping track of the duration primitives invoked by each of the controllers (2) keeping track of the control primitives invoked of each of the controllers, (3) deriving the nature and the timing of the influences that result from executing the controllers.

- *Support consistent simulation runs.* The simulation platform should encapsulate the functionality to carry out simulation runs that are consistent with the described simulation model. The simulation model specifies the causal relations between all influences, activities, reaction and interaction laws by means of simulation time, e.g. by means of specifying the duration of the various controllers in simulation time, specifying the start and duration of each activity in simulation time, etc. To obtain causal relations in accordance to the specification of the simulation model, the progress of all parts of the simulation, i.e. the progress of the various controllers and environment sources of dynamism and of applying the various reaction and interaction laws, should happen in the order of increasing simulation time. Given the unpredictable delays introduced by the underlying execution platform on which the simulation platform runs, an explicit synchronization between all parts of the simulation is necessary to regulate their relative progress.

We describe the main quality requirements of the simulation platform:

- *Flexibility of embedding the software of the control application.* The simulation platform should provide support for embedding the software of the control application in a flexible way, i.e. with minimal effort from the developer. The fact that some simulation concerns crosscut with the control application's functionality hampers embedding the real controllers in a flexible way. We rely on state-of-the-art software engineering technology to modularize crosscutting simulation concerns in order to insert and remove them in the control application in a plug-and-play manner.

- *Modifiability of the simulation platform.* Modifying core parts of the simulation platform should be relatively easy, and the impact of such modifications should be as local as possible. Core parts of the simulation platform include the simulation engine and the functionality to support simulated environment.

- *Performance of the simulation platform.* The simulation platform should support *as-fast-as-possible simulation*, to enable executing simulation runs faster than real time. Simulation platforms that support software-in-the-loop simulations are typically limited to *real-time simulation*, i.e. simulation time advances in pace with wallclock time during a simulation run (see Section 2.1).

5.3 Top-Level Module Decomposition View of the Simulation Platform

The goal of a module decomposition view is to show how the simulation platform is decomposed into manageable software implementation units. A module decomposition view is a static view on a system's architecture. The elements depicted in a module decomposition view are *modules*. A module is an implementation unit of software that provides a coherent unit of functionality. The relationship between the modules is is-part-of that defines a part/whole relationship between a submodule and the aggregate module. Modules are recursively refined, revealing more details in each decomposition step. The basic criteria for module decomposition is the achievement of quality requirements. For example, parts of a system that are likely to change, are encapsulated in separate modules to support modifiability. Another example is the separation of functionality of a system that has higher performance requirements from other functionality.

The module decomposition view includes a description of the interfaces of each module that documents how a module is used in combination with other modules. The interface description distinguishes between provided and required interfaces. A provided interface specifies what functionality the module offers to other modules. A required interface specifies what functionality the module needs from other modules; it defines constraints of a module in terms of the services a module requires to provide its functionality.

The top-level module decomposition view of the simulation platform is depicted in Figure 5.1. We first discuss the main elements and their properties. Afterwards, we describe their interfaces and explain how important qualities are realized.

5.3.1 Elements and Their Properties

The simulation system is decomposed in two main subsystems: Controller and Simulation Platform.

- `Controller` is a module that of the real distributed control application that is embedded in the simulation platform in order to test or configure it. A distributed control application consists of several controllers working in parallel and cooperating to solve a problem in the environment. A controller has

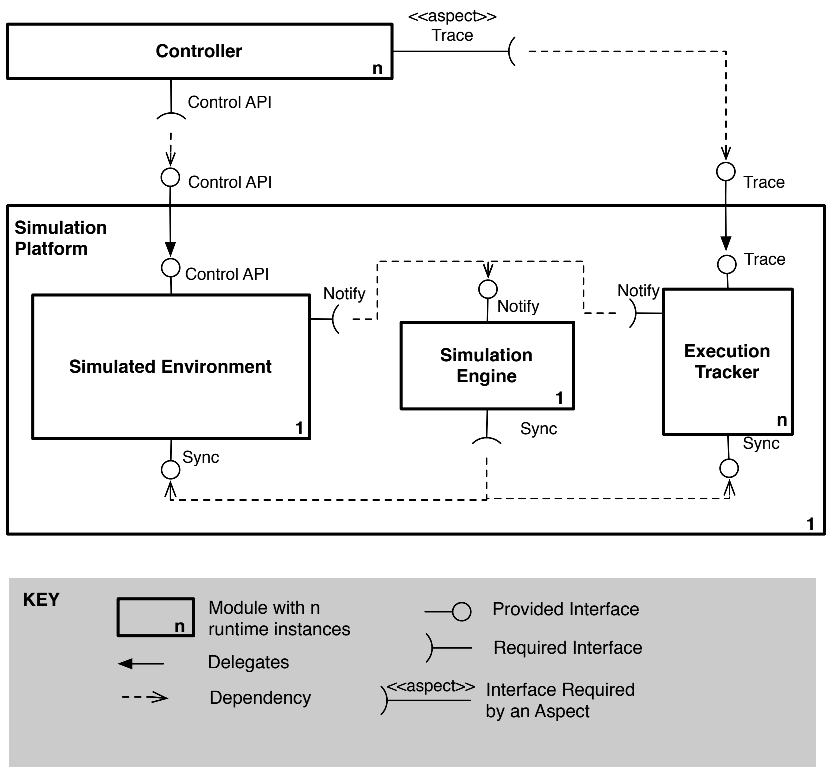


Figure 5.1: Top-level module decomposition view of the simulation platform

well-defined ways to sense the environment and to act upon it. An example of a controller is a program to controls a particular robot in a manufacturing plant.

- `Simulation Platform` is the medium in which controllers of a distributed control application are embedded in order to test or configure them. The main responsibilities of the simulation platform are:
  - To simulate the real dynamic environment of the control application.
  - To manage the execution of all controllers of the control application according to the specified duration model.
  - To execute simulation runs as-fast-as-possible, thus enabling simulations faster than real time.

The simulation platform is further decomposed in three different modules: Simulated Environment, Simulation Engine and Execution Tracker.

- `Simulated Environment` is responsible for managing a simulation model of the real environment of the distributed control application. The Simulated Environment encapsulates all functionality to support the modeling constructs described in Chapter 3.
- `Simulation Engine` is responsible for managing the evolution of all parts of the simulation in correspondence to the specifications of the simulation model. The simulation engine encapsulates all functionality to synchronize the progress of the simulated environment with the progress of all execution trackers of the controllers that are embedded in the simulation. This guarantees correct causal relations in correspondence to the specifications of the simulation model.
- `Execution Tracker` is responsible for tracing the execution of a particular controller of the distributed control application. This module encapsulates all functionality to support the modeling constructs for the control software described in Chapter 4. Tracing the execution of a controller includes (1) determining the execution time consumed by a particular controller of the distributed control application according to the duration mapping, and (2) synchronizing the execution of that controller with the simulation engine, which is necessary to enable as-fast-as-possible simulations. At runtime, there is an instance of the execution tracker module for each controller.

5.3.2 Interface Descriptions

The simulation platform module provides two interfaces to the controller: `Control API` and `Trace`.

- `Control API` supports the *application* concerns of a controller. `Control API` is the control interface required by the controller to interact with its environment. The `Control API` provided by the simulation platform is identical to the control interface the controller uses to interact with its sensors and actuators in the real environment. By providing the `Control API` interface, the simulation platform cannot be distinguished from the real environment from point of view of a controller.

- `Trace` supports the *simulation* concerns for a controller. `Trace` is the interface provided by the simulation platform to manage the execution of a controller. The `Trace` interface enables (1) monitoring the execution time consumed by the controller and (2) intercepting and synchronizing the execution of the controller with the simulation engine. The `Trace` interface is further explained in Section 5.6.

The simulation platform module delegates the `Control API` interface to the simulated environment module, and the `Trace` interface to the execution tracker module.

The simulation engine governs the progress of the simulation by means of the provided `Notify` and required `Sync` interfaces. We elaborate on Notify and Sync in Section 5.4.

5.3.3 Design Rationale

Each module in the decomposition encapsulates a particular functionality of the simulation platform. By minimizing the overlap of functionality among modules, the architect can focus on one particular part of functionality. Allocating different functionalities of the simulation platform to separate modules results in a clear design. It helps to accommodate change and to update one module without affecting the others, and it supports reusability. We elaborate on the core architectural decisions.

Low coupling between Controller and Simulation Platform. As we are concerned with software-in-the-loop simulations, one of the main architectural decisions is a low coupling between the software application on the one hand, i.e. the controllers, and the simulation platform in which it is embedded on the other hand. The `Control API` interface enables all communication, sensing and acting to be directed to the simulation platform transparently. The `Trace` interface connects the controller with a dedicated Execution Tracker in the Simulation Platform. Section 5.6 illustrates an aspect-oriented approach to provide existing controllers with support for the `Trace` interface.

Two advantages of a low coupling between Controller and Simulation platform are (1) reuse, i.e. the simulation platform can be reused for testing various controllers, and (2) modifiability, i.e. the controllers can be modified without affecting the simulation platform.

Low coupling between Simulated Environment and Simulation Engine. In the simulation platform, we make an explicit distinction between the simulated environment on the one hand, and the simulation engine on the other hand. The simulated environment maintains a model of the real environment. The simulation engine manages the simulation main loop, i.e. advancing simulation time by synchronizing the progress of all parts of the simulation. Simulated Environment and Simulation Engine are coupled by means of well-defined interfaces, i.e. `Notify` and `Sync`. This enables (1) the Simulated Environment to make abstraction of how and with whom synchronization is required, and (2) the Simulation Engine to focus on reliable and efficient synchronization, without knowledge of the internal working of each party that needs synchronization.

Two advantages of the low coupling between Simulated Environment and Simulation Engine are (1) reuse, i.e. it facilitates the integration of a different simulation engine into the simulation platform, and (2) manageability, i.e. the design of the Simulated Environment is facilitated because abstraction can be made of all synchronization issues.

Explicit support for as-fast-as possible simulations. In as-fast-as possible simulations, there is no fixed relation between the progress of the simulation engine and wallclock time. This enables simulation runs faster than real time. To support as-fast-as-possible simulation, the execution of each controller must be synchronized explicitly with the simulation engine in the simulation platform. Execution Trackers and the Trace interface encapsulate the functionality to Trace and synchronize the execution of the controllers with the simulation engine in an explicit manner.

5.4 Component and Connector View of the Simulated Environment

A component and connector view [CBB+02, ICG+04] shows a system as a set of cooperating units of execution. A component and connector view is a run-time view on a system's architecture. The elements of the component and connector view are run-time elements of computation and data storage, such as repositories and components. Components are run-time instances that perform calculations that typically require data from one or more data repositories. Data repositories store data and mediate the interactions among components. A data repository can provide a trigger mechanism to signal data consumers of the arrival of interesting data. Besides reading and writing data, a data repository may provide additional support, such as support for concurrency and persistency. The relationship between elements within a component and connector view are connectors. A connector is a path for communication that links connecting ports on two or more elements. A port is an interaction point on a run-time element through which data is sent and received according to a specific interface. A port is similar to an interface in that it describes how an element interacts with its environment, but is different in that each port is a distinct interaction point of its element [ICG+04].

The component and connector view of the Simulated Environment is depicted in Figure 5.2. This view gives a detailed perspective on the Simulated Environment module of Figure 5.1. The Simulated Environment supports the modeling constructs for dynamic environments described in Chapter 3. We first discuss the main elements and their properties. Afterwards, we describe how they are connected and explain how important qualities are realized.

5.4.1 Elements and Their Properties

The Simulated Environment contains various components that are connected to five possible repositories: State, Activities, Influences, Reaction Laws and Interaction Laws. We elaborate on each of the five repositories. Afterwards, we describe the components they are connected to.

- `State` repository contains values for all variables to describe the state of the environment. The values of the state describe a snapshot of the environment at a particular instant of simulation time, i.e. the snapshot time. The state of the environment includes the state of all environmental entities and properties of the environment. Examples are the position and battery level of each robot in the environment, the position of various objects in the environment, the temperature of the environment.

- `Activity` repository maintains the activities as first-class elements. Activities describe the evolution of the state of the environment over time. Activities are always expressed relative to the snapshot of the state stored in the States repository. Examples of activities are an activity that describes the driving of a robot and an activity describing the rolling of a ball in a RoboCup Soccer environment.

- `Influence` repository contains the influences as first-class elements. Influences are attempts to start, stop or alter activities. Influences originate from the controllers of the distributed control application on the one hand, and from environment sources external to the distributed control application on the other hand.

- `Reaction Law` repository maintains the reaction laws of the environment model as first-class elements. The reaction laws determine the way influences have an impact on the activities in the simulation.

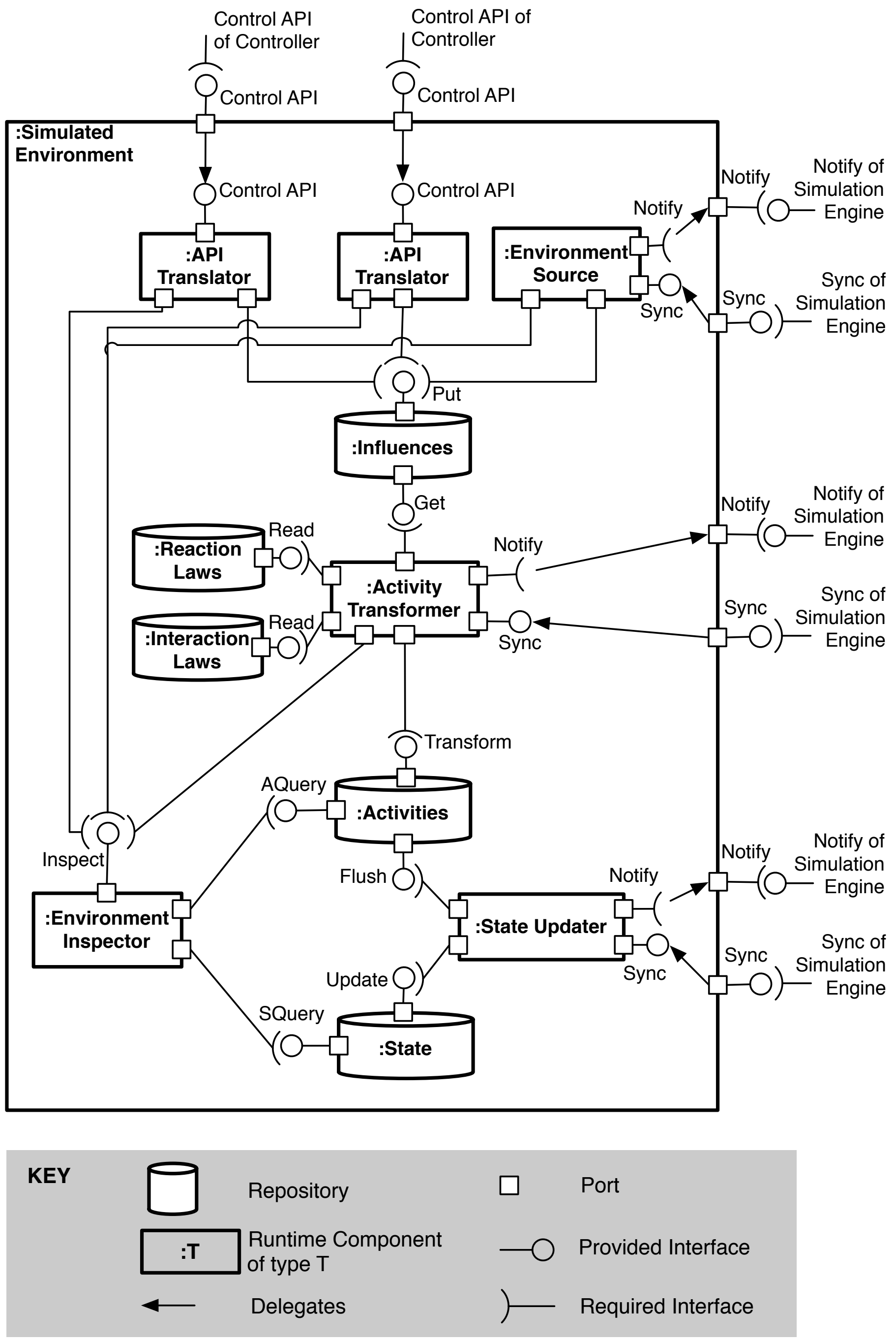


Figure 5.2: Component and connector view of the simulated environment

- `Interaction Law` repository maintains the interaction laws of the environment model as first-class elements. The interaction laws determine the way activities may interact in the environment.

The components are runtime instances of corresponding modules within the Simulated Environment:

- `Environment Inspector` acts as the facade that regulates all inspections of both state and dynamics of the environment. This includes supporting the State function (described in Section 3.4.3) to retrieve the state of a part of the environment at any particular point in simulation time, based on the actual content of the `State` repository and `Activity` repository.

- `State Updater` prevents activities from piling up in the Activity repository during a simulation run. The `State Updater` periodically flushes activities from the `Activity` repository and updates the corresponding values in the `State` repository such that they represent the state at a later snapshot time.

- `Activity Transformer` is responsible for applying all laws present in the `Reaction Law` repository and `Interaction Law` repository. The laws are black-box elements for the `Activity Transformer`, which only orchestrates applying all laws. Applying the laws includes (1) checking whether laws are applicable and (2) manipulating the activities in the `Activity` repository in correspondence to the applicable laws. To check whether laws are applicable, the `Activity Transformer` verifies for each law whether its conditions are satisfied. For interaction laws, this involves contacting the `Environment Inspector`; for reaction laws, this this also involves contacting the `Influence` repository besides the `Environment Inspector`. To apply a reaction law, the `Activity Transformer` removes the respective influences from the `Influence` repository, and performs the activity transformation proposed by the reaction law on the activities in the `Activity` repository. To apply an interaction law, the `Activity Transformer` performs the activity transformation proposed by the interaction law on the activities in the `Activity` repository.

- `API Translator` is responsible for translating a controller's invocations on the `Control API` interface into the concepts of the environment model. More specifically, the `API Translator` maps all triggering of actuators (e.g. (de-)activating motors or sending communication messages) into influences that are stored in the `Influence` repository, according to the control parameter mapping (see Section 4.5.2.2) and control name mapping (see Section 4.5.2.1). The `API Translator` realizes all triggering of sensors (e.g. readout of sensor values or received communication messages) by querying the `Environment Inspector`. In Figure 5.2, two `API translators` are depicted. Each `API translator` is connected to a controller of the distributed control application.

- `Environment Source` is responsible to mimic the behavior of a source of dynamism in the environment that is external to the distributed control application. `Environment Sources` are capable of performing influences and sensing the environment. Examples of `Environment Sources` are other machines or humans that reside in the environment of the distributed control application. In Figure 5.2, one instance of an `Environment Source` is depicted.

5.4.2 Interface Descriptions

Figure 5.2 depicts the interconnections between the repositories and the internal components of the simulated environment.

The State repository provides two interfaces:

- `SQuery` is the interface provided by the State repository to read the current values of the variables.
- `Update` is the interface provided by the State repository to enable updating the state to a new snapshot time.

The Activity repository provides three interfaces:

- `AQuery` is the interface for inspecting activities. Inspection is based on matching: the requester specifies a condition that must hold for all activities that are returned.
- `Flush` is the interface to (partially) empty the Activity repository. The requester specifies a point in simulation time. `Flush` returns all activities that finish before the specified time instant. In contrast to the `AQuery` interface, the activities returned by the `Flush` interface are removed from the Activity repository.
- `Transform` is the interface to manipulate the activities in the Activity repository. The requester specifies an activity transformation to be performed on the activities. This corresponds to the *ApplyTrans* function described in Section 3.5.3.1.

The Reaction Law repository and Interaction Law repository provide one interface:

- `Read` is the interface that can be used to access the laws in the corresponding repository.

The Influence repository provides two interfaces:

- `Put` is the interface for storing new influences in the Influence repository.
- `Get` is the interface for returning influences out of the Influence repository. The requester can specify (1) a condition that must be satisfied by each influence that is returned, and (2) whether the returned influences should be removed from the Influence repository.

The Environment Inspector provides the `Inspect` interface that assembles a snapshot of the state of the environment at any particular instant of simulation time. The `Notify` and `Sync` interfaces are described in Section 5.5, when the simulation engine is discussed.

5.4.3 Design Rationale

Low coupling due to data repositories. The use of data repositories decouples the various components within the simulated environment. Low coupling improves modifiability (as the changes in one element do not affect other elements), and reuse (as elements are not dependent on too many other elements). Decoupled elements do not require detailed knowledge about the internal structures or operations of other elements. Furthermore, decoupled elements are easier to understand due to clear and coherent responsibilities.

For example, the `Influence` repository gathers all influences, regardless of whether these influences originate from controller actions that are translated by `API translators`, or from `Environment Sources` external to the control application. As such, the `Influence` repository decouples the `Activity Transformer` from the various sources of influences, i.e. `API Translators` and various `Environment Sources`.

Decoupling synchronization from the Simulated Environment. The elements that need synchronization are `Environment Source`, `Activity Transformer` and `State Updater`. All synchronization is delegated to the simulation engine by means of the `Notify` and `Sync` interfaces. Synchronization will be discussed in Section 5.5.

Customizable presentation of the environment state. The `Environment Inspector` acts as a facade to hide the internal representation of the environment state in terms of state and activities. As such, the internal representation is decoupled from the way the state is presented towards other components, such as the `API Translators`, `Environment Sources` and the various laws managed by the `Activity Transformer`. The `Inspect` interface enables the use of a custom representation for each component it is connected to.

Customizable state updating strategy. The strategy to update the state is encapsulated in the `State Updater`. This offers the developer the ability to apply a custom updating strategy. For example, in case the execution trace should be logged, the `State Updater` can be deactivated easily, such that all activities are aggregated in the Activity repository and can be inspected afterwards.

Reusable Infrastructure. Finally, we emphasize the reusability of the architecture of the simulated environment. The internal components and repositories of the simulated environment comprise the infrastructure necessary to handle influences, activities, reaction laws and interaction laws. This infrastructure can be reused for all simulation studies whose simulation models are described in terms of these constructs.

5.5 Component and Connector View of the Simulation Engine

The Simulation Engine is responsible for advancing simulation time by synchronizing all parts of the simulation such that everything happens in the order of increasing simulation time. This guarantees correct causal relations in correspondence with the specifications of the simulation model.

We focus on the way the Simulation Engine regulates the progress of all parts of the simulation. The component and connector view of the Simulation Engine is depicted in Figure 5.3. We first discuss the main elements and their properties. Afterwards, we describe how they are connected and explain how important qualities are realized.

5.5.1 Elements and Their Properties

The `Simulation Engine` is responsible for executing a simulation run by synchronizing the progress of various components. Synchronization is necessary to ensure causality, i.e. to enforce that everything happens in the order of increasing simulation time. The components the need synchronization are the following: the various sources of dynamism that act in parallel, i.e. the `Controllers` of the distributed control application and the `Environment Sources` external to the distributed control application, the `Activity Transformer` that applies the reaction and interaction laws and the `State Updater` that updates the state to a new snapshot time.

We discuss each component, explain why it needs synchronization and the way it relies on the `Simulation Engine` for synchronization.

- `Environment Source` (see Section 5.4.1). An `Environment Source` can access the environment in several ways, i.e. by performing an influence or sens-

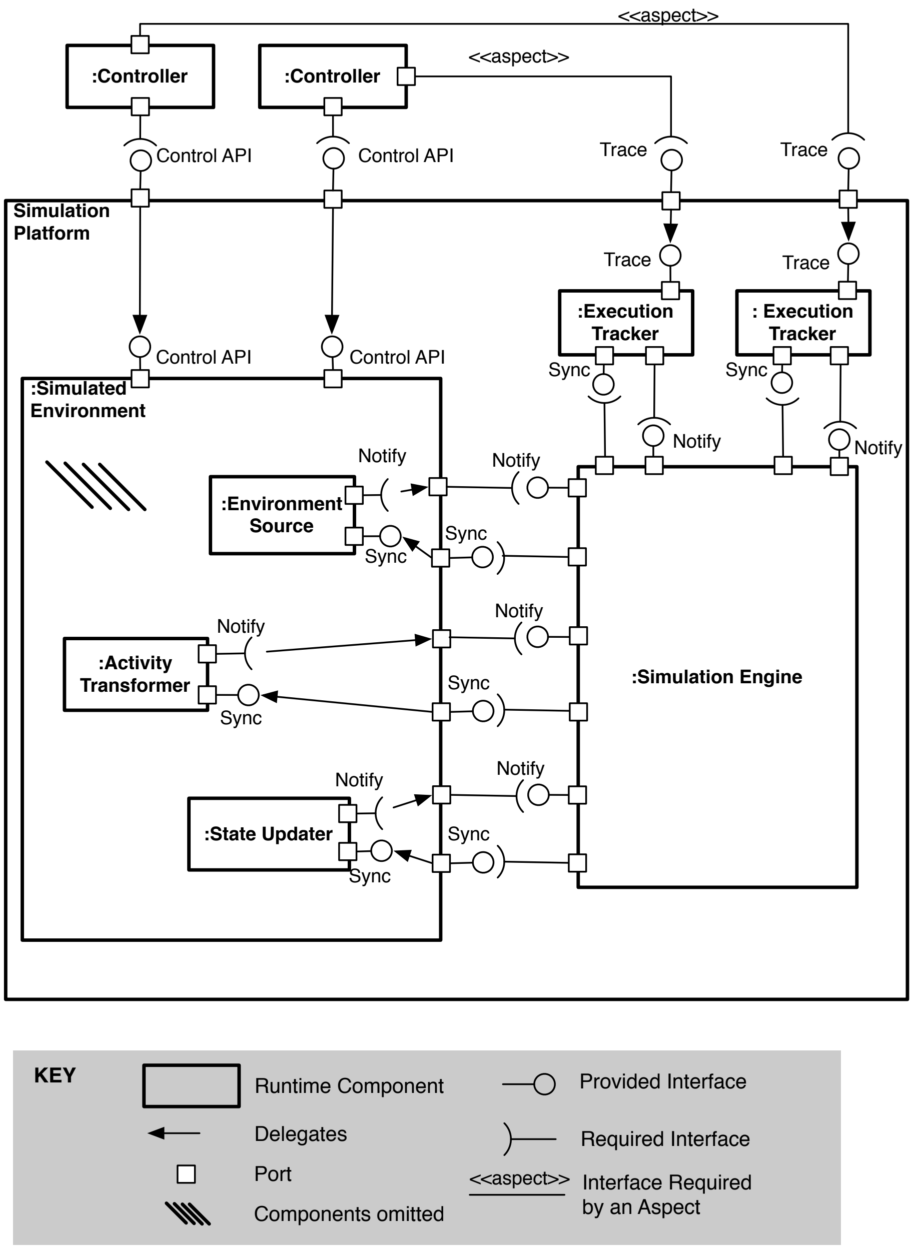


Figure 5.3: Component and connector view of the simulation engine

ing the environment. To ensure correct causal relations between its environment access and other things happening in the simulation, an `Environment Source` synchronizes its execution with the `Simulation Engine`: before performing an influence or sensing the environment, an `Environment Source` notifies the `Simulation Engine` at what moment in simulation time it wants to access the environment and suspends its execution until it is granted permission to proceed by the `Simulation Engine`.

- `Execution Tracker` (see Section 5.6). An `Execution Tracker` manages the execution of a `Controller`. An `Execution Tracker` keeps track of the execution time consumed by a `Controller` to deduce at what moment in simulation time a `Controller` accesses the environment. To determine whether a controller accesses the environment, an execution tracker keeps track of all control primitive invocations on the `Control API`. Synchronization is necessary to ensure correct causal relations between the access of a controller to the environment and other things happening in the simulation, Each time a control primitive of the `Control API` is invoked, the `Execution Tracker` notifies the `Simulation Engine` and suspends that `Controller's` execution until it is granted permission to proceed by the `Simulation Engine`.

- `Activity Transformer` (see Section 5.4.1). An `Activity Transformer` changes the activities in the `Activity` repository by applying the reaction and interaction laws. To ensure correct causal relations between activity transformations and other things happening in the simulation, the `Activity Transformer` notifies the `Simulation Engine` before applying an activity transformation and suspends its execution until it is granted permission to proceed by the `Simulation Engine`.

- `State Updater` (see Section 5.4.1). A `State Updater` updates the `State` repository to a new snapshot time by flushing activities. To guarantee correct causal relations with the rest of the simulation, the `State Updater` notifies the `Simulation Engine` of the new snapshot time it wants to update the `State` repository to and suspends its execution until permission to proceed is granted by the `Simulation Engine`.

5.5.2 Interface Descriptions

Figure 5.3 illustrates how the various components are connected with the `Simulation Engine`. The synchronization of all components happens through a uniform interface:

- `Notify` is the interface provided by the `Simulation Engine` to enable components to publish new events. To notify the Simulation Engine of a new event, a component specifies (1) the simulation time stamp of the event and

(2) a callback identifier of the component. The callback identifier is used for granting permission to that component when it is safe to execute that event.

- `Sync` is the interface required by the `Simulation Engine` to grant permission to a component for executing an event.

5.5.3 Design Rationale

The Simulation Engine encapsulates all synchronization. An important architectural decision is that the main components within the `Simulation Platform` can make abstraction of all synchronization with other components. The `Simulation Engine` encapsulates the actual synchronization algorithm (in our case a conservative synchronization algorithm [CM81]) that maintains and manages all synchronization partners. The `Simulation Engine` uses the `Notify` and `Sync` interfaces to synchronize various components. As such, the `Simulation Engine` does not depend upon the internal working and functionality of these components.

5.6 An Aspect-Oriented Approach to Embed Control Software

We explain the way the software of the real controllers is embedded in the simulation platform. Recall that a `Controller` is connected to the `Simulation Platform` by means of two interfaces: the `Control API` interface the `Trace` interface, as is depicted in Figures 5.1 and 5.3. The `Trace` interface enables monitoring the execution of a controller by tracking its duration primitive invocations and control primitive invocations (see Chapter 4). However, in contrast to the `Control API` interface, the `Trace` interface is not a native interface of a controller. The `Trace` interface is solely necessary for simulation purposes, i.e. to enable synchronizing the execution of a controller with the simulation. Consequently, embedding a controller in the simulation platform would require the developer to modify the design of the controller such that it supports the `Trace` interface. This would be a time-consuming and error-prone job, which we would like to avoid.

We describe an approach to extend a control application transparently, i.e. without requiring the developer to perform changes in the design of the controllers. Our approach uses aspect-oriented programming to achieve this. We first introduce aspect-oriented programming in Section 5.6.1. The way aspect-oriented programming is used in the simulation platform is described in Section 5.6.2. We emphasize how important qualities are realized in Section 5.6.3.

5.6.1 Aspect-Oriented Programming

Tracking the execution of a controller is a *crosscutting concern*, i.e. the functionality to do this crosscuts a control application's basic functionality. The problem of crosscutting concerns is that they can not be modularized with traditional object oriented techniques. This forces the functionality to monitor the execution of a controller to be scattered throughout the code of the control application, resulting in "tangled code" that is excessively difficult to develop and maintain. Aspect-oriented programming [KLM$^+$97, KHH$^+$01] handles crosscutting concerns by providing *aspects* for expressing these concerns in a modularized way. An aspect is a modular unit of crosscutting implementation. Aspect-oriented programming does not replace existing programming paradigms and languages, but instead, it can be seen as a co-existing, complementary technique that can improve the utility and expressiveness of existing languages. It enhances the ability to express the separation of concerns which is necessary for well-designed, maintainable software systems.

A language extension to Java which supports aspect-oriented programming, is AspectJ. In AspectJ, defining an aspect is based on two main concepts: pointcuts and advice. A *pointcut* is a language construct in AspectJ that selects particular join points, based on well-defined criteria. Each *join point* represents a particular point in the execution flow of a program where the aspect can interfere, e.g. a point in the flow when a particular method is called. As such, pointcuts are a means to express the crosscutting nature of an aspect. *Advice* on the other hand is a language construct in AspectJ that defines additional code that runs at join points specified by an associated pointcut. An aspect encapsulates a particular crosscutting concern and can contain several pointcut and advice definitions. The process of inserting all crosscutting code of an aspect at the appropriate join points within the original program code, is called *aspect weaving.* Aspect weaving is performed at compile-time in AspectJ.

5.6.2 Providing Support for Tracing a Controller's Execution through Aspect Weaving

We describe a way to flexibly embed a control application in the simulation platform, i.e. without requiring the developer to alter the design of the controllers. We use aspect-oriented programming technology to plug and unplug into a control application all functionality required for simulation purposes.

To embed the controller in a simulation platform, the controller must be extended with following tracing functionality:

- *Tracing duration primitive invocations.* The simulation platform tracks the execution time consumed by each controller according to the duration mapping, so it must be able to monitor all duration primitives invocations of a controller. To support this kind of monitoring, the controller should be extended with functionality that notifies the simulation platform each time the controller executes a duration primitive.

- *Tracing control primitive invocations.* The simulation platform synchronizes a controller's invocations on the `Control API` with the rest of the simulation (see Section 5.5.1). To support such synchronization, the simulation platform must be capable of intercepting a control primitive invocation and of temporarily suspending a controller's execution.

Figure 5.4 depicts the way the above tracing functionality is inserted in the controller software. This figure shows an example `Controller` that consists of a `Decision Taker` module and a `Plan Library` module.

The left hand side of the figure depicts the original controller. Note that this controller does not support the `Trace` interface. The left hand side of the figure also depicts an `Aspect`. The `Aspect` is a separate module that encapsulates all tracing functionality. The `Aspect` is generated from the specification of the duration primitives and control primitives. The *pointcut* definition of the aspect specifies all duration primitive invocations and control primitive invocations as join points. The *advice* of the aspect comprises a call to the `Trace` interface to notify the simulation platform.

The black arrow on the figure illustrates the process of aspect weaving. Aspect weaving happens at compile time, and automatically extends the `Controller` with all tracing functionality necessary to embed it in the simulation platform.

The right hand side of Figure 5.4 depicts the outcome of the weaving process. Within the `Controller`, the `Decision Taker` and `Plan Library` modules are now extended with additional tracing functionality that is the result of weaving the aspect's advice. The added tracing functionality crosscuts the modules of a controller, as depicted by the grey blocks. Note that due to aspect weaving, the controller now supports the `Trace` interface at the appropriate locations without requiring the developer to perform manual modifications to the control software.

5.6.3 Design Rationale

Flexibility of embedding a control application. Aspect weaving supports flexibly embedding the control application in a simulation: the developer is no longer bothered to modify a control application and manually insert or remove all code necessary for tracing its execution.

Separating simulation from application concerns. Aspect technology enables modularizing simulation concerns that crosscut the control application's functionality. This leads to a clean separation between application concerns and simulation concerns, as both are encapsulated in separate modules (as depicted on the left hand side of Figure 5.4).

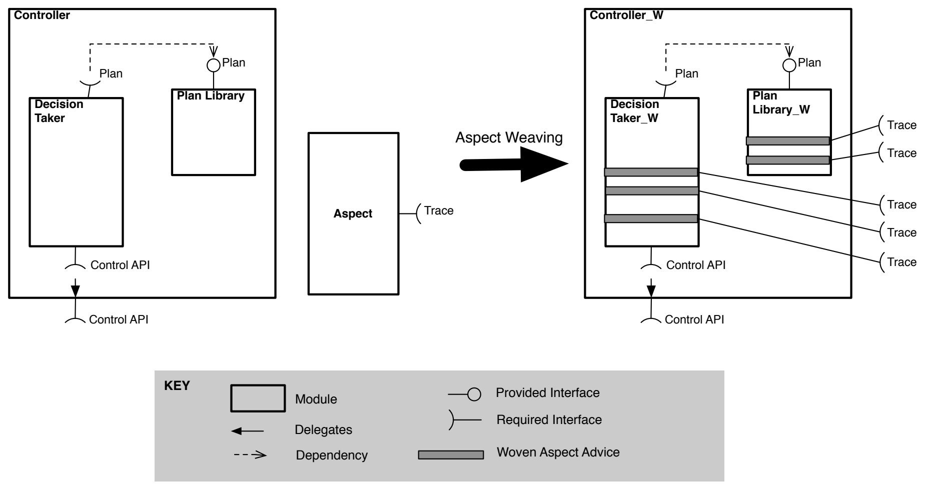


Figure 5.4: Controller before (left) and after (right) aspect weaving.

5.7 Component and Connector View of the Execution Tracker

We focus on the `Execution Tracker`. An Execution Tracker is responsible for tracing the execution of a particular controller of the distributed control application. Tracing the execution of a controller includes (1) determining the execution time consumed by a particular controller of the distributed control application according to the duration mapping, and (2) synchronizing the execution of that controller with the simulation engine, which is necessary to enable as-fast-as-possible simulations.

Figure 5.5 depicts a component and connector view of two controllers embedded in the simulation platform. The focus is on the Execution Trackers and the way they interact with a controller on the one hand, and with the simulation engine on the other hand. We first discuss an Execution Tracker's main elements and their properties. Afterwards, we describe how they are connected and explain how important qualities are realized.

5.7.1 Elements and Their Properties

Figure 5.5 depicts two `Execution Trackers`, each connected to a `Controller`. Each Execution Tracker comprises the following components and repositories:

- `Clock Manager` is responsible for managing the simulation clock of a par-

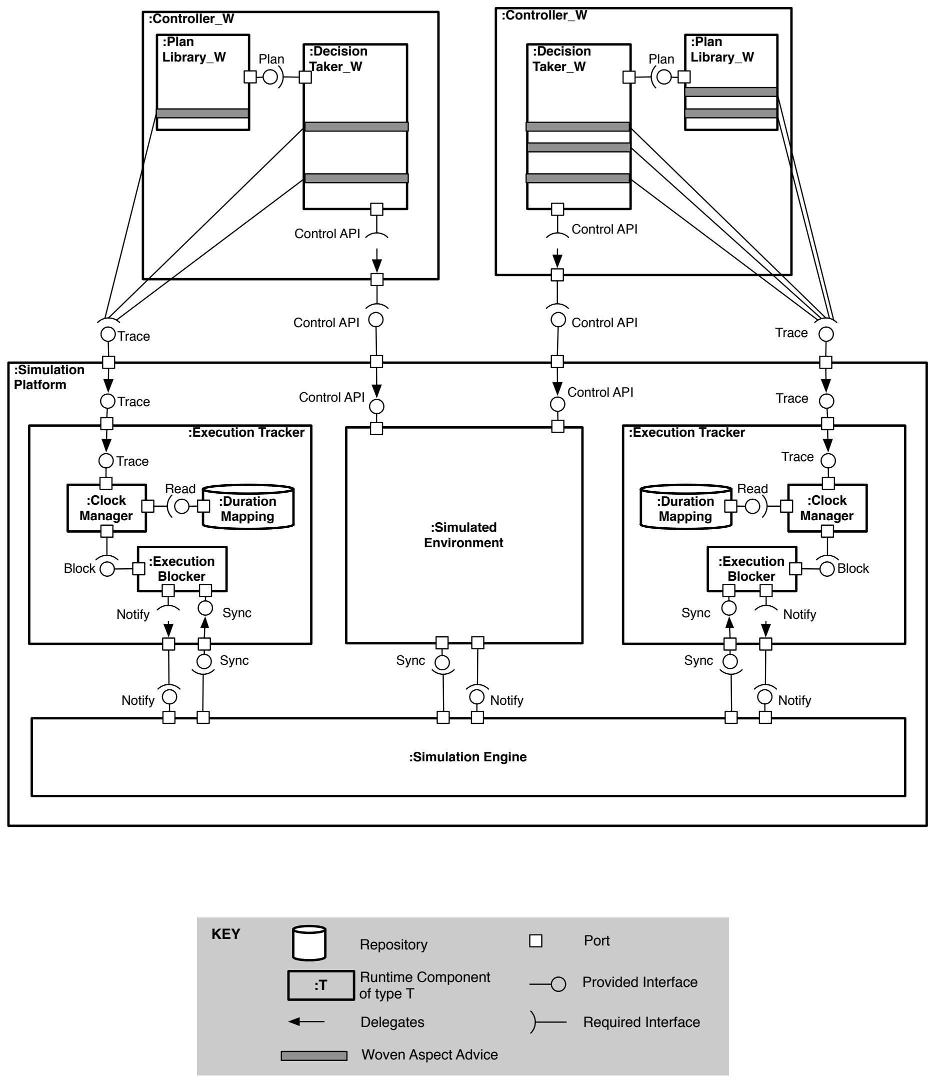


Figure 5.5: Component and connector view of controllers and execution trackers

ticular `Controller`. The simulation clock indicates how much execution time that particular `Controller` consumed. As a `Controller` executes, the `Clock Manager` is notified of the duration primitives invocations performed by that controller, and advances the simulation clock with the duration that is specified by the duration mapping (see Section 4.4.2). As such, the simulation clock of the `Clock Manager` is kept up-to-date with the execution time of the `Controller`.

- `Duration Mapping` repository is responsible for maintaining the duration mapping of a particular controller. For each duration primitive invocation, the `Duration Mapping` repository specifies a duration in simulation time.

- `Execution Blocker` is responsible for synchronizing the execution of a `Controller` with the `Simulation Engine`. For each control primitive invocation, the `Execution Blocker` can temporarily suspend the execution of a `Controller` until access is granted by the `Simulation Engine`.

5.7.2 Interface Descriptions

Figure 5.5 depicts the interconnections between the various elements of an `Execution Tracker`:

- `Trace` is the interface provided by the `Clock Manager` to keep track of the execution of a `Controller`. By means of aspect weaving (see Section 5.6) a `Controller's` execution is intercepted and redirected to the `Clock Manager` each time a duration primitive invocation or control primitive invocation is performed. The caller of the `Trace` interface specifies the characteristics of the duration primitive invocation or control primitive invocation (see Section 4.4.1 and Section 4.5.1).

- `Read` is the interface provided by the `Duration Mapping` repository to enable retrieving the duration in simulation time of various duration primitive invocations. The caller specifies the characteristics of the duration primitive invocation. `Read` returns the associated duration for that duration primitive invocation according to the duration mapping.

- `Block` is an interface provided by the `Execution Blocker` to synchronize the execution of a `Controller` with the `Simulation Engine`. The caller of the `Block` interface specifies a simulation time instant until which the execution of the `Controller` should be suspended. The `Clock Manager` calls the `Block` interface with the current value of its simulation clock in case it traces a control primitive invocation. This guarantees synchronization with the `Simulation Engine` each time a `Controller` accesses the environment.

5.7.3 Design Rationale

Separating monitoring from synchronization. The `Clock Manager` encapsulates all functionality to monitor the execution time of a `Controller`. The `Execution Blocker` encapsulates the functionality to synchronize the execution of a `Controller` with the `Simulation Engine`. Because both components have low coupling, a `Clock Manager` can make abstraction of synchronizing the execution of a `Controller`, whereas the `Execution Blocker` can make abstraction of monitoring a `Controller's` execution time.

Reuseable infrastructure. We emphasize the reusability of the architecture of the `Execution Tracker`. The internal components of the `Execution Tracker` are independent of the specified duration mapping. The duration mapping is encapsulated in the `Duration Mapping` repository, where it can be adapted easily.

5.8 Conclusions

We developed a simulation platform to demonstrate that the modeling framework is feasible for developing executable simulations. The simulation platform encapsulates the functionality to support the modeling constructs in an executable simulation. The simulation platform supports simulations (1) in which the software of real controllers can be embedded, and (2) of which the simulation model is described in terms of the proposed modeling constructs.

We put forward an architecture for such a simulation platform, and documented this architecture using several architectural views. The top-level module decomposition view of the architecture of the simulation platform comprises three main modules that each encapsulate a core functionality of the simulation platform:

1. The `Simulated Environment` module is responsible for managing the model of the environment. This module encapsulates all functionality to support the modeling constructs for dynamic environments.

2. The `Execution Tracker` module is responsible for tracing the execution of the software of a real controller of the distributed control application. This module encapsulates all functionality to support the modeling constructs for the control software.

3. The `Simulation Engine` module is responsible for managing the evolution of all parts of the simulation in correspondence to the specifications of the simulation model. The simulation engine encapsulates all functionality to synchronize the progress of the simulated environment with the progress of all execution trackers of the controllers that are embedded in the simulation.

The architecture uses aspect technology for plug-and-play integration of the control software in a simulation. Aspect technology is used to weave all tracing functionality required for the simulation into the control software. This contributes to a clean separation between application concerns and simulation concerns.

Chapter 6

Simulation of AGV Control Applications in Dynamic Warehouse Environments

In this chapter, we apply and evaluate the modeling framework and the simulation platform on a real-world case. We developed an automated guided vehicle (AGV) simulator that supports the evaluation of new or altered features of distributed control applications that control automated guided vehicles (AGVs) in warehouse environments.

6.1 Introduction

We demonstrate the modeling framework and the simulation platform in the context of a real-world case. The case comprises the development of distributed control applications that control automated guided vehicles (AGVs) in warehouse environments. An automated guided vehicle (AGV) control application was developed in the EMC^2 (Egemin Modular Controls Concept) project. Egemin N.V. is a Belgian manufacturer of AGVs, and develops control software for automating logistics in warehouses and manufactories using AGVs. AGVs are unmanned, battery-powered vehicles that are able to transport loads through a warehouse or factory.

In the context of the EMC^2 project, we developed an *AGV simulator*, i.e. a case-specific simulation platform that supports evaluating new or altered functionalities of the AGV control application in a simulated warehouse environment. The AGV simulator enables (1) safe experimentation and testing of AGV control applications without risk of damaging the real AGVs, (2) executing experiments faster than real-time, which is essential when investigating long-term scenarios (3) setting up and monitoring experiments in a less costly way, e.g. without the cost of buying AGVs or building particular warehouse layouts.

The goal of this chapter is to illustrate how the modeling framework and the simulation platform underpin the model formulation and the model translation phase for developing the AGV simulator, as well as to evaluate the flexibility and performance of the AGV simulator. Consequently, the goal of this chapter is not to evaluate the suitability of particular designs for AGV control applications. The focus is on the AGV simulator that supports such evaluations. The EMC^2 project is considered as providing the context and requirements for the AGV simulator developed with the aid of the modeling framework and the simulation platform.

This chapter is structured as follows. Section 6.2 describes the setup of an AGV transportation system within a warehouse, discusses the requirements of the AGV control application and the goal of the EMC^2 project. In Section 6.3, we elaborate on the requirements of the AGV simulator. In Section 6.4, we formulate a simulation model for the AGV simulator in terms of the modeling constructs of the framework. In Section 6.5, we describe the way the simulation platform supports translating the simulation model of the AGV simulator in an executable simulation. In Section 6.6, we evaluate the flexibility and performance of the AGV simulator. We draw conclusions in Section 6.7.

6.2 AGV Transportation System

An AGV transportation system is an industrial transport system using several automatic guided vehicles (AGVs) in a warehouse that are controlled by an AGV control application. AGV transportation systems are typically used for repackaging and distributing incoming goods to various branches, or distributing manufactured products to storage locations.

6.2.1 Physical Setup of an AGV Transportation System

Figure 6.1 shows a three dimensional view on an AGV transportation system. An AGV is an unmanned, computer-controlled transportation vehicle using a battery as its energy source. AGVs have to perform transports. A transport consists of picking up a load at a particular spot in the warehouse and bringing it to its destination. A load ranges from raw materials (e.g. wood, rolls of paper) to completed products (e.g. tyres, cheese).

The hardware of an AGV comprises the following. An AGV contains engines to move and turn and a lift to pick and drop loads. An AGV has sensors to observe its position and battery level. Finally, each AGV has a computer platform on which control software can be deployed. The computer platform of an AGV uses wireless communication.

The warehouse is a storage or manufacturing facility that contains various loads positioned at various locations across the warehouse. Loads are typically stored in racks. Racks are used to hold loads and are positioned across the warehouse, usually according a geometrical pattern that combines easy accessibility of the loads, as well as efficient use of the available room for storage purposes. Typically, also one or several battery chargers for the AGVs are positioned at particular locations across the warehouse.

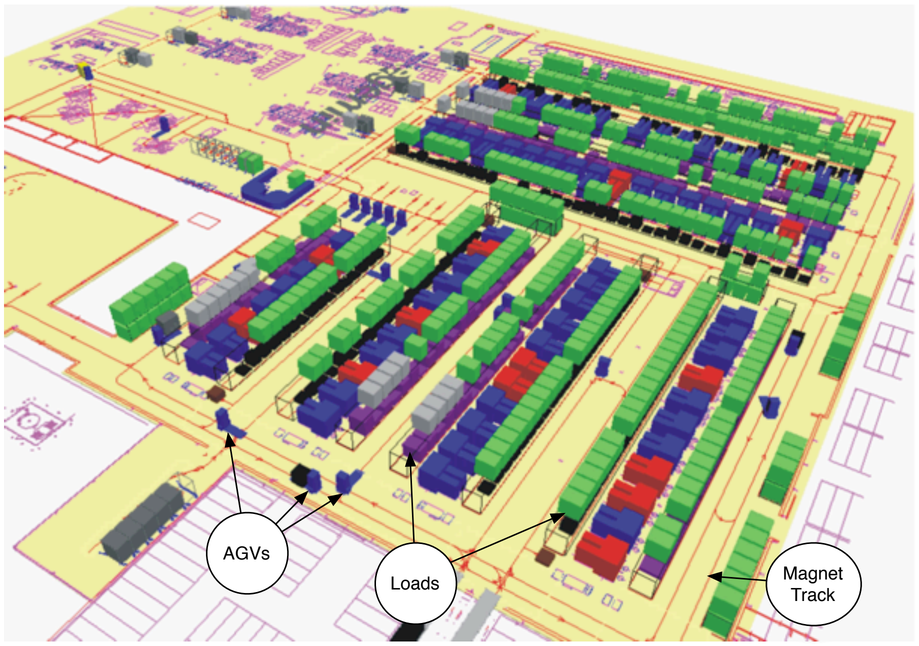


Figure 6.1: Three dimensional view on an AGV transportation system.

To support AGVs, the warehouse is usually customized. This typically involves a custom configuration of the racks. In addition, a complex layout of magnet strips is built into the warehouse floor to guide the AGVs to move from one spot in the warehouse to another. This *magnet track* allows AGVs to maneuver in an accurate manner according to predefined pathways. Moreover, as magnets are inexpensive and can be installed easily, magnet guided navigation is relatively cost-effective.

6.2.2 AGV Control Application

An AGV control application is a software system that controls a set of AGVs. We discuss the main functionalities of an AGV control application and elaborate on the AGV steering system that can be used by an AGV control application to instruct AGVs.

6.2.2.1 Functionalities of an AGV Control Application

The main functionality of an AGV control application is handling transports, i.e. moving loads from one place to an other. Transports are typically generated by order management software, but a transport can also be introduced manually by employees or operators. Abstracting from the origin of the transports, systems that generate transports for the AGV control application are called client systems. Client systems input transports to the AGV control application, and expect a confirmation from the AGV control application when the transport is done.

In order to handle transports, the AGV control application has to use the AGVs under its control efficiently. The main functionalities to be performed are the following.

- *Transport assignment:* transports originating from client systems must be assigned to an appropriate AGV. The goal is to assign transports in such a way that overall, transports are handled in an efficient and timely manner.
- *Routing:* in order to carry out transports, AGVs need to move to certain places. For the movement of all AGVs, efficient routes through the warehouse must be determined. Although the road network determined by the magnet track is static, the best route for an AGV is in general dynamic, and depends on the current conditions in the system. For example, the shortest route in distance may take a long time because there is a "traffic jam". So, routing in general may need to be adapted dynamically.
- *Collision avoidance:* while moving around, AGVs may not collide with each other. Collisions do not exclusively occur at intersections of paths; AGVs also need to avoid collisions while passing each other on closely located parallel paths.
- *Deadlock avoidance:* since AGVs cannot divert from their path, they are relatively constrained in their movement. Therefore, deadlocks can occur when a number of AGVs are in a situation where no AGV can move anymore without operator intervention. For example, on a bidirectional path AGVs may be standing head on towards each other. Since AGVs in general cannot drive in reverse, none of the two AGVs can move forward or backward. The AGV control application must ensure that manual intervention for such situations is avoided.

Besides handling transports efficiently, the AGV control application must also ensure the continued operation of the AGVs.

- *Maintenance:* AGVs need regular maintenance, which is typically scheduled in fixed time intervals. Furthermore, AGVs may need to calibrate their positioning system regularly.
- *Battery charging:* when an AGVs battery runs out, it must drive to a charging station. Either the AGV must wait until an operator exchanges the old battery for a full battery, or the battery is charged using contact points in the warehouse floor.
- *Resource saving:* AGVs are expensive and must be used as efficiently as possible. AGVs that are idle must save their resources, and get out of the way of the active AGVs. Therefore, idle AGVs are parked at park nodes.

6.2.2.2 AGV Steering System

To control an individual AGV, it is equipped with an AGV steering system developed by Egemin, called E'nsor[1]. E'nsor handles the low-level control of an AGV on the level of reading out sensors and driving actuators. Main functionalities of E'nsor are keeping the AGV on a path, turning, determining the AGVs current position, reading out the battery level, etc.

E'nsor can handle a number of actions on its own. These actions are called jobs. For example, picking up a load is a pick job, dropping it is a drop job and moving over a specific distance is a move job. A transport typically starts with a pick job, followed by a series of move jobs and ends with a drop job. The AGV control application gives jobs to E'nsor, which in turn controls the AGV to handle the jobs autonomously.

To be able to indicate to E'nsor where to pick and move, the layout (i.e. all the possible paths the AGVs can follow in the system) of the warehouse is divided into logical elements: segments and nodes. Segments determine the path an AGV can follow through the warehouse, and can be either straight or curved with lengths of typically three to five metres. A segment can either be unidirectional or bidirectional. In the latter case AGVs can drive over the segment in both directions. Nodes are at the beginning or end of segments. Nodes are the places where an AGV can stand still, or do an action like picking up a load. In normal operation, an AGV can only be at rest when standing on a node. Each segment and node is given a unique identifier. E'nsor is able to steer an AGV on a segment per segment basis. An AGV can stop on every node, possibly to change direction. E'nsor can handle five jobs. None of these jobs route the AGV, so the segment given as argument must be accessible from the node on which the AGV is currently standing.

[1]E'nsor is an acronym for Egemin Navigation System On Robot.

- Move(segment): this instructs E'nsor to drive the AGV over the given segment.
- Pick(segment): instructs E'nsor to drive the AGV over the given segment and pick up the load at the node at the end of the segment.
- Drop(segment): the same as pick, but drops a load the AGV is carrying.
- Park(segment): instructs E'nsor to drive the AGV over the given segment and park at a park node at the end of the segment.
- Charge(segment): instructs E'nsor to drive the AGV over a given segment to a battery charging node and start charging batteries there.

Furthermore, E'nsor allows the readout of sensor values of the AGV, of which the most important are battery level; position in coordinates on the floor; position in terms of segment and node on the layout; orientation; speed.

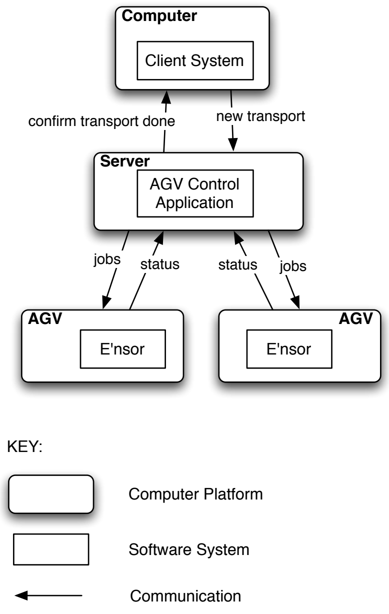
6.2.3 The EMC^2 Project

Egemin developed an AGV control application that is used in their existing installations. To investigate the long term possibilities for AGV control applications, the EMC^2 project has two main goals: studying the feasibility of a decentralized architecture of the AGV control application, and increasing the flexibility of the existing solution. Each of these goals is sketched in turn.

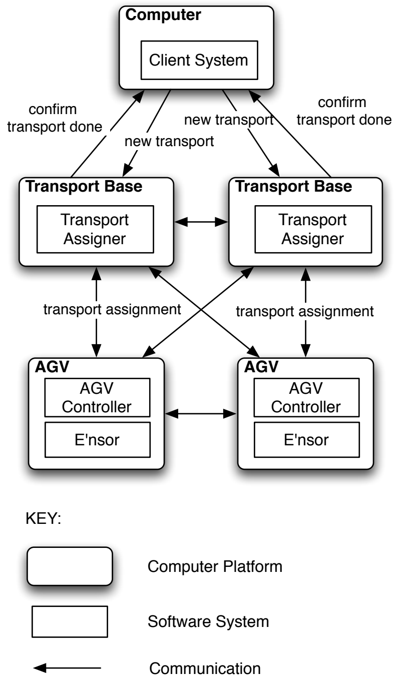
Decentralisation. Traditionally, the AGV control application is deployed on one central server, which uses wireless communication to hand out jobs to each AGV, see Figure 6.2(a). The AGV control application receives transport requests from the client systems. According to the incoming transports, the AGV control application plans routes for AGVs and instructs AGVs to perform the jobs, so that all transports are done and so that AGVs do not collide or enter a deadlock situation. The server continuously polls the AGVs about their status to monitor the progress of the transports. When a transport is finished, the server reports the completion of the transport to the corresponding client system.

In a decentralized approach the AGV control application would be distributed over the various AGVs and other systems. This puts more autonomy in the AGVs: decisions are made locally on each AGV, and each AGV coordinates with other AGVs to ensure the system as a whole processes transports in time.

Figure 6.2(b) shows a possible decentralized architecture for an AGV control application, i.e. the architecture that was chosen in the course of the project. The AGV control application is decomposed in *transport assigners* and *AGV controllers.* AGV controllers control a single AGV and are deployed on each AGV.



(a) Deployment diagram of centralized solution.



(b) Deployment diagram of decentralized solution.

Figure 6.2: Deployment diagram of centralized versus decentralized solution.

AGV controllers are responsible for driving the AGV, avoiding collisions and deadlocks, and executing the transport assigned by transport assigners. AGV controllers thus raise the autonomy of AGVs further, by building on E'nsor and using the wireless network to coordinate with each other.

Transport assigners are deployed on one or more transport bases, and are responsible for interacting with client systems, and to assign the incoming transports to an appropriate AGV. The functionality of assigning and monitoring the progress of transports can not be completely decentralized, since client systems need a fixed node to send transport requests to. Also, the transport assigner needs to make sure that a transport is never lost, and so needs to be deployed on reliable, fixed infrastructure.

The above shows that neither the centralized, nor the decentralized architecture is respectively completely under central control, or completely without central control. In the centralized architecture, E'nsor gives each AGV a considerable amount of autonomy, since it handles steering the AGV and executing high level actions such as pick and drop. This functionality is practically impossible to centralize. On the other hand, the decentralized architecture needs to keep an element of central control in the form of transport bases, since client systems and operators need a fixed point where they can monitor the AGV system. The aim of the EMC^2 project was to decentralize the system as much as practically possible, while reusing as much functionality as possible from the existing AGV control application (i.e. E'nsor, representation of layout, monitoring,...).

Flexibility. Flexible AGVs should adapt their behaviour with changing circumstances in the AGV transportation system. One aspect of flexibility is that AGVs should be able to exploit opportunities. For example, while an AGV is on its way to pick up a particular load, it could be beneficial for the AGV to switch to a new transport that pops up. AGVs should also be able to anticipate possible difficulties. For example, when the battery level of an AGV decreases, the AGV should anticipate this and prefer a zone near a charge node. Another aspect of flexibility is that AGVs should be able to cope with exceptional situations. For example, if a segment is blocked, the AGVs should avoid that segment.

Introducing more flexibility in the system is in principle independent of whether a centralized or decentralized approach is used. In a centralized solution, the plan for the further evolution of the system may be re-evaluated at regular intervals. In a decentralized solution, flexibility is introduced by designing adaptive protocols and coordination between the entities in the system. The entities that constitute the decentralized system must be able to adapt to changing circumstances.

A complete description of the course and conclusions of the EMC^2 project is out of the scope of this text. In the following, we focus the way simulation supports the development of a decentralized solution for the AGV control application.

6.3 Requirements of the AGV Simulator

We elaborate on the requirements of an AGV simulator that was developed in the context of the EMC^2. The goal of the AGV simulator is to support evaluating new or altered features of a decentralized AGV control application by means of software-in-the-loop simulation of AGV controllers in a simulated warehouse environment. Software-in-the-loop simulation enables evaluating the actual implementation (or parts thereof) of the AGV controllers.

The AGV simulator focusses on evaluating routing, collision avoidance, transport assignment and battery charging.

- *Support for routing.* To evaluate or compare routing behaviors of AGV controllers, the AGV simulator should simulate the movements of real AGVs and realistic layouts of the warehouse. This enables monitoring the path followed by an AGV, its travel time, the appearance of traffic jams, etc.

- *Support for collision avoidance.* To evaluate the appropriateness of collision avoidance techniques of AGV controllers, the AGV simulator should simulate the movements of AGVs on a warehouse layout and detect situations in which AGVs could collide. Moreover, to test the robustness of collision avoidance techniques, the AGV simulator should simulate unreliable communication between AGVs. This enables a developer to evaluate the adequacy of collision avoidance techniques under a variety of circumstances.

- *Support for transport assignment.* To evaluate transport assignment among AGV controllers, the AGV simulator should simulate several transport profiles generated by client systems.

- *Support for battery charging.* To evaluate the charging strategy of AGV controllers, the AGV simulator should simulate the energy consumption of an AGV, the charging of its battery at a charging station and the interruption of an AGV's operation in case it runs out of energy.

When building an AGV control application, these functionalities are typically developed iteratively. The AGV simulator should enable testing partial AGV control applications in which some of these functionalities are present and others not yet (fully) operational.

To support evaluating different functionalities of AGV control applications in a variety of settings, modifying core parts of the AGV simulator should be relatively easy and the impact of such modifications should be as local as possible. Important modifications that should be supported by the AGV simulator include altering the AGV controller software; the layout of the warehouse; the number of AGVs and their characteristics (e.g. characteristics of movements, energy consumption, etc.); the quality of service of communication between AGVs; the accuracy of collision detection; the transport profile of the client systems.

6.4 Model Formulation: Simulation Model of the AGV Simulator

We formulate a simulation model for the AGV simulator. The simulation model is described in terms of the modeling constructs of Chapter 3 and Chapter 4. We first focus on the simulation model of the warehouse environment in Section 6.4.1. Afterwards, we elaborate on the simulation model for integrating the AGV controller software in Section 6.4.2.

6.4.1 Simulation Model of the Warehouse Environment

Figure 6.3 gives a graphical overview of the environment part of the simulation model of the AGV simulator. This figure shows specific instantiations of the modeling constructs of Figure 3.1. The simulation model is organized in four parts, in analogy with Figure 3.1:

1. the part representing the *structure of the warehouse environment*;
2. the part representing *dynamism in the warehouse environment*;
3. the part representing the *manipulation of dynamism in the warehouse environment*;
4. the part representing *sources of dynamism in the warehouse environment.*

We give an overview of each of these parts of the simulation model for warehouse environments.

Structure of the Warehouse Environment. We describe the way the structure of the warehouse is captured in a simulation model. The structure of the simulated warehouse environment is modeled in terms of `environmental entities` and an `environmental layout` that arranges the entities with respect to each other.

The following environmental entities of the warehouse are captured in the simulation model:

- *AGVs.* AGVs are robotic vehicles controlled by an embedded AGV controller. AGVs can drive across the warehouse, have a lift to carry a load and a wireless WiFi module to communicate.
- *Loads.* Loads are materials, products or goods that are stored in the warehouse and can be transported by AGVs.
- *Warehouse floor.* The warehouse floor is the flat area with a particular size. AGVs can move across the warehouse floor. Loads are positioned on the warehouse floor.
- *Segments.* The magnet track is modeled in terms of the logical map representation that is employed by the AGVs, i.e. in terms of segments and stations. A segment can be unidirectional or bidirectional and has a particular length. Each segment connects two stations.
- *Stations.* Stations are locations that connect adjacent segments. Each station can be used for one or several purposes, i.e. as routing location, as storage location for loads, as parking location and/or as battery charging location.
- *WiFi access points.* Wireless access points support the communication among AGVs. A WiFi access point enables communication between AGVs and transport bases or among several AGVs.

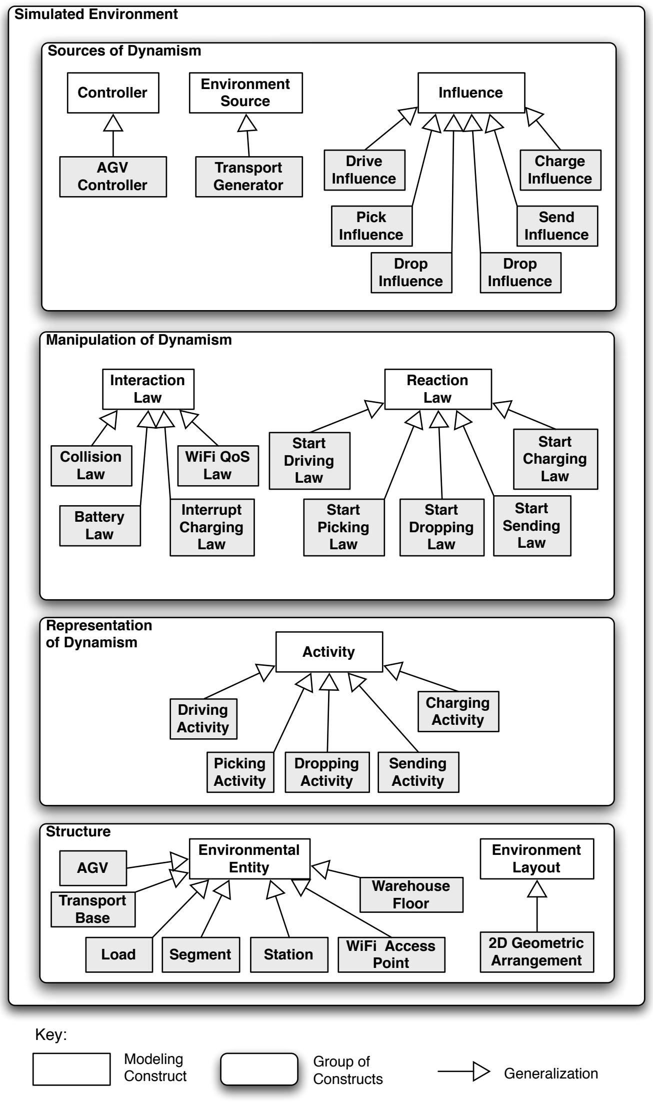


Figure 6.3: Overview of the simulation model of the simulated warehouse environment. The grey parts are specific instantiations of the modeling constructs for the AGV simulator.

- *Transport base.* A transport base is a computer that can be used to broadcast new transport tasks to the AGVs. A transport generator is embedded in a transport base.

We arrange the entities in the simulated warehouse environment according to a continuous two dimensional geometric layout. This layout expresses the spatial positioning of all entities with respect to each other.

Dynamism in the Warehouse Environment. We describe the way dynamism in the warehouse environment is captured in a simulation model. Dynamism in the simulated warehouse environment is modeled in terms of `activities`. In the simulated warehouse environment, the following activities can occur:

- *Driving activities.* Driving activities represent that AGVs drive across a segment on the warehouse floor until the station at the other end of that segment is reached.
- *Picking activities.* Picking activities represent that AGVs use their lift to pick up a load at a station.
- *Dropping activities.* Dropping activities represent that AGVs use their lift to put down the load they carry at a station.
- *Charging activities.* Charging activities represent that AGVs recharge their battery at a charging station.
- *Sending activities.* Sending activities represent that a WiFi access point is used to transmit messages from a transport base to AGVs or among AGVs.

Sources of Dynamism in the Warehouse Environment. In the warehouse environment, several sources of dynamism reside. We make a distinction between `controllers` and `environment sources`. Sources of dynamism can manipulate the environment by means of performing `influences`.

We consider the following sources of dynamism:

- *AGV controllers.* AGV controllers are the control software that is embedded in an AGV. AGV controllers constitute the AGV control application. Each AGV controller are responsible for controlling an AGV and for coordinating with other AGVs for routing, collision avoidance, transport assignment and battery charging.
- *Transport generator.* A transport generator broadcasts transport tasks to the AGVs. A transport generator generates transports according to a transport profile that specifies the characteristics of the stream of transport tasks

that should be handled by the AGVs. Transport generators are external to the AGV control application. Consequently, transport generators are environment sources of dynamism. A transport generator is deployed on a transport base.

We consider the following influences:

- *Drive influence.* A drive influence represents that attempt of an AGV controller to start driving over a given segment in a given direction.
- *Pick influence.* A pick influence represents that attempt of an AGV controller to start driving over a given segment in a given direction and to pick up a load at the station at the end of that segment.
- *Drop influence.* A drop influence represents that attempt of an AGV controller to start driving over a given segment in a given direction and to put down the load it carries at the station at end of that segment.
- *Send influence.* A send influence represents the attempt of an AGV controller or a transport generator to send a message.
- *Charge influence.* A charge influence represents the attempt of an AGV controller to start driving over a given segment in a given direction and to charge its batteries at the station at the end of that segment.
- *Park influence.* A park influence represents that attempt of an AGV controller to start driving over a given segment in a given direction and to park at the station at the end of that segment.

Manipulation of Dynamism in the Warehouse Environment. We describe the way manipulations of dynamism in the warehouse environment are captured in the simulation model. The way dynamism in the warehouse environment can be manipulated is modeled by means of `reaction laws` and `interaction laws`.

We consider the following reaction laws in the warehouse environment:

- *Start driving law.* Start driving law determines the reaction of the environment in response to a *drive influence* or a *park influence.* A real AGV does not always start driving when it is instructed to do so. Therefore, start driving law checks a number of conditions before adding a new *driving activity.* These conditions are that the AGV is not already involved in a driving, picking or dropping activity at the time of the influence; that the segment is adjacent to the station of the AGV; that the AGV is allowed to drive over the given segment in the given direction (as segments can be unidirectional). Start driving law does not define an activity in case one of these conditions does not hold, to reflect that E'nsor discards the instruction in these cases.

- *Start picking law.* Start picking law determines the reaction of the environment in response to a *pick influence*. Start picking law adds two activities, i.e. a new *driving activity* and a subsequent *picking activity*, in case a number of conditions apply. These conditions capture that a real AGV does not always start picking when it is instructed to do so. These conditions are that the AGV is not already involved in a driving, picking or dropping activity at the time of the influence; that the segment specified by the influence is adjacent to the station of the AGV; that the AGV is allowed to drive over the given segment in the given direction (as segments can be unidirectional). Start driving law does not add activities in case one of these conditions does not hold, to reflect that E'nsor discards the instruction in these cases.

- *Start dropping law.* Start dropping law determines the reaction of the environment in response to a *drop influence*. Start dropping law adds two activities, i.e. a new *driving activity* and a subsequent *dropping activity*, in case a number of conditions apply. Start dropping law is analogous to start picking law.

- *Start sending law.* Start sending law determines the reaction of the environment in response to a *send influence*. Start sending law adds a new *sending activity*.

- *Start charging law.* Start charging law determines the reaction of the environment in response to a *charge influence*. Start charging law adds two activities, i.e. a new *driving activity* and a subsequent *charging activity*, in case a number of conditions apply. These conditions capture that a real AGV in some cases discards the instruction to go charging. These conditions are that the AGV is not already involved in a driving, picking or dropping activity at the time of the influence; that the segment specified by the influence is adjacent to the station of the AGV; that the AGV is allowed to drive over the given segment in the given direction (as segments can be unidirectional). Start charging law does not add activities in case one of these conditions does not hold.

We consider the following interaction laws in the warehouse environment:

- *Collision law.* A collision law enforces collisions of AGVs in the warehouse environment. Based on the driving activities, a collision law determines whether AGVs collide. In case the collision law detects a collision, it returns an activity transformation that replaces the driving activity/activities involved with driving activity/activities that stop at the time the collision occurs.

- *Battery law.* A battery law enforces that all activities of an AGV are preempted in case it runs out of energy. Based on the energy consumption of driving, picking and dropping activities, a battery law preempts all activities as soon as an AGV runs out of energy.

- *WiFi QoS law.* A WiFi QoS law enforces a particular quality of service for the wireless communication. For example, to enforce a limited communication range, a WiFi QoS law removes sending activities of which the distance between sender/receiver and the WiFi access point exceeds this range. To enforce message loss, a WiFi QoS law can also remove sending activities with a given probability which could depend on the distance to the WiFi access point or the current communication load.
- *Interrupt charging law.* An interrupt charging law enforces that the charging activity is preempted as soon as an AGV is no longer located at a charging station, i.e. it started driving again.

6.4.2 Simulation Model for Integrating the AGV Controller Software

Figure 6.4 gives a graphical overview of the control software part of the simulation model of the AGV simulator. This figure shows specific instantiations of the modeling constructs of Figure 4.1. We elaborate on the way the AGV controller software interacts with the environment and the way the execution time of the AGV controller software is captured in the simulation model.

Control Interface of AGV Controllers. We describe the way AGV controllers interact with the warehouse environment. The interaction of the AGV controller software with the warehouse environment is modeled in terms of `control primitives` and a `control name mapping` and `control parameter mapping`.

The following control primitives are captured in the simulation model:

- *Ensor.move(segment).* Ensor.move(segment) is an E'nsor control primitive that instructs E'nsor to drive the AGV over the given segment.
- *Ensor.pick(segment).* Ensor.pick(segment) is an E'nsor control primitive that instructs E'nsor to drive the AGV over the given segment and pick up a load at the node at the end of the segment.
- *Ensor.drop(segment).* Ensor.drop(segment) is an E'nsor control primitive that instructs E'nsor to drop the load the AGV is carrying at the node at the end of the segment.
- *Ensor.park(segment).* Ensor.park(segment) is an E'nsor control primitive that instructs E'nsor to drive the AGV over the given segment and park at a park node at the end of the segment.

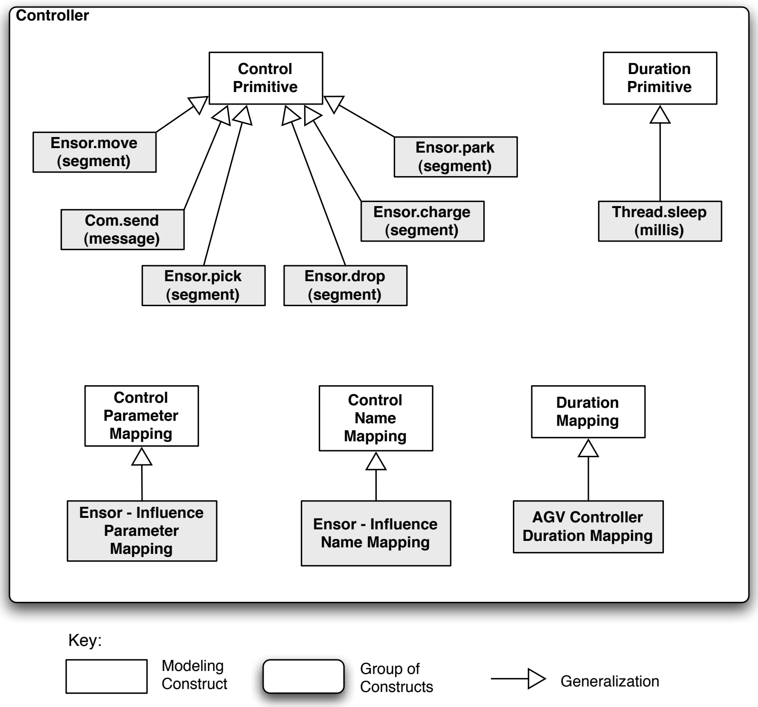


Figure 6.4: Overview of the simulation model for integrating the AGV control software in a simulation. The grey parts are specific instantiations of the modeling constructs for the AGV simulator.

- *Ensor.charge(segment).* Ensor.charge(segment) is an E'nsor control primitive that instructs E'nsor to drive the AGV over a given segment to a battery charging node and start charging batteries there.
- *Com.send(message).* Com.send(message) is a control primitive that instructs an AGV's onboard wireless communication module to send a message.

We employ an *Ensor-influence name mapping* to determine the name of the influences that result from the control primitive invocations. The mapping between control primitive invocations and influences is a straightforward one-to-one mapping. For example, invocations of the control primitive *Ensor.charge* will be mapped on *charge influences.*

We employ an *Ensor-influence parameter mapping* to determine the parameters of the influences that result from the control primitive invocations. For example, for all control primitives that take a *segment* as argument, the corresponding influence requires two parameters: the *segment* on the one hand, and one of both end stations of that segment on the other hand (to indicate the direction in which an AGV will drive over that segment). For invocations of the control primitive *Com.send(message)*, the corresponding *send influence* requires the receiver which is encapsulated in the message as an explicit parameter. The Ensor-influence parameter mapping takes care of determining all parameters needed for the influences.

Execution Time of AGV Controllers. We describe the way the execution time of AGV controllers is captured in a simulation model. The execution time of AGV controller software is modeled in terms of `duration primitives` and a `duration mapping`.

The duration primitives are typically dependent upon the AGV controller software. Therefore, the developer should specify custom duration primitives for the AGV control software that is to be embedded in the simulation. By default, only the following duration primitive is captured in the simulation model:

- *Thread.sleep(millis)*. Thread.sleep(millis) suspends the execution of an AGV controller for the specified number of milliseconds. This control primitive is used extensively in controllers, as the real environment typically evolves several orders of magnitude slower than the control application.

We employ an *AGV controller duration mapping* to specify the duration of invocations of duration primitives. By default, the *AGV controller duration mapping* only associates a duration to invocations of the duration primitive *Thread.sleep(millis)*. That duration corresponds to the amount of time specified by the argument *millis*.

6.5 Model Translation: Building Executable Simulations of the AGV Simulator

We describe how the simulation model of the AGV simulator can be translated into an executable simulation. We employ the simulation platform described in Chapter 5 to support executable simulations based on the simulation model of the AGV simulator.

We illustrate the way a number of core parts of the simulation model of the AGV simulator are designed. We focus on the design of driving activities of AGVs and the design of a collision law. The simulation platform encapsulates all functionality to support these constructs in an executable simulation.

6.5.1 Designing Driving Activities

We describe a way of designing driving activities.

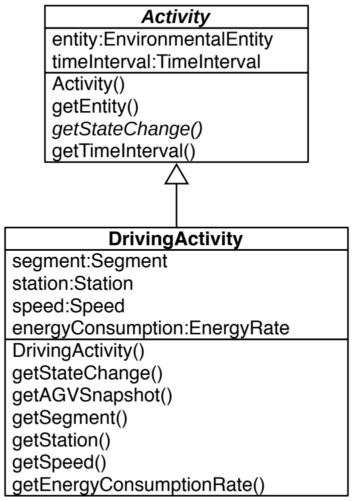


Figure 6.5: Class diagram depicting the DrivingActivity class.

Figure 6.5 shows that the class `DrivingActivity` inherits from the abstract class `Activity`. Driving activities encapsulate the following state:

- `entity` is the environmental entity that is involved in the drive activity. For a DrivingActivity, the subject is an AGV. `getEntity()` returns the entity of this DrivingActivity.

- `timeInterval` is the time interval during which the driving of the AGV happens. `getTimeInterval()` returns the time interval of this DrivingActivity.

- `segment` is the segment on the warehouse floor over which the AGV drives. `getSegment()` returns the segment of this DrivingActivity.

- `station` is the target station on that segment, which indicates the direction the AGV drives over the given segment. `getStation()` returns the station of this DrivingActivity.

- `speed` is the velocity of the movement of the AGV. `getSpeed()` returns the speed of this DrivingActivity.

- `energyConsumption` is the rate of energy the AGV consumes for this driving activity. `getEnergyConsumptionPerTimeUnit()` returns the amount of consumed energy per second for this driving activity.

In addition to the methods described above, driving activities offer the following methods:

- `DrivingActivity(EnvironmentalEntity e, TimeInterval i, Segment s, Station st, Speed v, EnergyRate r)` creates a new driving activity with given entity, time interval, segment, station, speed and energy rate.

- `getStateChange()` returns a StateChange object that encapsulates the state change of this activity for the time instant this activity completes. The state change encapsulates a state update of (1) the position of the AGV and (2) the battery level of the AGV.

- `getAGVSnapshot(TimePoint t)` returns a new AGV object that encapsulates the AGV state at time instant `t`. The snapshot time `t` belongs to the time interval of this driving activity.

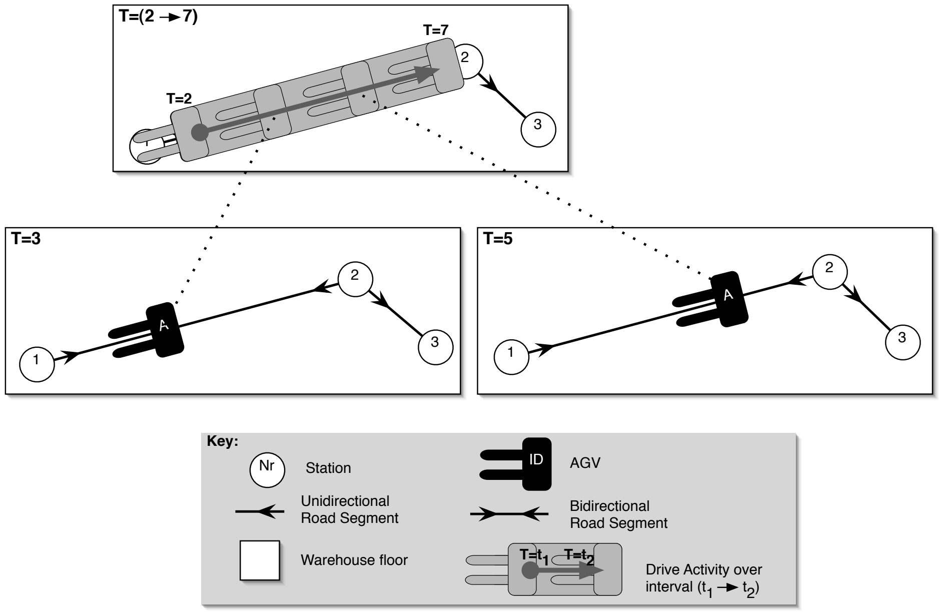

T=(2 → 7)
T=7
T=2
T=3
T=5
1
2
3
A
Key:
Nr
Station
ID
AGV
Unidirectional Road Segment
Bidirectional Road Segment
Warehouse floor
T=t1
T=t2
Drive Activity over interval (t1 → t2)

Figure 6.6: Example of a drive activity in the simulated warehouse environment.

In Figure 6.6, an example of a drive activity of AGV A over time interval $(2 \rightarrow 7)$ is depicted. We depict each drive activity as a hull that wraps the intermediate positions that are taken by the AGV over time. In Figure 6.6, the evolution represented by the drive activity is illustrated using two snapshots of the AGV within time interval $(2 \rightarrow 7)$: `getAGVSnapshot(3)` and `getAGVSnapshot(5)`, showing the instant position of the AGV at time instants 3 and 5 respectively.

6.5.2 Designing a Collision Law

We describe a way of designing a collision law. We focus on detecting whether and when collisions occur.

Snapshot-Based Collision Detection: Example. Figure 6.7 illustrates a collision law by means of an example. The example comprises two AGVs. `AGV B` is involved in a driving activity over time interval $(1 \rightarrow 7)$. `AGV C` is involved in a driving activity over time interval $(4 \rightarrow 9)$.

To detect collisions, a collision law takes consecutive snapshots of the intermediate positions of AGVs and determines whether the bounding boxes of the AGVs overlap. Figure 6.7 depicts six snapshots with a one second snapshot interval. In the snapshot at time instant $T = 6$, a collision occurs as the bounding boxes of both AGVs overlap. When detecting a collision, the collision law proposes an activity transformation that preempts both driving activities at the moment the collision occurs. The right hand side of Figure 6.7 illustrates the result of applying the activity transformation. `AGV B` and `AGV C` are involved in driving activities that end at time instant $T = 6$, i.e. the time instant the collision occurs.

The time interval between two consecutive snapshots determines the accuracy of detecting collisions. Suppose we want to detect overlap of the bounding boxes of AGVs with an accuracy of 10 centimeter. In case AGVs drive at a maximum speed of 1 meter per second, it takes an AGV 0.1 seconds to move over 10 centimeter. In case two AGVs travel at top speed, their relative position changes at a maximum rate of 2 meters per second. Consequently, to detect collisions with an accuracy of 10 centimeter, a snapshot to detect collisions must happen at least every 0.05 seconds.

Snapshot-Based Collision Detection: Implementation. Figure 6.8 shows the implementation of the `applyLaw()` method of the `CollisionLaw` class. `applyLaw()` uses the principle of snapshot-based collision detection.

The `applyLaw()` method is implemented as a nested loop.

- The outer `while`-loop is executed once for each snapshot time.
- The inner nested `for`-loop performs pairwise checks to determe whether the bounding boxes of some of the AGVs overlap at the snapshot time. The

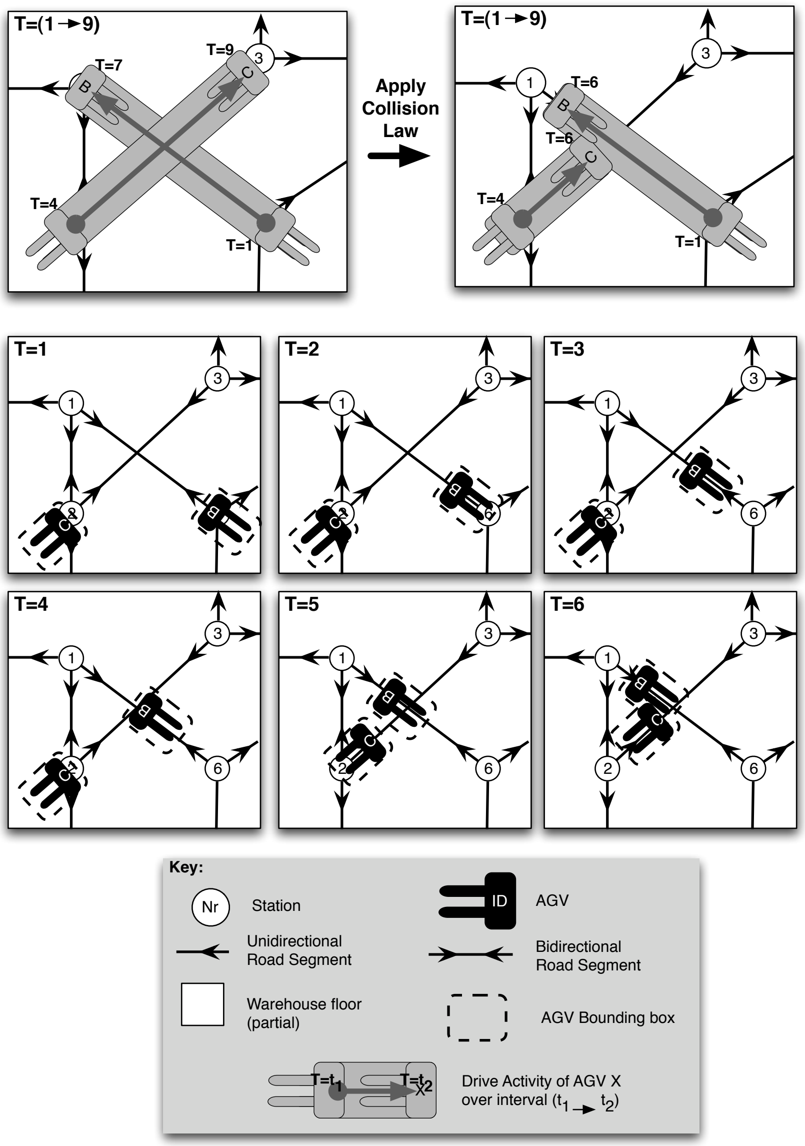

T=(1→9)
T=7
T=9
T=4
T=1
Apply Collision Law
T=6
T=1
T=2
T=3
T=4
T=5
T=6
Key:
Station
AGV
Unidirectional Road Segment
Bidirectional Road Segment
Warehouse floor (partial)
AGV Bounding box
Drive Activity of AGV X over interval (t1 → t2)

Figure 6.7: Collision detection based on snapshots. The bounding boxes of the AGVs overlap at time t=6, indicating a collision.

```
/**
 * Returns an array of activity transformations that occur within a
 * given time interval.
 *
 * @param       inspector The environment inspector
 * @param       interval The time interval in which this law should
 *              check for occurrences of collisions
 * @return      An array of activity transformations
 */
public ActivityTransformation[] applyLaw(EnvironmentInspector inspector,
           TimeInterval interval) {

  //This law is executed timestep based. The timestep
  //determines the accuracy of detecting collisions

  TimePoint snapshotTime=interval.getStartTime();
  AGV[] agvs = inspector.getAGVs();
  BoundingBox box1;
  BoundingBox box2;
  Vector<Collision> collisions = new Vector<Collision>();

  //loop to check snapshots within the given time interval for collisions
  while (! snapshotTime.after(interval.getEndTime())) {
      //nested loop to check collisions
      for (int i = 0; i < agvs.length; i++) {
           //retrieve the bounding box of the AGV
           box1=inspector.getState(agvs[i],snapshotTime).getBoundingBox();
           //check for overlap with bounding boxes of remaining AGVs
           for (int j = i+1; j < agvs.length; j++){
                box2 = inspector.getState(agvs[j],snapshotTime).getBoundingBox();
                //check whether a collision occurs
                if (box1.overlapsWith(box2)){
                     collisions.add(new Collision(agvs[i],agvs[j],snapshotTime));
                }
           }
      }
      //increase the snapshot time with a given interval
      //the snapshot interval determines the accuracy of detecting collisions
      snapshotTime.increase(getSnapshotInterval());
   }
  //determine activity transformations for the collisions that were detected.
  return getActivityTransformations(collisions);
}
```

Figure 6.8: Extract of Java code: the applyLaw() method of the class CollisionLaw.

`EnvironmentInspector` is responsible for composing the snapshots of the AGVs, as described in Section 5.4.

The `applyLaw()` method relies on the `getActivityTransformations()`

method to return an activity transformation for each of the detected collisions. The activity transformation of a collision encapsulates (1) the removal of the driving activity/activities that are involved in the collision and (2) the addition of a new driving activity or new driving activities that represent the movement of the AGV(s) until the time instant the collision occurs.

6.6 Evaluating the AGV Simulator

In the previous sections, we illustrated the way the modeling framework and the simulation platform underpin the *development* of the AGV simulator. We now focus on the AGV simulator and evaluate its flexibility and performance.

In Section 6.6.1, we discuss the flexibility of the AGV simulator. We demonstrate both flexibility and performance of the AGV simulator by means of experiments in Section 6.6.2. Finally, in Section 6.6.3 we discuss research on AGV control applications in the EMC^2 project that was supported by the AGV simulator.

6.6.1 Flexibility of the AGV simulator

To use the AGV simulator, the developer specifies the characteristics of the AGV control software and the simulated warehouse environment.

- To embed the AGV control software in the simulation, the developer specifies the execution time of the control software. This is done by identifying duration primitives and configuring a duration mapping for these primitives. The control primitives and the control mapping are predefined for all E'nsor control primitives.

- The developer can specify the characteristics of the simulated warehouse environment in which the control software is embedded. Besides the physical setup of the warehouse (i.e. the number and positioning of AGVs, nodes, segments, etc.), the developer can also select appropriate environment sources of dynamism (e.g. the transport generator), reaction laws and interaction laws.

Flexibility of the AGV simulator is important to enable experiments with AGV control software of which the functionality is not yet fully operational. We give a number of examples of core parts of the AGV simulator that can be customized to suit the needs of a particular simulation study.

- The *battery law* can be disabled when performing tests with AGV control software of which the battery charging functionality is not yet operational. This prevents AGVs from running out of energy.

- The quality of service of the communication channel can be adjusted by means of the *WiFi QoS law*. Disabling this law ensures reliable transmission of all messages. To simulate degraded quality of service of the communication channel, the law can be configured with the desired behavior, e.g. reduced communication range, message loss, message delay, etc.

- Collision detection can be configured by means of the *collision law*. The collision law can be configured with the accuracy that is required for detecting collisions. By deactivating the collision law, AGVs can drive across the warehouse without affecting each other.

- The activities can be customized to reflect the physical characteristics of the AGVs. For example, *driving activities* encapsulate the specific velocity or acceleration profile of the AGVs.

- The transport profile of the transport generator can be customized to suit the needs of a particular simulation study.

6.6.2 Measurements of the AGV Simulator

We measure the performance of the AGV simulator and demonstrate its flexibility. We focus the collision law of Section 6.5.2, as this law is a dominant factor for the performance of the AGV simulator. It is not our goal to apply optimizations to increase the performance of the AGV simulator.

6.6.2.1 Setup of the Experiments

The goal of the experiments is to illustrate both flexibility and performance of the collision law in the AGV simulator. We performed experiments with 4 different configurations with respect to the collision law:

1. The collision law *deactivated*. In this particular configuration, the collision law is not used in the simulation. This setting is typically used in simulation studies in which collision avoidance is out of focus.

2. The collision law configured with an accuracy of *10 centimeters*. As AGVs drive at a maximum speed of 1 meter per second, it takes an AGV 0.1 seconds to move over 10 centimeter. In case two AGVs travel at top speed, their relative position changes at a maximum rate of 2 meters per second. Consequently, to detect collisions with an accuracy of 10 centimeter, the snapshot frequency is 0.05 seconds.

3. The collision law configured with an accuracy of *25 centimeters*. This corresponds to a snapshot frequency of 0.125 seconds.

4. The collision law configured with an accuracy of *100 centimeters*. This corresponds to a snapshot frequency of 0.5 seconds.

The setup of the experiments is the following:

- The warehouse consists of 40 stations connected by 69 segments over an area of 1400 by 900 meters.
- The number of AGVs varies from 2 to 12. These are typical sizes of AGV warehouse transportation systems.
- We use lightweight AGV controllers. This enables us to measure the computation time consumed by the AGV simulator itself, with minimal bias from the controllers that are embedded in it. Each AGV controller is programmed to poll the status of its AGV every second. AGVs drive around randomly: as soon an AGV controller notices it has reached the next station, it randomly selects a next segment to drive on. AGVs rely on segment locking for avoiding collisions.

The simulations are executed on an Intel Pentium 4 computer with a processor speed of 2.8GHz and a 512MB of memory.

6.6.2.2 Measurements

Figure 6.9 shows the measured performance of the AGV simulator for each of the four configurations of the collision law discussed above. Each configuration of the collision law was tested in 11 different settings, i.e. from 2 to 12 AGVs. Each point in the graph is the average of 40 measurements, of which the 99% confidence interval is depicted. We discuss a number of observations.

From the measurements it is clear that the AGV simulator enables simulation runs faster than real-time. Even for detecting collisions of 12 AGVs with an accuracy of 10 centimeters, the *simulation speedup* is about factor 5, i.e. to simulate 100 seconds of (simulation) time in the AGV transportation system, the computer consumes about 20 seconds of wallclock time.

From the measurements it is clear that the collision law dominates the performance of the AGV simulator. The configuration in which the collision law is deactivated scales linearly as the number of AGVs increases, whereas all configurations with the collision law activated scale quadratically with the number of AGVs. This is within the line of expectations, as the complexity of the collision law is $O(n^2)$, with n the number of AGVs. This can be derived from its implementation depicted in Figure 6.8, which contains a double `for`-loop over the AGVs.

The performance that can be gained by reducing the accuracy of the collision law seems less than expected. For example, the snapshot frequency of the 10 centimeter law is 2.5 times smaller than the snapshot frequency of the 25 centimeter law, i.e. 0.05 seconds versus 0.125 seconds respectively. So one could expect a

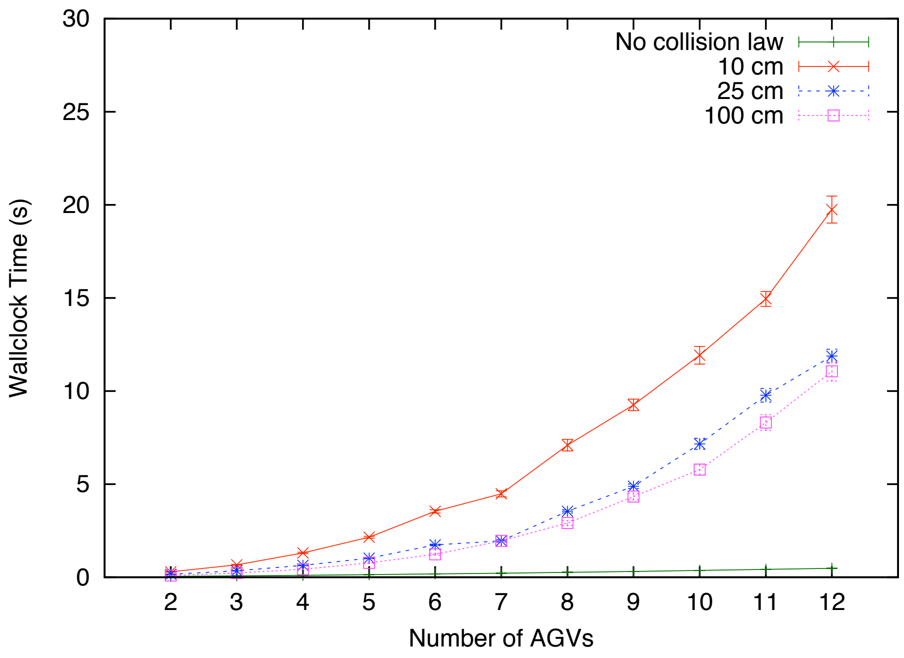


Figure 6.9: Performance (in seconds of wallclock time) for simulating 100 seconds of simulation time with the AGV simulator. The four lines correspond to four different configurations of the collision law: the collision law deactivated and the collision law detecting with an accuracy of 10 centimeters, 25 centimeters and 100 centimeters respectively. Each point in the graph is the average of 40 measurements, of which the 99% confidence interval is depicted.

performance difference of factor 2.5. Nevertheless, the difference in performance between both is only about a factor two. Moreover, the snapshot frequency of the 25 centimeter law is 4 times smaller than the snapshot frequency of the 100 centimeter law, i.e. 0.125 seconds versus 0.5 seconds respectively. However, the difference in performance between both is small, and certainly not a factor 4. This diminished performance gain of more coarse-grained laws can be explained as follows. From the description of the evolution of the model in Section 3.7, it is clear that the laws must be checked each time new influences are available. In our experiments, each AGV controller interacts with the environment on average once a second. Consequently, for an experiment with n AGVs, on average n AGVs access the environment during each second of simulation time. This means that new influences are generated on average each $1/n$ second of simulation time.

Consequently, the collision law is triggered on average each $1/n$ second of simulation time, i.e. the start time of the argument `interval` of an invocation of the `applyLaw()` method (see Figure 6.8) is on average only $1/n$ seconds later than the start of the interval that was given as an argument in the previous invocation. Decreasing the accuracy of a collision law only leads to a significant gain of performance as long as the snapshot frequency remains significantly smaller than the arrival rate of the influences. For example, for 8 AGVs each acting on average once every second, there is little use of employing a collision law with a snapshot interval larger than 0.125 seconds.

6.6.3 EMC^2 Research Supported by the AGV Simulator

The AGV simulator was extensively used during the development of AGV controllers in the EMC^2 project. The AGV simulator provides the necessary support for a developer to evaluate different functionalities an AGV control application in isolation, or to compare alternative solutions. We give a number of examples.

- *Virtual environment based routing* [WSH05]. In this approach, AGV controllers use a middleware, called virtual environment, for routing purposes. The virtual environment provides a graph-like map of the paths through the warehouse that the AGV controllers use for routing. Signs on the map specify the cost for the AGVs to drive to a given destination. To warn other AGVs that certain paths are blocked or have a long waiting time, AGV controllers mark segments with a dynamic cost on the map in the virtual environment. The middleware ensures consistency of the state of the virtual environment on neighboring AGVs. The simulated warehouse environment enables AGV controllers to drive over the warehouse layout and it handles the exchange of messages of the middleware.

- *Hull-based collision avoidance.* [WSHL05] AGV controllers avoid collisions by coordinating with other AGVs using the virtual environment. AGV controllers mark the path they are going to drive using hulls in their virtual environment. The hull of an AGV is the physical area the AGV occupies. A series of hulls then describes the physical area an AGV occupies along a certain path. If the area is not marked by other hulls (the AGV's own hulls do not intersect with others), the AGV can move along and actually drive over the reserved path. Afterwards, the AGV removes the markings in the virtual environment. In case of a conflict, the virtual environments execute a mutual exclusion protocol to determine which of AGVs involved can move on. The simulated warehouse environment handles the exchange of messages between virtual environments.

- *Field-based transport assignment* [WBH06, Sch05]. In this approach, transport tasks emit fields into the virtual environment that attract idle AGVs.

To avoid multiple AGVs driving towards the same transport, AGVs emit repulsive fields. AGVs combine received fields and follow the gradient of the combined fields that guide them towards locations of transports. The AGVs continuously reconsider the situation in the environment and task assignment is delayed until the load is picked, which improves the flexibility of the system. The simulated warehouse environment provides the infrastructure to add new tasks in the system and it handles the exchange of messages to spread fields in the virtual environment.

- *Protocol-based transport assignment* [WBHS06]. Besides field-based transport assignment, a dynamic version of the Contract Net protocol [Smi80] was developed to assign transports to AGVs. This protocol, called DynCNET, allows AGV controllers to reconsider the assignment of transports while they drive towards a transport. An extensive series of simulation tests with real world warehouse layouts and order profiles show that both approaches have similar performance characteristics.

The AGV simulator also supports evaluating the integration of different functionalities of an AGV control application. For example, a modular AGV controller [DL07] was developed that manages combinations of functionalities. A combination consists of a particular approach for routing, a particular approach for collision avoidance, a particular approach for transport assignment and/or a particular approach for battery charging.

6.7 Conclusions

In this chapter, we applied the modeling framework and the simulation platform in an industrial case. We developed an AGV simulator that supports software-in-the-loop simulation of distributed control applications that control AGVs in warehouse environments.

The *modeling framework* underpins the simulation model of the AGV simulator. The constructs of the modeling framework are used to capture key characteristics of the AGV system in a first-class manner. A developer can adapt the model of the AGV simulator to the needs of a particular simulation study by activating, deactivating and customizing first-class elements of the simulation model.

The *simulation platform* underpins the execution of the AGV simulator. The simulation platform encapsulates the functionality to execute simulation models that are customized for a particular AGV simulation study.

A real AGV control application encapsulates several complex functionalities, such as routing, collision avoidance, transport assignment and battery charging. These functionalities are typically developed incrementally, focussing on particular functionalities and abstracting from others. The AGV simulator provides the necessary support for the developer (1) to evaluate different functionalities of the AGV controllers in isolation, (2) to compare several approaches for each functionality, and (3) to enable the systematic integration of the functionalities.

Chapter 7

Conclusions

The research presented in this dissertation focusses on the development of software-in-the-loop simulations for distributed control application in dynamic environments. This family of simulations has the following characteristics:

- The environment to-be-simulated is dynamic. In a dynamic environment, the operating conditions of a distributed control application are continuously changing.
- The control software of the real distributed control application is embedded in the simulation.

We started from the observation that developing this family of simulations is complex. An important reason for this complexity is the lack of special-purpose modeling constructs to support such simulations. Existing support comprises either (1) general-purpose modeling constructs of which the meaning is formally specified, but which fail to provide support explicitly targeted at this family of simulations, or (2) informal abstractions of which the meaning is implicit and coupled to the design of a simulation platform.

In this dissertation, we presented an approach to support the development of such simulations. Core ingredients of the approach are special-purpose modeling constructs for this family of simulations, an explicit modeling framework that formally specifies these constructs and a simulation platform that supports the modeling constructs in an executable simulation. These ingredients enable a developer to handle the complexity that is involved in building the target family of simulations as follows.

Our approach enables a developer to disentangle the *model formulation phase* from the *model translation phase*. The model formulation phase is supported by special-purpose modeling constructs that are formally specified to enable a modeler to formulate a simulation model without taking into account a particular simulation platform. The model translation phase is supported by means of a simulation platform that can be used to execute simulation models expressed in terms of the modeling constructs.

In the remainder of this concluding chapter, we first summarize the contributions of our research in Section 7.1. Afterwards, we put forward suggestions for further research in Section 7.2 and a closing reflection in Section 7.3.

7.1 Contributions

The main contribution of the research described in this dissertation is the introduction of an explicit modeling framework for software-in-the-loop simulations of distributed control applications. The modeling framework offers constructs for formulating a simulation model for this family of simulations and captures core characteristics of these simulations in a first-class manner. Moreover, the modeling constructs are formally specified, which is crucial to decouple the simulation model from the simulation platform to execute the model.

Specific contributions of the research described in this dissertation are (1) the introduction of a modeling framework with special-purpose modeling constructs to support the development software-in-the-loop simulations of distributed control applications in dynamic environments, (2) the development of a formal specification of the modeling framework, (3) the development of a simulation platform that supports the modeling constructs in an executable simulation, and (4) the evaluation of the usability of all constructs of the modeling framework in an industrial case. We elaborate on each of the contributions.

The introduction of a modeling framework for software-in-the-loop simulations of distributed control applications in dynamic environments [HHB05, HVUM07, HHW04a, HHW04b]. The modeling framework enables a developer to capture key characteristics of this family of simulations in a first-class manner. The modeling framework comprises two complementary parts:

- The environment part [HHB05, HVUM07] of the modeling framework comprises modeling constructs that capture in an explicit manner a number of key characteristics and relations that are pertinent for modeling dynamic environments of distributed control applications:
  - A dynamic environment has a particular *scope*. We put forward `environmental entities`, `environmental properties` and `environment layout` as modeling constructs to capture all constituting parts in the environment and the way they are arranged with respect to each other.

  - A dynamic environment *encapsulates dynamism.* We put forward `activities` to capture the evolution of all entities in the environment in an explicit manner.
  - A dynamic environment *embeds sources of dynamism.* We put forward `controllers` and `environment sources` to capture sources of dynamism that are part of the distributed control application and sources of dynamism that are external to the distributed control application, respectively. Controllers and environment sources are embedded in environmental entities. Controllers and environment sources are restricted in their ability to affect dynamism in the environment. We put forward `influences` to reify the attempt of a source of dynamism to affect the environment.
  - A dynamic environment *regulates dynamism.* We put forward `reaction laws` to capture the way the environment reacts to influences originating from the sources of dynamism, and `interaction laws`, to capture interactions of dynamism in the environment.

- The control application part [HHW04a, HHW04b] of the modeling framework comprises modeling constructs that capture key characteristics of the software of the distributed control application that is embedded in the simulation:
  - The software of a distributed control application has a particular *execution time* in the real world. We put forward `duration primitives` and a `duration mapping` as modeling constructs to specify the real-world execution time of a distributed control application in an explicit model.
  - The software of a distributed control application has a particular *control interface* to access the environment. We put forward `control primitives` and a `control mapping` to capture the control interface and the influences that result from invoking that interface in the simulation model.

The development of a formal specification of the modeling framework [HVUM07, HHWB05a]. We developed a formal specification based on set theory to underpin the modeling framework. The formal specification decouples the modeling constructs from their implementation in a specific simulation platform. The advantage of the formal specification is twofold.

On the one hand, the formal specification enables using the modeling constructs for formulating a simulation model while making abstraction of the simulation platform to execute the model. The formal specification unambiguously specifies the meaning of the modeling constructs, and describes the way the constructs are related to each other.

On the other hand, the formal specification enables a developer to consider several design alternatives for translating a simulation model into an executable simulation. The formal specification specifies the functionality that is needed to support the constructs in an executable simulation, without commitment to particular design decisions. As such, the formal specification guides the development of an executable simulation and prevents reinventing its functionality from scratch.

The development of a simulation platform that supports the modeling constructs [HHW04b, WHH05]. We developed a simulation platform to demonstrate that the modeling framework is feasible for developing executable simulations. The simulation platform encapsulates the functionality to support the modeling constructs in an executable simulation. The simulation platform supports simulations (1) in which the software of real controllers can be embedded, and (2) of which the simulation model is described in terms of the proposed modeling constructs.

We put forward an architecture for such a simulation platform, and documented this architecture using several architectural views. The top-level module decomposition view of the architecture of the simulation platform comprises three main modules that each encapsulate a core functionality of the simulation platform:

1. The `Simulated Environment` module is responsible for managing the model of the environment. This module encapsulates all functionality to support the modeling constructs of the environment part of the modeling framework.

2. The `Execution Tracker` module is responsible for tracing the execution of the software of a real controller of the distributed control application. This module encapsulates all functionality to support the modeling constructs of the control application part of the modeling framework.

3. The `Simulation Engine` module is responsible for managing the evolution of all parts of the simulation in correspondence to the specifications of the simulation model. The simulation engine encapsulates all functionality to synchronize the progress of the simulated environment with the progress of all execution trackers of the controllers that are embedded in the simulation.

The architecture uses aspect technology for plug-and-play integration of the control software in a simulation. Aspect technology is used to weave all tracing functionality required for the simulation into the control software. This contributes to a clean separation between application concerns and simulation concerns.

A validation in an industrial case [HHB06, HHWB05b]. We applied the modeling framework and the simulation platform in an industrial case. We developed an AGV simulator that supports software-in-the-loop simulation of distributed control applications that control AGVs in warehouse environments.

The modeling framework underpins the simulation model of the AGV simulator. The constructs of the modeling framework are used to capture key characteristics of the AGV system in a first-class manner. A developer can adapt the model of the AGV simulator to the needs of a particular simulation study by activating, deactivating and customizing first-class elements of the simulation model.

The simulation platform underpins the execution of the AGV simulator. The support offered by the simulation platform enables executing simulation models that are customized for a particular simulation study.

A real AGV control application encapsulates several, complex functionalities, such as routing, collision avoidance, transport assignment and battery charging. These functionalities are typically developed incrementally, focussing on particular functionalities and abstracting from others. The AGV simulator provides the necessary support for the developer (1) to evaluate different functionalities of the AGV controllers in isolation, (2) to compare several approaches for each functionality, and (3) to enable the systematic integration of the functionalities.

7.2 Future work

We give a number of directions for future research.

Extending the Modeling Framework. The modeling framework could be extended with additional constructs. We suggest a number of avenues for future research.

- *Supporting perception.* Currently, the modeling framework does not provide modeling constructs to capture the way a distributed control application senses or perceives the environment. Consequently, modeling perception must still be tackled by the modeler without explicit support. In analogy with dynamism, perception is not limited to a mere state snapshot of the environment. Perception is affected by the environment [WSH04]. For example, a camera cannot perceive entities that are positioned behind other entities, and its perception could be affected by the amount of ambient light. Moreover, perception is not limited to a static state snapshot of a part of the environment, but closely related with dynamism. For example, sensors can be capable of registering the movement of entities in the environment, rather than their momentary position. Investigating the relation between perception and dynamism is an interesting challenge.

- *Supporting environment sources.* Currently, the modeling framework does not provide explicit modeling constructs to model the internals of an environment source. An environment is a black box source of influences, and its behavior is specific for a particular simulation study. More elaborate support for environment sources could focus on modeling constructs for various kinds of behaviors, such as reactive [Bro91, WH06], behavior-based [Mae91, WSHG05] or cognitive behaviors [HS96, RG95].

Supporting Model Transformations. The modeling framework offers explicit modeling constructs for modeling software-in-the-loop simulations of distributed control applications in dynamic environments. The resulting simulation model is independent of the design of a particular simulation platform. A model transformation would enable transforming a simulation model described in terms of the modeling constructs of the modeling framework into a simulation model described in terms of general-purpose modeling constructs. As such, general-purpose simulation platforms can be reused to support an executable simulation. Supporting model transformations in an explicit manner poses an interesting challenge (as was discussed in Section 3.8.2).

Model transformations are closely related to *MDA (Model-Driven Architecture)* [Gro03, KWB03], a software development approach that relies on a platform independent model (PIM) to reify the conceptual design of the functionality of a system in a way that is independent of a particular implementation technology. The technical realization is supported by *model transformations* that translate the PIM into one or several platform specific models (PSM), that support a particular technology (e.g. CORBA, .Net) to run on computers.

Extending the Simulation Platform. We indicate two directions for extending the simulation platform.

- *Distribution.* Currently, the simulation platform does not incorporate support for distribution. The challenges of distributing the simulation platform are not of pure technical nature: as all parts of the simulation are explicitly synchronized with the simulation engine, they could technically be distributed across different hosts. The main challenge is determining which distribution scheme is most suitable for a particular simulation study. On the one hand, distribution adds computing power which speeds up a simulation, on the other hand, distribution requires synchronization to happen over a network, which slows down a simulation. Simulations with computation-intensive controllers should benefit more from an expanded distribution scheme than lightweight, highly interactive controllers. Consequently, distribution of simulations should be supported in a flexible manner [EHU06], with distribution schemas that can be adapted or self-adapt to a particular simulation study.

- *Execution tracing.* Currently, the simulation platform relies on AspectJ to trace the execution of each controller. However, AspectJ can only trace the execution of duration primitives up to a particular level of granularity. An interesting challenge is developing support to trace the execution of more fine-grained duration primitives, e.g. at the level of machine instructions.

7.3 Closing Reflection

Formulating a simulation model is a matter of identifying the core characteristics or features of the real system that are sufficient to serve the purpose of a specific simulation study. A model should neither oversimplify the system nor carry so much detail that is becomes costly to build and run. Therefore, simulation modeling is often referred to as an art, instead of a science [Sha98].

However, as the demand for distributed control applications increases, more and more simulations are built to support their development. The way the simulation models for such systems are constructed, becomes common knowledge. This kind of common knowledge can be reified in a modeling framework.

For software-in-the-loop simulations of distributed control applications in dynamic environments, simulations have been studied and built for a long time. The efforts of many prominent researchers have laid the foundation on which our research contributions are built. As such, the modeling framework we put forward reifies the knowledge and expertise we have acquired during our research that is founded on decades of domain maturing.

The modeling framework demonstrates the way knowledge and practices with simulating distributed control applications in dynamic environments can be systematically documented and maturated in a form that has proven its value for simulation development. Therefore, we believe that simulation modeling is an art *supported* by science.

Bibliography

[AC96] Scott D. Anderson and Paul R. Cohen. Timed Common Lisp: the duration of deliberation. *SIGART Bull.*, 7(2):11–15, 1996.

[And97] Scott D. Anderson. Simulation of multiple time-pressured agents. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 397–404, 1997.

[And00] John Anderson. A generic distributed simulation system for intelligen agent design and evaluation. In *10th International Conference on AI, Simulation, and Planning in High Autonomy Systems*, pages 36–44, 2000.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *Computer Graphics*, 28(Annual Conference Series):23–34, 1994.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, April 2003.

[BCNN00] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation.* Prentice-Hall, Upper Saddle River, N.J., third edition, 2000.

[BDD03] Michael Batty, Jake Desyllas, and Elspeth Duxbury. The discrete dynamics of small-scale spatial events: Agent-based models of mobility in carnivals and street parades. *International Journal of Geographical Information Science*, 17(7):673–697, 2003.

[BHvR05] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576, 2005.

[BJNT06] Ansgar Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors. *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Computer Science.* Springer, 2006.

[BKB$^+$05] Neil Byrne, Peter Koonce, Robert L Bertini, Chris Pangilinan, and Matt Lasky. Using hardware-in-the-loop simulation to evaluate signal control strategies for transit signal priority. *Transportation Research Record: Journal of the Transportation Research Board*, (1925):227–234, 2005.

[BKW06] Thomas Bräunl, Andreas Koestler, and Axel Waggershauser. Fault-tolerant robot programming through simulation with realistic sensor models. *International Journal of Advanced Robotic Systems*, 3(2):99–106, 2006.

[BMP$^+$] Günter Bruns, Peter Mössinger, Daniel Polani, Ralf Schmitt, Rene Spalt, Thomas Uthmann, and Stefan Weber. Xraptor - a simulation environment for continuous virtual multi-agent systems - user manual.

[Bor06] Jan Borgers. "Hoe realistisch is een simulatie?": Studie aan de hand van LEGO Mindstorms. Master's thesis, K.U.Leuven, Department of Computer Science, 2006.

[Bro91] Rodney A. Brooks. Intelligence without reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Bru00] Sven A. Brueckner. *Return From The Ant - Synthetic Ecosystems For Manufacturing Control.* PhD thesis, Humboldt University Berlin, Department of Computer Science, 2000.

[BS91] R.L. Bagrodia and C.-C. Shen. Midas: Integrated design and simulation of distributed systems. *IEEE Transactions on Software Engineering*, 17(10):1042–1058, 1991.

[BT03] Brett Browning and Erick Tryzelaar. Ubersim: A realistic simulation engine for robot soccer. In *Proceedings of Autonomous Agents and Multi-Agent Systems, AAMAS'03*, Australia, July 2003.

[Car03] John S. Carson. Introduction to simulation: introduction to modeling and simulation. In *Winter Simulation Conference*, pages 7–13, 2003.

[CBB$^+$02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, September 2002.

[CK99] S. G. Choi and W. H. Kwon. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Proc. of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 115–119, 1999.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

[Cla96] Andy Clark. *Being There: Putting Brain, Body, and World Together Again.* MIT Press, Cambridge, MA, USA, 1996.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[DL07] Wouter Delbaere and Bart Lamberigts. Ontwikkeling van een gedecentralizeerd controle systeem voor autonome voertuigen. Master's thesis, Katholieke Universiteit Leuven, Belgium, 2007.

[DS05] Kurt Dresner and Peter Stone. Multiagent traffic management: An improved intersection control mechanism. In Frank Dignum, Virginia Dignum, Sven Koenig, Sarit Kraus, Munindar P. Singh, and Michael Wooldridge, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, NY, July 2005. ACM Press.

[DYP06] Pedro DeLima, George York, and Daniel Pack. Localization of ground targets using a flying sensor network. *sutc*, 1:194–199, 2006.

[EHU06] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *WSC '06: Proceedings of the 37th conference on Winter simulation*, pages 848–855. Winter Simulation Conference, 2006.

[EK04] Joel M. Esposito and Vijay Kumar. An asynchronous integration and event detection algorithm for simulating multi-agent hybrid systems. *ACM Trans. Model. Comput. Simul.*, 14(4):363–388, 2004.

[EKP01] Joel Esposito, Vijay Kumar, and George J. Pappas. Multi-agent hybrid simulation. In *Proceedings of IEEE Conference on Decision and Control*, December 2001.

[FBT$^+$03] D. Finkenzeller, M. Baas, S. Thüring, S. Yigit, and A. Schmitt. Visum: a vr system for the interactive and dynamics simulation of mechatronic systems. In *Virtual Concept 2003*, Biarritz, France, November 2003.

[FM96] J. Ferber and J.P. Müller. Influences and reaction: A model of situated multiagent systems. In *Proceedings of the Second International Conference on Multi-agent Systems*, pages 72–79. AAAI Press, 1996.

[FMM77] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1977.

[FT94] Alois Ferscha and Satish K. Tripathi. Parallel and distributed simulation of discrete event systems. Technical report, University of Maryland, College Park, MD, USA, 1994.

[Fuj98] R. Fujimoto. Time management in the high level architecture. *Simulation, Special Issue on High Level Architecture*, 71(6):388–400, 1998.

[GH04] D. Gu and H. Hu. Teaching robots to coordinate their behaviours. In *Proceedings of IEEE International Conference on Robotics and Automation*, New Orleans, LA, May 2004. Riverside Hilton & Towers.

[GL04] William Glover and John Lygeros. A stochastic hybrid model for air traffic control simulation. In Rajeev Alur and George J. Pappas, editors, *HSCC*, volume 2993 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[GLM96] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[Gom01] M. Gomez. Hardware-in-the-loop simulation. Embedded Systems Programming, December 2001.

[GPDV06] O. Gietelink, J. Ploeg, B. De Schutter, and M. Verhaegen. Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations. *Vehicle System Dynamics*, 44(7):569–590, July 2006.

[Gro03] Object Management Group. Mda guide version 1.0.1. Misc, 2003.

[GVH03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR 2003*, pages 317–323, Coimbra, Portugal, June 2003.

[HBZ90] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time-dependent parametric surfaces. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics* *and interactive techniques*, pages 39–48, New York, NY, USA, 1990. ACM Press.

[HCd05] Simon Hallé and Brahim Chaib-draa. A Collaborative Driving System based on Multiagent Modelling and Simulations. *Journal of Transportation Research Part C (TRC-C): Emergent Technologies*, 13(4):320–345, 2005.

[HHB05] Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Simulating actions in dynamic environments. In *Conceptual Modeling and Simulation Conference, CMS2005, Track on Agent Based Modeling and Simulation in Industry and Environment*, 2005.

[HHB06] Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Testing AGVs in Dynamic Warehouse Environments. In D. Weyns, V. Parunak, and F. Michel, editors, *Environments for Multiagent Systems II*, volume 3830 of *Lecture Notes in Computer Science*, pages 270–290. Springer-Verlag, 2006.

[HHW04a] Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management adaptability in multi-agent systems. In *Proceedings of the AISB 2004 Fourth Symposium on Adaptive Agents and Multi-Agent Systems*, pages 20–30. University of Leeds, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 2004.

[HHW04b] Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management support for simulating multi-agent systems. In *Joint workshop on multi-agent and multi-agent-based simulation*, pages 31–40. Columbia University, 2004.

[HHWB05a] Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Extending time management support for multi-agent systems. In *Multi-Agent and Multi-Agent-Based Simulation: Joint Workshop MABS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3415 / 2005 of *Lecture Notes in Computer Science*, pages 37–48. Springer-Verlag, GmbH, 2005.

[HHWB05b] Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Towards time management adaptability in multi-agent systems. In *Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning*, volume 3394 / 2005 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, GmbH, 2005.

[HJJ03] Ian J. Hayes, Michael Jackson, and Cliff B. Jones. Determining the specification of a control system from that of its environment. In

Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2003.

[HKSP03] Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido, and Wolfgang Pree. From control models to real-time code using giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

[HRU03] J. Himmelspach, M. Röhl, and A.M. Uhrmacher. Simulation for testing software agents - an exploration based on JAMES. In *Proc. of the 2003 Winter Simulation Conference, New Orleans, USA*, December 2003.

[HS96] Afsaneh Haddadi and Kurt Sundermeyer. Belief-desire-intention agent architectures. *Foundations of distributed artificial intelligence*, pages 169–185, 1996.

[HSKM97] Dirk Helbing, Frank Schweitzer, Joachim Keltsch, and Péter Molnár. Active walker model for the formation of human and animal trail systems. *Physical Review E*, 56(3):2527–2539, January 1997.

[Hub96] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[HVUM07] Alexander Helleboogh, Giuseppe Vizzari, Adelinde Uhrmacher, and Fabien Michel. Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems: Special issue on environments for multi-agent systems*, 14(1):87–116, February 2007.

[HW91] J. H. Hubbard and B. H. West. *Differential equations: a dynamical systems approach. Part I: ordinary differential equations.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[HZ05a] X. Hu and B. P. Zeigler. A simulation-based virtual environment to study cooperative robotic systems. *Integrated Computer-Aided Engineering (ICAE)*, 12(4):353 – 367, 2005.

[HZ05b] Xiaolin Hu and Bernard P. Zeigler. Model continuity in the design of dynamic distributed real-time systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(6):867–878, 2005.

[ICG$^+$04] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, 2004.

[Kam93] V. V. Kamat. A survey of techniques for simulation of dynamic collision detection and response. *Computers & Graphics*, 17(4):379–385, 1993.

[KB04] Franziska Klügl and Ana L. C. Bazzan. Route decision behaviour in a commuting scenario: Simple heuristics adaptation and effect of traffic forecast. *Journal of Artificial Societies and Social Simulation*, 7(1), 2004.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LGLM05] Andrea Lecchini, William Glover, John Lygeros, and Jan Maciejowski. Air-traffic control in approach sectors: Simulation examples and optimisation. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 433–448. Springer, 2005.

[LGS98] J. Lygeros, D. N. Godbole, and S. Sastry. Verified hybrid controllers for automated vehicles. *IEEE Transactions on Automatic Control*, 43(4):522—539, April 1998.

[LTS99] John Lygeros, Claire Tomlin, and Shankar Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, pages 349–370, 1999.

[Mae91] Pattie Maes. The agent network architecture (ana). *SIGART Bull.*, 2(4):115–120, 1991.

[Mic04] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[Mir98] Brian Mirtich. V-clip: fast and robust polyhedral collision detection. *ACM Trans. Graph.*, 17(3):177–208, 1998.

[Mir00] Brian Mirtich. Timewarp rigid body simulation. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 193–200, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.

[Mos99] Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 165–177, London, UK, 1999. Springer-Verlag.

[Neu04] Stephen Neuendorffer. Modeling real-world control systems: beyond hybrid systems. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 240–248. Winter Simulation Conference, 2004.

[NRSSV05] Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors. *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Computer Science*. Springer, 2005.

[OR04] Oliver Obst and Markus Rollmann. SPARK – A Generic Simulator for Physical Multiagent Simulations. In Gabriela Lindemann, Jörg Denzinger, Ingo J. Timm, and Rainer Unland, editors, *Multiagent System Technologies – Proceedings of the MATES 2004*, volume 3187, pages 243–257. Springer, September 2004.

[PR90] Martha Pollack and Marc Ringuette. Introducing the tileworld: experimentally evaluating agent architectures. In Thomas Dietterich and William Swartout, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Menlo Park, CA, 1990. AAAI Press.

[PVR04] Leslie Pack Kaelbling Paulina Varshavskaya and Daniela Rus. Learning distributed control for modular robots. In *International Conference on Intelligent Robots and Systems, Sendai, Japan*, 2004.

[RG95] A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proc. of 1st International Conference on Multi-Agent Systems (ICMAS)*, pages 313–319. AAAI Press/MIT Press, 1995.

[Ril03] Patrick Riley. MPADES: Middleware for parallel agent discrete event simulation. In Gal A. Kaminka, Pedro U. Lima, and Raul Rojas, editors, *RoboCup-2002: Robot Soccer World Cup VI*, number 2752 in Lecture Notes in Artificial Intelligence, pages 162–178. Springer Verlag, Berlin, 2003.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Roo99] D. Roozemond. Using intelligent agents for urban traffic control control systems. In *Proceedings of the International Conference on Artificial Intelligence in Transportation Systems and Science*, pages 69–79, 1999.

[RR03] Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference Proceedings*, volume 1, pages 817–825, 2003.

[Rut06] Matthew J. Rutherford. *Adequate System-Level Testing of Distributed Systems.* PhD thesis, Department of Computer Science, Univeristy of Colorado at Boulder, August 2006.

[SA06] Carlos M. Velez S. and Andres Agudelo. Control and parameter estimation of a mini-helicopter robot using rapid prototyping tools. *WSEAS Transactions on Systems*, 5(9):2250–2257, September 2006.

[SB99] Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 72–80. ACM Press, 1999.

[Sch05] Wannes Schols. Gradient Field Based Order Assignment in AGV Systems. Master's thesis, Katholieke Universiteit Leuven, Belgium, 2005.

[Sha98] Robert E. Shannon. Introduction to the art and science of simulation. In *Winter Simulation Conference*, pages 7–14, 1998.

[Smi80] R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, 1980.

[SPLK01] Rajarishi Sinha, Christiaan J. J. Paredis, Vei-Chung Liang, and Pradeep K. Khosla. Modeling and simulation methods for design of engineering systems. *J. Comput. Info. Sci. Eng.*, 1(1):84–91, 2001.

[S.S99] S.S.Sastry. *Nonlinear Systems: Analysis, Stability and Control.* Springer Verlag, New York, 1999.

[SSS+03] B. Sinopoli, C. Sharp, L. Schenato, S. Schaffert, and S. Sastry. Distributed control applications within sensor networks. In *Proceedings of the IEEE, Special Issue on Sensor Networks and Applications*, August 2003.

[SU01] Bernd Schattenberg and Adelinde M. Uhrmacher. Planning agents in james. *Proceedings of the IEEE*, 89(2):158–173, February 2001.

[TLSS00] C.J. Tomlin, J. Lygeros, and S. Shankar Sastry. A game theoretic approach to controller design for hybrid systems. *Proceedings of the IEEE*, 88(7):949–970, July 2000.

[TPS98] C. Tomlin, G. J. Pappas, and S. Sastry. Conflict resolution for air traffic management : A study in muti-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, April 1998.

[Uhr01] A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Trans. Model. Comput. Simul.*, 11(2):206–232, 2001.

[UK00] A.M. Uhrmacher and B.G. Kullick. "Plug and Test" - software agents in virtual environments. In *Proceedings of the 2000 Winter Simulation Conference*, volume 2, pages 1722–1729. Wyndham Palace Resort & Spa, Orlando, Florida, USA, December 2000.

[USC] The Network Simulator: NS2, http://www.isi.edu/nsnam/ns/.

[vdSS98] A.J. van der Schaft and J.M. Schumacher. Complementarity modeling of hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):483–490, April 1998.

[VGVVB06] Paul Verstraete, Bart Germain, Paul Valckenaers, and Hendrik Van Brussel. On applying the prosa reference architecture in multiagent manufacturing control applications. In *Multiagent Systems and Software Architecture*, 2006.

[VHL01] Regis Vincent, Bryan Horling, and Victor Lesser. An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator. *Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, 1887, January 2001.

[Wan05] Fei-Yue Wang. Agent-based control for networked traffic management systems. *IEEE Intelligent Systems*, 20(5):92–96, 2005.

[WBH06] D. Weyns, N. Boucké, and T. Holvoet. Gradient Field Based Transport Assignment in AGV Systems. In *5th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, Hakodate, Japan*, 2006.

[WBHS06] Danny Weyns, Nelis Boucké, Tom Holvoet, and Kurt Schelfthout. DynCNET: a protocol for flexible task assignment applied in an AGV transportation system. In *Proceedings of the Fourth European Workshop on Multi-Agent Systems*, volume 223, pages 359–370. Universidade de Lisboa & Do Minho, Portugal, Universidade de Lisboa & Do Minho, Portugal, 2006.

[WGW90] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 11–21, New York, NY, USA, 1990. ACM Press.

[WH06] Danny Weyns and Tom Holvoet. From reactive robotics to situated multiagent systems: A historical perspective on the role of environment in multiagent systems. In *Engineering Societies in the Agents World VI, Revised Selected and Invited Papers*, volume 3963 of *Lecture Notes in Computer Science*, pages 63–88. Springer, 2006. Invited paper.

[WHH05] Danny Weyns, Alexander Helleboogh, and Tom Holvoet. The Packet-World: A testbed for investigating situated multiagent systems. In *Software Agent-Based Applications, Platforms, and Development Kits*, Whitestein Series in Software Agent Technologies, pages 383–408. Birkhauser Verlag, Basel - Boston - Berlin, September 2005.

[Woo01] Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[WSH04] Danny Weyns, Elke Steegmans, and Tom Holvoet. Towards active perception in situated multi-agent systems. *Applied Artificial Intelligence*, 18(9-10):867–883, October 2004.

[WSH05] Danny Weyns, Kurt Schelfthout, and Tom Holvoet. Exploiting a virtual environment in a real-world application. In D. Weyns, V. Parunak, and F. Michel, editors, *2nd International Workshop on Environments for Multiagent Systems*, pages 1–18, Utrecht, The Netherlands, 2005.

[WSHG05] Danny Weyns, Kurt Schelfthout, Tom Holvoet, and Olivier Glorieux. Towards adaptive role selection for behavior-based agents.

In *Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning*, volume 3394 of *Lecture Notes in Computer Science*, pages 295–314. Springer-Verlag, GmbH, 2005.

[WSHL05] D. Weyns, K. Schelfthout, T. Holvoet, and T. Lefever. Decentralized control of E'GV transportation systems. In *4th Joint Conference on Autonomous Agents and Multiagent Systems, Industry Track*, Utrecht, The Netherlands, 2005. ACM Press, New York, NY, USA.

[WVvVK04] M. Wiering, J. Vreeken, J. van Veenen, and A. Koopman. Simulation and optimization of traffic in a city. In *IEEE Intelligent Vehicles symposium (IV'04)*, 2004.

[ZP00] Bernard Zeigler and Herbert Praehofer. *Theory of Modeling and Simulation.* Academic Press, January 2000.

List of Publications

Articles in international reviewed journals

1. Alexander Helleboogh, Giuseppe Vizzari, Adelinde Uhrmacher, and Fabien Michel. Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems: Special issue on environments for multi-agent systems*, 14(1):87–116, February 2007[1].

Parts of books

1. Danny Weyns, Alexander Helleboogh, and Tom Holvoet. The Packet-World: A testbed for investigating situated multiagent systems. In R. Unland, M. Klush, and M. Calisti, editors, *Software Agent-Based Applications, Platforms, and Development Kits*, Whitestein Series in Software Agent Technologies, pages 383–408. Birkhauser Verlag, Basel - Boston - Berlin, September 2005.

Contributions at international conferences, published in proceedings

1. Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Testing AGVs in Dynamic Warehouse Environments. In D. Weyns, V. Parunak, and F. Michel, editors, *Environments for Multiagent Systems II*, volume 3830 of *Lecture Notes in Computer Science*, pages 270–290. Springer-Verlag, 2006.

2. Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Towards time management adaptability in multi-agent systems. In D. Kudenko, D. Kazakov, and E. Alonso, editors, *Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning*, volume 3394/2005 of

[1] Journal Impact Factor 2.605, ISI Web of Knowledge

Lecture Notes in Computer Science, pages 88–105. Springer-Verlag, GmbH, 2005.

3. Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Simulating actions in dynamic environments. In F. Barros, A. Bruzzone, C. Frydman, and N. Giambiasi, editors, *Conceptual Modeling and Simulation Conference, CMS 2005, Track on Agent Based Modeling and Simulation in Industry and Environment*, 2005.

4. Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Extending time management support for multi-agent systems. In P. Davidsson, B. Logan, and K. Takadama, editors, *Multi-Agent and Multi-Agent-Based Simulation: Joint Workshop MABS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3415 / 2005 of *Lecture Notes in Computer Science*, pages 37–48. Springer-Verlag, GmbH, 2005.

5. Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management adaptability in multi-agent systems. In D. Kudenko, E. Alonso, and D. Kazakov, editors, *Proceedings of the AISB 2004 Fourth Symposium on Adaptive Agents and Multi-Agent Systems*, pages 20–30. University of Leeds, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 2004.

6. Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management support for simulating multi-agent systems. In P. Davidsson, L. Gasser, B. Logan, and K. Takadama, editors, *Joint workshop on multi-agent and multi-agent-based simulation*, pages 31–40. Columbia University, 2004.

7. D. Weyns, A. Helleboogh, E. Steegmans, T. De Wolf, K. Mertens, N. Boucké, and T. Holvoet, Agents are not part of the problem, agents can solve the problem, In C. Gonzales-Perez, editor, *Proceedings of the OOPSLA Workshop on Agent-Oriented Methodologies*, pages 101–112, 2004.

8. K. Schelfthout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns, Agent Implementation Patterns, In J. Debenham, B. Henderson-Sellers, N. Jennings, and J. Odell, editors, *Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, pages 119–130, 2002.

Biography

Alexander Helleboogh was born on January 21, 1978 in Bornem (Belgium). He received a *Bachelor of Applied Sciences and Engineering: Civil Engineering* degree (Kandidaat Burgerlijk Ingenieur), a *Bachelor of Philosophy* degree (Kandidaat Wijsbegeerte) and a *Master of Applied Sciences and Engineering: Computer Science* degree (Burgerlijk Ingenieur in de Computerwetenschappen) from the Katholieke Universiteit Leuven in Belgium. He graduated magna cum laude in July 2001 with the thesis "A configurable electronic stock market agent: framework and prototype", supervised by Prof. Tom Holvoet.

He started working as a Ph.D. student at the DistriNet (Distributed systems and computer Networks) research group of the Department of Computer Science at the Katholieke Universiteit Leuven in September 2001, as one the first members of the AgentWise taskforce, under supervision of Prof. Tom Holvoet. During the last two years of his research, he worked on an IBBT project on E-Health Information Platforms (E-HIP).

Simulatie van Gedistribueerde Controle Applicaties in Dynamische Omgevingen

Nederlandse samenvatting

Beknopte Samenvatting

Gedistribueerde controle applicaties zijn softwaresystemen die ontworpen zijn om de werking van verschillende gedistribueerde machines te controleren en coördineren. Een voorbeeld van een gedistribueerde controle applicatie is een softwaresysteem om productiemachines in een fabrieksomgeving te controleren. De omgeving van een gedistribueerde controle applicatie is typisch dynamisch. In een dynamische omgeving veranderen de werkingsomstandigheden van de controle applicatie voortdurend. Bijvoorbeeld, dynamiek in een fabrieksomgeving omvat het toekomen van nieuwe materialen, de werking van andere machines, voertuigen of mensen, etc. Het is essentieel dat een gedistribueerde controle applicatie rekening houdt met de omgeving waarin ze zich bevindt.

Simulatie is essentieel voor de ontwikkeling van gedistribueerde controle applicaties. Simulatie biedt een veilige en kosteneffectieve manier om het gedrag van een gedistribueerde controle applicatie te bestuderen, te evalueren of te configureren in een gesimuleerde omgeving, voordat die applicatie in gebruik genomen wordt in de echte wereld. In dit proefschrift ligt de nadruk op *software-in-de-lus simulatie* van gedistribueerde controle applicaties in dynamische omgevingen. Software-in-de-lus simulatie betekent dat de software van een echte gedistribueerde controle applicatie ingebed wordt in de simulatie. Met andere woorden: de controle software zelf maakt deel uit van de simulatie-lus. Bestaande aanpakken om deze familie van simulaties te ondersteunen, maken gebruik van (1) ofwel generische modelleringsconcepten die formeel onderbouwd zijn, maar die geen ondersteuning bieden die specifiek gericht is op deze familie van simulaties, (2) ofwel informele abstracties die specifieke ondersteuning bieden voor deze familie van simulaties, maar waarvan de betekenis gekoppeld is met de implementatie van een bepaald simulatieplatform.

We stellen een formeel onderbouwd modelleringsraamwerk voor ter ondersteuning van software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen. De concepten van het modelleringsraamwerk bieden specifieke ondersteuning voor deze familie van simulaties. Bovendien zijn de modelleringsconcepten formeel onderbouwd. Dit is cruciaal om een simulatiemodel te ontkoppelen van het simulatieplatform om het model uit te voeren. Het modelleringsraamwerk beschrijft de kernkarakteristieken van deze familie van simulaties op een expliciete manier. Het modelleringsraamwerk omvat een omgevingsdeel en een controle applicatiedeel. Het *omgevingsdeel* omvat modelleringsconcepten voor het beschrijven van dynamische omgevingen. Deze modelleringsconcepten beschrijven (1) de structuur van de omgeving, (2) dynamiek in de omgeving, (3) de manier waarop dynamiek beïnvloed wordt door de verschillende bronnen van dynamiek, en (4) de manier waarop interactie van dynamiek mogelijk is. Het *controle applicatiedeel* omvat modelleringsconcepten om de controle software van een gedistribueerde controle applicatie te integreren in een simulatie. Deze modelleringsconcepten beschrijven (1) de uitvoeringstijd van de controle software, en (2) de interface van de controle software met de omgeving.

Ter validatie van het modelleringsraamwerk, hebben we een simulatieplatform ontwikkeld dat de modelleringsconcepten ondersteunt in een uitvoerbare simulatie. Tevens hebben we de modelleringsconcepten toegepast in een simulator voor een industriële toepassing, meer bepaald een gedistribueerde controle applicatie om onbemande voertuigen te controleren in een fabrieksomgeving. De simulator omvat een simulatiemodel dat ontkoppeld is van het simulatieplatform om het uit te voeren. Dit vergemakkelijkt het aanpassen van het simulatiemodel, wat essentieel is om uiteenlopende functionaliteiten van de gedistribueerde controle applicatie te kunnen evalueren.

1 Inleiding

1.1 Achtergrond

Dit proefschrift handelt over simulatie van gedistribueerde controle applicaties in dynamische omgevingen. Eerst gaan we dieper in op gedistribueerde controle applicaties. Nadien lichten we het gebruik van simulatie voor dergelijke applicaties toe.

1.1.1 Gedistribueerde Controle Applicaties

We gebruiken de term *gedistribueerde controle applicaties* om te verwijzen naar een familie van software toepassingen die een aantal eigenschappen gemeenschappelijk hebben.

Een *controle applicatie* is een software systeem dat verbonden is met een onderliggende omgeving [6]. De omgeving is het deel van de externe wereld waarmee de controle applicatie interageert en waarin de effecten van de controle applicatie waargenomen worden. De taak van een controle applicatie is om een bepaalde functionaliteit te verwezenlijken in de omgeving. De interactie tussen de controle applicatie en haar omgeving gebeurt via sensoren en actuatoren. Een voorbeeld van een controle applicatie is de snelheidsregelaar van een wagen. De omgeving van deze controle applicatie omvat de wagen en de weg waarop de wagen rijdt. De controle applicatie interageert met deze omgeving via een sensor die de actuele snelheid van de wagen kan meten en een actuator die het vermogen van de motor kan aanpassen. De taak van de controle applicatie is om ervoor te zorgen dat de wagen met constante snelheid over de weg rijdt.

Een *gedistribueerde controle applicatie* is een controle applicatie waarvan het software systeem een gedistribueerde applicatie is. Een gedistribueerde applicatie is een software systeem dat bestaat uit meerdere componenten die uitvoeren op verschillende computers verbonden door een netwerk. Een voorbeeld van een gedistribueerde controle applicatie is een applicatie om een team van RoboCup Soccer [15] robots te laten voetballen. De taak van deze controle applicatie is om de robots van het eigen team te doen scoren en tegelijk het andere team te verhinderen om te scoren. Deze controle applicatie is gedistribueerd omdat software componenten uitgevoerd worden op elk van de robots van het team. Elk van de componenten heeft toegang tot de sensoren en actuatoren van een bepaalde robot, en coördineert de werking van die robot door te communiceren met andere robots van het team teneinde het gewenste globale gedrag te realiseren.

De omgeving van een gedistribueerde controle applicatie is typisch *zeer dynamisch*. Een dynamische omgeving is een omgeving die voortdurend verandert [17]. In een dynamische omgeving wijzigen de omstandigheden waarin een gedistribueerde controle applicatie moet werken voortdurend. Dynamiek in de omgeving kan afkomstig zijn van verschillende bronnen. Bijvoorbeeld, dynamiek in een RoboCup Soccer omgeving omvat het rollen van de bal, de bewegingen van de andere robots van het team, de bewegingen van de tegenstanders en zelfs afwijkingen omwille van de beperkte nauwkeurigheid waarmee de bal getrapt kan worden of omwille van defecten aan een robot.

Een dynamische omgeving heeft een significante impact op de acties van een gedistribueerde controle applicatie [5, 22]. In een dynamische omgeving verlopen de acties van een controle applicatie niet altijd zoals voorzien. Neem het voorbeeld van een component die een bepaalde RoboCup robot aanstuurt om vooruit te rijden met het doel om de bal die voor de robot ligt, naar een medespeler te trappen. In een dynamische omgeving kan deze actie op verschillende manieren beïnvloed worden. Zo kan de robot botsen met andere robots die hem proberen te verhinderen om tegen de bal te trappen, wat kan resulteren in schade aan de robot. Of de bal kan van zijn baan afwijken door kleine onnauwkeurigheden in de mechanische delen van de robot. Zelfs indien de robot erin slaagt de bal in de juiste richting te trappen, dan nog kan de bal onderweg onderschept worden door een andere robot die zich in de baan van de rollende bal positioneert.

1.1.2 Simulatie

Het is duidelijk dat een gedistribueerde controle applicatie rekening dient te houden met dynamiek die voorkomt in de omgeving en de potentiële impact hiervan op de uitgevoerde acties. Vooraleer een gedistribueerde controle applicatie in werking wordt genomen, is het cruciaal om het gedrag van deze applicatie te testen in scenario's die typische voorkomen in dynamische omgevingen.

Simulatie kan gedefinieerd worden als "het ontwerpen van een model van een echt systeem en het uitvoeren van experimenten op dit model met als doel het begrijpen van het gedrag van het systeem en/of het evalueren van verschillende strategieën voor de werking van het systeem" [19]. Twee belangrijke fases tijdens een simulatiestudie zijn de *modelleringsfase*, d.w.z. het bouwen van een simulatiemodel van het echte systeem, en de *vertaalfase*, d.w.z. het vertalen van het simulatiemodel naar een uitvoerbare simulatie.

Simulatie is essentieel voor het bestuderen en testen van het gedrag van een gedistribueerde controle applicatie in scenario's die typisch voorkomen in dynamische omgevingen [20, 14, 16]. Simulatie laat toe om (1) op een veilige manier te experimenteren met risicovolle scenario's, (2) experimenten uit te voeren sneller dan *real-time*, en (3) het opzetten en opvolgen van experimenten op een relatief kosteneffectieve manier. Bijvoorbeeld, neem een experiment waarbij een gedistribueerde controle applicatie betrokken is die robots aanstuurt in een fabrieksomgeving. Het doel van het experiment is om na te gaan of de robots botsingen kunnen vermijden met elkaar in een scenario waarbij communicatie onbetrouwbaar of tijdelijk onbeschikbaar is. Het uitvoeren van een dergelijk experiment met echte robots is moeilijk haalbaar omwille van (1) het hoge risico om robots te beschadigen, (2) de hoeveelheid tijd nodig om langdurige scenario's te testen, en (3) de hoge kost om grootschalige experimenten met vele robots op te zetten en te monitoren.

Door onderzoek en ontwikkeling rond simulatie is ondersteuning ontwikkeld voor zowel de modelleringsfase als de vertaalfase:

- *Modelleringsconcepten bieden ondersteuning voor de modelleringsfase.* Verschillende simulatieparadigma's bieden gevestigde modelleringsconcepten aan die het beschrijven van een simulatiemodel ondersteunen. Bijvoorbeeld, *discrete event* simulatie biedt concepten als *toestand* en *events* aan om een simulatiemodel uit te drukken. De toestand van een model is een lijst van waarden die voldoende zijn om de status van het systeem te definiëren [3]. Een event is een verandering in de toestand van de simulatie die ogenblikkelijk gebeurt op een welbepaald tijdstip in simulatie tijd [18].

- *Simulatieplatformen ondersteunen de vertaalfase.* Simulatieplatformen bevatten de functionaliteit die nodig is om de modelleringsconcepten te ondersteunen in een uitvoerbare simulatie. De functionaliteit van een simulatieplatform kan hergebruikt worden voor elk simulatiemodel dat beschreven is in termen van de ondersteunde modelleringsconcepten. Op die manier dient de functionaliteit van een simulatieplatform niet opnieuw uitgevonden te worden voor elke nieuwe simulatiestudie.

In dit proefschrift ligt de focus op software-in-de-lus simulatie van gedistribueerde controle applicaties in dynamische omgevingen. Dergelijke simulaties worden gebruikt om een gedistribueerde controle applicatie te testen of fijn te stemmen in een gesimuleerde omgeving voordat de software in gebruik wordt genomen in de echte omgeving [4]. Software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen hebben de volgende eigenschappen:

- De te simuleren omgeving is dynamisch. Een dynamische omgeving kan bronnen van dynamiek bevatten extern aan de gedistribueerde controle applicatie. Deze bronnen van dynamiek kunnen een significante impact hebben op de gedistribueerde controle applicatie omdat ze de werkingsomstandigheden van de applicatie veranderen.

- De software van de echte gedistribueerde controle applicatie wordt ingebed in de simulatie. De gedistribueerde controle applicatie wordt niet gesubstitueerd door een model, maar de controle software zelf maakt deel uit van de simulatielus, zoals vervat zit in de term *software-in-de-lus* simulatie.

1.2 Probleemstelling

Het ontwikkelen van software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen is complex. Het systeem dat gesimuleerd moet worden bestaat uit twee delen: een dynamische omgeving enerzijds en anderzijds een gedistribueerde controle applicatie die ingebed wordt in die omgeving. We bespreken kort enkele uitdagingen die gepaard gaan met het bouwen van dergelijke simulaties:

- *Het simuleren van dynamische omgevingen is complex.* Bijvoorbeeld, in een dynamische omgeving kan het resultaat van acties van een controle applicatie niet a priori bepaald worden [5, 7]. Andere activiteiten die actief zijn in de omgeving kunnen een significante impact hebben op de uitkomst van acties. Neem het voorbeeld van een robot die aangestuurd wordt om te beginnen rijden in een bepaalde richting. In een dynamische omgeving kan deze actie op verschillende manieren beïnvloed worden. Bijvoorbeeld, een andere machine kan in het pad van de robot bewegen en de robot blokkeren of opzij duwen; of door onnauwkeurigheden in het mechanische gedeelte van de robot kan het pad van de robot licht afwijken van het bedoelde pad; of de batterij van de robot kan leeg geraken en de beweging van de robot voortijdig beëindigen. Zelfs een combinatie van deze fenomenen kan voorkomen. Het is niet triviaal om de variëteit aan mogelijk samengestelde interacties van een dynamische omgeving te reproduceren in een simulatie om zo de precieze impact van deze interacties op acties te bepalen.

- *Het integreren van de software van een echte gedistribueerde controle applicatie in een simulatie is complex.* Bijvoorbeeld, de apparaten waarop een gedistribueerde controle applicatie zich bevindt in de echte wereld bepalen hoe snel die applicatie uitvoert en hoeveel tijd de applicatie nodig heeft om te reageren op veranderingen in de omgeving. Echter, de karakteristieken van het computerplatform waarop de simulatie uitgevoerd wordt, kunnen sterk verschillen van de apparaten waarop de controle applicatie uitvoert in de echte wereld. Bovendien kan een simulatie sneller of trager dan real-time worden uitgevoerd. Het is niet evident om de tijdskarakteristieken van een gedistribueerde controle applicatie in de echte wereld te reproduceren in een simulatie.

Om de ontwikkeling van software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen te ondersteunen, kan een ontwikkelaar gebruik maken van generische simulatieplatformen of van specifieke simulatieplatformen.

- Generische simulatieplatformen bieden ondersteuning voor het uitvoeren van simulatiemodellen die beschreven zijn in termen van generische modelleringsconcepten. Bijvoorbeeld, JAMES [14] is een simulatieplatform dat discrete event modellen ondersteunt die beschreven zijn met de modelleringconcepten van DEVS (Discrete EVent System specification) [23]. DEVS is een modelleringsraamwerk dat atomische en gekoppelde discrete event simulatiemodellen ondersteunt. Atomische modellen worden beschreven aan de hand van modelleringsconcepten als toestandsverzameling, input en output poorten, interne en externe transitiefuncties, etc.

  De betekenis van generische modelleringsconcepten is formeel gespecificeerd. Dit is cruciaal om het simulatiemodel te ontkoppelen van het simulatieplatform dat gebruikt wordt om het model uit te voeren. Op die manier kan een ontwikkelaar de modelleringsconcepten toepassen voor het beschrijven van een simulatiemodel zonder dat kennis vereist is van het simulatieplatform dat gebruikt zal worden om het simulatiemodel uit te voeren.

Desalniettemin bieden generische modelleringsconcepten geen ondersteuning die specifiek gericht is op software-in-de-lus simulatie van gedistribueerde controle applicaties in dynamische omgevingen. Generische modelleringsconcepten zijn toepasbaar op een breed spectrum van simulaties, en hun toepasbaarheid is niet beperkt tot simulaties van gedistribueerde controle applicaties. Bijgevolg bieden generische modelleringsconcepten geen ondersteuning voor de specifieke uitdagingen die gepaard gaan met het bouwen van simulaties voor gedistribueerde controle applicaties.

- Specifieke simulatieplatformen zijn gericht op het simuleren van gedistribueerde controle applicaties in het bijzonder. Bijvoorbeeld, XRaptor [2] is een simulatieplatform om het gedrag te bestuderen van grote aantallen agenten in twee- of driedimensionale continue virtuele omgevingen. XRaptor omschrijft een agent als een punt, een cirkelvormig oppervlak of een sferisch volume. Een agent beschikt over een sensoreenheid bevat om de wereld waar te nemen, een actuatoreenheid om acties uit te voeren en een controlekern voor actieselectie. Differentiaalvergelijkingen worden gebruikt om bewegingen te modelleren.

  Specifieke simulatieplatformen bieden ondersteuning die specifiek gericht is op software-in-de-lus simulatie van gedistribueerde controle applicaties in dynamische omgevingen. Bijvoorbeeld, XRaptor ondersteunt simulaties die bestaan uit een wereld waarin agenten, die gecontroleerd worden door een controlekern, kunnen waarnemen en handelen. In vergelijking met generische simulatieplatformen bieden specifieke simulatieplatformen ondersteuning voor de uitdagingen die gepaard gaan met het bouwen van simulaties voor gedistribueerde controle applicaties.

  Specifieke simulatieplatformen bieden enkel informele abstracties aan om een simulatiemodel te beschrijven. De precieze betekenis van en relatie tussen deze abstracties is niet formeel gespecificeerd, maar is enkel impliciet vervat in het ontwerp en de implementatie van het simulatieplatform. Bijvoorbeeld, de abstracties "controlekern", "actuator eenheid" of "cirkelvormig oppervlak" van XRaptor zijn niet onderbouwd met een formele specificatie, zodat hun precieze betekenis en onderlinge relatie vaag zijn. Zo is het bijvoorbeeld onduidelijk hoe de controlekern bewegingen in de omgeving kan activeren, of op welke manier de tijdskarakteristieken van de controlekern ondersteund worden. Omwille van het ontbreken van een formele specificatie vereist het bouwen van een simulatie met behulp van een specifiek simulatieplatform gedetailleerde kennis van het onderliggende ontwerp en de implementatie van dit simulatieplatform. Dit resulteert in een sterke koppeling tussen het simulatiemodel enerzijds, en het platform dat gebruikt worden om dat model uit te voeren anderzijds.

Samengevat stellen we dat de ondersteuning die geboden worden door bestaande aanpakken tekort schiet voor simulatie van gedistribueerde controle applicaties. De reden is dat bestaande aanpakken gebruik maken van (1) ofwel generische modelleringsconcepten die wel degelijk formeel onderbouwd zijn, maar die geen ondersteuning bieden specifiek voor simulaties van gedistribueerde controle applicaties, (2) ofwel informele abstracties die gericht zijn op simulatie van gedistribueerde controle applicaties, maar waarvan de betekenis impliciet vervat zit in het ontwerp en implementatie van het simulatieplatform.

We besluiten dat er een gebrek is aan formeel onderbouwde modelleringsconcepten die specifieke ondersteuning bieden voor de karakteristieken van gedistribueerde controle applicaties in dynamische omgevingen.

1.3 Aanpak

In dit proefschrift stellen we een modelleringsraamwerk voor, ondersteund door een simulatieplatform, dat specifiek gericht is op het ontwikkelen van software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen.

Het modelleringsraamwerk omvat een verzameling van formeel gespecificeerde modelleringsconcepten die specifiek gericht zijn op deze familie van simulaties, en die de kernkarakteristieken van deze familie van simulaties op een expliciete manier beschrijven. De modelleringsconcepten van het modelleringsraamwerk zijn formeel beschreven om hun betekenis en onderlinge relatie ondubbelzinnig voor te stellen. Dit is cruciaal om de modelleringconcepten te ontkoppelen van hun specifieke implementatie in een bepaald simulatieplatform. Op die manier wordt het mogelijk om een simulatiemodel te formuleren zonder dat kennis vereist is van het ontwerp en de implementatie van het simulatieplatform.

Het modelleringsraamwerk is enerzijds het resultaat van onze eigen ervaring met simulatie van gedistribueerde controle applicaties in dynamische omgevingen en wordt anderzijds onderbouwd door actueel onderzoek over het modelleren van gedistribueerde controle applicaties in dynamische omgevingen. Het modelleringsraamwerk omvat twee delen:

- Een omgevingsdeel dat specifieke modelleringsconcepten aanbiedt om dynamische omgevingen te beschrijven in een simulatiemodel. De modelleringsconcepten van het omgevingsdeel worden beschreven in sectie 2.

- Een controle-applicatiedeel dat specifieke modelleringsconcepten aanbiedt om de software van een echte gedistribueerde controle applicatie expliciet te integreren in het simulatiemodel. De modelleringsconcepten van het controle-applicatiedeel worden beschreven in sectie 3.

We onderzoeken twee alternatieven om een simulatiemodel dat beschreven is in termen van de concepten van het modelleringsraamwerk, te vertalen naar een uitvoerbare simulatie.

- Ten eerste, de formele beschrijving van het modelleringsraamwerk specificeert de kernfunctionaliteit die nodig is om de modelleringsconcepten te ondersteunen in een uitvoerbare simulatie. De formele beschrijving biedt enkel een specificatie aan om de vertaalfase te ondersteunen, zonder bepaalde ontwerpbeslissingen of algoritmes op te leggen. Dit maakt het mogelijk om bepaalde ontwerpbeslissingen of algoritmes te gebruiken die optimaal afgestemd zijn op een bepaalde simulatiestudie.
- Ten tweede hebben we een simulatieplatform ontwikkeld dat alle functionaliteit aanbiedt die nodig is om de modelleringsconcepten te ondersteunen in een uitvoerbare simulatie. Het simulatieplatform illustreert de haalbaarheid van het modelleringsraamwerk om software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen te ontwikkelen. Bovendien maakt het simulatieplatform gebruik van geavanceerde softwaretechnologie om een aantal moeilijke ontwerpuitdagingen aan te pakken. Een voorbeeld is het gebruik van aspect-georiënteerde technologie om (1) een aantal simulatieconcerns die de functionaliteit van de gedistribueerde controle applicatie doorsnijden, te modularizeren, en om (2) de gedistribueerde controle applicatie op een gebruiksvriendelijke manier te integreren in de simulatie.

1.4 Overzicht

Deze samenvatting is als volgt gestructureerd.

In sectie 2 introduceren we het omgevingsdeel van het modelleringsraamwerk. De nadruk ligt op de specificatie van modelleringsconcepten om een dynamische omgeving te beschrijven in een simulatiemodel. Ontwerp- en implementatiebeslissingen om de concepten te ondersteunen, worden besproken in sectie 4.

In sectie 3 introduceren we het controle-applicatiedeel van het modelleringsraamwerk. De nadruk ligt op de specificatie van modelleringsconcepten om de controle software te integreren in het simulatiemodel. Ontwerp- en implementatiebeslissingen om de concepten te ondersteunen, worden besproken in sectie 4.

In sectie 4 illustreren we de haalbaarheid van de concepten van het modelleringsraamwerk beschreven in sectie 2 en 3. We beschrijven de software architectuur van een simulatieplatform dat de modelleringsconcepten ondersteunt in een uitvoerbare simulatie.

In sectie 5 tonen we de bruikbaarheid van de modelleringsconcepten en het simulatieplatform aan door ze toe te passen op een echte wereld toepassing: software-in-de-lus simulatie van gedistribueerde controle applicaties die onbemande voertuigen aansturen in een dynamische fabrieksomgeving.

In sectie 6 besluiten we met een kort overzicht van de voornaamste bijdragen van het onderzoek.

2 Modellering van Dynamische Omgevingen

In deze sectie focussen we op het omgevingsdeel van het modelleringsraamwerk.

2.1 Inleiding

We introduceren modelleringsconcepten die specifiek gericht zijn op het modelleren van dynamische omgevingen. De modelleringsconcepten beschrijven de karakteristieken van een dynamische omgeving op een expliciete manier. Bovendien is de betekenis, relatie en uitvoeringssemantiek van alle modelleringsconcepten formeel beschreven. De formele beschrijving van de concepten laat toe om de modelleringsconcepten te ontkoppelen van hun specifieke implementatie in een bepaald simulatieplatform. Dit maakt het mogelijk om de modelleringsconcepten toe te passen om een simulatiemodel te beschrijven zonder rekening te houden met het simulatieplatform dat gebruikt zal worden om het simulatiemodel uit te voeren.

De onderbouw van de concepten van het modelleringsraamwerk is tweeledig. Enerzijds is het modelleringsraamwerk het resultaat van onze eigen ervaring met het ontwikkelen van simulaties van gedistribueerde controle applicaties in dynamische omgevingen. Anderzijds zijn de modelleringsconcepten gefundeerd in de actuele praktijk van het modelleren van dynamische omgevingen.

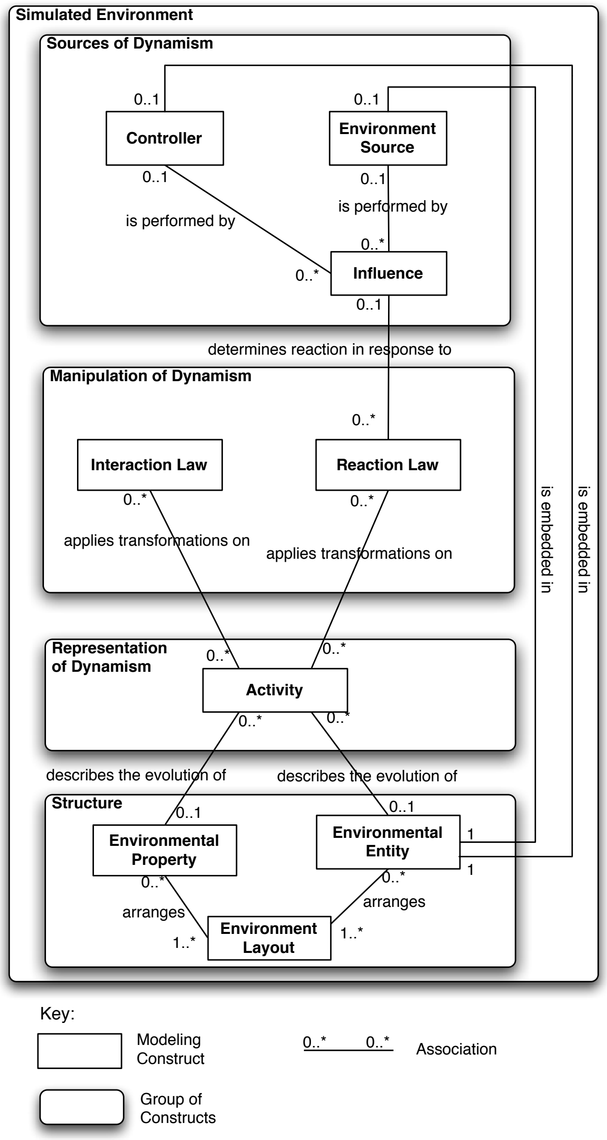
In deze samenvatting beperken we ons tot een kort overzicht van de verschillende modelleringsconcepten. Voor een gedetailleerde beschrijving, meer uitgebreide voorbeelden en een formele specificatie verwijzen we naar de Engelstalige tekst van dit proefschrift.

2.2 Overzicht van het Modelleringsraamwerk

Figuur 1 toont een grafisch overzicht van het modelleringsraamwerk voor dynamische omgevingen. De figuur toont de verschillende modelleringsconcepten en de relaties tussen deze concepten. De modelleringsconcepten zijn onderverdeeld in vier groepen:

1. Concepten die de *structuur van de omgeving* (`Structure`) voorstellen in het simulatiemodel.
2. Concepten die *dynamiek in de omgeving* (`Representation of Dynamism`) voorstellen in het simulatiemodel.
3. Concepten die *manipulatie van dynamiek in de omgeving* (`Manipulation of Dynamism`) voorstellen in het simulatiemodel.
4. Concepten die de *bronnen van dynamiek in de omgeving* (`Sources of Dynamism`) voorstellen in het simulatiemodel.

We geven een overzicht van de modelleringsconcepten in elke groep.



Figuur 1: Overzicht van de modelleringsconcepten voor het modelleren van dynamische omgevingen.

2.2.1 Structuur van de Omgeving

Een eerste groep van modelleringsconcepten dient om de structuur van de omgeving te beschrijven. Om de samenstellende delen van de omgeving voor te stellen in een simulatiemodel, introduceren we de concepten *omgevingsentiteit* (`Environmental Entity`) en *omgevingseigenschap* (`Environmental Property`). Voorbeelden van omgevingsentiteiten zijn de verschillende objecten in de omgeving, zoals de robots die aangestuurd worden door een gedistribueerde controle applicatie. Een voorbeeld van een omgevingseigenschap is de temperatuur van de omgeving. Om de fysische of logische structuur voor te stellen die de verschillende omgevingsentiteiten en omgevingseigenschappen ordent ten opzichte van elkaar, introduceren we het modelleringsconcept *omgevingslayout* (`Environment Layout`). Een voorbeeld van een omgevingslayout is een tweedimensionale geometrische schikking van de entiteiten.

2.2.2 Dynamiek in de Omgeving

Een tweede groep van modelleringsconcepten dient om dynamiek in de omgeving op een expliciete manier voor te stellen in het simulatiemodel. We introduceren het concept *activiteit* (`Activity`) om dynamiek expliciet voor te stellen in het simulatiemodel van de omgeving. De associatie tussen *activiteit* (`Activity`) en *omgevingsentiteit* (`Environmental Entity`) en tussen *activiteit* (`Activity`) en *omgevingseigenschap* (`Environmental Property`) drukt uit dat een activiteit de evolutie van een bepaalde omgevingsentiteit of omgevingseigenschap overheen de tijd beschrijft. Voorbeelden van activiteiten zijn de beweging van een robot of het rollen van een bal.

2.2.3 Manipulatie van Dynamiek in de Omgeving

Een derde groep van modelleringsconcepten dient om te beschrijven hoe dynamiek in de omgeving kan veranderen, meer bepaald hoe activiteiten ontstaan, in interactie treden en eindigen. We introduceren de concepten *reactiewet* (`Reaction Law`) en *interactiewet* (`Interaction Law`) om te beschrijven hoe activiteiten in de omgeving gemanipuleerd worden.

Een reactiewet is een modelleringsconcept dat dient om de reactie van de omgeving op een bepaalde manipulatiepoging van een bron van dynamiek te specificeren. Een voorbeeld is een reactiewet die specificeert wat er gebeurt in de omgeving als reactie op een poging van een controller om de motoren van een robot te starten. De reactiewet specificeert wat voor activiteit hierdoor geïnitieerd wordt, bijvoorbeeld een activiteit die een beweging van die robot voorstelt met een bepaalde snelheid en in een bepaalde richting.

Een interactiewet is een modelleringsconcept dat dient om te specificeren hoe dynamiek in interactie kan treden in de omgeving. Bijvoorbeeld, een interactiewet kan gebruikt worden om te specificeren wat er gebeurt in het geval een robot die onderhevig is aan een bewegingsactiviteit een muur of een andere robot raakt.

De associatie tussen *reactiewet* (`Reaction Law`) en *activiteit* (`Activity`) enerzijds en tussen *interactiewet* (`Interaction Law`) en *activiteit* (`Activity`) anderzijds, drukt uit dat reactiewetten en interactiewetten de activiteiten die aanwezig zijn in de omgeving, kunnen beïnvloeden.

2.2.4 Bronnen van Dynamiek in de Omgeving

Een vierde groep van modelleringsconcepten dient om de bronnen van dynamiek in de omgeving te beschrijven. We introduceren de concepten *controller* (`Controller`) en *omgevingsbron* (`Environment Source`) om het gedrag van de verschillende bronnen van dynamiek voor te stellen.

Een controller is een bron van dynamiek die deel uitmaakt van de gedistribueerde controle applicatie. Een voorbeeld van een controller is het software programma dat een bepaalde robot aanstuurt. Een omgevingsbron is een bron van dynamiek die deel uitmaakt van de omgeving zelf en die extern is aan de gedistribueerde controle applicatie. Een voorbeeld van een omgevingsbron is het gedrag van een machine in de omgeving die bestuurt wordt door een mens. Controllers en omgevingsbronnen zijn ingebed in bepaalde omgevingsentiteiten. Bijvoorbeeld, een robot bevat een bron van dynamiek, namelijk zijn controller, terwijl een bal passief is en geen bron van dynamiek bevat.

Controllers en omgevingsbronnen kunnen dynamiek in de omgeving initiëren, veranderen of beëindigen. We introduceren het modelleringsconcept *invloed* (`Influence`) om de poging voor te stellen van een controller of omgevingsbron om de omgeving te beïnvloeden. Een voorbeeld van een invloed is de poging van een controller om de beweging van een robot te starten of te stoppen. De associatie tussen *omgevingsbron* (`Environment Source`) en *invloed* (`Influence`), en tussen *controller* (`Controller`) en *invloed* (`Influence`) drukt uit dat dynamiek enkel indirect gemanipuleerd kan worden, namelijk door het uitoefenen van invloeden op de omgeving. Reactiewetten bepalen de eigenlijke reactie van de omgeving op deze invloeden. Dit is voorgesteld door de associatie tussen *reactiewet* (`Reaction Law`) en *invloed* (`Influence`).

3 Modellering van de Integratie van de Controle Software

In deze sectie focussen we op het controle-applicatiedeel van het modelleringsraamwerk.

3.1 Inleiding

In software-in-de-lus simulaties wordt de software van de echte controllers van een gedistribueerde controle applicatie ingebed in een gesimuleerde omgeving. We introduceren modelleringsconcepten die het mogelijk maken op een expliciete manier te beschrijven hoe de controle software van een gedistribueerde controle applicatie geïntegreerd wordt in het simulatiemodel. De betekenis, relatie en uitvoeringssemantiek van de modelleringsconcepten zijn formeel beschreven. De formele beschrijving ontkoppelt de modelleringsconcepten van hun implementatie in een bepaald simulatieplatform.

In deze samenvatting beperken we ons tot een kort overzicht van de verschillende modelleringsconcepten. Voor een gedetailleerde beschrijving, meer uitgebreide voorbeelden en de formele specificatie verwijzen we naar de Engelstalige tekst van dit proefschrift.

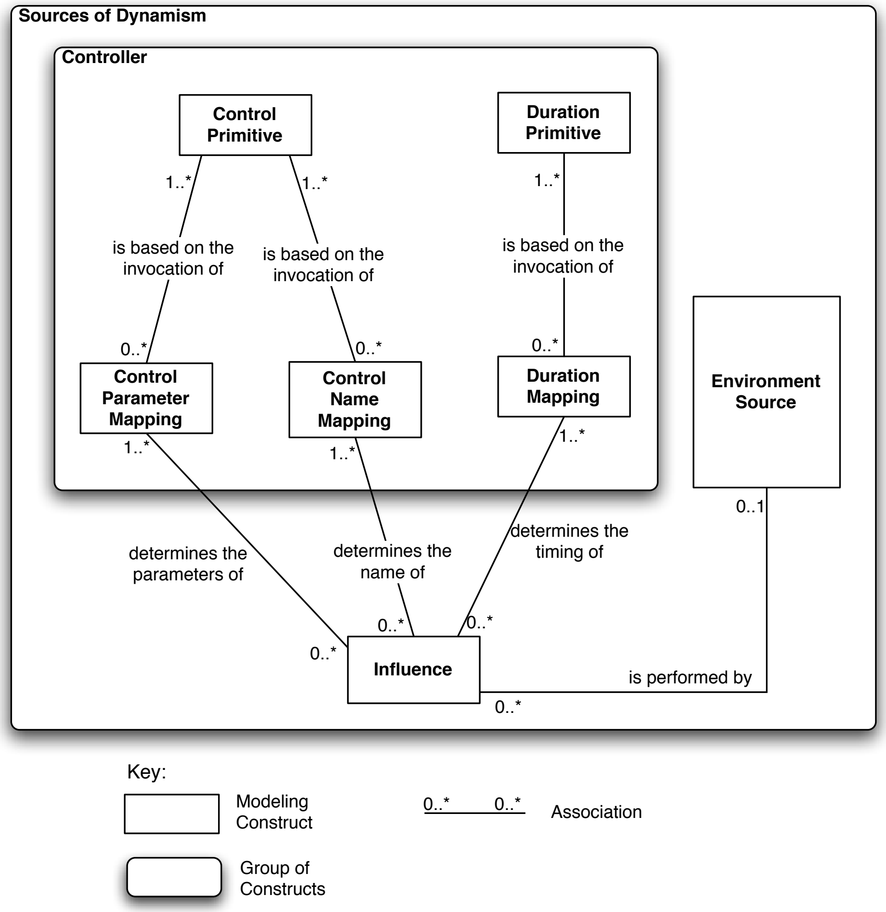
3.2 Overzicht van het Modelleringsraamwerk

Figuur 2 geeft een grafisch overzicht van het modelleringsraamwerk. Deze figuur toont de groep van concepten voor de bronnen van dynamiek (`Sources of Dynamism`) in Figuur 1, met bijkomende modelleringsconcepten voor de controller. De modelleringsconcepten stellen de volgende karakteristieken van de controle software expliciet voor in het simulatiemodel:

- *Voorstelling van de uitvoeringstijd van de controle software in het simulatiemodel.* De uitvoeringstijd van een controller in de echte wereld is de hoeveelheid tijd die verstrijkt totdat die controller zijn volgende actie initieert. De uitvoeringstijd van een controller bepaalt de timing van de acties. In een dynamische omgeving is de timing van acties cruciaal omdat opportuniteiten typisch komen en gaan.

  We introduceren de modelleringsconcepten *tijdsduurprimitief* (`Duration Primitive`) en *tijdsduurmapping* (`Duration Mapping`) om de uitvoeringstijd van een controller te beschrijven in het simulatiemodel. Een tijdsduurprimitief identificeert een codesegment dat voor de simulatie een relevante hoeveelheid uitvoeringstijd vraagt in de echte wereld. Een voorbeeld van een tijdsduurprimitief is een bepaalde Java methode *foo()* in de software van een bepaalde controller. Een tijdsduurmapping is een modelleringsconcept dat specificeert hoeveel uitvoeringstijd de invocatie van tijdsduurprimitieven inneemt. Bijvoorbeeld, een tijdsduurmapping kan specificeren dat het uitvoeren van de methode *foo()* 0.338 seconden duurt.

- *Voorstelling van de interactie van de controle software met de omgeving.* De software van een controller treedt in interactie met de omgeving. De uitvoering van de controle software kan bepaalde dingen laten gebeuren in de omgeving. Indien de software van controllers geïntegreerd wordt in een gesimuleerde omgeving, is het cruciaal om de software instructies te identificeren die door een controller gebruikt worden om in interactie te treden met de omgeving, en om te specificeren wat de precieze gevolgen zijn in de omgeving indien deze instructies uitgevoerd worden.



Figuur 2: Overzicht van de modelleringsconcepten voor het modelleren van de controle software.

We introduceren de modelleringsconcepten *controleprimitief* (`Control Primitive`), *controlenaammapping* (`Control Name Mapping`) en *controleparametermapping* (`Control Parameter Mapping`) om de interactie van de software met de omgeving voor te stellen in het simulatiemodel. Een controleprimitief stelt een bepaalde software instructie voor die gebruikt wordt door de controle software om in interactie te treden met de omgeving. Een voorbeeld van een controleprimitief is een Java methode *bar()* die de motor van een robot aanstuurt om op volle kracht te draaien. De modelleringsconcepten controlenaammapping en controleparametermapping specificeren de naam en de parameters van de invloed (`Influence`) die resulteert van het uitvoeren van een controleprimitief. Een controlenaammapping en controleparametermapping ontkoppelen de signatuur van de controleprimitieven van de specifieke representatie van invloeden (`Influences`) die gebruikt wordt in de gesimuleerde omgeving. Bijvoorbeeld, een controlenaammapping kan specificeren dat een invocatie van *bar()* overeenkomt met een invloed met naam *startRijden*; een controleparametermapping kan specificeren dat een invocatie van *bar()* resulteert in de toekenning van de waarde *10* aan die parameter van *startRijden* die de snelheid voorstelt van de beweging.

4 Architectuur van het Simulatieplatform

In deze sectie beschrijven we de architectuur van een simulatieplatform dat ondersteuning biedt voor de modelleringsconcepten van het modelleringsraamwerk beschreven in Sectie 2 en Sectie 3. Het simulatieplatform kan gebruikt worden om simulatiemodellen uit te voeren die beschreven zijn aan de hand van de modelleringsconcepten.

4.1 Inleiding

De software architectuur van een systeem realiseert de functionaliteit van een systeem zodanig dat aan de kwaliteitsvereisten voldaan is.

We gebruiken verschillende architecturale views om de architectuur van het simulatieplatform te documenteren. Een view is een voorstelling van een coherente set van architecturale elementen en de relaties ertussen [1]. Elk view beschouwt de architectuur (of een deel ervan) vanuit een bepaald perspectief. De architectuurdocumentatie van het simulatieplatform bestaat uit een module-decompositie view en verschillende component-en-connector views.

Eerst beschrijven we de vereisten van het simulatieplatform. Daarna bespreken we ter illustratie het module-decompositie view van het simulatieplatform op het hoogste niveau.

4.2 Vereisten

De belangrijkste functionele vereisten van het simulatieplatform zijn de volgende:

- *Ondersteuning bieden voor de modelleringsconcepten voor dynamische omgevingen.* Dit omvat (1) het beheren van bronnen van dynamiek en de invloeden die het resultaat zijn van hun uitvoering, (2) het toepassen van de reactiewetten om de reactie van de omgeving op verschillende invloeden te bepalen, (3) het beheren van alle activiteiten in de omgeving tijdens het uitvoeren van een simulatie, (4) het toepassen van de interactiewetten om interacties van activiteiten af te dwingen.

- *Ondersteuning bieden voor de modelleringsconcepten voor de controle software.* Dit omvat (1) het opvolgen van de tijdsduurprimitieven die uitgevoerd worden door de verschillende controllers, (2) het opvolgen van de controleprimitieven die uitgevoerd worden door de controllers, (3) het bepalen van de aard en de timing van de invloeden die voortkomen uit de uitvoering van de controllers.
- *Ondersteuning bieden om simulaties uit te voeren die consistent zijn met het beschreven simulatiemodel.* Dit omvat het regelen van de uitvoering van de verschillende delen van de simulatie, namelijk de verschillende controllers, omgevingsbronnen van dynamiek, reactiewetten en interactiewetten, zodat de causale relaties overeenstemmen met de specificaties van het simulatiemodel.

De belangrijkste kwaliteitsvereisten voor het simulatieplatform zijn de volgende:

- *Flexibiliteit om de software van een controle applicatie in te bedden in de simulatie.*
- *Aanpasbaarheid van het simulatieplatform.* Het aanpassen van de belangrijkste delen van het simulatieplatform moet relatief makkelijk kunnen gebeuren, en de impact van dergelijke wijzigingen moet zo lokaal mogelijk zijn. Voorbeelden zijn het aanpassen van de simulatie-engine en de functionaliteit om de gesimuleerde omgeving te ondersteunen.
- *Performantie van het simulatieplatform.* Het simulatieplatform moet het mogelijk maken simulaties uit te voeren sneller dan real-time.

4.3 Module-Decompositie View van het Simulatieplatform

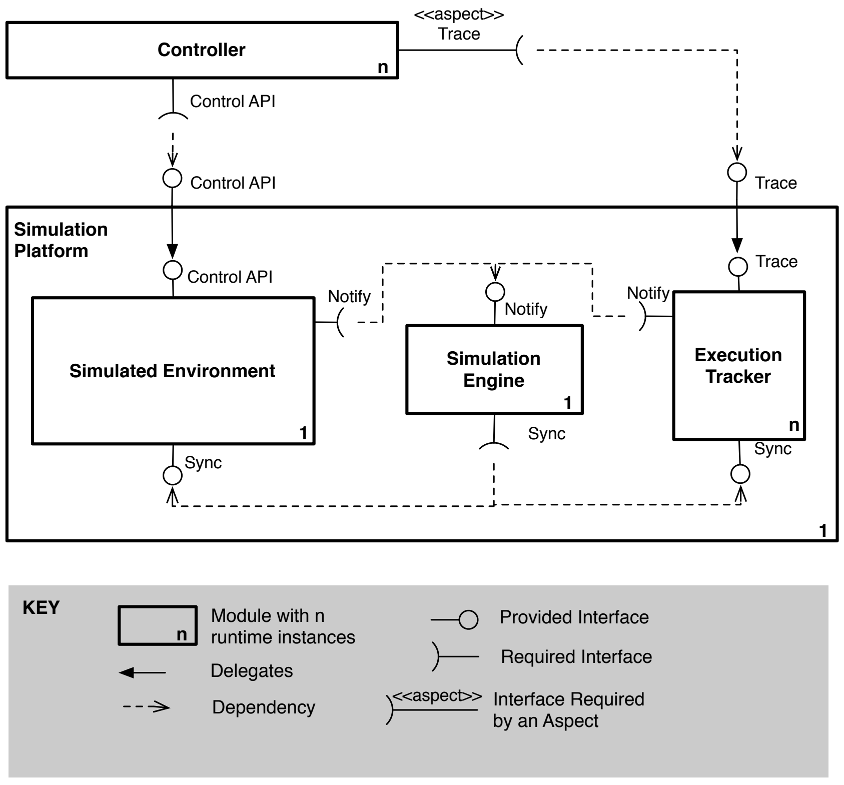
Een module-decompositie view is een statisch perspectief op de software architectuur van een systeem. Het module-decompositie view toont een decompositie van het simulatieplatform in verschillende modules. Een module is een implementatie-eenheid die een coherentie functionaliteit aanbiedt. De relatie tussen modules is “is een deel van” tussen een deelmodule en de overkoepelende module. Modules kunnen recursief verfijnd worden.

Figuur 3 toont de module-decompositie view van het simulatieplatform op het hoogste niveau. We beschrijven eerst de verschillende modules. Achteraf leggen we uit hoe belangrijke kwaliteiten gerealiseerd worden.

4.3.1 Bespreking van de Elementen

De decompositie van het simulatiesysteem bestaat uit twee grote subsystemen: Controller en Simulation Platform.

- `Controller` is een module van de echte gedistribueerde controle applicatie die ingebed is in het simulatieplatform met de bedoeling die module te testen of te



Figuur 3: Module-decompositie view van het simulatieplatform

configureren. Een gedistribueerde controle applicatie bestaat uit verschillende controllers die parallel uitvoeren en die samenwerken om een probleem op te lossen. Een controller heeft welbepaalde mogelijkheden om de omgeving waar te nemen en er acties in uit te voeren.

- `Simulation Platform` is het medium waarin de controllers van een gedistribueerde controle applicatie ingebed worden. De belangrijkste verantwoordelijkheden van het simulatieplatform zijn:
  - Het simuleren van de echte omgeving van de controle applicatie.
  - Het regelen van de uitvoering van alle controllers van de controle applicatie volgens het gespecificeerde model van de uitvoeringstijd.
  - Het uitvoeren van een simulatie, mogelijk sneller dan real-time.

  Het simulatieplatform is verder opgesplitst in drie verschillende modules: Simulated Environment, Simulation Engine en Execution Tracker.

  - `Simulated Environment` is verantwoordelijk voor het simulatiemodel van de echte omgeving van de controle applicatie. De Simulated Environment module omvat alle functionaliteit om de modelleringsconcepten van het omgevingsdeel(Sectie 2) te ondersteunen.
  - `Simulation Engine` is verantwoordelijk om de evolutie van alle delen van de simulatie te regelen in overeenstemming met de specificatie van het simulatiemodel. De Simulation Engine module omvat alle functionaliteit om de uitvoering de Simulated Environment en de verschillende Execution Trackers te synchroniseren met elkaar. Dit is nodig om correcte causale relaties te garanderen die in overeenstemming zijn met het gedefinieerde simulatiemodel.
  - `Execution Tracker` is verantwoordelijk om de uitvoering van een bepaalde controller van de gedistribueerde controle applicatie op te volgen. Deze module omvat alle functionaliteit om de modelleringsconcepten van het controle applicatiedeel (Sectie 3) te ondersteunen. Het opvolgen van de uitvoering van een controller omvat (1) het bepalen van de uitvoeringstijd die gebruikt is door die controller volgens de specificatie van het tijdsduurmodel, en (2) het synchroniseren van de uitvoering van die controller met de Simulation Engine.

4.3.2 Motivatie voor het Ontwerp

Elke module omvat een bepaald deel van de functionaliteit van het simulatieplatform. We lichten de voornaamste ontwerpbeslissingen toe.

Lage koppeling tussen de Controller en het Simulation Platform. Voor software-in-de-lus simulatie is het belangrijk om een lage koppeling te hebben tussen de controle software enerzijds, en het simulatieplatform waarin de controle software ingebed wordt, anderzijds. De Controller is met het Simulation Platform verbonden via twee interfaces: `Control API` en `Trace`. De Control API interface laat toe om alle communicatie, actie en perceptie van de Controller op een transparante manier om te leiden naar het Simulation Platform. De Trace interface verbindt de controller met een specifieke Execution Tracker in het Simulation Platform.

Twee voordelen van een lage koppeling tussen de Controller en het Simulation Platform zijn (1) hergebruik, namelijk het hergebruik van het Simulation Platform voor verschillende Controllers, en (2) aanpasbaarheid, namelijk dat het mogelijk wordt de Controllers te wijzigen zonder impact op het Simulation Platform.

Lage koppeling tussen de Simulated Environment en de Simulation Engine. In het Simulation Platform maken we een expliciet onderscheid tussen de Simulated Environment enerzijds, en de Simulation Engine anderzijds. De Simulated Environment en de Simulation Engine zijn verbonden met twee welbepaalde interfaces, namelijk de `Notify` en de `Sync` interfaces. Dit laat toe om (1) abstractie te maken van synchronisatie bij het ontwikkelen van de Simulated Environment, en om (2) abstractie te maken van de interne werking van de verschillende te synchroniseren partijen bij de ontwikkeling van de Simulation Engine.

Twee voordelen van een lage koppeling tussen de Simulated Environment en de Simulation Engine zijn (1) hergebruik, namelijk het vergemakkelijkt de integratie van een andere Simulation Engine in het Simulation Platform, en (2) beheersbaarheid, namelijk het ontwerp van de Simulated Environment wordt vergemakkelijkt omdat abstractie kan gemaakt worden van synchronisatie.

5 Simulatie van AGV Controle Applicaties in Dynamische Fabrieksomgevingen

In deze sectie passen we het modelleringsraamwerk en het simulatieplatform toe op een echte wereld probleem.

5.1 Inleiding

We ontwikkelden een simulator die toelaat om nieuwe of veranderde functionaliteit te evalueren van een gedistribueerde controle applicatie voor het aansturen van AGV's in een fabrieksomgeving.

Een AGV is een onbemand elektrisch voertuig dat dient om ladingen te transporteren van de ene naar de andere plaats in een fabriek. Een echte AGV controle applicatie omvat verschillende, complexe functionaliteiten, zoals:

- Transporttoewijzing: transporten worden gegenereerd door klantsystemen (typisch een warehouse management systeem) en dienen te worden toegewezen aan AGV's.
- Routering: AGV's dienen een efficiënte route te vinden op de layout van de fabrieksvloer, AGV's mogen enkel langs voorgedefinieerde paden bewegen. Om efficiënt naar een bestemming te rijden dienen de AGV's rekening te houden met de wijzigende verkeerstoestand in het systeem.
- Vermijden van botsingen: vanzelfsprekend mogen AGV's niet op hetzelfde ogenblik een kruispunt oversteken; doch botsingen moeten ook vermeden worden wanneer twee AGV's mekaar passeren in dicht bij elkaar gelegen parallelle wegen.
- Herladen van de batterij: om te vermijden dat AGV's zonder energie vallen, moeten ze op geregelde tijdstippen naar een laadstation bewegen om hun batterij ter herladen.

De AGV simulator laat toe om (1) verschillende functionaliteiten van AGV controllers te evalueren, (2) verschillende aanpakken voor een bepaalde functionaliteit te vergelijken, en (3) de functionaliteiten op een systematische manier te integreren.

Het doel van deze sectie is om aan te tonen hoe het modelleringsraamwerk ondersteuning biedt voor de modelleringsfase van de ontwikkeling van de AGV simulator, en om te evalueren hoe flexibel en performant de AGV simulator is.

5.2 Simulatiemodel van de AGV Simulator

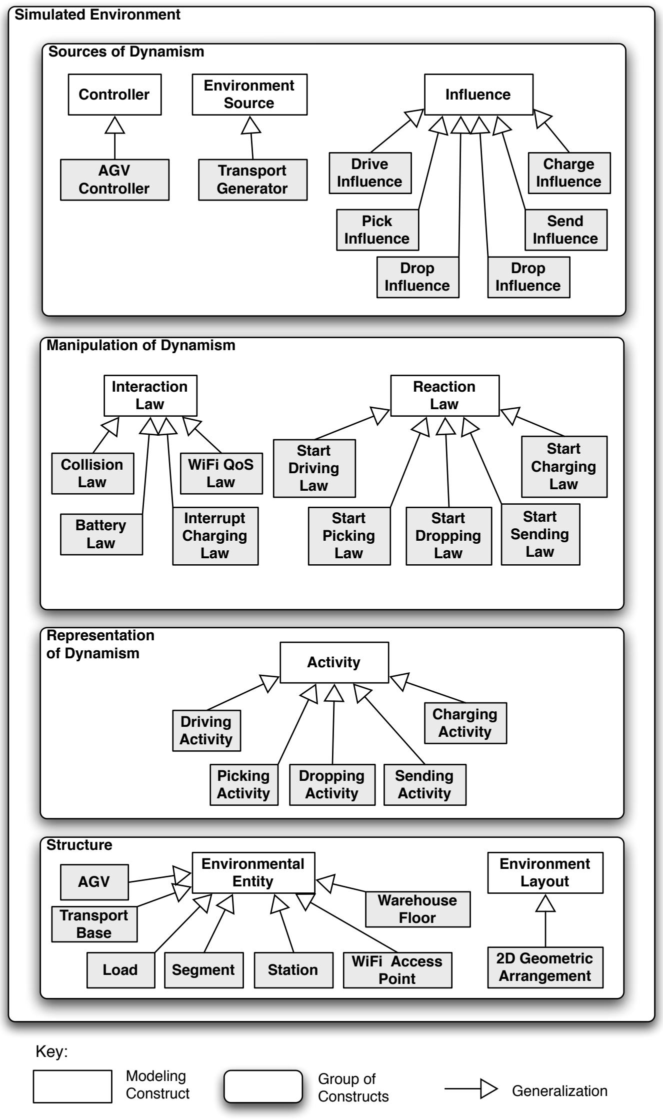
We formuleren een simulatiemodel voor de AGV simulator. Het simulatiemodel is beschreven aan de hand van de modelleringsconcepten van het omgevingsdeel (Sectie 2) en het controle applicatiedeel (Sectie 3). In deze samenvatting beperken we ons tot een overzicht van het model van de fabrieksomgeving.

Figuur 4 geeft een overzicht van het omgevingsdeel van het simulatiemodel van de AGV simulator. Deze figuur toont specifieke instanties van de modelleringsconcepten uit Figuur 1. Het simulatiemodel is gestructureerd in vier delen, analoog aan Figuur 1. We bespreken enkele instanties van elk deel van het simulatiemodel.

5.2.1 Structuur van de Fabrieksomgeving.

De structuur van de gesimuleerde fabrieksomgeving is gemodelleerd met behulp van omgevingsentiteiten (`Environmental Entity`) en *omgevingslayout* (`Environment Layout`). We bespreken enkele omgevingsentiteiten:

- *Fabrieksvloer* (`Warehouse Floor`). De fabrieksvloer is een vlak oppervlak van een bepaalde grootte. AGVs rijden over de fabrieksvloer en ladingen worden geplaatst op de fabrieksvloer.



Figuur 4: Overzicht van het simulatiemodel van de gesimuleerde fabrieksomgeving. De grijze delen zijn specifieke instanties van de modelleringsconcepten voor de AGV simulator.

- *Segmenten* (`Segments`). AGV's rijden over voorgedefinieerde magneetpaden, Een segment kan unidirectioneel of bidirectioneel zijn en heeft een bepaalde lengte. Elk segment verbindt twee stations.
- *Stations* (`Stations`). Stations zijn locaties die naburige segmenten verbinden. Een station kan gebruikt worden voor een of meerdere doeleinden, namelijk voor routering, als locatie om ladingen te plaatsen, als parkeerplaats en/of als herlaadstation voor batterijen.
- *Transportbasis* (`Transport Base`). Een transportbasis is een computer die gebruikt kan worden om nieuwe transporttaken te verspreiden onder de AGV's. Een transportgenerator is het ingebed in elke transportbasis.

De omgevingsentiteiten in de gesimuleerde fabrieksomgeving worden geordend op een tweedimensionale geometrische layout. Deze layout drukt de ruimtelijke positionering van alle entiteiten met betrekking tot elkaar uit.

5.2.2 Dynamiek in de Fabrieksomgeving

We bespreken enkele activiteiten die kunnen voorkomen in de fabrieksomgeving:

- *Rijd-activiteiten* (`Driving activities`). Een rijd-activiteit stelt voor dat een AGV over een segment van de fabrieksvloer rijdt tot hij het station op het einde van dat segment bereikt.
- *Oppik-activiteiten* (`Picking activities`). Een oppik-activiteit stelt voor dat een AGV zijn vorklift gebruikt om een bepaalde lading op een station op te pikken.
- *Herlaad-activiteiten* (`Charging activities`). Een herlaad-activiteit stelt voor dat een AGV zijn batterij herlaadt op een herlaadstation.

5.2.3 Bronnen van Dynamiek in de Fabrieksomgeving

In de fabrieksomgeving bevinden zich verschillende bronnen van dynamiek:

- *AGV controllers.* AGV controllers stellen de controle software voor die ingebed wordt in een AGV. Alle AGV controllers samen vormen de gedistribueerde AGV controle applicatie. Elke AGV controller is verantwoordelijk voor de sturing van een AGV, en om de werking van die AGV te coördineren met andere AGVs.
- *Transport generator.* Een transport generator zendt transporttaken naar de AGV's. Een transport generator genereert transporttaken volgens een bepaald transportprofiel. Dit transportprofiel bepaalt de karakteristieken van de stroom transporttaken die door de AGV's afgehandeld moet worden. Transport generators zijn omgevingsbronnen die extern zijn aan de AGV controle applicatie. Een transport generator in ingebed in een transportbasis.

We bepreken enkele invloeden (`Influences`):

- *Rijd-invloed* (`Drive influence`). Een rijd-invloed stelt de poging voor van een AGV controller om te beginnen rijden over een gegeven segment.
- *Oppik-invloed* (`Pick influence`). Een oppik-invloed stelt de poging van een AGV controller voor om te beginnen rijden over een gegeven segment en om de lading op te pikken op het station aan het einde van dit segment.

5.2.4 Manipulatie van Dynamiek in de Fabrieksomgeving

We bepreken de volgende reactiewet (`Reaction Laws`) in de gesimuleerde fabrieksomgeving:

- *Start rijden wet* (`Start driving law`). Deze wet bepaalt de reactie van de omgeving op een *rijd-invloed* (`drive influence`) of een *parkeer-invloed* (`park influence`). Een echte AGV begint niet altijd te rijden wanneer hij hiertoe wordt aangestuurd. Om dit te modelleren controleert de wet een aantal voorwaarden vooraleer een nieuwe rijd-activiteit (`driving activity`) wordt toegevoegd. Deze voorwaarden controleren of de AGV al niet bezig is met een rijd-, oppik- of neerzet-activiteit op het moment van de invloed; of het segment grenst aan het huidige station van de AGV en of de AGV in de gegeven richting over dat segment mag rijden (segmenten kunnen unidirectioneel zijn). De wet bepaalt geen nieuwe activiteit in geval aan een van deze voorwaarden niet geldig is, aangezien een echte AGV dan ook niet start met rijden.

We bespreken de volgende interactiewet (`Interaction Law`) in de gesimuleerde fabrieksomgeving.

- *Botsingswet* (`Collision law`). Een botsingswet controleert of AGV's in de gesimuleerde fabrieksomgeving botsen. In het geval de botsingswet een botsing detecteert, verandert het de activiteit(en) van de AGV's die betrokken zijn in de botsing, zodanig dat de AGV's stoppen met rijden op het moment van de botsing.

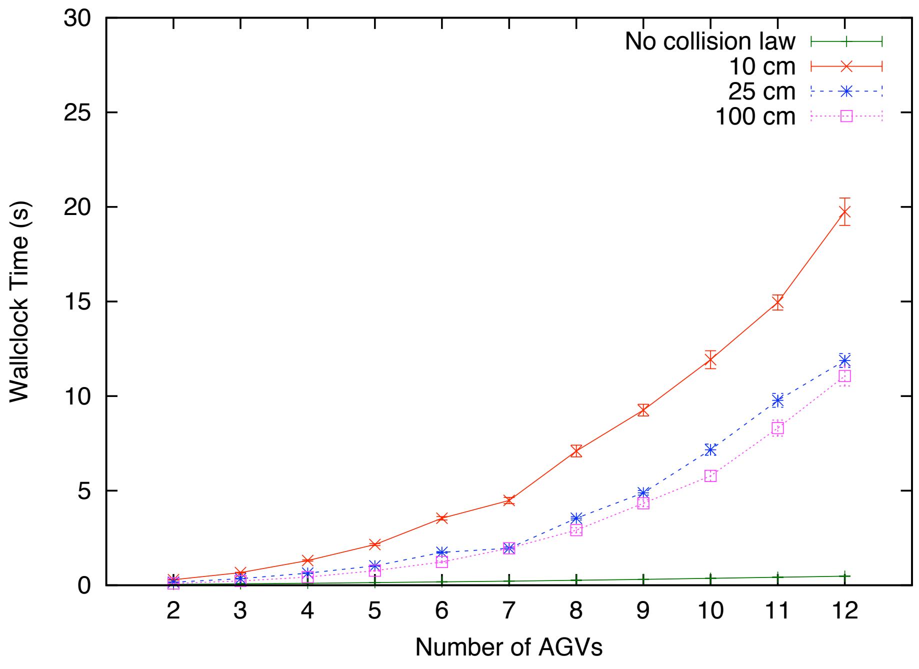
5.3 Evaluatie van de AGV Simulator

Ter evaluatie illustreren we de flexibiliteit van de AGV simulator en meten we de performantie.

Flexibiliteit van de AGV simulator is belangrijk om te kunnen experimenteren met AGV controle software in verschillende situaties. Door de verschillende wetten in te stellen kunnen de karakteristieken van bewegingen, energieverbruik, communicatie, botsingen, transporttaken, etc. in de gesimuleerde fabrieksomgeving afgestemd worden aan de noden van de simulatie. Bijvoorbeeld:

- Activiteiten kunnen aangepast worden naargelang de fysische karakteristieken van de echte AGV's. Bijvoorbeeld rijd-activiteiten kunnen ingesteld worden met de snelheids- en acceleratiekarakteristieken van de AGV's.

- Botsingsdetectie kan ingesteld worden aan de hand van de botsingswet (`Collision Law`). De botsingswet kan ingesteld worden om botsingen te detecteren met een vereiste nauwkeurigheid. Of in geval de AGV controllers nog geen botsingsvermijding ondersteunen, kan de botsingswet gedeactiveerd worden zodat AGV's elkaar niet hinderen.



Figuur 5: Performantie (in seconden) om 100 seconden simulatietijd te simuleren met de AGV simulator. De vier lijnen komen overeen met vier verschillende instellingen van de botsingswet: de botsingswet gedeactiveerd en de botsingswet ingesteld om botsingen te detecteren met een nauwkeurigheid van 10 centimeter, 25 centimeter en 100 centimeter. Elk punt in de figuur is het gemiddelde van 40 simulaties, waarvan tevens het 99% betrouwbaarheidsinterval is afgebeeld.

We hebben experimenten uitgevoerd om de flexibiliteit en de performantie van de AGV simulator te illustreren, zie Figuur 5. De experimenten gebruiken vier verschillende instellingen van de botsingswet: (1) de botsingswet gedeactiveerd;

de botsingswet ingesteld om botsingen te detecteren met een nauwkeurigheid van respectievelijk (2) 10 centimeter, (3) 25 centimeter en tenslotte (4) 100 centimeter.

We bespreken twee bevindingen die duidelijk uit de experimenten naar voren komen. Ten eerste tonen de experimenten aan dat de botsingswet een dominante factor is met betrekking tot de performantie van de AGV simulator. De complexiteit van de botsingswet is $O(n^2)$, met n het aantal AGV's. Ten tweede zien we dat de AGV simulator in staat is om simulaties uit te voeren sneller dan real-time. Immers, zelfs voor 12 AGV's met een nauwkeurigheid van 10 centimeter worden 100 seconden gesimuleerd met 20 seconden rekentijd.

6 Besluit

We geven een overzicht van de bijdragen van het onderzoek beschreven in dit proefschrift.

De belangrijkste bijdrage is de introductie van een expliciet modelleringsraamwerk voor software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen. Het modelleringsraamwerk biedt modelleringsconcepten aan die specifiek gericht zijn op het beschrijven van een simulatiemodel voor deze familie van simulaties. Deze modelleringsconcepten bieden expliciete ondersteuning voor de kernkarakteristieken van deze familie van simulaties. Bovendien zijn de modelleringsconcepten formeel gespecificeerd. Dit is cruciaal om het simulatiemodel te ontkoppelen van het simulatieplatform om het model uit te voeren.

Specifieke bijdragen van het onderzoek beschreven in dit proefschrift zijn (1) de introductie van een modelleringsraamwerk met specifieke modelleringsconcepten die de ontwikkeling van software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen ondersteunen, (2) de beschrijving van een formele specificatie van het modelleringsraamwerk, (3) de ontwikkeling van een simulatieplatform dat de modelleringsconcepten ondersteunt in een uitvoerbare simulatie, (4) de evaluatie van de bruikbaarheid van alle modelleringsconcepten van het modelleringsraamwerk in een industriële toepassing. We lichten deze bijdragen kort toe.

De introductie van een modelleringsraamwerk voor software-in-de-lus simulaties van gedistribueerde controle applicaties in dynamische omgevingen [7, 13, 9, 10]. Het modelleringsraamwerk laat toe om belangrijke karakteristieken van deze familie van simulaties op een expliciete manier te beschrijven. Het modelleringsraamwerk omvat twee delen:

- Het omgevingsdeel van het modelleringsraamwerk omvat modelleringsconcepten die op een expliciete manier ondersteuning bieden voor het modelleren van karakteristieken die typisch zijn voor dynamische omgevingen:
  - Modelleringsconcepten om de structuur van een dynamische omgeving te beschrijven in een simulatiemodel: `Environmental Entity`,

  `Environmental Property` en `Environment Layout`.

  – Modelleringsconcepten om de dynamiek in een dynamische omgeving te beschrijven in een simulatiemodel: `Activity`.
  – Modelleringsconcepten om de bronnen van dynamiek in een dynamische omgeving te beschrijven in een simulatiemodel: `Controller` en `Environment Source`.
  – Modelleringsconcepten om de wetmatigheden van dynamiek in dynamische omgeving te beschrijven in een simulatiemodel: `Reaction Law` en `Interaction Law`.

- Het controle applicatiedeel van het modelleringsraamwerk omvat modelleringconcepten die op een expliciete manier ondersteuning bieden voor het modelleren van belangrijke karakteristieken van de controle software die ingebed wordt in de simulatie:

  – Modelleringsconcepten om de uitvoeringstijd van de controllers van een gedistribueerde controle applicatie te beschrijven in een simulatiemodel: `Duration Primitive` en `Duration Mapping`.
  – Modelleringsconcepten om de controle interface van een gedistribueerde controle applicatie te beschrijven in een simulatiemodel: `Control Primitive`, `Control Parameter Mapping` en `Control Name Mapping`.

De beschrijving van een formele specificatie van het modelleringsraamwerk [13, 11]. De formele specificatie ontkoppelt de modelleringsconcepten van hun implementatie in een specifiek simulatieplatform. Het voordeel hiervan is tweeledig.

Enerzijds laat de formele specificatie toe om een simulatiemodel te beschrijven aan de hand van de modelleringsconcepten zonder dat kennis vereist is van het simulatieplatform dat gebruikt wordt om het simulatiemodel uit te voeren. Immers, de formele specificatie beschrijft op ondubbelzinnige wijze de betekenis en de onderlinge relatie van alle modelleringsconcepten.

Anderzijds laat de formele specificatie toe om verschillende alternatieve ontwerpbeslissingen tegen elkaar af te wegen wanneer een uitvoerbare simulatie gebouwd wordt. Immers, de formele beschrijving specificeert enkel de functionaliteit die nodig is om een simulatiemodel uit te voeren, zonder bepaalde ontwerpbeslissingen op te leggen.

De ontwikkeling van een simulatieplatform dat de modelleringsconcepten ondersteunt in een uitvoerbare simulatie [10, 21]. We hebben een simulatieplatform gebouwd dat aantoont dat het modelleringsraamwerk bruikbaar is om uitvoerbare simulaties te ontwikkelen. Het simulatieplatform omvat de functionaliteit om de modelleringsconcepten te ondersteunen in een uitvoerbare simulatie. Het simulatieplatform biedt ondersteuning voor simulaties (1) waarin de software van echte controllers kan ingebed worden, en (2) waarvan het simulatiemodel beschreven is aan de hand van de voorgestelde modelleringsconcepten.

We hebben een architectuur voorgesteld voor zo een simulatieplatform, en we hebben deze architectuur gedocumenteerd met verschillende architecturale views. De architectuur gebruikt aspect technologie om de controle software op een gebruiksklare manier te integreren in een simulatie, en om simulatieconcerns en applicatieconcerns te scheiden.

Een validatie in een industriële toepassing [8, 12]. We hebben het modelleringsraamwerk en het simulatieplatform toegepast in een industriële toepassing. We hebben een AGV simulator ontwikkeld die ondersteuning biedt voor software-in-de-lus simulatie van gedistribueerde controle applicaties die AGV's aansturen in fabrieksomgevingen.

Het modelleringsraamwerk biedt ondersteuning voor het simulatiemodel van de AGV simulator. De modelleringsconcepten van het modelleringsraamwerk bieden ondersteuning om de karakteristieken die typisch zijn voor een AGV systeem op een expliciete manier te beschrijven in een simulatiemodel. De ontwikkelaar kan het simulatiemodel van de AGV simulator aanpassen aan de noden van een bepaalde simulatie door *first-class* elementen van het simulatiemodel te activeren, deactiveren of fijn te stemmen.

Het simulatieplatform ondersteunt de uitvoering van de AGV simulator. Het simulatieplatform maakt het mogelijk om simulatiemodellen uit te voeren die specifiek aangepast zijn aan de noden van een bepaalde simulatie.

Een echte AGV controle applicatie omvat verschillende, complexe functionaliteiten, zoals routering, botsingsvermijding, transporttoekenning en batterij herlading. Deze functionaliteiten worden typisch incrementeel ontwikkeld, waarbij de nadruk ligt op bepaalde functionaliteiten en waarbij er van andere functionaliteiten abstractie wordt gemaakt. De AGV simulator maakt het mogelijk om (1) verschillende functionaliteiten afzonderlijk te evalueren, (2) verschillende aanpakken voor één enkele functionaliteit te vergelijken, en (3) functionaliteiten op een systematische manier met elkaar te integreren.

Slotbemerking

Bij het bouwen van een simulatiemodel dient men enkel die karakteristieken van het echte systeem te beschrijven, die volstaan voor het doel van de simulatie. Een simulatiemodel mag het echte systeem niet over-simplificeren, noch zo gedetailleerd zijn dat het duur wordt om het model te bouwen en uit te voeren. Daarom wordt modelleren vaak een kunst genoemd, in plaats van een wetenschap [19].

Echter, naarmate de vraag naar gedistribueerde controle applicaties toeneemt, worden meer en meer simulaties gebouwd om hun ontwikkeling te ondersteunen. De manier waarop simulatiemodellen voor degelijke systemen ontwikkeld worden, wordt meer en meer gemeenschappelijke kennis. Dit soort van gemeenschappelijke kennis kan expliciet beschreven worden in een modelleringsraamwerk.

Het werk van veel vooraanstaande onderzoekers vormt het fundament waarop onze onderzoeksbijdragen steunen. In het voorgestelde modelleringsraamwerk zit kennis en expertise vervat die we hebben opgedaan tijdens ons onderzoek, en die op zijn beurt steunt op tientallen jaren ervaring die opgebouwd werd in de onderzoeksgemeenschap.

Het modelleringsraamwerk toont aan hoe kennis en ervaring omtrent simulatie van gedistribueerde controle applicaties in dynamische omgevingen op systematische wijze kan gedocumenteerd worden, en we hebben aangetoond dat dit kan bijdragen tot het ontwikkelen van nieuwe simulaties. Daarom zijn we van mening dat modelleren geen kunst is *in plaats van* een wetenschap, maar een kunst *ondersteund* door wetenschap.

Referenties

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[2] Günter Bruns, Peter Mössinger, Daniel Polani, Ralf Schmitt, Rene Spalt, Thomas Uthmann, and Stefan Weber. Xraptor - a simulation environment for continuous virtual multi-agent systems - user manual.

[3] John S. Carson. Introduction to simulation: introduction to modeling and simulation. In *Winter Simulation Conference*, pages 7–13, 2003.

[4] S. G. Choi and W. H. Kwon. Real-time distributed software-in-the-loop simulation for distributed control systems. In *Proc. of the 1999 IEEE International Symposium on Computer Aided Control System Design*, pages 115–119, 1999.

[5] J. Ferber and J.P. Müller. Influences and reaction: A model of situated multiagent systems. In *Proceedings of the Second International Conference on Multi-agent Systems*, pages 72–79. AAAI Press, 1996.

[6] Ian J. Hayes, Michael Jackson, and Cliff B. Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2003.

[7] Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Simulating actions in dynamic environments. In *Conceptual Modeling and Simulation Conference, CMS2005, Track on Agent Based Modeling and Simulation in Industry and Environment*, 2005.

[8] Alexander Helleboogh, Tom Holvoet, and Yolande Berbers. Testing AGVs in Dynamic Warehouse Environments. In D. Weyns, V. Parunak, and F. Michel, editors, *Environments for Multiagent Systems II*, volume 3830 of *Lecture Notes in Computer Science*, pages 270–290. Springer-Verlag, 2006.

[9] Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management adaptability in multi-agent systems. In *Proceedings of the AISB 2004 Fourth Symposium on Adaptive Agents and Multi-Agent Systems*, pages 20–30. University of Leeds, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 2004.

[10] Alexander Helleboogh, Tom Holvoet, and Danny Weyns. Time management support for simulating multi-agent systems. In *Joint workshop on multi-agent and multi-agent-based simulation*, pages 31–40. Columbia University, 2004.

[11] Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Extending time management support for multi-agent systems. In *Multi-Agent and Multi-Agent-Based Simulation: Joint Workshop MABS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, volume 3415 / 2005 of *Lecture Notes in Computer Science*, pages 37–48. Springer-Verlag, GmbH, 2005.

[12] Alexander Helleboogh, Tom Holvoet, Danny Weyns, and Yolande Berbers. Towards time management adaptability in multi-agent systems. In *Adaptive Agents and Multi-Agent Systems III: Adaptation and Multi-Agent Learning*, volume 3394 / 2005 of *Lecture Notes in Computer Science*, pages 88–105. Springer-Verlag, GmbH, 2005.

[13] Alexander Helleboogh, Giuseppe Vizzari, Adelinde Uhrmacher, and Fabien Michel. Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems: Special issue on environments for multi-agent systems*, 14(1):87–116, February 2007.

[14] J. Himmelspach, M. Röhl, and A.M. Uhrmacher. Simulation for testing software agents - an exploration based on JAMES. In *Proc. of the 2003 Winter Simulation Conference, New Orleans, USA*, December 2003.

[15] Daniele Nardi, Martin Riedmiller, Claude Sammut, and José Santos-Victor, editors. *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Computer Science*. Springer, 2005.

[16] Patrick Riley and George Riley. SPADES — a distributed agent simulation environment with software-in-the-loop execution. In S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, editors, *Winter Simulation Conference Proceedings*, volume 1, pages 817–825, 2003.

[17] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1995.

[18] Thomas J. Schriber and Daniel T. Brunner. Inside discrete-event simulation software: how it works and why it matters. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 72–80. ACM Press, 1999.

[19] Robert E. Shannon. Introduction to the art and science of simulation. In *Winter Simulation Conference*, pages 7–14, 1998.

[20] A.M. Uhrmacher and B.G. Kullick. "plug and test- software agents in virtual environments. In *Proceedings of the 2000 Winter Simulation Conference*, volume 2, pages 1722–1729. Wyndham Palace Resort & Spa, Orlando, Florida, USA, December 2000.

[21] Danny Weyns, Alexander Helleboogh, and Tom Holvoet. The Packet-World: A testbed for investigating situated multiagent systems. In *Software Agent-Based Applications, Platforms, and Development Kits*, Whitestein Series in Software Agent Technologies, pages 383–408. Birkhauser Verlag, Basel - Boston - Berlin, September 2005.

[22] Michael J. Wooldridge. *Introduction to Multiagent Systems.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[23] Bernard Zeigler and Herbert Praehofer. *Theory of Modeling and Simulation.* Academic Press, January 2000.