6

On the Role of Software Architecture for Simulating Multi-Agent Systems

Alexander Helleboogh
Katholieke Universiteit Leuven

Danny Weyns
Katholieke Universiteit Leuven

Tom Holvoet
Katholieke Universiteit Leuven

6.1 Introduction

A control system is a software system connected to an underlying environment. The environment is the part of the external world with which the control system interacts, and in which the effects of the control system will be observed [Jackson, 1997]. The task of a control system is to ensure that particular functionalities are achieved in the environment. A multi-agent control system is a control system of which the software is a multi-agent

0-8493-8597-0/01/$0.00+$1.50

system, i.e. a system that consists of a number of autonomous software components, called agents, that collaborate to achieve a common goal. Examples of multi-agent control systems include manufacturing control systems [Verstraete et al., 2006; Brueckner, 2000], collective robotic systems [Gu and Hu, 2004; P. Varshavskaya and Rus, 2004; Bredenfeld et al., 2006], traffic control systems [Wang, 2005; Roozemond, 1999; Dresner and Stone, 2005] and sensor networks [Sinopoli et al., 2003; DeLima et al., 2006].

Simulation can be used to support the development of multi-agent control systems. Simulation offers a safe and cost-effective way for studying, evaluating and configuring the behavior of a multi-agent control system in a simulated setting [Himmelspach et al., 2003]. In this chapter, we focus on software-in-the-loop simulations for multi-agent control systems in dynamic environments. This family of simulations has the following characteristics: (1) the environment to-be-simulated is dynamic. In a dynamic environment, the operating conditions of a multi-agent control system are continuously changing; (2) the control software of the real multi-agent control system is embedded in the simulation.

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

- *Simulating dynamic environments is complex.* In a dynamic environment an agent cannot determine the outcome of its actions a priori [Ferber and Müller, 1996; Helleboogh et al., 2005]. 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 multi-agent control system in a simulation is complicated.* The devices on which the multi-agent control system is deployed in the real world determine how fast the control software can execute and consequently how much time it takes the software 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 software 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 multi-agent control system in a simulation [Uhrmacher and Kullick, 2000; Anderson, 1997].

Special-purpose modeling constructs and simulation platforms incorporate a large body of expertise on building software-in-the-loop simulations of multi-agent control system in dynamic environments. In simulation platforms this expertise is primarily reified in terms of reusable code libraries and frameworks. In our own research, we have built up knowledge and best practices in several cases, including simulations of the Packet-World [Weyns et al., 2005a], Lego Mindstorms robots [Borgers, 2006], traffic control systems [Weyns et al., 2007]

and Automated Guided Vehicles [Helleboogh et al., 2006]. We experienced that we could reuse a lot of expertise across these cases. Nevertheless, a substantial part of this expertise was reused implicitly, as explicit reuse of code libraries and frameworks was rather limited.

In this chapter, we put forward a software architecture for a simulation platform targeted at software-in-the-loop simulation of multi-agent control systems in dynamic environments. This software architecture explicitly documents the knowledge and practice incorporated in a simulation platform in the form of a reusable artifact, clearly distinguished from the code of the simulation platform. More than reusable code libraries and software frameworks, a software architecture captures the essence of a complex software system by identifying key stakeholder concerns and by explicitly specifying how software needs to be structured and behave to address the concerns. The software architecture integrates the essential architectural building blocks for such a simulation platform and explicitly documents the rationale and tradeoffs that underpin its design. Software architecture provides a systematic way to capture and share expertise that was acquired across several cases in a form that has proven its value for software development.

This chapter is structured as follows. After presenting some essential background in Section 6.2, we introduce an industrial case in Section 6.3, i.e. a AGV transportation system that comprises a multi-agent control system for controlling automated guided vehicles (AGVs) that transport loads in a warehouse. This case will be used as an illustration throughout this chapter. In Section 6.4, we summarize our previous work on special-purpose modeling constructs for modeling software-in-the-loop simulations of multi-agent control systems in dynamic environments, and we apply these modeling constructs to describe a simulation model for the AGV transportation system. In Section 6.5, we describe the software architecture for a simulation platform that supports these special-purpose modeling constructs. We document the architecture and explain how important functional and quality requirements are achieved. In Section 6.6, we evaluate a simulation platform that implements this architecture and that was used for conducting simulations of the AGV transportation system. In Section 6.7 we discuss related work on software-in-the-loop simulations of multi-agent control systems in dynamic environments. Finally, we point out directions for future research and we draw conclusions in Section 6.8.

6.2 Background

In this section, we describe the necessary background information. We focus on (1) the relation between a system and its environment, (2) the characteristics of multi-agent control systems, and (3) software-in-the-loop simulation.

6.2.1 A System and Its Environment

Software systems are designed to satisfy particular functional and quality requirements. These requirements are issued by the group of stakeholders involved. For systems that interact with the external world via sensors and actuators, these requirements do not directly concern the software system [Jackson, 1997]. The requirements primarily concern the environment in which the system will be installed [Hayes et al., 2003]. The task of a system is to ensure that particular functionalities are achieved in the environment.

Jackson defines the environment as the part of the external world with which the system interacts, and in which the effects of the system will be observed and evaluated [Jackson, 1997]. The distinction between the environment and the system is partly a distinction between what is given and what is to be constructed.

Figure 6.1 depicts the relation between a system and its environment. The system interacts with the environment by means of shared phenomena that are directly accessible via sensors and actuators. However, influencing phenomena that are private to the environment can only be done in an indirect manner: using sensors and actuators, the system tries to bring about causal chains to observe and affect private phenomena in the environment.

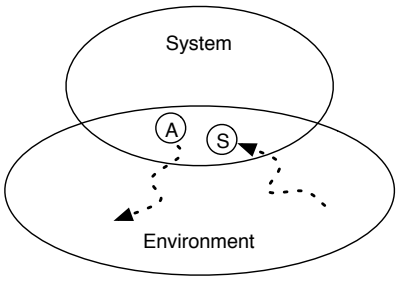

FIGURE 6.1 The system and its environment [Jackson, 1997]

As an example, consider a car's cruise control system. The environment of the system consists of the car, its driver, the atmospheric conditions, the road the car drives on, etc. The cruise control system interacts with its 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 requirements of the cruise control system are expressed in terms of phenomena in its environment. For example, the cruise control system should ensure that the car drives at a constant speed across the road. The cruise control system can only affect the car's speed indirectly, i.e. by relying on a causal chain between manipulations of the throttle actuator and alterations in the speed of the car.

Today, the environments in which software systems have to operate are typically *dynamic* [Issarny et al., 2007]. A dynamic environment is an environment that changes frequently. In a dynamic environment, the operating conditions of a system are continuously changing. For example, the environment of the cruise control system is dynamic: the road may go uphill or downhill, turbulence or wind may arise that causes additional or reduced drag. These phenomena affect the causal chains by means of which the cruise control system affects the environment. For example, in case the road goes uphill, changing the throttle will affect the car's speed in a different manner compared to the case that the road goes downhill.

As a dynamic environment can significantly affect the software system, it is generally considered good practice to capture properties and assumptions regarding the environment in an explicit model [Jackson, 1997; Hayes et al., 2003]. Such a model of the environment includes assumptions about the frequency and nature of the changes in the environment, the accuracy and latency of sensors and actuators, the assumptions about the causal chain from activation of an actuator to the changes in the actual environment, etc. Such a model describes essential characteristics and assumptions about the environment that must be checked for proper deployment of the system.

6.2.2 Characteristics of Multi-Agent Control Systems

A multi-agent control system is a distributed software application that continuously and autonomously acts in, and reacts to, an underlying environment. Examples of multi-agent

control systems include manufacturing control systems [Verstraete et al., 2006; Brueckner, 2000], collective robotic systems [Gu and Hu, 2004; P. Varshavskaya and Rus, 2004; Bredenfeld et al., 2006], traffic control systems [Wang, 2005; Roozemond, 1999; Dresner and Stone, 2005] and sensor networks [Sinopoli et al., 2003; DeLima et al., 2006]. Figure 6.2 gives a schematic overview of a multi-agent control system in an environment.

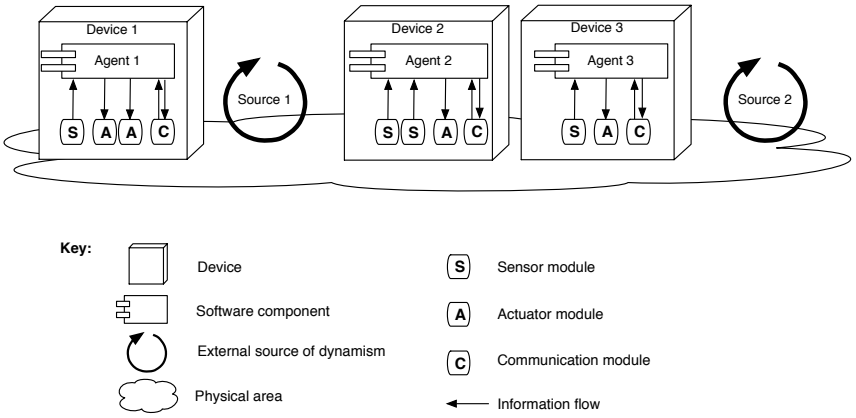

FIGURE 6.2 Schematic view of a multi-agent control system in an environment

A multi-agent control system consists of several agents. Agents are autonomous software components that are distributed in the environment and that cooperate to solve a particular problem in the environment. In Figure 6.2, three agents are depicted: `Agent 1`, `Agent 2` and `Agent 3`.

The agents of a multi-agent control system are deployed on particular devices in the environment. A device consists of a software and a hardware part. The software part is one of the agents that constitutes the multi-agent control system, whereas the hardware part comprises sensor, actuator and communication modules. An agent can use the sensor, actuator and communication modules of its device to interact with the environment. Figure 6.2 depicts three devices in the environment: `Device 1`, `Device 2` and `Device 3`.

The environment of a multi-agent control system is the part of the external world in which the problem resides and in which the effects of the control system, once installed and set in operation, will be observed [Hayes et al., 2003]. Typically, a multi-agent control system 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 multi-agent control system. Figure 6.2 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 multi-agent control system.

Designing and testing a multi-agent control system is complex as it requires an integrated approach that takes into account the environment in which the application is situated [Hu and Zeigler, 2005]. A multi-agent control system should take into account dynamism originating from other systems, processes or humans in the environment and react appropriately to their presence.

6.2.3 Software-in-the-loop Simulation Mode

Software-in-the-loop simulation mode denotes simulations in which the software of the real control system is embedded in the simulation loop. Software-in-the-loop simulation mode is depicted in Figure 6.3. The simulation contains parts of the real system, i.e. the control software, together with simulated parts, i.e. the device hardware and the environment. The executable code of the real control system is directly embedded in the simulation. In software-in-the-loop simulation mode, the software of the real control system is deployed on simulated devices that reside within a simulated environment with simulated sources of dynamism.

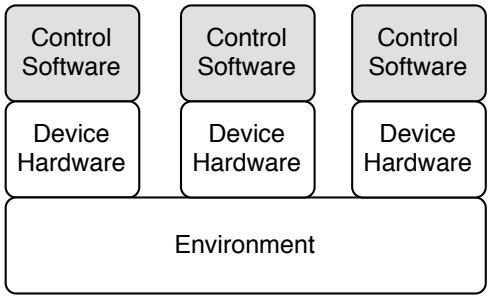

FIGURE 6.3 Schematic view of software-in-the-loop simulation mode. White blocks are simulated parts. Grey blocks are parts of the real system that are integrated in the simulation loop.

Software-in-the-loop simulation is typically used during the late stages of application development, i.e. after the software of the multi-agent control system (or parts thereof) has been implemented. Software-in-the-loop simulation enables experimenting with the agents of a multi-agent control system on simulated devices before deployment on real devices. Software-in-the-loop simulation is extensively used for the development of control systems for robots. For example, software-in-the-loop simulations enable testing the robustness and fault-tolerance of control systems for robots [Bräunl et al., 2006; Finkenzeller et al., 2003] or can facilitate parameter estimation of a control systems for robots [Velez and Agudelo, 2006].

6.3 AGV Transportation System

We introduce a real-world case that will be used as an example throughout this chapter. The case comprises the development of a multi-agent control system that controls automated guided vehicles (AGVs) in warehouse environments. An AGV is an unmanned, computer-controlled transportation vehicle using a battery as its energy source (see Figure 6.4). 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).

An AGV control system was developed in the EMC^2 (Egemin Modular Controls Concept) project. Egemin N.V. is a Belgian manufacturer of AGVs, and develops control software

FIGURE 6.4 An AGV in a cheese factory.

for automating logistics in warehouses and manufactories using AGVs.

6.3.1 Physical Setup of an AGV Transportation System

Figure 6.5 shows a three dimensional view on an AGV transportation system. 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.

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.

6.3.2 AGV Control System

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

Functionalities of an AGV Control System

The main functionality of an AGV control system 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. Ab-

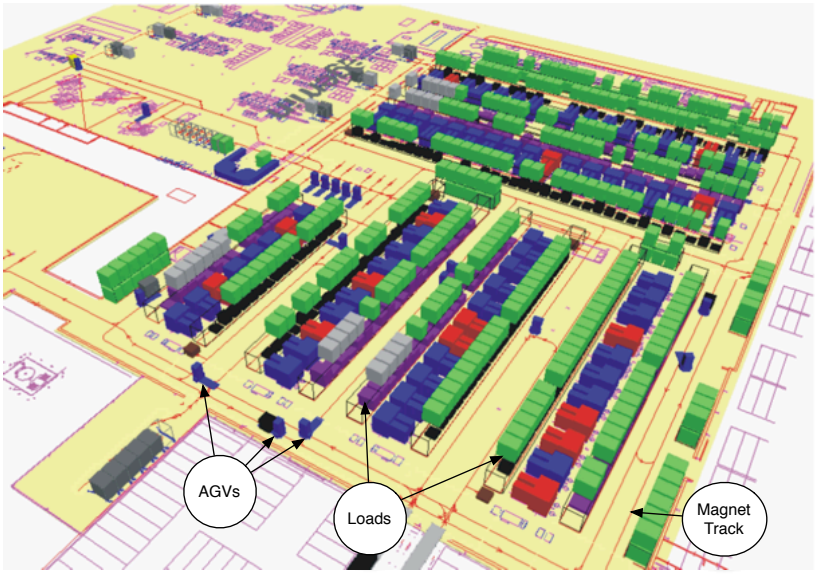

FIGURE 6.5 Three dimensional view on an AGV transportation system.

stracting from the origin of the transports, systems that generate transports for the AGV control system are called client systems. Client systems input transports to the AGV control system, and expect a confirmation from the AGV control system when the transport is done.

In order to handle transports, the AGV control system 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 toward each other. Since AGVs in general cannot drive in reverse, none of the two AGVs can move forward or backward. The AGV control system must ensure that manual intervention for such situations is avoided.

Besides handling transports efficiently, the AGV control system 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 AGV's 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.

AGV Steering System

To control an individual AGV, it is equipped with an AGV steering system developed by Egemin, called E'nsor*. 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 AGV's 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 system 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 meters. 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.

- 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

*E'nsor is an acronym for Egemin Navigation System On Robot.

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.

6.3.3 Requirements of an AGV Simulator

In the context of the EMC^2 project, we developed an *AGV simulator*. The AGV simulator enables (1) safe experimentation and testing of AGV control systems 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.

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

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

- *Support for routing.* To evaluate or compare routing behaviors of AGV agents, 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 agents, 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 agents, the AGV simulator should simulate several transport profiles generated by client systems.
- *Support for battery charging.* To evaluate the charging strategy of AGV agents, 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 system, these functionalities are typically developed iteratively. The AGV simulator should enable testing partial AGV control systems in which some of these functionalities are present and others not yet (fully) operational.

To support evaluating different functionalities of AGV control systems in a variety of settings, modifying core parts of the model 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 include altering the AGV agent 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 Modeling Multi-Agent Control Applications in Dynamic Environments

In this section, we (1) summarize our previous work on a modeling framework for software-in-the-loop simulations of multi-agent control systems in dynamic environments, and (2) apply this modeling framework to formulate a simulation model for the AGV transportation system.

6.4.1 Overview of Modeling Framework

The modeling framework offers special-purpose modeling constructs for formulating a simulation model for software-in-the-loop simulations of multi-agent control systems in dynamic environments. The modeling framework captures core characteristics of these simulations in a first-class manner.

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 multi-agent control systems in dynamic environments. Examples include simulations of the Packet-World [Weyns et al., 2005a], Lego Mindstorms robots [Borgers, 2006] and Automated Guided Vehicles [Helleboogh et al., 2006]. On the other hand, the modeling constructs are underpinned by existing practice on modeling dynamic environments of multi-agent control systems. For a detailed motivation, discussion and a formal description of all modeling constructs as well as of the evolution of the model, we refer to [Helleboogh et al., 2007; Helleboogh, 2007]

The modeling framework comprises two complementary parts: an environment part and a control software part. We give a brief brief overview of the modeling constructs in each part of the modeling framework.

Modeling Dynamic Environments

The environment part 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 multi-agent control systems.

Figure 6.6 gives a graphical overview of the modeling framework for dynamic environments, depicting all modeling constructs and the relations between the 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

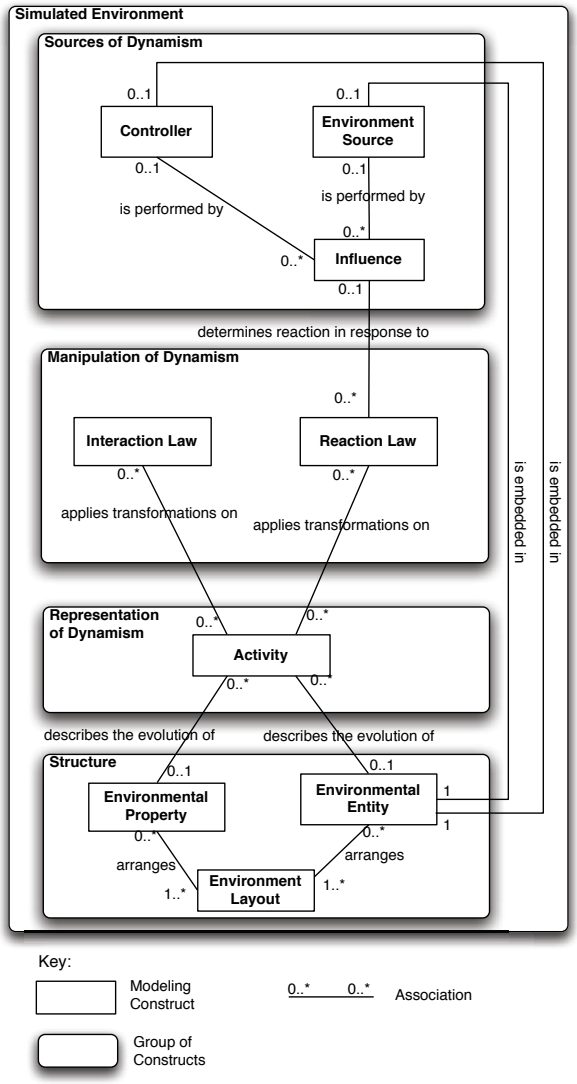

FIGURE 6.6 Overview of the constructs in the environment part of the modeling framework.

which a multi-agent control system 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 a particular evolution of a particular environmental entity or

property over a particular time interval. 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 an agent 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.

The `Controller` is a source of dynamism that is part of the multi-agent control system. An example of a controller is an agent 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 multi-agent control system. 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 the controller or of an environment source to affect the environment. An example of an influence is the attempt of an agent 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`.

Modeling the Software of a Multi-Agent Control System

In software-in-the-loop simulations, the software of the real controllers of the multi-agent control system is embedded in a simulated environment. The control software part of the modeling framework comprises modeling constructs that capture key characteristics of the software of the multi-agent control system that is embedded in the simulation.

Figure 6.7 is a detailed view on the group of modeling constructs to represent the sources of dynamism in Figure 6.6, with additional modeling constructs for the controller. We give

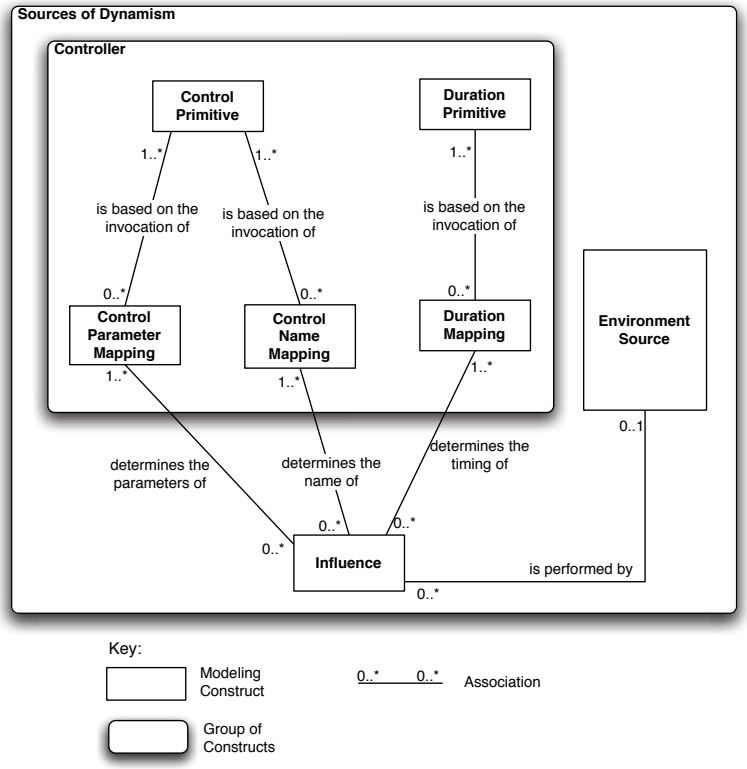

FIGURE 6.7 Overview of the modeling constructs for the control software and their associations.

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 wallclock 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 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 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.

6.4.2 Simulation Model of the AGV Transportation System

We apply the modeling framework to formulate a simulation model for an AGV simulator. The AGV simulator supports the evaluation of new or altered features of a multi-agent control system that controls automated guided vehicles in warehouse environments.

The constructs of the modeling framework are used to capture key characteristics of the AGV system in a first-class manner. This enables a developer to 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.

We start with an overview of the simulation model of the warehouse environment. Afterward, we focus on the simulation model for integrating the AGV control system. For a detailed discussion of the simulation model of the AGV Transportation System, we refer to [Helleboogh, 2007].

Simulation Model of the Warehouse Environment

Figure 6.8 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 6.6. The simulation model is organized in four parts, in analogy with Figure 6.6. We discuss a number of examples for each of the parts of the simulation model for warehouse environments.

Structure of the Warehouse Environment.

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. We give examples of environmental entities. *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; A *WiFi access point* enables communication between AGVs and transport bases or among several AGVs. 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.

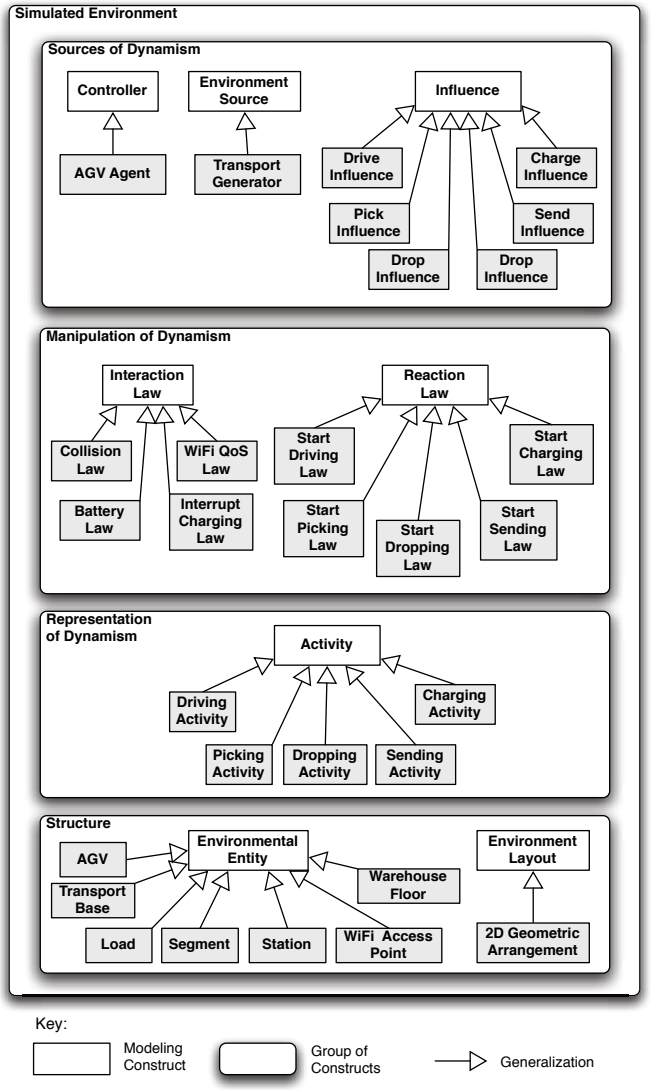

FIGURE 6.8 Overview of the simulation model of the simulated warehouse environment. The grey parts are specific instantiations of the modeling constructs for the AGV simulator.

We arrange the entities in the simulated warehouse environment according to a continuous *2D-geometric layout*. This layout expresses the spatial positioning of all entities with respect to each other.

Dynamism in the Warehouse Environment.

Dynamism in the simulated warehouse environment is modeled in terms of `activities`. For example, *driving activities* represent the driving of an AGV across a segment on the warehouse floor until the station at the other end of that segment is reached. Each driving activity is characterized by the AGV involved in the movement, the segment over which the AGV moves, the direction of the movement over that segment, the time interval during

which the movement takes place and the acceleration profile of the AGV during that movement. *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`. *AGV agents* are the control software that is embedded in an AGV. AGV agents constitute the AGV control system. Each AGV agent is responsible for controlling an AGV and for coordinating with other AGVs for routing, collision avoidance, transport assignment and battery charging. 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 system. Consequently, transport generators are environment sources of dynamism. A transport generator is deployed on a transport base.

Sources of dynamism can manipulate the environment by means of performing `influences`. For example, a *drive influence* represents that attempt of an AGV agent to start driving over a given segment in a given direction. A *send influence* represents the attempt of an AGV agent or a transport generator to send a message.

Manipulation of Dynamism in the Warehouse Environment.

The way dynamism in the warehouse environment can be manipulated is modeled by means of `reaction laws` and `interaction laws`.

An example of a reaction law is a *start driving law*. Start driving law defines 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.

An example of an interaction law is a *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 transforms the driving activity/activities involved with driving activity/activities that stop at the time the collision occurs. 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.

Simulation Model for Integrating the AGV Agent Software

Figure 6.9 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 6.7. We elaborate on the way the AGV agent software interacts with the environment and the way the execution time of the AGV agent software is captured in the simulation model.

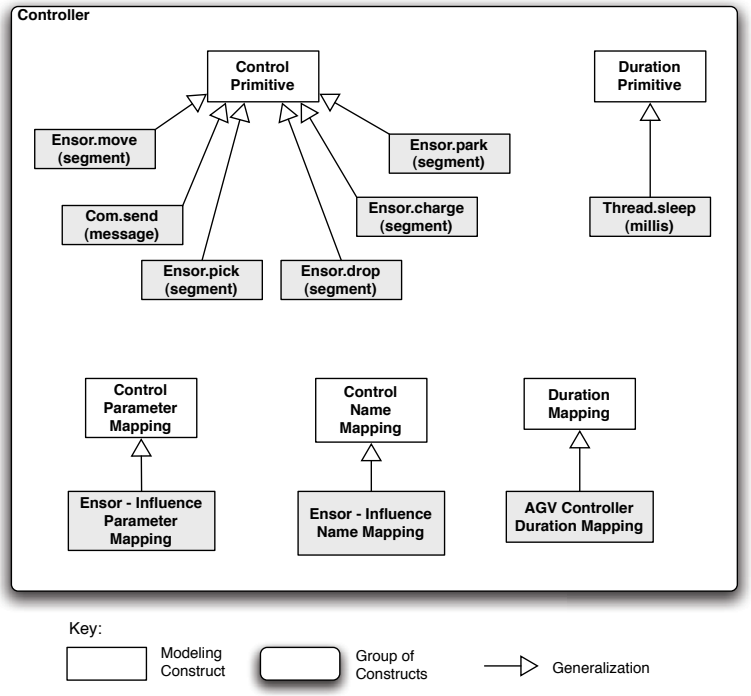

FIGURE 6.9 Overview of the simulation model for integrating the AGV control software in a simulation. The grey parts are specific instantiations of the modeling constructs for the AGV simulator.

Control Interface of AGV Agents.

The interaction of the AGV agent software with the warehouse environment is modeled in terms of `control primitives` and a `control name mapping` and `control parameter mapping`.

We discuss two examples of control primitives. *Ensor.move(segment)* is an E'nsor control primitive that instructs E'nsor to drive the AGV over the given segment. *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 Agents.

The execution time of AGV agent software is modeled in terms of `duration primitives` and a `duration mapping`.

The duration primitives are typically dependent upon the AGV agent software. Therefore, the developer should specify custom duration primitives for the AGV control software that is to be embedded in the simulation. There is one default duration primitive captured in the simulation model: *Thread.sleep(millis)*. *Thread.sleep(millis)* suspends the execution of an AGV agent 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 system.

We employ an *AGV agent duration mapping* to specify the duration of invocations of duration primitives. By default, the *AGV agent 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 Architecture of the Simulation Platform

In this section, we describe the software architecture of a simulation platform that supports the modeling constructs of the modeling framework described in Section 6.4.

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" [Bass et al., 2003]. The software architecture captures the essence of a complex software system by identifying key stakeholder concerns and by explicitly specifying how software needs to be structured and behave to address the concerns. As such, a software architecture is a reusable artifact for the creation of such a simulation platform. We developed a simulation platform that implements this architecture and we applied this simulation platform to support software-in-the-loop simulations for evaluating, comparing and integrating several functionalities of a multi-agent control system for steering AGVs.

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 [Bass et al., 2003]. Each view presents a particular perspective on the architecture or a part thereof. 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. Afterward, we document each view for the simulation platform using a graphical notation and we explain how important quality requirements are realized.

This section is structured as follows. In Section 6.5.1, 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 6.5.2. We describe a component and connector view of the functionality to support dynamic environments in Section 6.5.3. We elaborate on a component and connector to explain the simulation engine that synchronizes all parts of the simulation in Section 6.5.4. In Section 6.5.5, 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 an agent in Section 6.5.6.

6.5.1 Requirements

The goal of the simulation platform is to provide run-time support for software-in-the-loop simulations of multi-agent control systems in dynamic environments, of which the simulation model is described in terms of the modeling constructs proposed in Section 6.4. 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 Section 6.4.1. 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 Section 6.4.1. 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 system.* The simulation platform should provide support for embedding the software of the control system in a flexible way, i.e. with minimal effort from the developer. The fact that some simulation concerns crosscut with the control system'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 system 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.

6.5.2 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 6.10. We first discuss the main elements and their properties. Afterward, we describe their interfaces and explain how important qualities are realized.

Elements and Their Properties

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

- `Controller` is a software module of the real multi-agent control system that is embedded in the simulation platform in order to test or configure it. A multi-agent control system consists of several controllers, i.e. agents, working in parallel and cooperating to solve a problem in the environment. An controller has well-defined ways to sense the environment and to act upon it. An example of an controller is an agent program to controls a particular robot in a manufacturing plant.
- `Simulation Platform` is the medium in which controllers of a multi-agent control system 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 multi-agent control system.
  - To manage the execution of all controllers of the multi-agent control system according to the specified duration model.

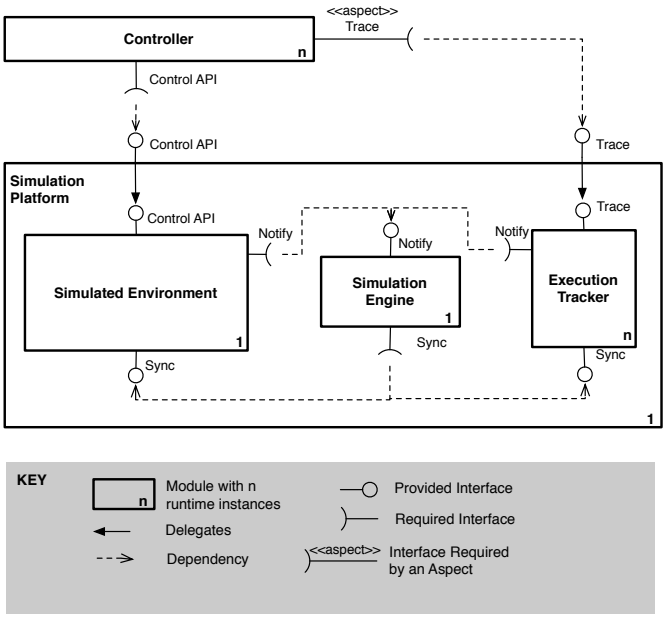

FIGURE 6.10 Top-level module decomposition view of the simulation platform.

– 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 multi-agent control system. The Simulated Environment encapsulates all functionality to support the modeling constructs described in Section 6.4.1.
- `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 multi-agent control system. This module encapsulates all functionality to support the modeling constructs for the control software described in Section 6.4.1. Tracing the execution of a controller includes (1) determining the execution time consumed by a particular controller of the multi-agent control system 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.

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 6.5.5.

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 6.5.3.

Architectural 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 control software 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 6.5.5 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 En-

vironment 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.

6.5.3 Component and Connector View of the Simulated Environment

A component and connector view [Clements et al., 2002; Ivers et al., 2004] 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 [Ivers et al., 2004].

The component and connector view of the Simulated Environment is depicted in Figure 6.11. This view gives a detailed perspective on the Simulated Environment module of Figure 6.10. The Simulated Environment supports the modeling constructs for dynamic environments described in Section 6.4.1. We first discuss the main elements and their properties. Afterward, we describe how they are connected and explain how important qualities are realized.

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. Afterward, 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

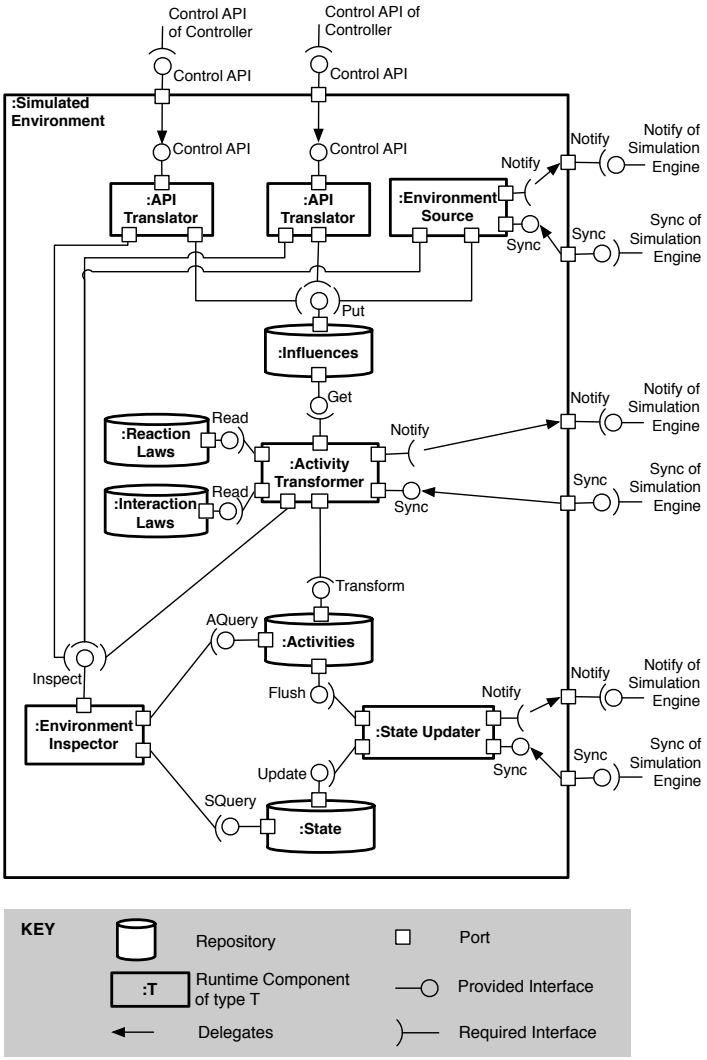

FIGURE 6.11 Component and connector view of the simulated environment.

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 con-

trollers of the multi-agent control system on the one hand, and from environment sources external to the multi-agent control system 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.
- `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 functionality 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 and control name mapping (see Section 6.4.1). 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 6.11, two `API translators` are depicted. Each `API translator` is connected to a controller of the multi-agent control system.
- `Environment Source` is responsible to mimic the behavior of a source of dynamism in the environment that is external to the multi-agent control system. `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 multi-agent control system. In Figure 6.11, one instance of an `Environment Source` is depicted.

Interface Descriptions

Figure 6.11 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.

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 6.5.4, when the simulation engine is discussed.

Architectural 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 multi-agent control system. 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 6.5.4.

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 toward 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 afterward.

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.

6.5.4 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 6.12. We first discuss the main elements and their properties. Afterward, we describe how they are connected and explain how important qualities are realized.

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 that need synchronization are the following: the various sources of dynamism that act in parallel, i.e. the `Controllers` of the multi-agent control system and the `Environment`

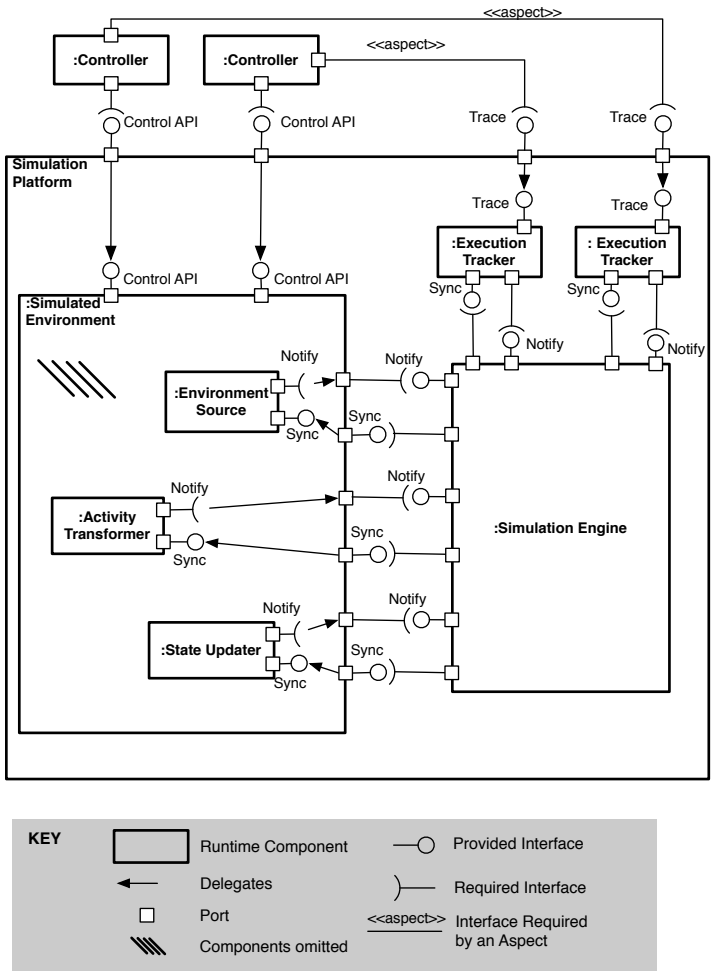

FIGURE 6.12 Component and connector view of the simulation engine.

`Sources` external to the multi-agent control system, 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 6.5.3). An `Environment Source` can access the environment in several ways, i.e. by performing an influence or sensing 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 6.5.5). 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 6.5.3). 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 6.5.3). 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`.

Interface Descriptions

Figure 6.12 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.

Architectural 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 discrete event synchronization algorithm [Chandy and Misra, 1981]) 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.

6.5.5 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 6.10 and 6.12. The `Trace` interface enables monitoring the execution of a controller by tracking its duration primitive invocations and control primitive invocations (see Section 6.4.1). 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 multi-agent control system 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 6.5.5. The way aspect-oriented programming is used in the simulation platform is described in Section 6.5.5. We emphasize how important qualities are realized in Section 6.5.5.

Aspect-Oriented Programming

Tracking the execution of a controller is a *crosscutting concern*, i.e. the functionality to do this crosscuts a multi-agent control system'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 multi-agent control system, resulting in "tangled code" that is excessively difficult to develop and maintain. Aspect-oriented programming [Kiczales et al., 1997, 2001] 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.

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

We describe a way to flexibly embed a multi-agent control system 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 multi-agent control system 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 6.5.4). 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 6.13 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 6.13 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.

Architectural Rationale

Flexibility of embedding a multi-agent control system.

Aspect weaving supports flexibly embedding the multi-agent control system in a simulation: the developer is no longer bothered to modify a multi-agent control system 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 multi-agent control system'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 6.13).

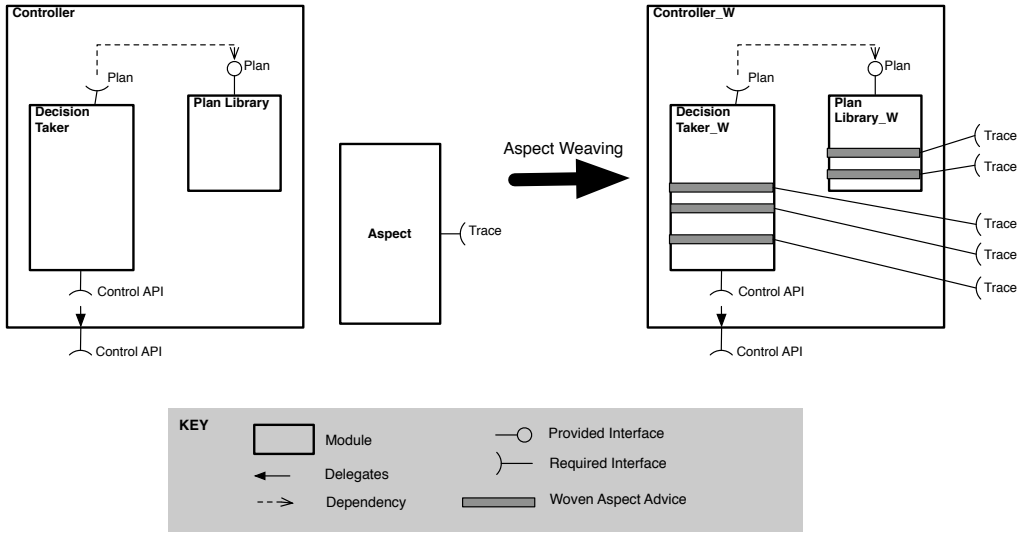

FIGURE 6.13 Controller before (left) and after (right) aspect weaving.

6.5.6 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 multi-agent control system. Tracing the execution of a controller includes (1) determining the execution time consumed by a particular controller of the multi-agent control system 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 6.14 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. Afterward, we describe how they are connected and explain how important qualities are realized.

Elements and Their Properties

Figure 6.14 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 particular `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 6.4.1). 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`

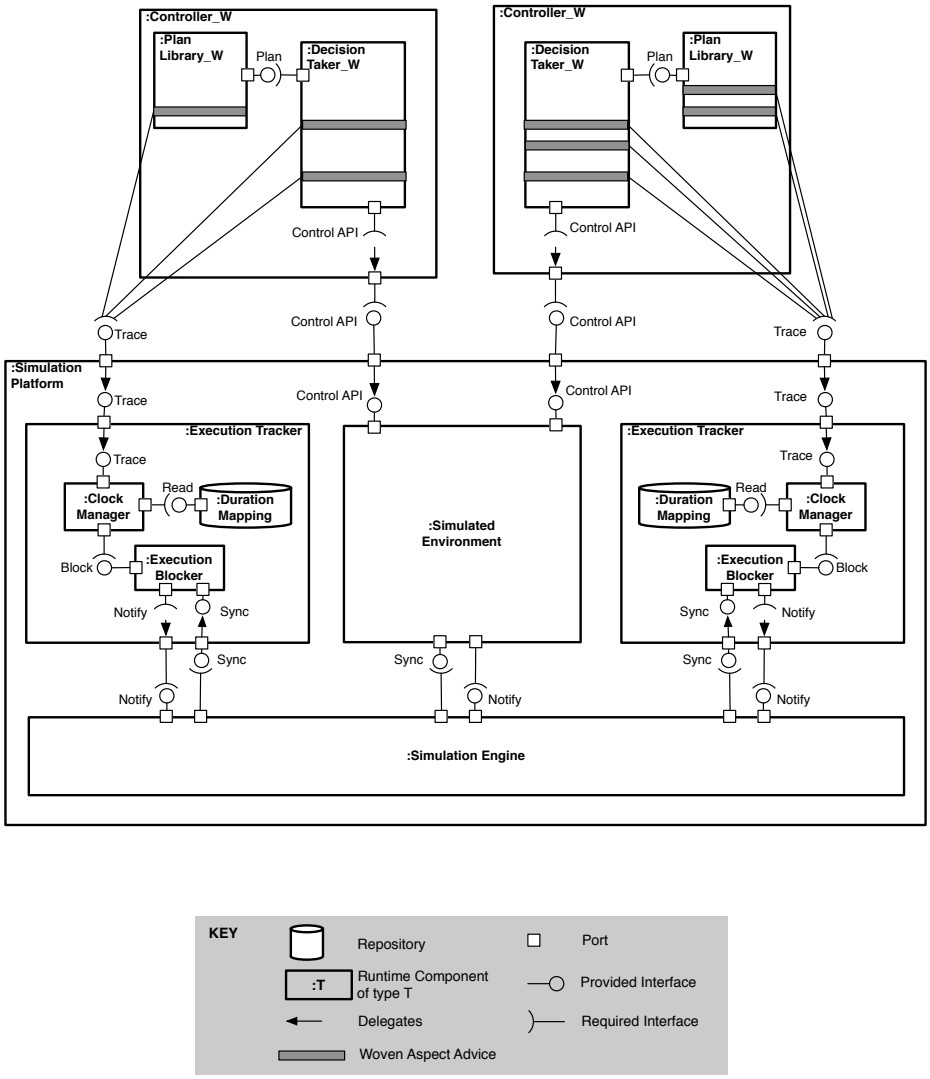

FIGURE 6.14 Component and connector view of controllers and execution trackers.

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`.

Interface Descriptions

Figure 6.14 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 6.5.5) a `Controller's` execution is intercepted and redirected to the `Clock Manager` each time a duration primitive invocation or control primitive invocation is per-

formed. The caller of the `Trace` interface specifies the characteristics of the duration primitive invocation or control primitive invocation (see Section 6.4.1).

- `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.

Architectural 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.

6.6 Evaluating the AGV Simulator

We developed a simulation platform that implements this architecture and we applied this simulation platform to support software-in-the-loop simulations for evaluating, comparing and integrating several functionalities of a multi-agent control system for steering AGVs. We now focus on evaluating flexibility and performance of the AGV simulator.

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 multi-agent control systems 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, as this law is a dominant factor for the performance of the AGV simulator.

Setup of the Experiments

The goal of the experiments is to illustrate both flexibility and performance of 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 to detect collisions 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 agents. 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 agent is programmed to poll the status of its AGV every second. AGVs drive around randomly: as soon an AGV agent 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 the following computer platform*: Intel Pentium 4, 2.8GHz, 512MB of memory, Java 1.5.0.

Measurements

Figure 6.15 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 is not limited to *real-time simulation*, but supports *as-fast-as-possible simulations*. For example, 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.

6.6.3 Multi-Agent System Development Supported by the AGV Simulator

The AGV simulator was extensively used during the development of a multi-agent control system in the EMC^2 project. The AGV simulator provides the necessary support for a developer to evaluate different functionalities an AGV control system in isolation, or to

*SciMark 2.0 benchmark score: 174.5 Mflops

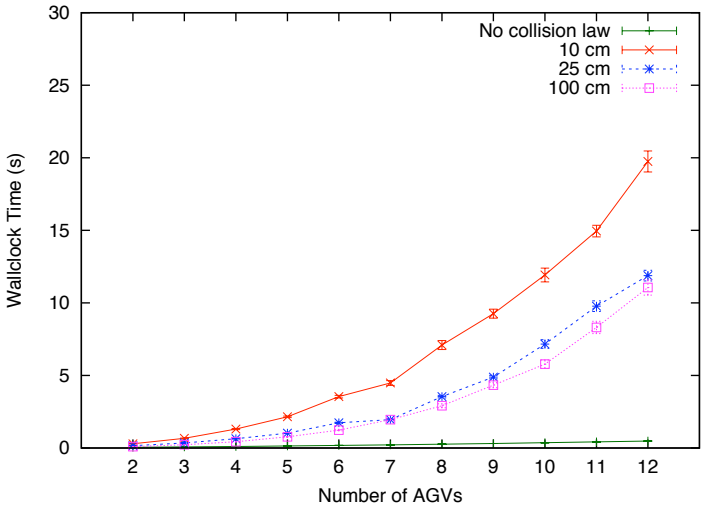

FIGURE 6.15 Performance (in seconds of wallclock time) for simulating 100 seconds of simulation time with the AGV simulator. The four lines correspond to four different configurations of the collision law: the collision law deactivated and the collision law detecting with an accuracy of 10 centimeters, 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.

compare alternative solutions. We give a number of examples.

- *Virtual environment based routing* [Weyns et al., 2005b]. In this approach, AGV agents 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 agents 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 agents 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 agents to drive over the warehouse layout and it handles the exchange of messages of the middleware.
- *Hull-based collision avoidance* [Weyns et al., 2005d]. AGV controllers avoid collisions by coordinating with other AGVs using the virtual environment. AGV agents 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. Afterward, 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* [Weyns et al., 2006; Schols, 2005]. In this approach, transport tasks emit fields into the virtual environment that attract idle AGVs. To avoid multiple AGVs driving toward the same transport, AGVs emit

repulsive fields. AGVs combine received fields and follow the gradient of the combined fields that guide them toward 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* [Weyns et al., 2008]. Besides field-based transport assignment, a dynamic version of the Contract Net protocol [Smith, 1980] was developed to assign transports to AGVs. This protocol, called DynCNET, allows AGV agents to reconsider the assignment of transports while they drive toward 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 system. For example, a modular AGV agent [Delbaere and Lamberigts, 2007] 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 Related Work

We focus our discussion of related work on two facets. First, in Section 6.7.1, we compare our approach with existing simulation platforms that are specifically aimed at software-in-the-loop simulation of multi-agent control systems in dynamic environments. Second, in Section 6.7.2, we zoom in on integrating the control software in a simulation and compare our work with existing approaches. For both these facets, we start with an overview of related approaches, and afterward we compare these approaches with our work.

6.7.1 Special-Purpose Simulation Platforms

Simulation platforms that are specifically aimed at software-in-the-loop simulation of multi-agent control applications in dynamic environments include:

- XRaptor [Bruns et al.] 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.
- SPARK [Obst and Rollmann, 2004] is a simulation platform for physical multi-agent systems in three dimensional environments. *Agent programs* are external processes for the SPARK simulator. There are *bodies* (i.e. mass and a mass distribution) for the physical simulation. SPARK relies the Open Dynamics Engine [Smith, 2006], a free library for simulating rigid body dynamics. 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. SPARK relies on SPADES [Riley and Riley, 2003] to track an agent's execution time between sense and act events.

- Webots [Michel, 2004] is a commercial agent simulation platform that offers support for mobile robots. Like Spark, it uses the Open Dynamics Engine for simulation of physical movements. The focus of Webots is on simulating existing robot platforms. Webots incorporates fine-grained models of low-level sensors and actuators that match their real life counterparts.
- Übersim [Browning and Tryzelaar, 2003] is a multi-robot simulation engine for simulating games of robot soccer for the RoboCup small-size soccer league. Übersim is a simulator specifically designed as a robot development tool. It provides a set of predefined robot models. Like SPARK, Übersim is an Open Source project and uses the Open Dynamics Engine.
- Player/Stage/Gazebo [Gerkey et al., 2003; Koenig and Howard, 2004] is a distributed multi-robot simulator and can simulate a variety of different robots, with a range of conventional sensors, interacting in a complex environment. The Player part of the simulator supports interfaces for the integration of the control software for a variety of robot hardware models. The Stage part of the simulator is a 2D environment with low-fidelity dynamics models that are computationally cheap. The Gazebo part of the simulator is a 3D environment with high-fidelity dynamics based on the Open Dynamics Engine. Stage and Gazebo devices present a standard Player interface.

We make the following observations when comparing these simulation platforms with our approach.

First, similarly to our approach, the aforementioned simulation platforms rely on special-purpose modeling constructs that are targeted at software-in-the-loop simulations of multi-agent control systems in dynamic environments. However, in contrast to our approach, the semantics of the modeling constructs supported by the aforementioned simulation platforms is only described in an informal manner. Consequently, formulating a simulation model that complies with these simulation platforms requires detailed knowledge of the design and implementation of these simulation platforms. In contrast, our approach relies on modeling constructs that are formally specified to unambiguously define their meaning and relations, which is crucial to decouple the simulation model from the simulation platform that is used to execute the model.

Second, the aforementioned simulation platforms support one particular way to simulate dynamism in the environment, either customly developed or by reusing an existing physics library, e.g. the Open Dynamics Engine. Whereas the Open Dynamics Engine simulates dynamism in an accurate way, its high level of detail entails a trade-off in terms of modeling effort and computational efficiency. By putting forward explicit modeling constructs for dynamism (activities, reaction and interaction laws, etc.) we support the modeler to capture dynamism at a level of detail that can be customized to fit the needs of a particular simulation study.

Third, in contrast to the aforementioned simulation platforms, we take a rigorous, view-based approach to documenting the software architecture of our simulation platform. We put forward this software architecture as a reusable artifact, clearly distinguished from the code. The software architecture explicitly documents the knowledge and practice incorporated in the simulation platform. In contrast to code libraries and software frameworks, a software architecture *explicitly* documents (1) stakeholder concerns, (2) how software needs

to be structured and behave to address the concerns, and (3) the rationale and tradeoffs that underpin the design of the system. As such, a software architecture captures the essence of the simulation platform and provides a systematic way to capture and share expertise in a form that has proven its value for software development.

6.7.2 Embedding the Control Software

Simulation platforms use various approaches to integrate the software of a (multi-agent) control system in a simulation [Uhrmacher et al., 2003]. We focus on the way simulation platforms support the execution time of a control system. We make a distinction between approaches that incorporate execution time based on direct measurement and approaches that rely on a specification of execution time.

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/Gazebo [Gerkey et al., 2003; Koenig and Howard, 2004] 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 [Anderson, 2000] 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 [Riley and Riley, 2003] 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 [Anderson, 1997].

First, 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.

Second, 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.

Third, 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.

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 [Anderson and Cohen, 1996] 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 [Bräunl et al., 2006] 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 [Weyns et al., 2005a] 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.

- In Webots [Michel, 2004], the code that deals with execution time is tangled with the functionality of the control software. The control software must proceed in a step-like fashion. After each step, the controller must return the amount of time consumed during that step of the control loop.

We make the following observation when comparing these approaches with our work. By relying on aspect-oriented programming, our approach has an increased flexibility compared to the aforementioned approaches. A modeler can declaratively define the duration primitives of the control software in a single aspect, clearly separated from the rest of the code. We rely on aspect weaving to automatically enforce the tracing of these duration primitives during the simulation. Aspect weaving avoids that the code to trace the invocation of duration primitives needs to be written and inserted by hand each time the control software is embedded in a simulation and each time the duration primitives are adjusted by the modeler. The tradeoff of our approach is that the granularity of the duration primitives that can be defined is constrained by the expressiveness of the pointcut language, i.e. by he granularity of the join points that can be specified in AspectJ.

6.8 Conclusions and Future Work

In this concluding section, we first put forward concrete suggestions for future research with respect to our own work in Section 6.8.1. Afterward, we reflect on the way our work could stimulate future research on multi-agent simulation in a broader setting in Section 6.8.2.

6.8.1 Concrete Directions for Future Research

We suggest two main areas for future research in the context of our own work: extending the modeling framework and extending the software architecture.

Extending the Modeling Framework.

The modeling framework for dynamic environments could be extended with additional constructs. We suggest a number of avenues for future research.

- *Supporting perception in dynamic environments.* Currently, the modeling framework does not provide modeling constructs to capture the way agents sense or perceive a dynamic environment. In a dynamic environment, 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 sources of dynamism in the environment.* Currently, the modeling framework does not provide explicit modeling constructs to model the internals of a source of dynamism in the environment. The internal machinery of a source of dynamism in the environment is black box, and its behavior is specific for a particular simulation study. More elaborate support for sources of dynamism could focus on modeling constructs for various kinds of behaviors, such as reactive [Brooks, 1991; Weyns and Holvoet, 2006], behavior-based [Maes, 1991; Weyns et al., 2005c] or cognitive behaviors [Haddadi and Sundermeyer, 1996; Rao and Georgeff, 1995].

Extending the Software Architecture.

We indicate a direction for extending the software architecture.

- *Distribution of the Simulated Environment.* The simulated environment can become a bottleneck in large-scale simulations involving many agents. Currently, the architecture does not incorporate support for distribution of the simulated environment. The challenges of distributing the simulated environment are not of pure technical nature: as the parts of the simulated environment 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. Distribution of simulations should be supported in a flexible manner [Ewald et al., 2006], with distribution schemas that can be adapted or self-adapt to a particular simulation study. The distribution scheme of the architecture can be documented using deployment views.

6.8.2 Closing Reflection

As demand for multi-agent control applications increases, more and more simulations are built to support their development. The way such simulations are built becomes common knowledge. In recent research, we observe two trends to consolidate common knowledge on developing such simulations.

With respect to building simulation models, research puts forward special-purpose modeling constructs to reify common knowledge. Special-purpose modeling constructs support the modeler by capturing key characteristics of such systems in a first-class manner.

With respect to building simulation platforms, common knowledge is typically reified in reusable code libraries and software frameworks. We put forward software architecture in addition to such code libraries and software frameworks. A software architecture captures the essence of a simulation platform in an artifact that amplifies reuse beyond traditional code libraries and software frameworks. The software architecture we propose explicitly captures (1) functional and quality requirements (2) how software needs to be structured to address the requirements, and (3) the rationale and tradeoffs that underpin the design of the software. We developed a simulation platform that implements this architecture and we applied this simulation platform to support software-in-the-loop simulations for evaluating, comparing and integrating several functionalities of a multi-agent control system for steering AGVs.

We strongly believe that multi-agent simulation can benefit from a more systematic approach to software architecture. Software architecture supports consolidating and sharing expertise in the domain of multi-agent simulation in a form that has proven its value for software development.

References

J. 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.

S. D. Anderson. Simulation of multiple time-pressured agents. In *WSC '97: Proceedings*

of the 29th conference on Winter simulation, pages 397–404, 1997. ISBN 0-7803-4278-X. doi: http://doi.acm.org/10.1145/268437.268515.

S. D. Anderson and P. R. Cohen. Timed Common Lisp: the duration of deliberation. *SIGART Bull.*, 7(2):11–15, 1996. ISSN 0163-5719. doi: http://doi.acm.org/10.1145/242587.242590.

L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003. ISBN 0321154959. URL `http://www.amazon.co.uk/exec/obidos/ASIN/0321154959/citeulike-21`.

J. Borgers. "Hoe realistisch is een simulatie?": Studie aan de hand van LEGO Mindstorms. Master's thesis, K.U.Leuven, Department of Computer Science, 2006.

T. Bräunl, A. Koestler, and A. Waggershauser. Fault-tolerant robot programming through simulation with realistic sensor models. *International Journal of Advanced Robotic Systems*, 3(2):99–106, 2006.

A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors. *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Computer Science*, 2006. Springer. ISBN 3-540-35437-9.

R. A. Brooks. Intelligence without reason. In J. Myopoulos and R. 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. ISBN 1-55860-160-0.

B. Browning and E. Tryzelaar. Ubersim: A realistic simulation engine for robot soccer. In *Proceedings of Autonomous Agents and Multi-Agent Systems, AAMAS'03*, Australia, July 2003.

S. A. Brueckner. *Return From The Ant - Synthetic Ecosystems For Manufacturing Control*. PhD thesis, Humboldt University Berlin, Department of Computer Science, 2000.

G. Bruns, P. Mössinger, D. Polani, R. Schmitt, R. Spalt, T. Uthmann, and S. Weber. Xraptor - a simulation environment for continuous virtual multi-agent systems - user manual. URL `citeseer.ist.psu.edu/bruns03xraptor.html`.

K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/358598.358613.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, September 2002. ISBN 0201703726. URL `http://www.amazon.co.uk/exec/obidos/ASIN/0201703726/citeulike-21`.

W. Delbaere and B. Lamberigts. Ontwikkeling van een gedecentralizeerd controle systeem voor autonome voertuigen. Master's thesis, Katholieke Universiteit Leuven, Belgium, 2007.

P. DeLima, G. York, and D. Pack. Localization of ground targets using a flying sensor network. *sutc*, 1:194–199, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/SUTC.2006.84.

K. Dresner and P. Stone. Multiagent traffic management: An improved intersection control mechanism. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *The Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, NY, July 2005. ACM Press.

R. Ewald, J. Himmelspach, and A. 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. ISBN

1-4244-0501-7.

J. Ferber and J. 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.

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.

B. P. Gerkey, R. T.Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *ICAR 2003*, pages 317–323, Coimbra, Portugal, June 2003.

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.

A. Haddadi and K. Sundermeyer. Belief-desire-intention agent architectures. *Foundations of distributed artificial intelligence*, pages 169–185, 1996.

I. J. Hayes, M. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2003. ISBN 3-540-40828-2.

A. Helleboogh. *Simulation of distributed control applications in dynamic environments*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2007. 169 + xxxi pages.

A. Helleboogh, T. Holvoet, and Y. 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.

A. Helleboogh, T. Holvoet, and Y. 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.

A. Helleboogh, G. Vizzari, A. Uhrmacher, and F. 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.

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.

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.

V. Issarny, M. Caporuscio, and N. Georgantas. A Perspective on the Future of Middleware-Based Software Engineering. In *Future of Software Engineering, 29th International Conference on Software Engineering*, Minneapolis, USA, 2007.

J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, 2004.

M. Jackson. The Meaning of Requirements. *Annals of Software Engineering*, 3:5–21, 1997.

G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241,

pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. URL `citeseer.ist.psu.edu/kiczales97aspectoriented.html`.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/383845.383858.

N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004. URL `http://cres.usc.edu/cgi-bin/print_pub_details.pl?pubid=394`.

P. Maes. The agent network architecture (ana). *SIGART Bull.*, 2(4):115–120, 1991. ISSN 0163-5719. doi: http://doi.acm.org/10.1145/122344.122367.

O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004. URL `http://www.ars-journal.com/ars/SubscriberArea/Volume1/39-42.pdf`.

O. Obst and M. Rollmann. SPARK – A Generic Simulator for Physical Multiagent Simulations. In G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland, editors, *Multiagent System Technologies – Proceedings of the MATES 2004*, volume 3187, pages 243–257. Springer, Sept. 2004.

L. P. K. P. Varshavskaya and D. Rus. Learning distributed control for modular robots. In *International Conference on Intelligent Robots and Systems, Sendai, Japan*, 2004.

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.

P. Riley and G. 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.

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.

W. Schols. Gradient Field Based Order Assignment in AGV Systems. Master's thesis, Katholieke Universiteit Leuven, Belgium, 2005.

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.

R. Smith. *Open Dynamics Engine: User Guide*, 2006.

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.

A. Uhrmacher and B. 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.

A. M. Uhrmacher, M. Röhl, and J. Himmelspach. Unpaced and paced simulation for testing agents. In SCS, editor, *Proc. of the 15th European Simulation Symposium*, pages 71–80, 2003. ISBN 3-936150-28-1.

C. M. Velez and A. Agudelo. Control and parameter estimation of a mini-helicopter robot using rapid prototyping tools. *WSEAS Transactions on Systems*, 5(9): 2250–2257, September 2006.

P. Verstraete, B. Germain, P. Valckenaers, and H. Van Brussel. On applying

the prosa reference architecture in multiagent manufacturing control applications. In *Multiagent Systems and Software Architecture*, 2006. URL `http://people.mech.kuleuven.be/~pverstra/papers/On%20applying%20the%20PROSA%20reference%20architecture%20in%20multi-agent%20manufacturing%20control%20applications_camera_ready.pdf`.

F.-Y. Wang. Agent-based control for networked traffic management systems. *IEEE Intelligent Systems*, 20(5):92–96, 2005. ISSN 1541-1672. doi: http://dx.doi.org/10.1109/MIS.2005.80.

D. Weyns and T. 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.

D. Weyns, A. Helleboogh, and T. 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, Sept. 2005a.

D. Weyns, K. Schelfthout, and T. 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, 2005b.

D. Weyns, K. Schelfthout, T. Holvoet, and O. 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, 2005c.

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, 2005d. ACM Press, New York, NY, USA.

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.

D. Weyns, T. Holvoet, and A. Helleboogh. Anticipatory vehicle routing using delegate multi-agent systems. In *Intelligent Transportation Systems Conference, 2007 (ITSC 2007)*, pages 87–93. IEEE, 2007.

D. Weyns, N. Boucké, and T. Holvoet. A Field-Based Versus a Protocol-Based Approach for Adaptive Task Assignment. *Autonomous Agents and Multi-Agent Systems*, 2008. (in press).