Decentralized Runtime Norm Enforcement

SIKS Dissertation Series No. 2017-45
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Ridderprint BV, The Netherlands

ISBN 978-94-6299-773-8

Decentralized Runtime Norm Enforcement

Gedecentraliseerde Runtime Handhaving van Normen

(met een samenvatting in het Nederlands)

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. G.J. van der Zwaan, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op maandag 20 november 2017 des middags te 12.45 uur

door

Bas Johannes Gerrit Testerink

geboren op 22 juli 1989 te Biddinghuizen

Promotor: Prof. dr. J.-J.Ch. Meyer

Copromotor: Dr. M.M. Dastani

Dit proefschrift werd (mede) mogelijk gemaakt met financiële steun van de Next Generation Infrastructures Foundation.

Contents

1 Introduction

It is anticipated that in the near future a significant portion of traffic will consist of autonomous vehicles (cf. [2, 92, 1]). The companies that produce such vehicles will not likely be keen on sharing the source code and training data of their vehicles. However, these vehicles still have to operate together as a traffic application where throughput and safety are among the highest priorities. In current traffic we control the behavior of humans by enforcing traffic rules. For autonomous vehicles we can also formulate traffic rules, though these need not be the same as for humans. An autonomous vehicle can handle a more continuous stream of information regarding the regulations compared to human drivers. As with human drivers, we must consider the control of autonomous vehicles since we cannot guarantee that they will comply to the traffic regulations. Smart roads are a proposed solution to coerce autonomous vehicles into desirable behavior [136][1]. Such infrastructures support the coordination of vehicles by communicating directives to the vehicles (infrastructure-to-vehicle communication) and by providing relevant information to the vehicles such as the traffic situation ahead. The directives that the infrastructure sends to vehicles can, for instance, concern speed limitations and lane positions. Such directives are not always of a voluntary nature. If a speed limit is imposed for safety reasons, then we must consider sanctioning mechanisms for enforcing the limit. The design of a smart roads application therefore has to consider the challenge of formulating regulations for autonomous vehicles, a monitoring mechanism for detecting violations of the regulations, and a sanctioning mechanism to compensate for violations. Smart roads are also inherently decentralized given their geographic distribution and the fact that many traffic systems are connected across borders, but regulated by local policies.
The challenge of coordinating autonomous vehicles on smart roads is an example of the challenge to control complex systems with autonomous software components. If an autonomous program may violate the rules of a complex system such as smart roads, then this is likely the result of either an unintentional mistake/error, or because the desired behavior (which aims to improve global traffic properties such as throughput) does not always align with the goals of the autonomous program (which aims to improve individual properties such as traveling time). In the latter case, we may speak of an autonomous program that generates proactive behavior to achieve its goals. In

[1] Smart roads are also called "Intelligent Vehicle/Highway System", IVHS.

the field of artificial intelligence we refer to such programs as agents [29, 30]. We consider autonomous vehicles to be agents which have the goal of arriving at a given destination with some preferences (travel time, comfort, etc.). A system of agents is called a multi-agent system. An open multi-agent system is a multi-agent system where agents may freely come and go and where the agents may not be designed, programmed, or known to the party that maintains the multi-agent system. This corresponds to a smart roads application, as the agents (i.e., autonomous vehicles) are developed by companies that do not share the full details of the agents with whomever decides upon the traffic rules of the smart roads. The research community of agent systems addresses among other challenges the challenge of controlling open multi-agent systems. Control mechanisms for open multi-agent systems are often based on high level concepts and abstractions that are inspired by human organizations or institutions. The main regulatory concept is norms. Norms can be seen as rules of behavior. The field of normative multi-agent systems concerns itself, amongst other topics, with how we can translate the concept of norms to control mechanisms that ensure that open multi-agent systems behave within bounds that are deemed acceptable [27]. Such a control mechanism is often referred to as an institution, which is also a term that we will often use. The task of an institution is to realize norm enforcement: the manipulation of system behavior such that either no norms are violated (called regimentation) or the violation of norms is compensated through corrective actions (called sanctioning).

The research in this thesis started with the observation that if smart roads applications correspond to open multi-agent systems, then we can use open multi-agent system control mechanisms to control autonomous traffic. Since institutions are intended as control mechanisms for open multi-agent systems, they might also be a suitable solution for the challenge of controlling smart roads. In this thesis we focus on the general challenge of controlling open multi-agent systems and use smart roads applications both as a motivation and an example throughout most of the chapters. We are especially interested in the decentralized nature of smart roads applications and the question whether the work on open multi-agent system control is suitable for such settings. We also address how we can translate the theory of controlling open multi-agent systems to practical applications. In this chapter we shall outline the motivation behind the research in this thesis, the research questions that we address, our approach, and finally an overview of the research collaborations that have brought about this thesis.

1.1 Motivation and Research Question

In this section we motivate why we are in particular interested in decentralized norm enforcement for runtime settings and why the research field of normative systems should pay attention to the transfer of theory to practice. Furthermore, we also pose the main research question that this thesis aims to answer.

Models of norm enforcement are often defined to investigate the regulatory concepts of open multi-agent systems. This research provides insight in concepts such as obligations and prohibitions. As such, most work on norm enforcement is not con-

cerned with decentralization of norm enforcement in applications. We are interested in decentralized norm enforcement because centralized software systems come with plenty of potential disadvantages such as single point of failures, scalability issues and maintenance of always-on applications. Many multi-agent systems are decentralized by nature. Imposing a centralized control mechanism on a multi-agent system would diminish the system's advantages that are obtained from distribution (e.g., parallel processing). Sometimes it can also be the case that security requirements prevent the use of a centralized system. For instance, certain sensitive data might be constrained in terms of who may share it to what entities. We also see this in human organizations. Within the government for instance there are strict rules on what department may access or obtain which data. If this is the case for a software application as well, then we may not be allowed to construct a centralized enforcement mechanism that gathers all sensitive data in a single location. Instead, we may have to resort to local processing of data and only share information that is allowed to be transferred across enforcing entities. Given that the need for decentralized enforcement exists, one can wonder whether the theories/techniques/methodologies regarding norm enforcement that we have are appropriate. We believe this question to be answered in the negative for the general case. To our best knowledge all current work on decentralized enforcement is focused on techniques with which one can make applications that have some form of distribution, but no work explicitly addresses decentralized norm enforcement.

There are different ways to approach the analysis of decentralized norm enforcement. For instance, one may focus on offline model-theoretic investigations of normative multi-agent systems as Max Knobbout has done in his PhD thesis [82]. A benefit of an offline model analysis is that a system can be verified to be correct before it is deployed. Hence, no computational effort is needed at runtime to ensure that a system works as intended. As an example, we may apply offline model analysis to determine for a specific system whether, if norm enforcement is correctly applied, a specific norm cannot be complied with without leading to a violation of another norm in some situations. However, we are particularly interested in the enforcement mechanisms themselves, and such a norm conflict analysis does not help us much in determining how to develop the enforcement mechanism. An enforcement mechanism can still work as intended regardless of whether the norms that it enforces are conflicting. This is why we focus on runtime analyses to investigate decentralized norm enforcement. We still make use of formal methods to discuss decentralized norm enforcement since they provide insight on the topic, and can guide us to implementations. As it turns out, there is a fairly limited amount of work regarding formal investigations of runtime norm enforcement, much less with a focus on decentralization.

Formal investigations can give us guidance for when we want to apply theory to practice. However, a formal model does not give us a program that we can execute on an execution environment. For this we require means to specify an executable decentralized runtime norm enforcement mechanism. We hypothesize two main user groups when it comes to programming mechanisms for decentralized runtime norm enforcement. On the one hand we have academics and students who want to familiarize themselves with the theoretical side of decentralized norm enforcement. Their programs are likely to serve as prototypes and proof-of-concepts for ideas. On the other hand we have industry software engineers. Their focus lies on implementing

software using standard industry practices. Moreover, these programmers often do not have the theoretical background and interest that helps them to quickly grasp the theoretical side of decentralized norm enforcement. Hence, if we want to stimulate a transfer of technology from academia to industry, then we have to work on a bridge between standard industry practices and academic contributions [142]. A transfer of technology is important for the research field since industry interest can spark new funding for the field. More importantly, the aim of academic research endeavors is not to just provide a collection of theory, it is also aimed at solving real world challenges. Many solutions to real world challenges are implemented by industry companies.

These concerns: runtime norm enforcement, decentralization, models, formal analysis and development lead us to our main research question:

Main Research Question: *How can a decentralized runtime norm enforcement mechanism be modeled, analyzed and developed?*

1.2 Approach and Scope of this Thesis

By answering the main research question we aim to create a body of work that discusses the topic of decentralized runtime norm enforcement from the conceptual level to implementation. We organize the chapters in such a way that we start from a discussion of concepts regarding norm enforcement, then move to runtime models for norm enforcement, then discuss the implementation of normative multi-agent systems and finally present a proof-of-concept smart roads application. In this section we discuss the relevant subquestions that we aim to answer, in order to answer the main research question.

In order to model decentralized runtime norm enforcement mechanisms we need to know what they consist of. Therefore our first research question is:

Research Question 1: *What are the core concepts for modeling decentralized runtime norm enforcement mechanisms?*

This question is mainly addressed in Chapter 2, where we discuss related work on norm enforcement that has at least some form of decentralization and identify where decentralized enforcement differs from centralized enforcement. Among other concepts that are discussed in Chapter 2, we observe that decentralized monitoring and control mechanisms are integral to decentralized runtime norm enforcement mechanisms. Monitoring and control mechanisms must detect norm violations and realize corrective measures, respectively. Models of such mechanisms help us to analyze them and provide guidelines for development. For the latter it is important therefore that the models are computationally feasible. For instance, it would not do to have a monitor for a norm of which the violations cannot even be monitored in theory. The second research question is:

Research Question 2: *How to computationally model runtime monitoring and control mechanisms for decentralized runtime norm enforcement?*

We answer this question in Chapters 2, 3, 4 and 5. In Chapter 2 we provide a high-level model, including monitoring and control, that is used to highlight the core

concepts of decentralized runtime norm enforcement mechanisms. One of the challenges that Chapter 2 highlights is that a decentralized runtime norm enforcement mechanism has to overcome the partial observability and controllability that the monitors and controllers of the mechanism have. In Chapters 3 and 4 we describe detailed models of runtime monitoring mechanisms to deal with local observation capabilities and their topology in terms of what monitor may communicate with what other monitors. Chapter 5 focuses solely on modeling control mechanisms. For decentralized controllers it is a challenge that concurrently applied controllers may have conflicting changes that they want to make in the behavior of an open multi-agent system.

From an analysis point of view we want to know what the desirable properties are of a decentralized runtime norm enforcement mechanism, so that we can verify whether a model is indeed an appropriate model.

Research Question 3: *What are the desirable properties of decentralized runtime norm enforcement mechanisms?*

In Chapter 3 we specify for which norms a runtime monitoring mechanism can detect the norm violations at runtime. In Chapter 3 we verify for our proposed model of monitoring that it correctly detects norm violations and that all the violations of a norm are detected, for those norms for which violations can in principle be detected at runtime. Similarly, in Chapter 5 we discuss which norms are enforceable by a runtime control mechanism and show that our proposed model for decentralized control can enforce those norms that are enforceable in principle by runtime control mechanisms.

Robustness and security are two motivating concepts for why we may want to decentralize a norm enforcement mechanism. For instance in a smart roads application we transmit privacy sensitive information such as the location of vehicles. Hence we may want to provide extra security measures for monitors that deal with privacy sensitive information. Also the robustness of a smart roads application is important. If the system as a whole fails, then this increases the risk of compromised safety and throughput. Chapter 4 provides for a proposed runtime monitoring model an analysis of how to find the critical parts with regard to robustness and security.

Finally, the last part of our main research question concerns the development of runtime norm enforcement mechanisms. The last research question that we answer is:

Research Question 4: *How to develop a decentralized runtime norm enforcement mechanism?*

We approach this question from different angles throughout the thesis. The modeling and analysis parts of this thesis are aimed especially at runtime settings in order to make them suitable for development. The examples for these models also include scenarios and supporting explanations that indicate how the models are intended to execute once developed. If we know how a system ought to execute, then we can use this during its development as a requirement guideline for designing the software. For the control mechanism in Chapter 5 we specify operational semantics in order to formally explain how the mechanism executes. In Chapter 6 we discuss programming techniques and methodologies for the development of normative multi-agent systems. We describe how norm enforcement can be realized through object-

and aspect-oriented programming. Our approach is to describe the general solution structure as a design pattern.

The main motivational use case of this thesis are smart roads applications. Such applications are very expensive to investigate in 'the real world', since it requires many resources to create a controlled highway system. A more viable approach is to first use simulations. In Chapter 7 we provide a proof-of-concept application of decentralized runtime norm enforcement for smart roads applications in a well-known traffic simulator.

We would also like to emphasize what lies outside of the scope of this thesis. One topic has already been mentioned: offline model verification, instead of runtime. We do believe that a critical application should also be analyzed with offline model verification, but such a verification does not give us answers to our main research question regarding decentralized runtime norm enforcement. Another earlier mentioned topic is normative conflict. We focus on the enforcement of norms, and not on whether norms are inherently incompatible and can never be complied with without causing a violation. Neither do we investigate the social, philosophical and legal aspects of norms. Similarly, we do not discuss new contributions regarding norm awareness, though it is mentioned occasionally throughout this thesis. Norm awareness is a concept that describes how agents can be aware of norms and change their behavior accordingly. Often the notion of norm awareness goes hand in hand with organizational awareness. This means that an agent can reason about its role within an organization of agents. Since our focus lies solely on norm enforcement, we will not discuss organizational concepts such as roles and reorganization.

1.3 Smart Roads Scenarios

Smart roads applications are one of the main motivational systems for this thesis and we use them often in chapters to draw example scenarios from. In this section we go into more detail on the type of scenarios that we are considering.

Smart roads applications are an expanded version of existing traffic systems. Currently, humans drive vehicles on the road and are organized by a set of rules which are the traffic laws of the road. The aim of traffic rules is that we can have efficient (in terms of high throughput) traffic that is also safe. The rules may, however, conflict with the preferences of a specific driver. Such a conflict provides an incentive for many drivers to violate the traffic rules in order to optimize for instance their personal efficiency/travel time. We lose the benefits of regulated traffic if too many drivers violate the laws. We can apply sanctions such as monetary fines every time a rule is violated in order to decrease the incentive to violate the rules. We need to monitor traffic to check whether a rule is violated. Current traffic systems collect data with specialized sensors such as inductive loops and speed cameras. The sensor data has to be interpreted in order to determine whether a rule has been violated. This currently happens either by humans or fully automatic. Speed violations in the Netherlands are for instance fully automatically detected and fined. Determining whether a rule is violated (e.g.: is the sensed velocity violating a speed limit?), determining what the sanction is given a violation (e.g.: a large or a small fine?), and,

if applicable, imposing a sanction (e.g.: actually imposing a fine) can all be separate responsibilities that are handled by separate entities. Also the rules themselves can be localized. For example, some municipalities do not allow vehicles such as tractors to use certain roads.

The enrichment that a smart roads application provides to a regular traffic system lies in the automated components and the consideration of autonomous traffic. A key feature of a smart roads application with infrastructure-to-vehicle communication facilities is that the infrastructure is able to communicate with the traffic in a personalized manner. Right now, the only way that the infrastructure 'communicates' is through visible signs that are directed to all traffic. One avenue of smart roads application research focuses on the content of the communication towards vehicles that promotes throughput and safety. For example, the infrastructure may send directives to vehicles such that two separate traffic streams merge fluently. In Chapter 7 we also discuss such an example. This thesis, however, does not investigate the content of communication. Instead, we focus on the regulatory system that ensures that vehicles follow the directives from the infrastructure. An autonomous vehicle can be expected to maximize its efficiency, just like a human driver. We need to process the sensor data such as camera feeds, determine which rules were violated and find an appropriate compensation in addition to communicating traffic rules to autonomous vehicles.

There are various challenges which surround the enforcement of traffic laws in smart roads applications. We need to determine what monitoring capabilities we require to see whether any traffic laws have been violated. We need to determine what kind of compensations we want to apply and what capabilities are required for that. We have the practical challenge of ensuring that sensors report correct data and are continuously functioning. We have the security challenge of determining what information flows require special attention to protect for instance the privacy of agents. We have the operational challenge of having to deal with localized rules, a system that cannot be fully shut down for maintenance and geographical distribution. Finally, we have the software challenge of how to implement the system. We noted in the introduction of this chapter that we view smart roads applications as instances of decentralized runtime norm enforcement mechanisms. All challenges of smart roads are also challenges of decentralized normative systems in general. Throughout this thesis there will be example scenarios that will highlight how our research provides answers to these challenges.

1.4 Research Collaborations

Our research started with a logic-oriented programming approach for decentralized institutions in my master thesis [123] and the follow-up paper [126]. Ideas from these works have been fleshed out in various forms throughout the thesis and are briefly discussed in Chapter 2. Chapter 2 is further based on the work from [129]. In [129] we explored the important concepts regarding decentralized norm enforcement. We identified decentralized monitoring and control as the cornerstones for decentralized norm enforcement. In [128] we discussed decentralized monitoring where individual

institutions send observations to each other. This was our first investigation into decentralized monitoring for norm enforcement and is adopted in Section 3.5 as the framework for monitoring with delay. We later followed up on this research which has resulted in work presented in [124] and [125]. In these works we investigated a different monitoring model (resulting in the aggregation-based model from Section 4.2) and discussed basic notions of robustness and security regarding decentralized monitoring (which was adopted and expanded in Section 4.3 and 4.4). The basis for our decentralized control model, and thereby Chapter 5, can be found in [127]. Our work on object-oriented normative multi-agent system engineering in Chapter 6 is largely based on [49], where we describe how concepts from normative multi-agent systems can be captured with object-oriented design patterns. Finally, the basis of Chapter 7 can be found in [21], in which we applied norm-based control to traffic simulations.

2 On the Decentralized Enforcement of Norms

We focus on the meaning of the concept runtime decentralized norm enforcement as an initial step to answer the question how we can realize runtime decentralized norm enforcement. Runtime norm enforcement has been addressed in various works that are related to this thesis. The first contribution of this chapter is a discussion of such related works. We focus in particular on the proposals that discuss mechanisms to enforce norms at runtime and where decentralization plays some role. The second and main contribution of this chapter is to investigate how decentralized norm enforcement differs from centralized norm enforcement. We do this by providing a simple formal model in order to get a feel for the challenges when we want to design and develop a system with decentralized norm enforcement.

2.1 Introduction

In open multi-agent systems, organizations and/or institutions are used to promote global system behavior. Often the intended behavior of the agents ensures that some global goal will be achieved by the multi-agent system (e.g. [74]) or ensures that agents correctly interact with each other (e.g. [138, 96]) or with the environment (e.g. [46]). The specification of what constitutes correct behavior can be made with norms, which can be seen as regulatory rules. Throughout this thesis we will view norms as prescriptive regulations, and not as descriptions of behavior. The terms organization and institution are regularly used interchangeably. However, in the case of organizations the literature often includes other high level concepts alongside norms such as roles, organizational empowerment and reorganization (cf. [53]). Institutions tend to be focused solely on norms and their enforcement, which is also what we focus on in this thesis. Hence, we will mainly use the term institution. This thesis concerns normative institutions where norms are used to control and coordinate agents. Over the years there have been investigations in many aspects of norm-based control. This has led to different frameworks (e.g. [75, 74, 53, 57]) and programming languages for organizations and institutions (e.g. 2OPL [46] and the framework from Gaertner et al. [60]).

We consider only institutions that are exogenous to the agents and the environment and have explicit norms. This approach is suitable for open multi-agent systems, as we cannot ensure that agents will endogenously adopt the norms in those systems.

An exogenous institution is also useful because it separates the regulation concerns from the rest of the multi-agent system (i.e. agents, middleware, etc.). This in turn allows for independent maintenance and debugging of the institution. Also the use of norms that can be violated is often preferred over hard constraints. This is because norms preserve to a greater extent the autonomy of agents. Agents still have a choice whether or not to violate norms if they are aware of them. Also, if an agent is not aware of norms then still the institution can repair or compensate violations of norms if they occur. For instance a tourist can be incarcerated for violating a law during vacation, regardless of whether he or she was aware of the law. We note that norm-awareness is not a topic of this thesis.

There is an abundant collection of norm formalisms that allow us to specify norms. One of the most basic methods is to specify the desired behavior and the consequences if that specification is violated. One can implement all norms with this principle as was done in the programming framework 2OPL [46] and the framework from Gaertner et al. [60]. We shall also adopt this view in this chapter. Monitoring is required to detect the violation of norms, which we shall capture with rules that indicate which environment states should be considered as violations. From an agent behavior perspective we can hence consider all behavior that leads to an undesirable state as undesirable behavior. Sanctioning is needed to respond to violations, i.e. to enforce norms. For enforcement we shall use rules that tie a norm violation to a consequence. We separate the rules because they can be maintained and applied independently of each other, which also happens a lot in human society. For instance the velocity that counts-as speeding for a specific road can be determined by a local municipality whereas the sanction is determined by the court. In terms of software development we gain modularity by separating different concepts. Monitoring and sanctioning require two core capabilities of an institution given an environment: what can be observed and what can be controlled in that environment. We do not assume that an institution has full observation and control capabilities over the environment, unlike programming frameworks such as 2OPL [46]. However, shortcomings in terms of capabilities might be partially compensated if an institution can collaborate with other institutions in order to share observations or realize sanctions.

In this chapter we address decentralized institutions. These decentralized institutions are comprised of a network of local institutions, each of which has their own observation and control capabilities. There are different kinds of distribution in a decentralized institution. We can decentralize the monitoring of a norm, the application of a sanction to repair/compensate a violation, the task of determining violations based on monitored data and the task of determining the sanction based on a norm violation. In particular we are interested in the question whether, given a network of institutions, a set of norms *can* be enforced (e.g. are the required monitoring capabilities for detecting norm violations present?) and if we can enforce norms, what are our possibilities in terms of distributing the norm monitoring and violation sanctioning tasks? To answer these questions we will look at approaches from related literature (Section 2.2) and formally define norms, decentralized institutions and enforcement (Section 2.3). Then we shall describe how the norm monitoring and violation sanctioning tasks can be distributed in the network (Section 2.4). We draw our example scenario (Example 2.1) from the smart roads application which was described in Sec-

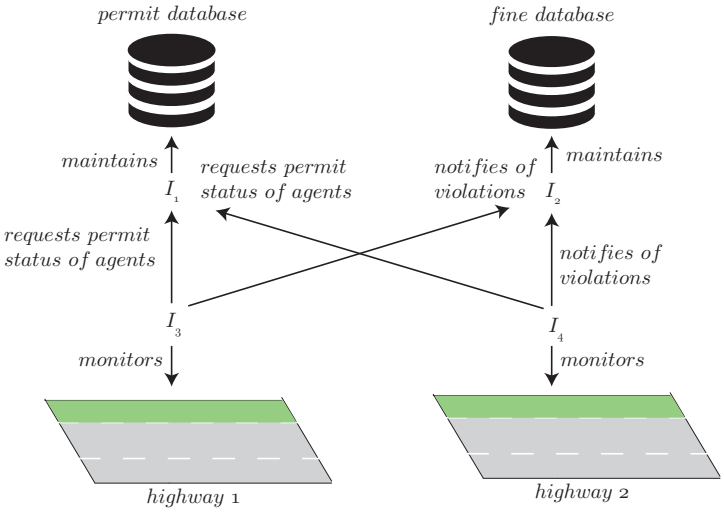

Figure 2.1: *Example scenario from the smart roads application. Institutions 1 and 2 maintain the permit and fine database, respectively. Institutions 3 and 4 monitor different highway segments. The top lane is a priority lane.*

tion 1.3.

Example 2.1 (Smart roads scenario): An example scenario is schematically depicted in Figure 2.1. Two highway segments are shown at the bottom of the picture where traffic flows from highway one to highway two. The upper lanes of the highways are priority lanes to increase the throughput of vehicles with a permit such as emergency services and car repair services. The priority lanes only come into effect when the traffic density on the highways is above a given threshold. The norm is: "*if the traffic density is high, then cars are only allowed to drive on the priority lane when they have a permit*". Violations of this norm will be sanctioned by a monetary fine.

Institutions commonly have a core responsibility. The example scenario contains four institutions: one manages a permit database, one manages a fine register, and two to manage the highway segments. We assume that institutions follow common software practices such as restrictions on what other institutions may receive data from the institution and from which other institutions data can be requested (and not, for instance, that data is publicly accessible by any other institution). Data sharing restrictions are captured by communication relations. The highway institutions are connected to the two other institutions. They can obtain from the permit managing institution whether a vehicle has a permit to use the priority lane. They can inform the fine maintaining institution whether a norm violation has occurred, so that it can issue a fine. Monitoring constitutes for the highway institutions the observation of vehicles while control constitutes, for instance, the possibility to change the speed limit. Having a limited area per institution where

it can monitor and control the highway leads to different situations in which it becomes problematic to monitor norms or impose sanctions. Our example norm can only be monitored if the institutions share their local observations. There is no institution that can both check if a car has a permit and whether the highways' densities are high. In this situation the institutions have to share this information in order to determine whether a violation is occurring. We will investigate the problem of how norms can be enforced in general within a network of institutions, which we call a decentralized institution.

2.2 Related Work

In this section we describe the main ideas from related work on organizations/institutions that also touch upon some form of decentralization. First, we discuss a view on norm enforcement in multi-agent systems that has some overlap, but is different from our approach towards norm enforcement. Second, we discuss a selection of works that are similar to the contributions of this thesis. These are proposals of norm enforcement mechanisms that are also accompanied with methods and techniques to implement them. In these related works, and like the approach in this thesis, norms are explicitly defined by the designer of the enforcing mechanism. For each of the selected works in this section we focus on the core idea, its relation to the other discussed works, and the implementation approach of the authors. With regard to our own implementation approach, we discuss a proposal in Chapter 6 using aspect-oriented programming.

We note that none of the related works analyze the issue of whether and how we can distribute the enforcement responsibility of a given norm set over a network of institutions, where the norm set is assumed to be developed independently of the network. Especially the issue of norms that cannot be locally enforced (checking whether the norm is violated and repairing/compensating for this violation within one institution) has not been addressed. In this chapter we complement the related work by highlighting this question. The results of our research will help to develop more general norm technologies. Also a deeper understanding of design choices when constructing a norm set will be obtained. This aids the process of designing a norm set for a decentralized institution.

2.2.1 Enforcement by Agents

In this thesis we take an approach towards norm enforcement where an enforcing entity contains an explicit specification of norms which it tries to enforce. In our approach, the enforcing entity is not prescribed or assumed to be an agent itself. Various examples can be found of norm enforcement approaches where the enforcement task is the responsibility of agents. In such cases we may view the enforcing entities as special agents that enforce explicit norms among their peers (cf. [41, 68, 25, 58, 66]). Our approach is compatible with the special-agent approach, with the exception of our implementation proposal in Chapter 6. Our implementation proposal follows the 2OPL [46] approach where the norms are integrated in the interfaces between the agents and the environment and possibly each other. Implementations of the discussed formal models in this thesis are not required to be non-agents. For any

application for which norm enforcement is considered it should be evaluated whether the norm enforcing entities ought to be agents or not. Section 6.2 discusses agent-oriented programming in more detail and when to apply agent programming. It is also possible to apply a hybrid approach. For instance, Modgil et al. [97] propose an approach where the enforcement mechanism relies on agents to gather observations and realize sanctions. The mechanism itself consists of monitors that are responsible for computing which norms are violated and what sanctions should follow. Note that the use of agents to gather information and realize sanctions automatically makes the enforcement mechanism decentralized, as the agents form together a decentralized system.

Another approach to decentralized norm enforcement concerns itself not with prescribed explicit norms but rather allows agents to create and enforce norms amongst each other. For example, in peer-to-peer file-sharing applications it can be impossible for an enforcement mechanism other than the agents themselves to observe the communication that takes place in the multi-agent system. In such a case a built-in reputation system may help the system as a whole to increase performance (cf. [66, 17]). Alechina et al. [9] address the issue of handling the detection of norm violations when the norms are not prescribed and the agents may even disagree on what constitutes a violation of the norms. In their system there are not appointed monitoring agents, so not only do agents have to be motivated to comply with the norms, they also have to be motivated to monitor whether other agents comply with the norms. Balke et al. [17] call this category of enforcement approaches 'economics-inspired', since they are usually related to utility-based methods. The motivation is that if agents act rationally, then complying with the norms (and/or volunteering for enforcement duties) is the preferred course of action. Although these enforcement mechanisms are inherently different from our approach, and are not further discussed in this thesis, they do highlight some challenges in open multi-agent system norm enforcement that we do not address in this thesis. Sanctions in these enforcement mechanisms are usually tied to the identity of an agent (e.g. its reputation). If in an open multi-agent system an agent can rejoin the system to gain a new identity, then it may use this tactic to get rid of accumulated sanctions and thereby circumvent the enforcement mechanism's intent. This is comparable to vehicles in our smart road examples changing their license plate, which prevents cameras to correctly identify them. The sanctions in this thesis are not always tied to agent identities (e.g. a sanction can be the temporal closure of a lane on a highway), hence identity changes are not always an issue. Also, practice has shown that for existing accumulating sanctions such as fines it is quite exceptional that agents change their identity. In other areas, such as electronic markets, the issue is more pressing.

2.2.2 Law-Governed Interaction

The work by Minksy and Ungureanu in [96] is closely related to norm-based control of open multi-agent systems, though open multi-agent systems and norm-based control are not explicitly mentioned. They do also call the entities that fall under the influence of a control mechanism agents. The control mechanism that Minksy and Ungureanu [96] propose is called law-governed interaction (LGI). LGI is to be used

to facilitate the enforcement of coordination policies among heterogeneous programs, i.e. agents. Minksy and Ungureanu also argue that a centralized control mechanism becomes a bottleneck as the group of agents grows and poses the danger of a single point of failure. Like our approach with exogenous norms, they also support the use of separating coordination policies from the internal specification of the agents that are controlled. Hence, due to the similarities, it is well worth investigating the proposed solutions for decentralized control that LGI offers.

In LGI, the law of a group of agents is an explicitly defined set of "rules of engagement", where we provide a set of explicit norms to represent desired behavior. The specification of a governed group of agents further consists of the messages that they may interchange and a set of control states, one for each agent. The core approach to LGI is to encapsulate each agent with a controller that can influence the agent's control state (the agent itself cannot change that state) and block illegal events/actions that the agent tries to bring about. The controller can also execute sanctions if obligations of the agent are violated. This is very similar to executing sanctions when norms are violated, as the norms are strongly related to the notion of obligation. On a conceptual level this is also quite similar to maintaining an explicit social or institutional state per agent. The encapsulation approach to controlling an agent, without changing its internals, is quite similar to edit automata [90], which form a foundation for the decentralized control model in this thesis (Chapter 5). The controllers in LGI are able to determine on their own for any action of the agent what the effect of the law is. There is no communication between the controllers for monitoring whether some law is about to be broken, and no possibility that a local control state change affects the control state of another agent. In this thesis we shall discuss monitoring and control where the enforcing entities do communicate with each other in order to detect when and how the control mechanism should intervene. Though there is not a formal execution model of LGI, there is an implementation toolkit called Moses[1]. As is common in normative technology, the choice programming paradigm for the implementation of LGI was to use declarative programming (Prolog).

2.2.3 Multi-Institutions

The controller specifications are implementations of runtime control in LGI [96]. Cliffe et al. [39] take an approach where norm-based controllers - they also use the term institutions - are modeled for offline design analysis and runtime reasoning about the normative state of a multi-agent system. Their approach is to describe institutions in an action language, and then translate it to answer set programming code for analysis and reasoning. There are a few overlapping aspects with the work by Cliffe et al. and the work in this thesis, though their work does not directly involve runtime mechanisms for decentralized normative control. Of interest to us is their notion of multi-institutions. These are models of norm-based control mechanisms that can directly or indirectly influence each other, as is our approach with norm-based controllers that collaborate.

A single institution is modeled by a quintuple of possible events, fluents to describe a state, a consequence function to describe the effect of events on the fluents, an event

[1] http://www.moses.rutgers.edu/

generation function and an initial state of fluents. The event generation function captures the notion of counts-as [117] by telling for a given state and event whether the event counts-as other events as well. For instance, the event of writing a signature might count-as the generation of a new contractual obligation if the signature was written on a contract. A multi-institution is modeled by a tuple of single institutions. Institutions have the possibility of one to initiate or terminate a fluent in another. This is captured by an empowerment specification in individual institutions. When one institution can initiate (or terminate) a fluent in another institution, then it can only do so if that event is locally initiated (or terminated, respectively). Hence, empowerment in [39] can be seen as a kind of state synchronization specification. There is no model of limited views of institutions on the environment in the work by Cliffe et al. [39]. A multi-institutional model can be programmed in the action language InstAl. A specification in this language can be automatically transformed in an answer set programming specification. Due to the action-oriented model it is possible to generate possible sequences of states of a multi-institution. The answer set program can answer queries about such sequences. Using this approach a designer or agent may query, given a multi-institution specification, whether there is a possible sequence of events such that no sanction will be incurred. In our monitoring and control framework (Chapters 3 and 5) we also rely heavily on the analysis of sequences of states and events.

2.2.4 Distributed Norm Management

Like the work from Minksy and Ungureanu [96] the work by Vasconcelos et al. [137], which is an expansion from the work by Gaertner et al. [60], also deals with the distributed enforcement of norms. They note that the LGI approach does not allow a local institutional change (i.e. a change in the controller of an agent) to propagate to other parts of the system (i.e. affect the controllers of other agents). In the multi-institution approach this kind of effect across multiple institutions could be modeled, but they did not provide a runtime methodology for how such systems can be programmed. In [137] there is a specification of a runtime distributed norm enforcement mechanism and an analysis of conflict resolution.

Recall that in the LGI approach an agent is encapsulated by a controller that governs all the actions that the agent undertakes. In the approach presented in [137] the control is not encapsulated around agents, but around the activities that they partake in. An activity is a coherent collection of actions over time such as the execution of an auction or an online purchase. Hence, norm-based control is distributed over the concurrent activities, rather than over the agents individually. The total structure of all the norm enforcement surrounding the activities is called a normative structure. The propagation of effects of the normative state of the systems is realized by 'normative positions' (obligations, permissions, prohibitions) that may flow from one activity to another. Such a flow may give rise to conflicts, e.g. an agent being both prohibited and obliged to execute the same action at a given moment. However, it is shown that conflict resolution at the design time of the normative structure is intractable and that hence runtime conflict resolution is required. In this thesis we also focus on runtime norm-based control, but not only because offline design analysis

might be intractable. It is also the case that in open multi-agent systems can only (partially) observe and control the possible actions that agents execute, but maybe have no insight in how those actions form a coherent activity. Our position is that certain temporal properties of sequences of those actions are undesired, regardless whether they are part of coherent activity, or complex of activities.

The model of a normative structure bears some resemblance with the model of a multi-institution. The state of a controlled activity is called a normative scene and consists of the utterances that agents made (a speech act approach to agent actions is adopted) and the current normative positions. Like the consequence function in [39], normative transition rules specify under what circumstances (conditions on the state of the activity) which normative positions are removed or added from a scene. For a rule, the scene that triggers the scene can be different than the scene where the effect takes places. For instance an action in one activity may cause an obligation in another. In order to implement a normative structure it is suggested to make the control component itself a multi-agent system. This allows the developer to make use of the distributed nature of agents to distribute norm enforcement. In particular it is suggested to adopt an expanded version of AMELI [58]. In AMELI a so called social layer of agents governs the interaction among agents, like controllers from the LGI approach. In [137] additional agents for this layer are suggested to implement the normative structure.

2.2.5 Situated Organizations

Okuyama et al. [100] note that in many real world organizations the norms and organizational empowerment of agents is dependent on their location or the objects with which they interact. They particularly consider organizations that are situated in (simulated) physical environments. In such organizations the publication of norms that apply to a part of the environment might be announced through a sign. For instance, in a train one might have a sign in some carriages that indicates whether present agents ought to be silent. The approach from [100] can also account for certain agents being empowered in only certain rooms. For instance, a conductor may give another agent a fine for not carrying a valid train ticket if they are both located in a train, but not when they are in a pub.

Distribution of norms in [100] entails distribution of norms over places and/or objects. A normative object or place announces to agents the norms with which they must comply. Hence this kind of distribution has some similarity to the work from Vasconcelos et al. [137], where norms were distributed over activities. The overlap is particularly strong if there are designated rooms for activities that are governed by norms. As in AMELI, the authors of [100] suggest the usage of agents to deal with the decentralized monitoring of norm compliance, and potential sanctioning in case of violations. Though the authors also suggest that sometimes the peers of violating agents may issue social sanctions, in case they benefit from other agents not violating any norms. In this thesis we discuss networks of monitors and controllers in the next chapters. The normative side of a situated organization can be modeled by modeling a separate monitor and controller for each normative object and place. However, the work in [100] also considers roles, which are not covered by this thesis.

The proposed implementation of situated organizations is to extend the existing environment modeling language ELMS [101], the organization framework MOISE+ [74] and the agent programming language Jason [31]. Hence this would fit quite well with the later-proposed JaCaMo [28] framework that also builds on similar technologies. There is no formal analysis of situated organizations.

2.2.6 Distribution in Moise

Moise, and its later extensions to $\mathcal{S} - \mathcal{M}$oise$^+$, is a framework for the development of organizations for agents [68, 76, 74, 75, 28]. In the Moise approach there is the core assumption that an organization of agents exists to achieve some goal or global plan. This is different from other organization-oriented programming frameworks like D2OPL from Section 2.2.7, where the organization is intended as a controlling entity to prevent and sanction unwanted behavior, but does not have any goals or overarching plans. Also, as the aim is coordination of agents in a global plan, Moise is strongly oriented towards roles of agents within that plan. Though we do not consider roles explicitly, in [130] there is an exploration of roles for 2OPL [46], which is also an organization-oriented programming language (see Section 2.2.7). As in 2OPL, the original Moise$^{(+)}$ approach specifies organizations explicitly, but in contrast to 2OPL, the specification has to be adopted by agents, and is not enforced by exogenous (to the agents) controlling entities [68, 76, 74].

The development framework $\mathcal{S} - \mathcal{M}$oise$^+$ for implementing Moise organizations does provide exogenous enforcement mechanisms. These are mainly put in place to ensure that no *organizational* norms are violated such as the exclusion of an agent adopting two conflicting roles. These entities are called OrgBoxes and are placed between an agent and its communication channel, and hence provide a form of distribution. This is much like the LGI approach from Minksy and Ungureanu [96]. The OrgBoxes, however, also provide their corresponding agents with information regarding the obligations, goals, etc, that come with their current roles. The Moise framework was also adopted in a later effort to support full multi-agent level development for the environment, agents and organizations called JaCaMo [28]. In JaCaMo the agents are made in Jason [31] and the environment in CArtAgO [111]. CArtAgO's model for environments is to create separate modules that can be linked together. In the JaCaMo framework the distribution of a Moise organization is realized by attaching organization modules to CArtAgO's environment modules, and then use the links from CArtAgO to allow communication among organization modules. This communication is not detailed in [28], except that it is used to maintain a consistent state for the organizations, by which we may assume that at least communication regarding the state of the environment takes place. A Moise model in a JaCaMo project is executed by compiling it to the norm oriented programming language NPL, which is a declarative language for norm programming [73]. Though Moise does not include enforcement of norms by itself, NPL does allow for the specifications of sanctions. These take form in the obligation for some agent to execute a sanction action.

2.2.7 (D)2OPL

One of our own earlier proposals towards implementing decentralized normative institutions, named D2OPL, can be found in [123] and [126]. Our approach was to take the existing non-distributed language "Organization Oriented Programming Language", 2OPL [46], and update it to a distributed version. 2OPL was originally proposed to implement norms by utilizing counts-as rules and sanction rules. We also explain and use this type of norm representation in this chapter. In later work, Tinnemeier et al. expanded the language with various constructs such as conditional obligations and prohibitions, roles, and norm dynamics [134, 133, 131, 132, 130]. In D2OPL only the conditional obligations and prohibitions have been transferred from Tinnemeier's extensions.

2OPL's approach to norm-based control is that a programmer programs an insitutional artifact between the environment of a multi-agent system and the agents. Note that the term 'organization', as used by the original authors of 2OPL, is appropriate since the complete body of literature on 2OPL includes organizational concepts and abstractions such as roles, empowerment and re-organization. The institutional artifact is exogenous to the agents and has no guaranteed insight in their specification. This approach is ideal for open multi-agent systems where the agents may come and go as they please and can be developed by parties other than the institution designer(s) (such as smart roads applications).

The main features of D2OPL are that institutions can conditionalize norms on the state of environments that are not locally observable, conditionalize sanction measures on the violations of norms which are not locally observable, and realize sanctions which cannot be realized locally. How these features are related to this chapter is discussed in Section 2.6. D2OPL follows the logic-oriented programming paradigm and is implemented with Prolog at its core. Its operational semantics is given in [123] and [126].

2.3 A Model for Decentralized Institutions

In this section we shall define a simple model for decentralized institutions. This requires us to define norms, institutions and norm enforcement. Part of the model will involve the description of environment states and observation and control capabilities. For this we use propositional logic. We assume that the reader is familiar with propositional logic.

2.3.1 Defining Norms

Active monitoring is required to detect the violation of norms in open multi-agent systems and sanctioning realizes the consequences of violations. There are many ways to represent norms using preconditions, deadlines, deontic concepts, etc. For our purposes we take the most basic representation which is counts-as rules. We use sanction rules for describing the compensations of violations.

We model all possible norm violations as an assumed set V of violation atoms, where $v \in V$ is an identifiable norm violation. We further assume that the state of

an environment can be modeled by a conjunction of propositional literals in which no atoms from V occur. A norm is modeled by a counts-as rule of the form $\varphi \Rightarrow v$ where φ is a propositional formula regarding the environment, and v is a violation atom. The reading of a counts-as rule $\varphi \Rightarrow v$ is that the system states that satisfy φ are violation states. Or to put it differently, states that satisfy φ are forbidden states. The violation of this prohibition is identified by the atom v. A sanction rule has the form $v \Rightarrow \psi$, where v is again a violation atom, and ψ is a conjunction of literals about the environment. The reading of a sanction rule $v \Rightarrow \psi$ is that states that lead to violation v have to be updated with ψ to compensate for the violation. Violation atoms are considered to be institutional facts. Often the use of counts-as/sanction rules is combined with a context description that tells when the counts-as/sanction rules are applicable. We omit such a context for simplicity. Thoughts on incorporating context in counts-as rules can be found in Section 2.6. A matching norm and sanction pair is a pair where the norm's violations are sanctioned by that sanction (Definition 2.1).

Definition 2.1 (Matching Norms and Sanctions, $\varphi \Rightarrow v, v \Rightarrow \psi$): *A norm is represented by a counts-as rule of the form $\varphi \Rightarrow v$, where φ is a conjunction of literals and v is a violation atom. A sanction is represented by a sanction rule of the form $v \Rightarrow \psi$, where v is a violation atom and ψ is a conjunction of literals that has to be made true in case violation v has occurred. A norm n matches a sanction s iff $n = \varphi \Rightarrow v$ and $s = v \Rightarrow \psi$, i.e. sanction s responds to violations of norm n.*

Example 2.2 (Ex. 2.1 Cont.: Matching Norms and Sanctions): In Section 2.1 we described a network of institutions to regulate the traffic on highways. The example norm was "*if the traffic density is high, then cars are only allowed to drive on the priority lane when they have a permit*". The sanction for riding on a priority lane when it is forbidden is a fine. We will model this scenario with the following atoms with respect to an arbitrary vehicle:

- d_1 and d_2 stand for "*high density*" on highway one and two, respectively.
- $prioritylane_1$ and $prioritylane_2$ stand for "*the vehicle is driving on the priority lane*" on highway one and two respectively.
- $permit$ stands for "*the vehicle has a permit*".
- $fine$ stands for the"*the vehicle is fined*".

The norm holds for both segments. We model the norm by the following counts-as rules:

- $c_1 = d_1 \wedge \neg permit \wedge prioritylane_1 \Rightarrow v$
- $c_2 = d_2 \wedge \neg permit \wedge prioritylane_2 \Rightarrow v$

I.e. the reading of the first counts-as rule is that a high density at highway one and the vehicle not having a permit and the vehicle being on the priority lane

counts-as a violation that is identified with atom v. As a sanction rule s we use $s = v \Rightarrow fine$, which indicates that the fine is applied when the antecedent c_1 or c_2 hold given an environment state. Note that (c_1, s) and (c_2, s) are matching norm-sanction pairs.

We limit the use of institutional facts to violations. Complex regulative systems often use constitutive norms to define institutional facts [117]. However, as violations are the only institutional facts, and they are already defined by counts-as rules, we do not include the use of constitutive norms.[2]

2.3.2 Defining (Decentralized) Institutions

The core components of an institution are its observation and control capabilities. These are derived from sensors in the environment and actuators. Defining the capabilities of an institution can by itself be quite complex if we want to model precisely the sensors and actuators. In this chapter we shall abstract away from such details in order to maintain focus on norm enforcement. We model observation and control capabilities as entities that can be queried regarding the observability or controllability of a specific formula. An observable formula is a formula of which the truth value can be derived given any environmental state. Controllable formulas are formulas that can be made true at any time[3]. We use the entailment operator $\vDash_{o/c}$ that tells for given capability whether it follows that a given formula is observable or controllable. For observation capabilities it is likely the case in many applications that if it can be checked whether a formula is true, then it can also be checked whether its negation is true. However, this does not always need to be the case. For instance we may have a system where formulas are not either evaluated to true or false, but can be evaluated to unknown as well. Our definition abstracts away from such considerations.

Definition 2.2 (Institution Specification, I): *An institution I is specified by (Γ_o, Γ_c). Γ_o is a specification of the observation capability of I. Γ_c is a specification of the control capability of I. For an institution $I = (\Gamma_o, \Gamma_c)$ and a conjunction of literals φ we use $\Gamma_o(\varphi)$ or $I \vDash_o \varphi$ to indicate that φ can be observed by I. Similarly, for a conjunction of literals ψ we use $\Gamma_c(\psi)$ or $I \vDash_c \psi$ to indicate that ψ can be controlled by I.*

Example 2.3 (Ex. 2.2 Cont.: Institution Specification): The network of institutions from Figure 2.1 is depicted in more detail in Figure 2.2. As an example we shall model the capabilities of an organization with a set of atoms that is a subset of the described atoms in the previous example. Furthermore, we say $(\Gamma_o, \Gamma_c) \vDash_{o/c} \varphi$ iff each occurring atom in φ is a member of Γ_o or Γ_c, respectively. Figure 2.2 tells which atoms an institution can observe and control. Consider institution $I_1 = (\Gamma_{o1}, \Gamma_{c1})$. According to the figure we have that $\Gamma_{o1} = \{d_1, priority lane_1\}$ and $\Gamma_{c1} = \emptyset$. Also, $I_1 \vDash_o d_1 \wedge priority lane_1$.

[2] More on this in Section 2.6.

[3] A controllable formula cannot necessarily be made false. Consider for example a controller that can issue a fine but not retract it.

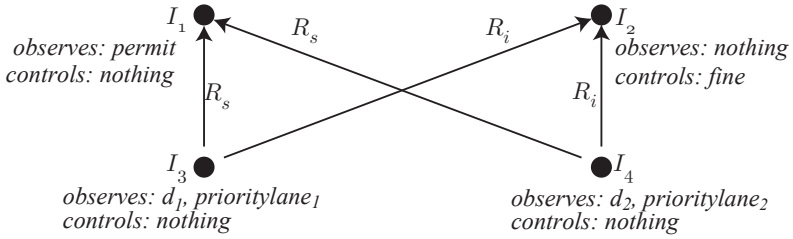

Figure 2.2: *More detailed depiction of the smart roads example. Arrows are labeled with whether they are state or institutional communication lines or both. See Examples 2.3 and 2.4.*

We observe two different types of communication that may occur among institutions in a collaborative setting. The first is communication concerning the state of the environment. These include observations in order to support the monitoring process of an institution, but also communication concerning changes that have to be realized in the environment. We model which institution may query which information from another institution, or request which changes, by a state communication relation. The second type of communication concerns communication regarding the violation of norms. As norm violations are considered institutional facts, we capture this type of communication with an institutional communication function. Hence the only messages in this type of communication are the relay of violation atoms. Consider for example the interaction between the police, courthouse and jail. Two police stations may share information in order to determine norm violations. This communication is observation communication. The police monitors norm violations and sends those to the court. That communication concerns a norm violation and is thus institutional communication. The court then determines the compensation/sanction. In case of incarceration the court informs the jail for how long and under which circumstances an agent is detained. We call the communication between the court and the jail also state communication, as it is communication that request changes in the environment state.

We separate the different communication types because often we want to constrain the information that one institution may query from another, or the control requests from one institution to another. This is related to institutional empowerment and is part of the security of an institution. Though we do not go into the details of empowerment in this chapter, our definition does capture such restrictions. A decentralized institution is a collection of institutions plus the communication relations.

Definition 2.3 (Decentralized Institution, D): *A decentralized institution D is specified by $(\mathbb{I}, R_s, R_i)$, where $\mathbb{I} = \{I_1, ..., I_k\}$ is a set of institutions, and $R_s, R_i \subseteq \mathbb{I} \times \mathbb{I}$ are the state communication relation and the institutional communication relation, respectively.*

Example 2.4 (Ex. 2.3 Cont.: Decentralized Institution): Our decentralized institution from Figure 2.2 is formally $(\{I_1, ..., I_4\}, R_s, R_i)$, where $I_j = (\Gamma_{oj}, \Gamma_{cj})$,

$j \in [1,4]$. The state communication relation is $R_s = \{(I_3, I_1), (I_4, I_1)\}$, and the institutional communication relation is $R_i = \{(I_3, I_2), (I_4, I_2)\}$. Hence, I_1 can for instance communicate about the permit that a vehicle has with I_4, but not about a norm violation. On the other hand, I_4 *can* communicate about norm violations with I_2.

2.3.3 Enforcement

It is required for norm enforcement to have the correct observation and control capabilities available. The presence of these capabilities can be local or available through communication. If the capabilities are not local then the state communication relation is used to request observations or environment changes. In case the capabilities are present, then there is also a choice which institutions do the monitoring and which institutions do the sanctioning. This leads to a distinction between centralized enforcement or a decoupled form of enforcement where one institution monitors a norm and another determines and realizes the potential sanction. Decentralized enforcement, where monitoring and sanctioning is performed by a number of institutions, uses the institutional communication relation. By splitting communication into state and institutional communication we can clearly describe the distribution of control and observe capabilities and the distribution of enforcement.

We will define the characteristics of each possible way that norms can be enforced in a decentralized institution. We begin with defining local and collaborative monitoring and sanctioning in an institution that is part of the decentralized institution. For a norm to be applicable it means that the norm can be monitored, and for a sanction to be applicable, it means that the sanction can be imposed. We assume that the sanctioning of violations is described in terms of what institution can do. For instance, an institution from our example can ultimately only give fines and not force agents to pay them (though not paying a fine could result in the notification of some other authority that can force payment).

Local monitoring and sanctioning in an institution can only happen if the right observation and control capabilities are locally present. If a norm or sanction is locally monitorable or realizable, respectively, then no communication needs to burden the network. We shall next define what it means for an institution to locally monitor and sanction given a set of norms and sanctions. For local monitoring it is required that all the antecedents of the norm counts-as rules are monitorable given the observation capability of the institution. Similarly, for local sanctioning it is required that all the consequences of the sanction rules are realizable given the control capability of the institution.

Definition 2.4 (Local Monitoring and Sanctioning): *Let N be a set of norms, S a set of sanctions and $(\mathbb{I}, R_s, R_i)$ a decentralized institution. An institution $I \in \mathbb{I}$ can locally monitor N iff $\forall(\varphi \Rightarrow v) \in N : I \vDash_o \varphi$, and I can locally sanction violations using S iff $\forall(v \Rightarrow \psi) \in S : I \vDash_c \psi$.*

Example 2.5 (Ex. 2.4 Cont.: Local Monitoring and Sanctioning): No institution from our example can locally monitor the example norm from Ex-

ample 2.2. The highway institution I_3 can locally monitor the atoms d_1 and $prioritylane_1$. Hence, it could locally monitor for instance $N = \{\neg d_1 \wedge prioritylane_1 \Rightarrow v\}$. Institution I_2 controls the fine atom and can hence locally sanction norm violations with the set of rules $S = \{v \Rightarrow fine\}$.

Collaborative monitoring and sanctioning concerns subparts of the network of institutions (Definition 2.6). An institution I can collaboratively monitor a norm or apply a sanction if the reachable institutions for I, given the state communication relation, together have the required observation and control capabilities. Given the state communication relation among institutions, one can treat observable/controllable atoms in a connected network as distributed knowledge/controllability. However, we did not specify the details of capabilities and hence have no formal definition of how capabilities can be combined through communication. Hence, we first define capability composition, of which we only demand that is monotonic (e.g. if a formula is observable for a capability in the composed capability, then it is also observable given the composed capability).

Definition 2.5 (Capability Composition, $\oplus$): *Let $I = (\Gamma_o, \Gamma_c)$ and $I' = (\Gamma'_o, \Gamma'_c)$ be two institutions. The composition of Γ_o, Γ'_o and Γ_c, Γ'_c is notated as $\Gamma_o \oplus \Gamma'_o$ and $\Gamma_c \oplus \Gamma'_c$, respectively. If for a conjunction of literals φ it holds that $\Gamma_o \vDash_o \varphi$ or $\Gamma'_o \vDash_o \varphi$ then also $\Gamma_o \oplus \Gamma'_o \vDash_o \varphi$. Similarly, if for a conjunction of literals ψ it holds that $\Gamma_c \vDash_c \psi$ or $\Gamma'_c \vDash_c \psi$ then also $\Gamma_c \oplus \Gamma'_c \vDash_c \psi$.*

Example 2.6 (Ex. 2.4 Cont.: Capability Composition): We model our example capabilities as sets of atoms. Hence, we may use the union operator for sets as the composition operator, i.e. $\Gamma \oplus \Gamma' \equiv \Gamma \cup \Gamma'$, for two observation or control capabilities Γ and Γ'. Note that this composition satisfies the monotonicity condition of the composition definition, as the set of atoms can only remain the same or grow, and hence the set of constructable formulas given a capability can also only remain the same or grow.

For collaborative monitoring and sanctioning it is required that the composition of capabilities of a group of connected institutions (through the state communication relation) can monitor a norm or control the sanction consequence.

Definition 2.6 (Collaborative Monitoring and Sanctioning): *Let N be a set of norms, S a set of sanctions and $(\mathbb{I}, R_s, R_i)$ a decentralized institution, $I \in \mathbb{I}$ an institution and $\mathbb{I}' = \{I' \in \mathbb{I} | (I, I') \in R_s\}$. Institution $I = (\Gamma_o, \Gamma_c)$ can collaboratively monitor N iff $\forall(\varphi \Rightarrow v) \in N : (\bigoplus_{(\Gamma'_o, \Gamma'_c) \in \mathbb{I}'} \Gamma'_o) \oplus \Gamma_o \vDash_o \varphi$. We also say that I can collaboratively sanction violations using S iff $\forall(v \Rightarrow \psi) \in S : (\bigoplus_{(\Gamma'_o, \Gamma'_c) \in \mathbb{I}'} \Gamma'_c) \oplus \Gamma_c \vDash_c \psi$.*

Example 2.7 (Ex. 2.6 Cont.: Collaborative Monitoring and Sanctioning): Consider I_3, for which $\mathbb{I}' = \{I_1\}$. The composition of $\Gamma_{o3} \oplus \Gamma_{o1} = \Gamma_{o3} \cup \Gamma_{o1} = \{prioritylane_1, d_1, permit\}$. The formula $\varphi = d_1 \wedge prioritylane_1 \wedge permit$ can be constructed from the atoms of $\Gamma_{o3} \oplus \Gamma_{o1}$, hence $\Gamma_{o3} \oplus \Gamma_{o1}(\varphi)$. Therefore, I_1 can collaboratively monitor the set of norms $N = \{d_1 \wedge prioritylane_1 \wedge permit \Rightarrow v\}$.

There is in the example no occurrence of collaborative sanctioning.

Observation 2.1: *Local monitoring and sanctioning is a subproperty of collaborative monitoring and sanctioning. This follows trivially as the composition of an institution's capability with other capabilities can only increase the capabilities and not shrink it. Hence a set of norms that is locally monitorable, or a set of sanction rules that is locally applicable, must also be collaboratively applicable according to the definition of collaborative monitoring and sanctioning.*

We stress that the capability composition definition does not mean that one institution necessarily has to reveal all observations that it can make to another institution. In our example scenario this is the case as we define capability compositions by disjunction of sets of observable atoms. However, it is possible to model a form of empowerment where for each institution pair $\oplus$ is separately defined in order to determine what information one institution gives to the other. Hence, collaborative monitoring and sanctioning does not always follow straightforwardly from the decentralized institution specification, but is also dependent on the chosen capability composition operator. We propose that empowerment modeling can be achieved by extending the definition of a decentralized institution with a specification of what institution can request what observations or control actions from which other institution.

To enforce norms, first monitoring takes place to determine violations. Then sanctioning takes place to compensate for the violations that occurred. There is a choice whether this process takes place within a single institution, or whether this process is decoupled among two institutions. Central enforcement of norms in an institution means that both the monitoring and the sanctioning of norms/violations is done by the same institution. The monitoring and sanctioning can however be either locally or collaboratively done (Definition 2.7). Note that for a norm set N to be enforceable using S it must hold that for each norm $\varphi \Rightarrow v \in N$ there must be at least one corresponding sanction $v \Rightarrow \psi \in S$, otherwise the violation cannot be sanctioned.

Definition 2.7 (Centralized Enforcement): *Let N be a set of norms, S a set of sanctions and $(\mathbb{I}, R_s, R_i)$ a decentralized institution. An institution $I \in \mathbb{I}$ can centrally enforce N by S iff I can locally or collaboratively monitor N, I can locally or collaboratively sanction violations using S, and for each norm $n \in N$ there exists a matching sanction rule $s \in S$.*

Example 2.8 (Ex. 2.7 Cont.: Centralized Enforcement): Our example norm and sanction cannot be centrally enforced by a single institution from the scenario. However, if the highway institutions were extended with the capability to create fines, then they could centrally enforce the example norm by the example sanction rule. However, note that they would still depend on I_1 to reveal information regarding the permits of vehicles.

Decoupled enforcement entails that monitoring and sanctioning can happen in two different institutions. This way it can happen that one institution detects a

violation of the norm and sends a violation notification through the institutional communication relation to another institution that determines the sanction. The separation of monitoring and sanctioning is quite common in for instance human societies. It resembles for instance the distinction between the police which reports crime and the court which determines the fine.

Definition 2.8 (Decoupled Enforcement): *Let N be a set of norms, S a set of sanctions and $(\mathbb{I}, R_s, R_i)$ a decentralized institution. Institutions $I \in \mathbb{I}$ and $I' \in \mathbb{I}$, $I \neq I'$, can enforce N by S in a decoupled manner iff $(I, I') \in R_i$ and I can locally or collaboratively monitor N, I' can locally or collaboratively sanction violations using S, and for each norm $n \in N$ there exists a matching sanction rule $s \in S$.*

> **Example 2.9 (Ex. 2.8 Cont.: Decoupled Enforcement):** Consider the example set of counts-as rules $N = \{d_1 \wedge \neg permit \wedge priority lane_1 \Rightarrow v\}$ and our set of sanction rules is $S = \{v \Rightarrow fine\}$. Institutions I_3 and I_2 can enforce N by S in a decoupled manner because I_3 can collaboratively monitor N and I_2 can locally sanction violations using S, and for each norm in N there is a matching sanction rule in S.

One of our aims is to specify for a set of norms, a set of sanction rules, and a decentralized institution whether those norms and sanctions rules can be monitored and applied by the decentralized institution. This specification is defined as decentralized enforcement in Definition 2.10. The definition of decentralized enforcement is built on top of the idea that the enforcement task given a set of norms and sanctions can be divided into subtasks (represented as pairs of subsets of the norms and sanctions) which have to be enforced either locally by some institution or in a decoupled manner by a pair of institutions. We refer to such a division of subtasks as a clustering and define it in Definition 2.9. If a clustering is to represent the enforcement tasks of institutions, then it is required that all matching norm-sanction pairs are covered by the clustering, otherwise some potential violations may not be sanctioned properly.

Definition 2.9 (Clustering): *Let (N, S) be a pair consisting of a set of norms N and a set of sanctions S. A clustering of (N, S) is a set of clusters $\{C_1, \ldots, C_n\}$, where $C_i = (N_i, S_i)$, $i \in [1, n]$, $N_i \subseteq N$ and $S_i \subseteq S$. Furthermore, for each matching norm-sanction pair (n, s), $n \in N$, $s \in S$, we require the existence of a cluster $C_i = (N_i, S_i)$ in the clustering such that $n \in N_i$ and $s \in S_i$.*

> **Example 2.10 (Ex. 2.9 Cont.: Clustering):** A clustering given our scenario is: $C = \{C_1, C_2\}$ such that $C_1 = (\{d_1 \wedge priority lane_1 \wedge permit \Rightarrow v\}, \{v \Rightarrow fine\})$ and $C_2 = (\{d_2 \wedge priority lane_2 \wedge permit \Rightarrow v\}, \{v \Rightarrow fine\})$.

We have defined enforcement of norms and their sanctions for a specific institution (local enforcement) or a pair of institutions (decoupled enforcement), but not yet for the decentralized institution as a whole (which we call decentralized enforcement). In Definition 2.10 we define when a set of norms and sanction rules is enforceable by a decentralized institution. This is the case if there is a clustering such that each of the norm subset and sanction subset pairs in that clustering are enforceable locally by some institution or enforceable in a decoupled manner by a pair of institutions.

Definition 2.10 (Decentralized Enforcement): *Let N be a set of norms, S a set of sanctions and $D = (\mathbb{I}, R_s, R_i)$ a decentralized institution. D can decentrally enforce N by S iff there exists a clustering $\{C_0, \ldots, C_n\}$ of (N, S) such that for each $C_i = (N_i, S_i)$, $i \in [0, n]$, there either exists an institution $I \in \mathbb{I}$ that can centrally enforce N_i by S_i or there exists a pair of institutions $(I, I') \in R_i$ such that (I, I') can enforce N_i by S_i in a decoupled manner.*

Example 2.11 (Ex. 2.10 Cont.: Decentralized Enforcement): Our example decentralized institution can decentrally enforce the set of norms $N = \{d_1 \wedge prioritylane_1 \wedge permit \Rightarrow v, d_2 \wedge prioritylane_2 \wedge permit \Rightarrow v\}$ by the set of sanction rules $S = \{v \Rightarrow fine\}$ given the clustering C from Example 2.10.

We have discussed local/collaborative availability of observation and control capabilities, and we have discussed centralized, decoupled and decentralized enforcement. In Definition 2.10 we required a clustering for which each element was either centrally enforced or in a decoupled manner, and each norm/sanction rule was either locally or collaboratively monitored/sanctioned. A decentralized institution is built from a set of institutions that are connected through a state and institutional communication relation. Some definitions are not applicable to a decentralized institution if we restrict the communication relations.

Collaborative monitoring and sanctioning (Definition 2.6) is impossible if $R_s = \emptyset$, so only local monitoring and sanctioning (Definition 2.4) then remains. This kind of design is likely to be seen in practical applications where the data for norm monitoring is too big to be handled by the communication infrastructure. In Chapter 3 we will also discuss other options to deal with large amounts of data when monitoring, such as intermediate data aggregation.

Decoupled enforcement (Definition 2.8) is impossible if $R_i = \emptyset$, so only centralized enforcement (Definition 2.7) then remains. This kind of enforcement is useful because we do not need to create an institutional infrastructure. It also increases security as important data concerning enforcement (the violations) cannot be intercepted. In Chapter 4 we will also discuss some security aspects of monitoring systems.

If both collaborative monitoring and sanctioning, and decentralized enforcement are impossible, i.e. $R_s = \emptyset$ and $R_i = \emptyset$, then there is no communication needed for enforcement. This kind of enforcement restriction is needed if we want to implement norms using a language that does not allow for inter-institutional communication (such as 2OPL [46] and NPL [73]).

Observation 2.2: In summary: If $D = (\mathbb{I}, R_s, \emptyset)$ can decentrally enforce a set of norms N by S then no subsets of N and S can be enforced in a decoupled manner. If $D = (\mathbb{I}, \emptyset, R_i)$ can decentrally enforce a set of norms N by S then only local monitoring and sanctioning of N and S respectively is possible. If $D = (\mathbb{I}, \emptyset, \emptyset)$ can decentrally enforce a set of norms N by S then only local monitoring and sanctioning of N and S respectively is possible and only centralized enforcement.

2.4 Assigning Norms and Sanctions to Institutions

Each norm has to be monitored when we implement an institution to enforce a set of norms. This requires computational effort. Similarly, each sanction to be imposed costs computational effort as well. We have the possibility in a decentralized institution to distribute this computational effort in the network of local institutions. We want to assign to institutions norms and sanctions that they must monitor/impose.

The assignment of a global set of norms and sanctions to local institutions should provide the information about which institution exactly monitors which norms and which institutions may impose what sanctions. We define this by creating two sets of pairs. The first set couples institutions with norms, the second set couples institutions with sanctions. All the given norms and sanctions must be assigned to at least one institution. Furthermore, all assigned norms must be (locally/collaboratively) monitorable by the institutions to which they are assigned. Equally, all assigned sanction rules must be applicable by the institutions to which they are assigned. Lastly it is needed that each detectable violation can be communicated to an institution that handles sanction rules for that violation.

Definition 2.11 (Assignment, A): *Let (N, S) be a pair consisting of a set of norms N and a set of sanctions S and $(\mathbb{I}, R_s, R_i)$ a decentralized institution. An assignment A is a pair (A_N, A_S), where $A_N \subseteq \mathbb{I} \times N$ and $A_S \subseteq \mathbb{I} \times S$. For every $(I, n) \in A_N$ and $(I, s) \in A_S$, n can be monitored locally or collaboratively by I and s can be locally or collaboratively applied by I to sanction. Furthermore for each matching norm-sanction pair (n, s), $n \in N$, $s \in S$, either there exists an institution $I \in \mathbb{I}$ s.t. $(I, n) \in A_N$ and $(I, s) \in A_S$ and I can centrally enforce n by s, or there exists $(I, I') \in R_i$ s.t. $(I, n) \in A_N$ and $(I', s) \in A_S$ and (I, I') can enforce n by s in a decoupled manner.*

> **Example 2.12 (Ex. 2.11 Cont.: Assignment):** For our scenario there is only one sensible assignment of the counts-as rules and the sanction rule. Due to the communication relations it must be the case that $d_1 \wedge \mathit{prioritylane}_1 \wedge \mathit{permit} \Rightarrow v$ is assigned to institution I_3 and $d_1 \wedge \mathit{prioritylane}_1 \wedge \mathit{permit} \Rightarrow v$ is assigned to institution I_4 and $v \Rightarrow \mathit{fine}$ is assigned to I_2. The only other assignments would include the addition of institution-norm/sanction rule pairs for which the institution cannot monitor the norm or apply the sanction locally or collaboratively.

Observation 2.3: Let N be a set of norms, S a set of sanctions and $D = (\mathbb{I}, R_s, R_i)$ a decentralized institution. There exists an assignment (A_N, A_S) if and only if D can decentrally enforce N by S. An assignment can be transformed into a clustering $\{C_0, \ldots, C_n\}$ by creating for each matching norm-sanction pair (n_i, s_i), $(I, n_i) \in A_N$, $(I, s_i) \in A_S$, a cluster $C_i = \{(\{n_i\}), (\{s_i\})\}$. Likewise the clustering $\{C_0, \ldots, C_n\}$ that is required for decentral enforcement can be transformed in an assignment. We know that for each matching norm pair (n, s) there exists a cluster $C_i = (N_i, S_i)$ and norm $n \in N_i$ and sanction $s \in S_i$. We also know that there exists an institution I s.t. I can centrally enforce n by s or two institutions I, I' s.t. they can enforce n by s in a decoupled manner. In the first case the assignment will contain $(I, n) \in A_N$ and $(I, s) \in A_S$. In the second case the assignment will contain $(I, n) \in A_N$ and $(I', s) \in A_S$.

2.5 Towards Practice

So far we have defined norms, (decentralized) institutions, the enforcement of norms and the assignment of norms to local institutions. To put this into practical use we would need assignment functions that given a norm and sanction rule set and a decentralized institution provides an assignment of the norms and sanction rules (or multiple candidate assignments). It is not our goal to provide such functions but we will give some examples of how they are related to our definition of norm and sanction assignments.

The solution space of assignment functions is the set of all possible assignments as per Definition 2.11. Depending on the requirements of a decentralized institution, an assignment function can access a subset of the solution space. Consider the following restrictions for an assignment of norms (A_N, A_S):

1. $\forall n \in N : \exists!(I, n) \in A_N$ and $\forall s \in S : \exists!(I, s) \in A_S$

2. $\forall n \in N : \exists(I, n) \in A_N, (I', n) \in A_N : I \neq I'$ and
   $\forall s \in S : \exists(I, s) \in A_S, (I', s) \in A_S : I \neq I'$

The first restriction is read as: 'For each norm there exist exactly one institution that monitors violations of that norm and for each sanction there exists exactly one institution that realizes the sanction, when applicable.' Assignments under the first restriction have no redundancy in the assignment of norms and sanctions, which increases efficiency. The second restriction is read as 'For each norm there are at least two institutions that monitor violations of that norm and for each sanction there are at least two institutions that realize the sanction, when applicable.'. Hence, assignments under the second restriction have at least two institutions per norm and sanction. This increases robustness of the decentralized institution. If one institution fails then the norms are still properly enforced. It depends on the designer of the decentralized institution which kind of properties should hold for assignments. Having a clear definition of assignments helps to think about what properties are possible. We discuss in Chapter 4 more aspects of robustness in the monitoring process.

If the accessed subset of the assignment solution space is not a singleton set, then an ordering is needed to determine the best assignments. For instance if we have a low bandwidth in the communication channels among institutions then those assignments are preferred where communication is kept at a minimum. Such a preference can be captured by for instance counting for norm assignments the number of norms that cannot be locally monitored by the institutions to which they are assigned.

In our definition of collaborative monitoring and sanctioning (Definition 2.6) we have not touched upon the issue of how to exactly get information from one institution to another. We stated the conditions such that it is possible. In an implementation it would be required to use a distribution annotation such that institutions know where to get which information. In [126] and Chapter 6 we discuss in more detail the implementation of institutions.

2.6 Discussion

Norms are our main focus in this thesis. We can also incorporate other organizational aspects. For instance there are interesting aspects to the use of constitutive norms when it comes to distribution. Constitutive norms relate brute facts to institutional facts. E.g. certain vehicles will count as trucks. The brute facts that are required for applying constitutive norms can also be locally or collaboratively acquired. For monitoring and sanctioning in an institution the applicability of norms and sanction rules would depend on whether the used institutional facts in those norms/sanctions rules can be derived by that institution.

We represented norms with counts-as rules that do not have context. Usually counts-as rules have the form $A \Rightarrow B$ *in* C, where C is the context in which A counts as B. Context specifies under which circumstances obligations or prohibitions hold. For instance an obligation to have a maximum velocity of 120 km/h might only hold at daytime. Adding context to our framework would require a description of how the context is evaluated and how the required information for this is gathered. If the information is possibly distributed then the applicability of norms and sanctions can alter.

Also empowerment as investigated in [39] forms an interesting topic in the context of decentralized institutions. If only a selection of institutions is empowered to determine an institutional fact, then this poses limits to the possible assignments of norms. Also information access among institutions can differ. We can filter out assignments where certain institutions need to acquire information for an observation to which they have no access. Such considerations can be added to our model by expanding upon the capability descriptions.

We have split communication in two different communication channels to clearly distinguish different types of distribution. The assumptions we made about communication, such as costless data transfer, do not allow for a sophisticated interaction model between institutions. In Chapter 3 we go into more detail on communication models and how this impacts the enforcement process.

We have not discussed an approach towards implementing the concepts in this chapter. We have proposed D2OPL [123, 126] earlier as a proof of concept. In D2OPL one programs an institution by programming the norms which that institution has to enforce. Hence, an assignment is necessary before starting to program. Each institution has its own brute state which it observes and controls (corresponding to its observation and control capabilities). However, when programming a norm, the programmer can tell the institution to get an observation from another institution. This enables collaborative monitoring. Similarly, when the institution has detected a violation and is executing a sanction, it can do this by instructing another institution to locally make changes. This enables collaborative sanctioning. Finally, sanction rules in institutions are specified separately from norms and can be conditioned on violations that are detected elsewhere. This corresponds to a realization of decoupled enforcement by a pair of institutions.

2.7 Conclusion

Literature on norm controlled multi-agent systems often calls the systems that enforce norms institutions. In this chapter we have investigated decentralized norm enforcement by investigating decentralized institutions. A decentralized institution is described as a network of institutions. Each institution has observation capabilities to observe the environment and control capabilities to manipulate the environment. Norms have to be monitored for determining whether violations have occurred, and their violations have to be sanctioned to be enforced. If institutions do not have the required local capabilities available for monitoring and sanctioning, then they may still be able to perform the monitoring and sanctioning task through collaborative monitoring (where information is shared among institutions) and collaborative sanctioning (where parts of the sanction are realized by different institutions).

We have described the possible ways in which the enforcement overhead of norms can be distributed in a decentralized institution. There are different approaches for the enforcement of norms. With centralized enforcement an institution applies both the norms and the sanctions. With decoupled enforcement a pair of institutions together enforces norms by letting one monitor the norms and the other determining and imposing the sanctions. Using these definitions we have described what proper norm assignments are. The result can be used as a basis for further investigations of decentralized institutions and as a guideline for functions that produce assignments of norms for decentralized institutions. In the rest of this thesis we will delve deeper into collaborative monitoring and control, the implementation of decentralized norm-controlled multi-agent systems and discuss example applications.

3 Decentralized Runtime Monitoring

In the previous chapter we highlighted that runtime monitoring is one of the main aspects to runtime norm enforcement. For decentralized enforcement we require decentralized monitoring. In this chapter we discuss decentralized runtime monitoring for the detection of norm violations. The first contribution is a discussion of the foundation of runtime decentralized monitoring. The second contribution is the description of a related monitoring framework and a new monitoring framework that we published earlier in [128]. Our main conclusion in this chapter is that runtime monitoring for norm violations is very closely related to monitoring the validity of linear temporal logic formulas at runtime.

3.1 Introduction

The behavior of multi-agent systems is hard to predict. This is especially the case for open multi-agent systems where we cannot obtain the specification of the agents. As a consequence, it is challenging for a system designer to guarantee that agents behave according to preset guidelines. Norms are a popular candidate solution to solve this challenge [27]. The enforcement of norms requires that violations are detected and that sanctions are applied to compensate for those violations. We focus in this chapter on the runtime detection of norm violations. Runtime monitoring for multi-agent systems can benefit a lot from decentralized techniques, because a multi-agent system is highly distributed by nature. Advantages of using decentralized monitoring include:

- Scalability. If the multi-agent system is expanded, then we may introduce new monitors to monitor the expansion. If this can be done in parallel with the existing monitors, then this is preferable over a non-parallel system. If a monitor can locally process the data that it retrieves from the multi-agent system, then this reduces communication costs compared to a system where all data is sent to a central data collection point.
- Modularity. In a decentralized monitor it is possible to selectively update monitors or take them down for maintenance. Also in terms of reusability it can be beneficial to have an approach where existing monitors can be tied together in a network.

- Graceful degradation. In a centralized approach, if the central hub fails, then the entire monitor fails. In a decentralized monitor it might be possible that part of its functionality can still be achieved if one or more local monitors fail.
- Safety. If data is processed locally, then there is no chance that communication to a central hub is intercepted.

Formal methods for decentralized monitoring help us to better analyze and design decentralized monitors. We are in particular interested in models to analyze runtime monitoring. There are different models in related work (cf. [19]). One of our contributions in this chapter is the proposal of a complementary monitoring framework. We also discuss and contrast our proposal with the proposal from Bauer and Falcone in [19]. Monitors in these approaches observe the execution trace of a system to verify properties of its behavior. These properties are expressed in linear-time temporal logic (LTL) formulas [107]. Temporal logics have been used in the past to analyze normative systems (e.g. [3, 7]). Hence we shall focus on LTL verification, with the knowledge that various normative frameworks rely on LTL. The basis of our method and that of Bauer and Falcone is a network of monitors where each monitor has its own local view on the environment. Each monitor is assigned a linear temporal formula that represents a norm, and the task of the monitor is to check whether this formula is true or false. In our proposed framework these formulas can differ between monitors. In [19] each monitor evaluates the same formula. The formulas represent norms which concern the multi-agent system that is being monitored. Because of the local view of monitors, it might be necessary that they communicate in order to detect norm violations. In [19] each local monitor can locally rewrite formulas with what is known as a progression function. Hence we call such local monitors progression monitors, and a network of such monitors a decentralized progression monitor. The communication between progression monitors consists of rewritten formulas that capture which information is still required to confirm a norm violation. In our method for decentralized monitoring we also work with communicating monitors. They propagate observations. Propagation causes delays in violation detection. Hence we call the local monitors in our second method delay monitors, and a network of such monitors a decentralized delay monitor. The proposed method is aimed at detecting a violation in the number of computation steps that it maximally takes for an observation to be propagated across the network. This type of propagation is suitable for scenarios where communication between monitors is relatively expensive and where new observations can be made in between any two consecutive messages. We make no assumptions on the topology of a decentralized delay monitor, other than that communication is acyclic. In contrast, a decentralized progression monitor needs to be fully connected.

The rest of this chapter is structured as follows: In Section 3.2 we explain linear temporal logic and its semantics for both infinite and finite sequences of states. In Section 3.3 we go deeper into the theory of runtime monitoring. In Section 3.4 we describe the monitoring approach from [19]. In Section 3.5 we describe our own monitoring approach from [128]. In Section 3.6 we relate the theory of runtime linear temporal logic monitoring to normative systems theory. Finally, we conclude this chapter in Section 3.7.

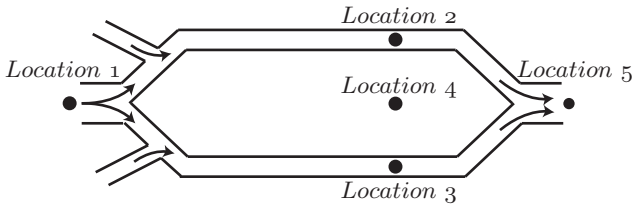

Figure 3.1: *Example scenario. Black dots indicate locations, arrows indicate traffic flow and double lines indicate roads.*

3.2 Linear Temporal Logic

The monitoring frameworks that we discuss in this chapter are not only suitable for multi-agent systems. To be a bit more general, we shall refer to a system that is being monitored as the target system. Also, we model the behavior of such a target system as a word, which is a sequence of letters from an alphabet. For the remainder of this chapter, let A be a finite, non-empty set of atomic propositions and $S = 2^A$ be the alphabet. A letter $s \in S$ in our alphabet is a set of atomic propositions which describes a state of the target system. The finite sequences of elements in S are notated as S^* and the infinite sequences of elements in S are notated as S^ω. The set of all sequences is given by $S^\infty = S^* \cup S^\omega$. We commonly refer to a sequence of elements as a word. For a word $w \in S^\infty$ we use $|w|$ to indicate the length of the word where $|w| = \infty$ if $w \in S^\omega$. The empty sequence (i.e. the word of length 0) is notated as ε. Note that $\varepsilon \in S^*$. An index i in a word $w \in S^\infty$ is a positive number $i \in [0, |w|]$ in case w is finite and otherwise $i \in [0, \infty)$. For a word $w = s_1 s_2 ... \in S^\infty$ element s_i at index i is indicated with $w[i]$. Also for a word $w = s_1 s_2 ... \in S^\infty$ and indices i and j, $i \leq j$, we use $w_{..i}$, $w_{i..j}$ and $w_{i..}$ to indicate the words $s_1 ... s_i$ (called prefix up to i), $s_i ... s_j$ (called the subsequence from i to j) and $s_i s_{i+1} ...$ (called the suffix from i), respectively. For two words $w \in S^*$ and $w' \in S^\infty$ we use $w \preceq w'$ and $w \prec w'$ to indicate that w is a prefix or strict prefix of w', respectively. Finally for two words $w \in S^*$ and $w' \in S^\infty$ we notate the concatenation of w and w' simply as ww'.

> **Example 3.1 (Scenario):** Throughout this chapter and the next we shall draw examples from the same smart infrastructure scenario. In the scenario we have various traffic streams that at some point merge together, see Figure 3.1. When modeling our example scenario we assume for simplicity that there is a single vehicle that can be at any of the locations on the road, which is indicated by the atomic proposition v_i for location $i \in \{1, 2, 3, 5\}$. Furthermore, there might be a traffic jam on locations two and three, which is modeled by the propositional atoms j_2 and j_3. Hence the full set of atomic propositions for our scenario is $A = \{v_1, v_2, v_3, v_5, j_2, j_3\}$. A state could for example be $\{v_2, j_2\} \in S$, which is read as "the vehicle is at location two and there is a traffic jam at location two". An example finite word is $w = \{v_1\}\{v_2\}\{v_5\}$, which indicates that the vehicle traveled from location one to two to five and no traffic jams occurred.

Linear Temporal Logic (LTL) [107] allows us to reason over infinite words by

specifying logic operators that represent notions of temporal reasoning. Classical LTL only has operators concerning the future; for instance a formula might say that an atomic proposition will hold somewhere in the future. LTL can be extended with past operators which does not extend the expressivity of LTL [89] but can make formulas exponentially more succinct [88]. In this chapter we focus on runtime verification of properties that are specified by future LTL formulas, but as we shall see, past operators will also be used in that regard. The syntax of LTL with past operators is defined below.

Definition 3.1 (LTL Syntax [89]): *The correctly formed LTL formulas are inductively defined by the grammar:*

$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \mathcal{X}\varphi \mid \varphi \mathcal{S} \varphi \mid \mathcal{Y}\varphi$, *where* $p \in A$.

We shall use common shorthands. For equivalence we use $\varphi_1 \equiv \varphi_2$ for $\neg(\neg\varphi_1 \vee \neg\varphi_2) \vee \neg(\varphi_1 \vee \varphi_2)$. Furthermore:

(false)	$false$	$\equiv$	$\neg true$
(conjunction)	$\varphi_1 \wedge \varphi_2$	$\equiv$	$\neg(\neg\varphi_1 \vee \neg\varphi_2)$
(implication)	$\neg\varphi_1 \vee \varphi_2$	$\equiv$	$\varphi_1 \rightarrow \varphi_2$
(sometime in the future)	$\Diamond\varphi$	$\equiv$	$true \mathcal{U} \varphi$
(always in the future)	$\Box\varphi$	$\equiv$	$\neg\Diamond\neg\varphi$
(sometime in the past)	$\blacklozenge\varphi$	$\equiv$	$true \mathcal{S} \varphi$
(always in the past)	$\blacksquare\varphi$	$\equiv$	$\neg\blacklozenge\neg\varphi$

We will often refer to the fragment of LTL formulas that do not make use of past operators. This fragment is defined below.

Definition 3.2 (Future LTL, LTL_f): *The correctly formed future LTL formulas are inductively defined by the grammar:*

$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \mathcal{X}\varphi$, *where* $p \in A$.

Example 3.2 (Ex. 3.1 Cont.: LTL Syntax): The use of past operators allows in some cases for more intuitive formulas (e.g. [83] contains such examples). Consider for example the sentence "if the vehicle passes location five then it has passed through either location two or three". With only future operators this would be translated to for instance $\Box((\neg v_5 \wedge \Diamond v_5) \rightarrow (\neg v_5 \mathcal{U} (v_2 \vee v_3)))$ which is read as "if sometime in the future (but not now) the vehicle passes location five then the vehicle does not pass location five until it has passed either location two or three", whereas with past operators we could translate the sentence to for instance $\Box(v_5 \rightarrow \blacklozenge(v_2 \vee v_3))$ which is read as "It is always the case that if the vehicle passes location five, then it passed location two or three sometime in the past", which is a more direct translation.

The semantics of LTL is defined over infinite words.

Definition 3.3 (LTL Semantics [107]): *Let $w \in S^\omega$ be an infinite word and $i \in [0, \infty)$ be an index. The semantics of LTL formulas is inductively defined as follows:*

$$
\begin{array}{lll}
w, i \models \mathit{true} & & \\
w, i \models p & \mathit{iff} & p \in w[i] \\
w, i \models \neg\varphi & \mathit{iff} & w, i \not\models \varphi \\
w, i \models \varphi_1 \vee \varphi_2 & \mathit{iff} & w, i \models \varphi_1 \text{ or } w, i \models \varphi_2 \\
w, i \models \varphi_1 \mathcal{U} \varphi_2 & \mathit{iff} & \exists k \in [i, \infty) \text{ such that } w, k \models \varphi_2 \text{ and} \\
 & & \forall l \in [i, k-1] : w, l \models \varphi_1 \\
w, i \models \mathcal{X}\varphi & \mathit{iff} & w, i+1 \models \varphi \\
w, i \models \varphi_1 \mathcal{S} \varphi_2 & \mathit{iff} & \exists k \in [0, i] \text{ such that } w, k \models \varphi_2 \text{ and} \\
 & & \forall l \in [k+1, i] : w, l \models \varphi_1 \\
w, i \models \mathcal{Y}\varphi & \mathit{iff} & i > 0 \text{ and } w, i-1 \models \varphi
\end{array}
$$

As a shorthand we write $w \models \varphi$ for $w, 0 \models \varphi$ and $\models \varphi$ if $\forall w \in S^\omega : w \models \varphi$.

> **Example 3.3 (Ex. 3.1 Cont.: LTL semantics):** Consider the infinite word $w = \{v_1\}\{v_2\}\{v_5\}s_4s_5...$ such that $s_i = \emptyset$ for $i \in [4, \infty)$. In this word the vehicle travels through locations one, two and five consecutively. For this word we have that $w \models \Box(v_5 \rightarrow \blacklozenge(v_2 \vee v_3))$ (the vehicle passed locations two or three before passing location five) and $w \not\models \Diamond((v_2 \wedge j_2) \vee (v_3 \wedge j_3))$ (i.e. the vehicle did not travel through a traffic jam).

A multi-agent system is a typical application that may run indefinitely and hence its behavior may be described adequately with infinite words. LTL is a suitable logic for reasoning over a multi-agent system's behavior. If we see the multi-agent system as a runtime process, then this system produces an infinite word from left to right. A runtime monitoring process will not have access to the infinite word that a multi-agent system is producing from the moment that the system starts. Instead, the word is revealed incrementally. For instance, let $\varphi = \Box p$, where $p \in A$, and let the system produce a word $w = \{p\}\{p\}\emptyset... \in S^\omega$ (where the continuation consists of empty states). We can determine whether $w \models \varphi$ without needing the whole word w, as from its prefix $\{p\}\{p\}\emptyset$ we can already conclude that $w, 3 \not\models p$ and hence $w \not\models \Box p$. Hence a runtime monitoring process can potentially after three observed state changes already determine that the property is violated. We need semantics for LTL on finite words in order to model this reasoning.

We follow the definition from Bauer et al. [20] for LTL semantics with respect to finite words. For formulas such as $\varphi = \Box p$, where $p \in A$, it might be the case that φ is always true in a finite prefix w of an infinite word w' that is revealed. In such cases it is not possible to tell whether $w' \models \varphi$ or whether $w' \not\models \varphi$. Therefore we need semantics for LTL that evaluate formulas with respect to finite words as being true (t), false (f) or unknown/inconclusive (u). If a formula φ is evaluated to t with respect to a finite word w, then there is no infinite word w' with w as prefix such that $w' \not\models \varphi$. If a formula φ is evaluated to f with respect to a finite word w, then there is no infinite word w' with w as prefix such that $w' \models \varphi$. If a formula φ is evaluated to u with respect to a finite word w, then there exist two different infinite words w', w'' with w as prefix and where $w' \models \varphi$ and $w'' \not\models \varphi$.

Definition 3.4 (Finite LTL Semantics): *Let $w \in S^*$ be a finite word and $i \in [1, |w|]$ be an index. The evaluation of an LTL formula φ with respect to w and an index i is defined as follows:*

$$[w, i \models \varphi] = \begin{cases} t & \text{if } \forall w' \in S^\omega : ww', i \models \varphi \\ f & \text{if } \forall w' \in S^\omega : ww', i \not\models \varphi \\ u & \text{otherwise} \end{cases}$$

As a shorthand we write $[w \models \varphi]$ for $[w, 0 \models \varphi]$ and $[\models \varphi] = v$ if $\forall w \in S^* : [w \models \varphi] = v$, where $v \in \{t, f, u\}$.

Note that a conclusive evaluation of a formula with respect to a finite word will be the same evaluation for that formula for any finite extension of that finite word. I.e. if $[w \models \varphi] = t$ (or $= f$) then for all finite words $w' \in S^*$ such that w is a prefix of w' we have that $[w' \models \varphi] = t$ (or $= f$, respectively).

Also note that formulas without future operations will never be evaluated to u. I.e. let φ be a formula without future operators and $w \in S^*$ be a finite word and $i \in [0, |w|]$ be an index, we have that: $[w, i \models \varphi] \neq u$.

Example 3.4 (Ex. 3.1 Cont.: Finite LTL Semantics): Let $\varphi = \Diamond(v_1 \wedge j_2 \wedge \neg(\neg v_5 \mathcal{U} v_3))$ be an example LTL formula. This formula is true for an infinite word $w \in S^\omega$, i.e. $w \models \varphi$, if and only if at some point the vehicle arrives at location one whilst there is a traffic jam at location two, and passes location five before it passes location three. This formula can be seen as a representation for the violation of an obligation to go through location three upon passing location one whilst there is a traffic jam at location two, before exiting through location five. Note that for each prefix w' of a word $w \in S^\omega$ where $w \models \varphi$ we have that $[w' \models \varphi] = u$. I.e., if the vehicle does not violate the obligation until some point, then it may still violate the obligation at a later point. Also, for each word $w \in S^\omega$ such that $w \not\models \varphi$ there is a prefix w' such that $[w' \models \varphi] = f$. I.e. if the obligation is violated, then there is an arbitrary long but finite prefix of the word that describes the behavior, such that given that prefix the violation can be detected. Finally, there exists no finite word $w \in S^*$ such that $[w \models \varphi] = t$, meaning that at no point can a runtime monitoring process determine whether φ will be satisfied by the behavior of the multi-agent system.

3.3 Monitorability

In this section we turn towards a fundamental analysis of runtime monitoring where the behavior of a target system is modeled with an infinite sequence of states and the properties to monitor are represented with LTL. The monitors that we want to engineer should be generally applicable and not be designed for a particular target system, so that we may reuse them. For instance we may have a monitor that outputs the evaluation of $\Diamond p \vee \Diamond q$. If we apply such a monitor on a target system, then we might be able to occasionally conclude at runtime that the monitor can be switched off and save computational resources. For instance, if a monitor that monitors $\Diamond p \vee \Diamond q$

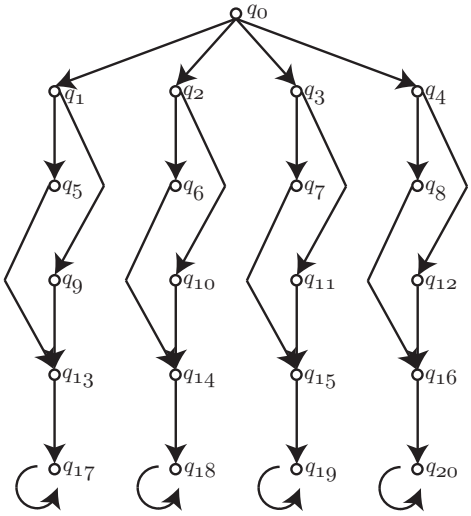

Figure 3.2: *Transition relation for Example 3.5*

	$\{v_1\}$	$\{v_2\}$	$\{v_3\}$	$\{v_5\}$	$\emptyset$
$\{j_2, j_3\}$	q_1	q_5	q_9	q_{13}	q_{17}
$\{j_2\}$	q_2	q_6	q_{10}	q_{14}	q_{18}
$\{j_3\}$	q_3	q_7	q_{11}	q_{15}	q_{19}
$\emptyset$	q_4	q_8	q_{12}	q_{16}	q_0, q_{20}

Table 3.1: *Valuation table for Example 3.5.*

observes q sometime in the future, then the monitor can be switched off since it is known that the target system's behavior satisfies the formula. There are also situations where we have a monitor for specific formula, a target system and the observed target system's behavior up to a finite point, such that we can determine that the monitor will never find out whether the target system satisfies the formula or not. For instance, this holds for a monitor that is monitoring whether there is always a new traffic jam somewhere in the future. The notion of monitorability, which we investigate in this section, captures the intuition of when we may switch off a monitor due to it being unable to ever reach a conclusive answer (true/false).

We define the model of a target system as a transition system where for a given set of propositional atoms A each state of the model can be mapped to a propositional system state in $S = 2^A$.

Definition 3.5 (Target System, T): *A target system is specified by $T = (Q, q_0, \delta, V)$ where Q is a set of states, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times Q$ is the transition relation, and $V : Q \to S$ is the valuation function.*

Example 3.5 (Ex. 3.4 Cont.: Target System): We shall specify our example target system $T = (Q, q_0, \delta, V)$. We assume that the vehicle cannot go against the direction of traffic and that the traffic jam status does not change whilst the vehicle moves through the scenario. I.e. if there is a jam when the vehicle reaches location one, then this jam will not go away. $Q = \{q_0, ..., q_{20}\}$ consists of twenty-one states. δ is depicted by arrows in Figure 3.2, where if an arrow goes from q to q' then $(q, q') \in \delta$. V is depicted in Table 3.1. The valuation of a state q is the union of the row and column set of atoms for that state. For instance $V(q_0) = \emptyset \cup \emptyset$ and $V(q_6) = \{v_2\} \cup \{j_2\}$.

Each target system has a set of sequences of states through which it may transition. Each of these state sequences can be mapped to words in S^∞. The resulting set of words is a language, which we refer to as the monitored language, as the words in the monitored language are the words that may be observed by monitors. The monitored language can be seen as the set of possible words that might be produced by the system that is monitored.

Definition 3.6 (Monitored Language, $\mathcal{L}_T$): *Let $T = (Q, q_0, \delta, V)$ be a target system. The language $\mathcal{L}_T \subseteq S^\infty$ of T consists of all words $w = s_1 s_2 ... \in S^\infty$ such that there is a sequence $q_1 q_2 ... \in Q^\infty$ where $q_1 = q_0$ and for each $i \in [1, |w|]$ (or $i \in [1, \infty)$ if $w \in S^\omega$) we have $V(q_i) = s_i$ and if $|w| > 1$ then for each $j \in [2, |w|]$ (or $j \in [2, \infty)$ if $w \in S^\omega$) we have that $(q_{j-1}, q_j) \in \delta$.*

Example 3.6 (Ex. 3.5 Cont.: Monitored Language): An example word in the monitored language $\mathcal{L}_T$ given our example target system T is $s_1 s_2 s_3 s_4 s_5 = V(q_0)V(q_2)V(q_6)V(q_{14})V(q_{18})$ (in which the vehicle passes through a jam). Note that for each infinite word $w \in \mathcal{L}_T$ it holds that $w \models \Diamond v_5$ whereas $\Diamond v_5$ is not a logical validity, i.e.: $\not\models \Diamond v_5$.

Note that $\mathcal{L}_T$ is prefix closed. I.e. let T be a target system, for each word $w \in \mathcal{L}_T$ and each prefix $w' \preceq w$ we have that $w' \in \mathcal{L}_T$. For a target system T we use $\mathcal{L}_T^{\preceq} = \mathcal{L}_T \cap S^*$ for the set of all words in $\mathcal{L}_T$ that are finite and are hence a prefix of behaviors that can be observed at runtime. We use $\mathcal{L}_T^{\omega} = \mathcal{L}_T \cap S^\omega$ for the set of all infinite words in $\mathcal{L}_T$, which are considered to be the possible behaviors of the target system of which prefixes can be observed.

Throughout this chapter we assume that the target system corresponds exactly with the system that is being modeled. However, if we want to assume that there is no knowledge aside from an initial state $s_{init} \in S$, then we can easily construct a target system $T = (Q, q_0, \delta, V)$ such that $\mathcal{L}_T = S^\infty$. We do this by first constructing Q and V such that for each $s \in S$ there is a $q \in Q$ where $V(q) = s$. Second, we specify $q_0 = s_{init}$ such that $V(s_0) = \varepsilon$. Third we specify $\delta = Q \times Q$.

The process of monitoring a system requires energy and computational resources. Therefore it is often useful to determine whether a monitor can be "switched off". In some cases we can also determine that a monitoring process will not be reaching a conclusive evaluation for a formula, in which case the process can be ignored all together. For instance, assume we are interested in the evaluation of $\varphi = \Diamond p$, $p \in A$,

given the behavior of a target system T. If $[w \models \varphi] = u$ for each $w \in \mathcal{L}_T^{\preceq}$, then it will be impossible to ever decide, based on a finitely produced word from the target system, whether the system's behavior satisfies φ. Hence any runtime monitor will never conclude something different than that the satisfaction of φ is inconclusive. To capture these intuitions we will define the notion of monitorability which is based upon the definitions from Pnueli and Zaks [108] and Bauer et al. [20]. The difference is that we consider a given transition system, which in those works is not assumed. We say that a formula is monitorable with respect to a target system and a word, if and only if there is a possibility that the target system will act in such a way that the formula becomes satisfied or false. This roughly corresponds with saying that a monitor should still remain active. Recall that at runtime a monitor has to decide upon the truth value of a property using a finite view (a prefix) of the infinite word that describes the target system's behavior. Hence monitorability is defined with respect to a finite word w. Monitorability for a property and finite word w is dependent on whether that finite word w is a prefix of an infinite word w' (describing possible target system behavior at runtime) such that if zero or finitely more states of w' are revealed (resulting in a new finite word w'') that then the truth value of the property can be determined. Note that we are interested in verifying LTL properties over infinite words, but that strictly speaking the monitored language may contain finite words without an infinite extension, which is why we require both the infinite word w' and prefix w''.

Definition 3.7 (Monitorable): *Let T be a target system and $w \in \mathcal{L}_T^{\preceq}$ be a word. An LTL formula φ is monitorable with respect to T and w iff there exists a word $w' \in \mathcal{L}_T^{\omega}$ such that $w \preceq w'$ which has a finite prefix $w'' \preceq w'$ where $w \preceq w''$ and $[w'' \models \varphi] \neq u$.*

> **Example 3.7 (Ex. 3.6 Cont.: Monitorable):** Consider the word $w = s_1 ... s_5 = V(q_0)V(q_2)V(q_6)V(q_{14})V(q_{18})$. It holds that $[w \models \Box\neg(v_2 \wedge j_2)] = f$ because $[w, 3 \models v_2 \wedge j_2] = t$ (note: $w[3] = V(q_6) = \{v_2, j_2\}$). The formula $\Box\neg(v_2 \wedge j_2)$ (the vehicle is never in a jam at location two) is therefore a monitorable formula with respect to our example target system T and the empty word ε. However, if for instance the target system was updated into T' such that state q_6 and all its connected transitions are removed, then this formula would not be monitorable with respect to T' and ε.

The consequence of regarding a target system as opposed to [108] and [20] is that a formula which might not be monitorable with respect to a target system T and word w, might become monitorable with respect to another target system T' and the same word w. Consider for example the formula $\Diamond p$ and let w be the empty word ε. Also assume that for T it holds that for each word $w' \in \mathcal{L}_T^{\preceq}$ we have that $[w' \models \Diamond p] = u$ and for T' there is a $w'' \in \mathcal{L}_{T'}$ such that $[w'' \models \Diamond p] = t$. In this case, $\Diamond p$ is not monitorable with respect to T and w, but is monitorable with respect to T' and w.

Also there are some formulas that are not monitorable with respect to any target system and word. These formulas correspond to the not monitorable formulas from [108] and [20]. An example is $\Box\Diamond p$. For instance, even if in the monitored language $\mathcal{L}_T$ we have for each word $w \in \mathcal{L}_T^{\omega}$ that $w \models \Box\neg p$, then still $\Box\Diamond p$ is not

monitorable. For a formula φ to be monitorable with respect to a target system and finite word implies that there is at least one finite word w' such that $[w' \models \varphi] \neq u$. The semantics of finite LTL is defined over all possible words, not only those of a target system[1]. There exists no finite word w' such that $[w' \models \Box\Diamond p] \neq u$, therefore $\Box\Diamond p$ cannot be monitorable with respect to any finite word and target system. If a formula is not monitorable then a runtime monitor will always evaluate the formula to u.

Note that once a formula is not monitorable with respect to a given target system and a word, then this formula will remain not monitorable with respect to the same target system and any extension of the word. I.e. a monitor that evaluates the formula at runtime can be switched off when it is determined that the formula has become not monitorable. On a side note, a monitor can also be switched off if it has determined that a formula is true or false, because the evaluation cannot change if the target system continues execution.

Proposition 3.1: *Let T be a target system and $w \in \mathcal{L}_T^{\preceq}$ be a word, and φ be an LTL formula. If φ is not monitorable with respect to T and w, then φ is not monitorable with respect to T and any $w' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w'$.*

Proof. From the definition of monitorable it follows that φ is not monitorable with respect to T and w iff for each $w' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w'$ it holds that $[w' \models \varphi] = u$. The set of words of which w is a prefix is a superset of the set of words of which a word w' such that $w \preceq w'$ is a prefix. Hence if φ is not monitorable with respect to T and w, then for an arbitrary $w' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w'$, φ is also not monitorable with respect to T and w'. □

It might be the case however that if a formula is monitorable, that then the target system still produces prefixes of an infinite word such that for each prefix the evaluation of the formula is inconclusive. The consequence is that a monitor which evaluates the formula can then never be switched off and hence runs infinitely. If we want to guarantee that a monitor for a formula eventually can be switched off after a word, then the formula must either non-monitorable or strictly monitorable after that word as defined below.

Definition 3.8 (Strictly Monitorable): *Let T be a target system and $w \in \mathcal{L}_T^{\preceq}$ be a word. An LTL formula φ is strictly monitorable with respect to T and w iff for each $w' \in \mathcal{L}_T^{\omega}$ such that $w \preceq w'$ there is a prefix $w'' \preceq w'$ where $w \preceq w''$ and $[w'' \models \varphi] \neq u$.*

> **Example 3.8 (Ex. 3.7 Cont.: Strictly Monitorable):** The aforementioned formula $\Box\neg(v_2 \vee j_2)$ is not strictly monitorable with respect to our example target system T and ε, because there exist infinite words where the vehicle never ends up in a traffic jam at location two. An example formula that is strictly monitorable with respect to T and the empty word ε is $\varphi = \Diamond(v_5 \wedge \blacklozenge v_2)$ (if the vehicle passes location five somewhere in the future, then it passed location two earlier). For

[1] This ensures that monitors which implement this semantics are generally applicable.

each infinite word in $\mathcal{L}_T$ there is a prefix such that φ is true or false according to finite LTL semantics.

As intuition demands, if a formula is monitorable with respect to a target system and a word, then it may become strictly monitorable in the future for an extension of the word.

Proposition 3.2: *Let T be a target system, $w \in \mathcal{L}_T^{\preceq}$ be a word and φ be a monitorable LTL formula with respect to T and w. There exists a word $w' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w'$ and φ is strictly monitorable with respect to T and w'.*

Proof. From the definition of monitorable it follows that if φ is monitorable with respect to T and w, then there is a $w' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w'$ and $[w' \models \varphi] \neq u$. As established before, if $[w' \models \varphi] \neq u$ then $[w' \models \varphi] = [w'' \models \varphi]$ for any extension w'' of w. Hence φ would become strictly monitorable with respect to T and w'. □

From this it also directly follows that if a formula is strictly monitorable with respect to some target system and word, then that formula must be monitorable with respect to the target system and each prefix of that word.

Proposition 3.3: *Let T be a target system, $w \in \mathcal{L}_T^{\preceq}$ be a word and φ be a strictly monitorable LTL formula with respect to T and w. For each word $w' \in \mathcal{L}_T^{\preceq}$ such that $w' \preceq w$ we have that φ is monitorable with respect to T and w'.*

Proof. If φ is strictly monitorable with respect to T and w, then there exists a $w'' \in \mathcal{L}_T^{\preceq}$ such that $w \preceq w''$ and $[w'' \models \varphi] \neq u$. Hence φ cannot be not monitorable with respect to T and any word w' such that $w' \preceq w$. □

The combination of monitorability and strict monitorability allows us to determine whether for a given target system and LTL formula it is guaranteed that no matter how the target system behaves, at some point a monitor that evaluates the formula will be able to conclude that the behavior will satisfy, not satisfy or remain inconclusive regarding the formula's evaluation. This is useful for practical considerations. If we can determine the prefix size such that a monitor concludes that its output will remain inconclusive if it is not conclusive already, then we can set this prefix size as an automatic turn-off moment.

Theorem 3.1: *Let T be a target system and φ be an LTL formula. If there is not an infinite word $w \in \mathcal{L}_T \cap S^\omega$ such that for each of its prefixes $w' \in \mathcal{L}_T^{\preceq}$, $w' \preceq w$, φ is monitorable but not strictly monitorable with respect to T and w', then there is an arbitrary but not infinite $k \in [0, \infty)$ where for each $w'' \in \mathcal{L}_T^{\preceq}$, $|w''| = k$, φ is either not monitorable with respect to T and w'', or $[w'' \models \varphi] \neq u$.*

Proof. Assume there is not an infinite word $w \in \mathcal{L}_T \cap S^\omega$ such that for each of its prefixes $w' \in \mathcal{L}_T^{\preceq}$, $w' \preceq w$, φ is monitorable but not strictly monitorable with respect to T and w. In this case we get a contradiction if there is not an arbitrary big but not infinite $k \in [0, \infty)$ where for each $w'' \in \mathcal{L}_T^{\preceq}$, $|w''| = k$, φ is either not monitorable with respect to T and w'', or $[w'' \models \varphi] \neq u$. Without such a k there must be an infinite word $w \in \mathcal{L}_T \cap S^\omega$ such that for each prefix $w' \preceq w$ φ is monitorable with respect to T and w' and $[w' \models \varphi] = u$. Because for every prefix $w' \preceq w$ we have that

$[w' \models \varphi] = u$, it follows that φ cannot be strictly monitorable with respect to T and any prefix of w. Therefore, w is an infinite word such that for each of its prefixes φ is monitorable but not strictly monitorable with respect to T and the prefix, which contradicts our assumption. □

3.4 Monitoring with Progression

The first monitoring framework that we discuss is the one which is proposed by Bauer and Falcone in [19] and which serves as an inspiration for the frameworks of Sections 3.5 and 4.2. We do not repeat the full formal model in this section, but only discuss the intuition behind their approach. This framework is suitable for scenarios where a system cannot be monitored by a single monitor and all communication takes place through a central bus. This implies that if multiple monitors are used, that they then share the same communication bus. A practical example from [19] is the case of monitoring the inside of a car. Different monitors can monitor different parts of the car (e.g. engine temperature, oil, brake status, etc), and communicate with each other through the car's communication bus. However, other components of the car also use this bus. The aim of their framework was to show that monitors only require a certain maximum number of messages to be sent (that being one per monitor) in order to monitor some global property of the system's behavior.

The monitoring approach from [19] is based on a syntactical rewriting method for LTL formulas. In a centralized setting where the monitor can see the entire state of the target system it would start with the LTL formula that is being monitored and rewrite it whenever a new state is revealed. If the formula becomes equivalent to $true$ or $false$ then the property is shown to be true or false, respectively, given the behavior of the target system. Otherwise the verdict remains inconclusive. As an example, consider the formula $\varphi = p \wedge \Diamond q$ and some word $w = s_1 s_2 ....$ If s_1 is revealed at runtime and $s_1 = \{p\}$ then φ is rewritten to $\varphi' = \Diamond q$, which can be read as "given the information of state one, $w, 0 \models \varphi$ if and only if $w, 1 \models \varphi'$". If state s_2 is revealed to be empty ($\emptyset$) then the formula is rewritten to $\varphi'' = \varphi'$, which can be read as "given the information of state one and two, $w, 0 \models \varphi$ if and only if $w, 2 \models \varphi''$". Finally assume that $s_3 = \{q\}$. In this case $\varphi''' = true$, which is read as "given the information of state one, two and three, $w, 0 \models \varphi$ if and only if $w, 3 \models \varphi'''$". Since necessarily $w, 3 \models true$ we can stop monitoring and conclude that φ holds for w.

The rewriting method that is used in the framework from [19] is called the progression function. The progression function was proposed by Bacchus and Kabanza in [13], originally deployed in a goal planning system, and has been an inspiration for other related works on monitoring as well, such as $hyMITL^{\pm}$ [42] and the framework from Havelund and Rosu [69]. The work in [42] and [69] does not deal with decentralized monitoring though. In a decentralized setting we assume that monitors may not have a full view on the target system. The progression function has to consider this limitation. Consider again the formula $\varphi = p \wedge \Diamond q$ and the same word $w = s_1 s_2 ...$ where $s_1 = \{p\}$. If the monitor is unable to observe p then we still want to obtain a formula φ' such that $w, 0 \models \varphi$ if and only if $w, 1 \models \varphi'$. Without knowing whether $p \in s_1$, the best that we can do is rewrite φ to $\varphi' = \mathcal{Y}p \wedge \Diamond q$, because $w, 0 \models \varphi$ if and

only if $w, 1 \models \varphi'$. If we then rewrite the formula again given $s_2 = \emptyset$ then the new formula becomes $\varphi'' = \mathcal{Y}\mathcal{Y}p \wedge \Diamond q$. Finally upon state $s_3 = \{q\}$ we can rewrite φ'' to $\varphi''' = \mathcal{Y}\mathcal{Y}\mathcal{Y}p$. Since it does not follow that $w, 3 \models \varphi'''$, we can see that the monitor remains inconclusive due to its limited view on the target system. Before we move on to how the decentralized framework from [19] deals with partial views we first define the progression function for completeness.

The progression function does not have the full set of all LTL formulas as its domain. For clarity we shall define the used fragment of LTL for progression, though Bauer and Falcone do not explicitly specify this fragment in [19]. We use $LTL_{\mathcal{P}}$ for the set of all LTL formulas that contain no since operator ($\mathcal{S}$), and where for each occurrence of the yesterday operator ($\mathcal{Y}$) the subject must be either another yesterday formula, or a propositional atom. We shall use $\mathcal{Y}^x p$ as shorthand for x nested yesterday operators, followed by $p \in A$. We note that constraining ourselves to this fragment of LTL is not an expressiveness limitation as LTL without the since operator is as expressive as LTL with the since operator.

Definition 3.9 (Progression LTL, $LTL_{\mathcal{P}}$): *The correctly formed progression LTL formulas are inductively defined by the grammar:*

$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid \mathcal{X}\varphi \mid \mathcal{Y}\psi$, *where* $p \in A$ *and:*

$\psi ::= p \mid \mathcal{Y}\psi$, *where* $p \in A$.

Progression of formulas is something that monitors do locally whilst the behavior of the target system is revealed. As mentioned, we assume that monitors may be limited to a partial view of the target system. This is represented by a set of atoms called a local view, which are the atoms of which a monitor can observe whether they are included or excluded in a given state of the target system. Let $\pi \subseteq A$ be a local view, $\varphi \in LTL_{\mathcal{P}}$ be a formula and $w \in S^{\omega}$ of length k be a finite word representing revealed behavior of the target system. The progression of φ given π and w, denoted as $\mathcal{P}(\varphi, \pi, w) = \varphi'$ is read as: "Given an infinite word $w' \in S^{\omega}$ it holds that $ww', k \models \varphi$ if and only if $ww', k+1 \models \varphi'$". The following function was given by Bacchus and Kabanza in [13].

Definition 3.10 (Progression Function [13], $\mathcal{P}$): *Let* $\pi \subseteq A$ *be a local view,* $p \in A$, $\varphi, \varphi_1, \varphi_2 \in LTL_{\mathcal{P}}$ *and* $w \in S^*$ *such that* $|w| = k$. *The progression function* $\mathcal{P} : LTL_{\mathcal{P}} \times 2^A \times S^* \to LTL_{\mathcal{P}}$ *is inductively defined as follows:*

$$
\begin{aligned}
\mathcal{P}(true, \pi, w) &= true \\
\mathcal{P}(false, \pi, w) &= false \\
\mathcal{P}(p, \pi, w) &= \begin{cases} true & \text{if } p \in w[k] \cap \pi \\ false & \text{if } p \in \overline{w[k]} \cap \pi \\ \mathcal{Y}p & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P}(\neg\varphi, \pi, w) &= \neg\mathcal{P}(\varphi, \pi, w) \\
\mathcal{P}(\varphi_1 \vee \varphi_2, \pi, w) &= \mathcal{P}(\varphi_1, \pi, w) \vee \mathcal{P}(\varphi_2, \pi, w) \\
\mathcal{P}(\varphi_1 \wedge \varphi_2, \pi, w) &= \mathcal{P}(\varphi_1, \pi, w) \wedge \mathcal{P}(\varphi_2, \pi, w) \\
\mathcal{P}(\varphi_1 \mathcal{U} \varphi_2, \pi, w) &= \mathcal{P}(\varphi_2, \pi, w) \vee (\mathcal{P}(\varphi_1, \pi, w) \wedge \varphi_1 \mathcal{U} \varphi_2) \\
\mathcal{P}(\mathcal{X}\varphi, \pi, w) &= \varphi \\
\mathcal{P}(\mathcal{Y}^x\varphi, \pi, w) &= \begin{cases} true & \text{if } p \in w[k-x] \cap \pi \\ false & \text{if } p \in \overline{w[k-x]} \cap \pi \\ \mathcal{Y}^{x+1}p & \text{otherwise} \end{cases}
\end{aligned}
$$

Example 3.9 (Ex. 3.8 Cont.: Progression Function): In this example we show how yesterday-operators are removed and added by the progression function by progressing a formula with a yesterday operator where the nested atom can be observed and another, not nested, atom cannot be observed. Let $\varphi = \mathcal{Y}^1 p \wedge (true \mathcal{U} q)$ (previously p and somewhere in the future q) be a formula, $\pi = \{p\}$ be a local view and $w \in S^*$ be a word such that $w[1] = \{p\}$ and $w[2] = \emptyset$. The progression of φ given π and $w_{..2}$ goes as follows:

1. $\mathcal{P}(\varphi, \pi, w_{..2}) = \mathcal{P}(\mathcal{Y}^1 p, \pi, w_{..2}) \wedge \mathcal{P}(true \mathcal{U} q, \pi, w_{..2})$
2. $\mathcal{P}(\mathcal{Y}^1 p, \pi, w_{..2}) = true$ because $p \in w[2-1] \cap \pi$ $(p \in \{p\} \cap \{p\})$
3. Therefore $\mathcal{P}(\varphi, \pi, w_{..2}) = true \wedge \mathcal{P}(true \mathcal{U} q, \pi, w_{..2})$
4. $\mathcal{P}(true \mathcal{U} q, \pi, w_{..2}) = \mathcal{P}(q, \pi, w_{..2}) \vee (\mathcal{P}(true, \pi, w_{..2}) \wedge true \mathcal{U} q)$
5. $\mathcal{P}(q, \pi, w_{..2}) = \mathcal{Y} q$ because $q \notin w[2] \cap \pi$ and $q \notin \overline{w[2]} \cap \pi$
6. Therefore $\mathcal{P}(true \mathcal{U} q, \pi, w_{..2}) = \mathcal{Y} q \vee (\mathcal{P}(true, \pi, w_{..2}) \wedge true \mathcal{U} q)$
7. $\mathcal{P}(true, \pi, w_{..2}) = true$
8. Therefore $\mathcal{P}(true \mathcal{U} q, \pi, w_{..2}) = \mathcal{Y} q \vee (true \wedge true \mathcal{U} q) = \mathcal{Y} q \vee (true \mathcal{U} q)$
9. Therefore $\mathcal{P}(\varphi, \pi, w_{..2}) = true \wedge \mathcal{Y} q \vee (true \mathcal{U} q) = \mathcal{Y} q \vee (true \mathcal{U} q)$

This derivation follows our intuition: the monitor observes p to hold in the first state, hence the only remaining question is whether somewhere in the future q holds. It cannot observe whether q is true currently, therefore, in the next state either q must have held previously or will hold somewhere in the future.

We will explain the rest of the progression based monitoring system by an informal example. For the full formal details of progression based monitoring see [19]. A decentralized progression monitor consists of a set of local monitors that have a local view on the target system. A view is modeled as a set of propositional atoms that indicate that the monitor can observe whether those propositional atoms are true or not in a target system state. Every local monitor has a formula assigned to it. Initially all monitors have the same assigned formula. For instance, we may have a

decentralized monitor that consists of two local monitors where the first one has as a view $\pi_1 = \{p\}$ and the second one has as a view $\pi_2 = \{q\}$. The initially assigned formula is $\varphi = p \wedge \Diamond q (= p \wedge (true \mathcal{U} q))$, which we used earlier as well. The monitor process works as follows.

Upon a reveal of a new state, each monitor progresses its assigned formula using its view on the newly revealed state and the past observations that it made. Should any monitor now have a formula that is equivalent to $true$ or $false$ then the initially assigned property is shown to be true or false, respectively, and the entire system halts. Otherwise it continues into a communication phase. For our example, if the first state equals $\{p\}$, then the first monitor progresses the formula as $\mathcal{P}(\varphi, \{p\}, \{p\}) = \mathcal{Y}q \vee \Diamond q$, and the second monitor as $\mathcal{P}(\varphi, \{q\}, \{p\}) = \mathcal{Y}p \wedge \Diamond q$.

If the progression of a formula by a monitor includes yesterday operators, then it can be seen as an indication that this monitor cannot verify the formula alone. After all, progression only introduces new yesterday operators when a propositional atom occurs in the assigned formula that is not in the view of the monitor. In [19] the monitor may then send the formula to another monitor. The recipient of the formula is determined by picking the monitor which in its local view has the atom that is nested in the greatest number of yesterday operators. This detail is important for practical reasons. In the worst case if there are k local monitors, then the greatest number of nested yesterday operators is k. This means that a local monitor only has to remember the last k observations. Furthermore, since each monitor may send maximally one message, we also guarantee that the communication bus is burdened with maximally k messages per observation round. In our example, the first monitor has $\mathcal{Y}q$ in its assigned formula and the second monitor $\mathcal{Y}p$. Therefore they send each other their formula. In general, if a monitor is to receive multiple formulas, then it will concatenate these by conjunction. The resulting formulas after communication are used as input for the progression function when the next state is revealed[2]. In the example though they simply swap their formulas and the next round begins.

If the second state in our example equals $\{q\}$ then the first monitor will progress its assigned formula as $\mathcal{P}(\mathcal{Y}p \wedge \Diamond q, \{p\}, \{p\}\{q\}) = \mathcal{Y}q \vee \Diamond q$, and the second monitor progress is assigned formula as $\mathcal{P}(\mathcal{Y}q \vee \Diamond q, \{q\}, \{p\}\{q\}) = true$. Since the second monitor now has $true$ assigned to it we can stop the decentralized monitor.

3.5 Monitoring with Delay

In this section we present an adaptation of the framework that we published earlier in [128]. We refer to this method as monitoring with delay, where delay refers to "delay of observations". A decentralized delay monitor consists of multiple local monitors which are referred to as delay monitors. Originally we modeled in [128] delay monitors as entities that report on the violation of norms. In order to be more consistent with the other material in this thesis, but without losing the original intent, we shall model delay monitors here as entities that provide an evaluation given a word and future LTL formula. In Section 3.6 we will discuss the relation between LTL and norms. As in the previous frameworks, a local monitor may have a partial

[2]Unless the monitor sent a formula, but received none, in which case it remains idle.

view of the target system. However, some observations might be propagated among delay monitors which causes some delay. We model observations as truth values of propositional atoms. The delay that is caused by observation propagation is modeled by a delay function which returns the delay for each atom. The delay of an observation is measured in abstract time units that correspond with the time between two states in a word.

We only focus on delays that are caused by the communication among delay monitors. However, delays can in practice also originate from for instance sensors that make observations some time after they happen. Extending the framework with additional causes of delay (e.g., delayed sensors) will impact some of the results regarding the delays of when a formula is conclusively evaluated. The main limitation for extending the model's delay specification is that the delay must be measured in equal time units.

We model a delay function as a function from atoms to positive natural numbers. The delay of observations causes a delay monitor to not be able to distinguish between different words. For instance, if atom p has a delay of two, then for any finite word that is being read it is unknown whether p holds for the final state or not. Given a finite word w, the indistinguishable words from $w = s_1...s_k$ are those words $w' = s'_1...s'_k$ where for each atom p and index $i \in [1, k]$ the valuation of p given s_i equals the valuation of p given s'_i if the delay has passed for p at moment i. For instance, assume that the delay of p is x and the target system has revealed the word $w = s_1...s_k$. In this case the monitor can observe whether or not $p \in s_i$ for $i \in [1, k-x]$, but for the other states it has not observed yet whether $p \in s_j$ where $j \in [x+1, k]$. Hence we can view the word as having an observed and unobserved part with respect to a propositional atom, as visualized by:

$$w = \overbrace{s_1...s_{k-x}}^{observed}\ \overbrace{s_{x+1}...s_k}^{unobserved}$$

where k is the length of w and x is the delay of the atom. Two words $w = s_1...s_k$ and $w' = s'_1...s'_k$ are indistinguishable if they are of equal length and for each propositional atom if s_i is in the observed part of w for that atom (the delay of the atom is less or equal to $k - i$) then the atom is either both in s_i and s'_i or in neither. Note that by this definition every word is indistinguishable from itself. Also, if the delay is zero for every atom for some delay function, then the indistinguishability relation for that function is equal to the identity function. Finally, if the delay for an atom is infinite (∞), then it means that the atom is unobservable and its truth value can never be determined.

Definition 3.11 (Word Indistinguishability, $\sim_\delta$): *Let $\delta : A \to \mathbb{N}^0 \cup \{\infty\}$ be a function called delay function. Two words $w, w' \in S^*$ of equal length k are indistinguishable given $\sim_\delta$, notated as $w \sim_\delta w'$, iff for each $i \in [1, k]$ and $p \in A$ if $\delta(p) \leq k - i$ then $p \in w[i] \cap w'[i]$ or $p \notin w[i] \cup w'[i]$.*

> **Example 3.10 (Ex. 3.4 Cont.: Word Indistinguishability):** We will illustrate indistinguishability by an example from our smart roads scenario where a monitor receives traffic information with a delay. Specifically, information from location one is observed instantly, information from location two is observed with

a delay of one and information from location three with a delay of two. Hence δ of this monitor is defined by: $\delta(v_1) = 0$, $\delta(v_2) = \delta(j_2) = 1$ and $\delta(v_3) = \delta(j_3) = 3$. Given our target system T from Example 3.5 consider the three words in our example target language $\mathcal{L}_T$ of length five in which the vehicle travels through locations one, two five and there is a traffic jam somewhere:

1. $w = V(q_0)V(q_1)V(q_5)V(q_{13})V(q_{17})$ (jams at location two and three)
2. $w' = V(q_0)V(q_2)V(q_6)V(q_{14})V(q_{18})$ (jam at location two only)
3. $w'' = V(q_0)V(q_3)V(q_7)V(q_{15})V(q_{19})$ (jam at location three only)

These words start with an empty state $q_0 = \emptyset$. Hence the prefixes of length one are all indistinguishable since the 'certain' part of all atoms are equal: $w_{..1} \sim_\delta w'_{..1} \sim_\delta w''_{..1}$. The prefixes of length two are also indistinguishable. The second states differ in their jam location, but the delay of j_2 and j_3 is greater than zero, hence for the prefixes of state one and two the monitor cannot distinguish between the words if they only differ in their final state and only in terms of the jam atoms j_2 and j_3. Thus: $w_{..2} \sim_\delta w'_{..2} \sim_\delta w''_{..2}$. However, w, w' and w'' are distinguishable from each other. The words are of length $k = 5$ and $\delta(j_2) \leq 5 - 2$ and $\delta(j_3) \leq 5 - 2$, therefore, these atoms should for indistinguishability coincide with their occurrences in $w[2]$, $w'[2]$ and $w''[2]$. But since they do not coincide it means that the words are distinguishable.

Before we formally define delay monitors we first describe intuitively what kind of process the delay monitor is a model of. A delay monitor is a tick-based process where each tick corresponds to a tick in the target system. With every target system tick the monitor observes the state of the target system given its local view. The observations that are made during that step have a delay of zero. After the instant observations, the monitor receives the delayed observations. With the (delayed) observations and the indistinguishability relation, the monitor can then construct the set of possible finite words that the target system has produced so-far. Initially this set will be equal to the set of the empty word, $\{\varepsilon\}$. At all times, the actual prefix of the word that is being revealed, and for which we want to verify an LTL property, is in the set of possible words. Hence if the LTL property is shown to be true of false for all the possible words, then it must be true (or false, respectively) for the actual revealed prefix, and by extension for the word that is being revealed. Hence, after the possible words are determined with the latest observations, the monitor checks whether or not the formula that is being monitored is true/false for all the possible words. If so, then it halts.

A delay monitor is fully specified by its delay function. Its purpose is to evaluate LTL formulas on the target system's behavior as the target system reveals more states. We therefore model a delay monitor as a function from finite words and LTL formulas to true, false and unknown. Due to the delay function it can only determine a formula φ to be true or false given a word w, if for all indistinguishable words from w the evaluation is true or false, respectively. If this is not the case, then the delay monitor cannot make a conclusive evaluation.

Definition 3.12 (Delay Monitor, m_δ): *Let $\delta : A \to \mathbb{N}^0 \cup \{\infty\}$ be a function called*

	v_1	v_2	v_3	v_5	j_2	j_3
$m_{\delta 1}$	0	1	3	∞	1	3
$m_{\delta 2}$	1	0	2	∞	0	2
$m_{\delta 3}$	3	2	0	∞	2	0
$m_{\delta 4}$	2	1	1	∞	1	1

Table 3.2: *Example delays for monitors. A combination of a monitor and atom gives the delay of that atom for the monitor, i.e. $m_{\delta 3}$ and j_2 gives 2, indicating that $\delta_3(j_2) = 2$.*

delay function. A delay monitor $m_\delta : S^* \times LTL_f \to \mathbb{B}_3$ is defined for a word $w \in S^*$ and formula $\varphi \in LTL_f$ as:

$$m_\delta(w, \varphi) = \begin{cases} t & \forall w' \in S^*, w' \sim_\delta w : [w' \models \varphi] = t \\ f & \forall w' \in S^*, w' \sim_\delta w : [w' \models \varphi] = f \\ u & otherwise \end{cases}$$

Example 3.11 (Ex. 3.4 Cont.: Delay Monitors): For our example scenario we use four delay monitors $m_{\delta 1}$, $m_{\delta 2}$, $m_{\delta 3}$, $m_{\delta 4}$, for which the delay functions are given in Table 3.2. Note that δ_1 corresponds to the delay function in Example 3.10. For an arbitrary word $w \in S^*$ we shall always have that $m_{\delta i}(w, \Diamond v_5) = u$ for each $i \in [1, 4]$, because the delay is infinite for v_5 and hence it will never be known for a word and state whether v_5 is contained in that state. From our example target system let $w = V(q_0)V(q_1)V(q_5)V(q_{13})V(q_{17})$ which is the word where the vehicle travels through location three and locations two and three are jammed. We have that $m_{\delta 2}(w_{..4}, \Diamond v_2) = t$, $i \in \{1, 2, 4\}$, because the delay of v_2 has passed in state three given $w_{..4}$ for each of these monitors. However, for monitor three it could still be hypothetically possible given $w_{..4}$ that $v_2 \notin w_{..4}[2]$ (it is already determined that v_2 is not in $w_{..4}[1]$). Hence $m_{\delta 3}(w, \Diamond v_2) = u$. For each of the monitors it holds that $m_{\delta i}(w, \Diamond v_2) = t$.

A decentralized delay monitor is a network of delay monitors that can propagate observations among themselves. The setting we assume is that as in decentralized progression monitoring [19] the network ticks at synchronous intervals that coincide with the change of states in the target system. Between each tick messages can be sent between monitors that are directly connected to each other. A delay monitor sends its received messages of the previous tick and its own observations to its connected neighbors after it has locally made its own observations. The step where a monitor receives the delayed observations, as described before Definition 3.12, can be interpreted as the moment where the delay monitor receives all messages that were sent to it. As in [125] we do not assume full connectedness, i.e. the network can have any topology.

Delays arise because it takes time to propagate observations. The delay functions of monitors therefore depend on the topology of the network. If a delay monitor observes an atom p to hold in the target system, then it sends this information to each directly connected neighbouring delay monitor in the network. If a delay

monitor receives an observation, then the next tick it also sends this information to each directly connected neighbouring delay monitor. The propagation delay between two directly connected monitors is one. If for a given delay monitor and atom the delay is zero, then it means that the delay monitor can directly observe that atom. Therefore, for a given delay monitor and atom, the delay of that atom is either zero if it is visible for the delay monitor, or the length of the shortest path to another delay monitor for which the delay of that atom is zero, or infinite if there is no such path to another delay monitor.

For decentralized progression monitors we discussed that the entire decentralized monitor is constructed to monitor a specific formula. Our framework allows for a specification where we assign separate formulas to the local monitors. However, for consistency with progression monitoring and aggregation monitoring (see the next chapter, Section 4.2) we will assign a specific local monitor to be a main monitor and assign a formula to be our main formula of interest. The decentralized monitor then becomes a function from finite words to true, false or unknown, notated as $[D^d, w]$, where D^d is the decentralized delay monitor as defined next. The output of the decentralized delay monitor given a word is the verdict of the main delay monitor for the assigned target formula given the word.

Definition 3.13 (Decentralized Delay Monitor, D^d): *A decentralized delay monitor D^d is specified by $(M, m_\delta, \varphi, E)$ where M is a set of delay monitors, $m_\delta \in M$ is a designated main delay monitor, $\varphi \in LTL_f$ is an LTL formula and $E \subseteq M \times M$ is a set of edges. Furthermore, for each $m_{\delta'} \in M$ and $p \in A$: $\delta'(p)$ equals either 0, or the length of the shortest path given E between m_δ and any $m_{\delta''} \in M$ such that $\delta''(p) = 0$, or, if no such path exists, $\delta'(p) = \infty$.*

The verdict of a decentralized delay monitor $D^d = (M, m_\delta, \varphi, E)$ given w is defined as $[D^d, w] = m_\delta(w, \varphi)$.

Example 3.12 (Ex. 3.11 Cont.: Decentralized Delay Monitor): We illustrate a decentralized delay monitor by defining one for our smart roads scenario. Let $M = \{m_{\delta 1}, m_{\delta 2}, m_{\delta 3}, m_{\delta 4}\}$ be the delay monitors from Example 3.11, let $\varphi = \Diamond v_1 \wedge \Box\neg((v_2 \wedge j_2) \vee (v_3 \wedge j_3))$ (the vehicle travels through location one and never ends up in a jam) and finally let E be the symmetric closure of $\{(m_{\delta 1}, m_{\delta 2}), (m_{\delta 2}, m_{\delta 4}), (m_{\delta 4}, m_{\delta 3})\}$. An example decentralized delay monitor is $D^d = (M, m_{\delta 1}, \varphi, E)$. Given a finite word w, $[D^d, w] = f$ if, and only if, in w it holds that (1) the vehicle travels through location one at some point and does not travel through a traffic jam and (2) if the vehicle does travel through a jam at location two then it does so at or before state $|w| - \delta_1(j_2)$ (which is $|w| - 1$) and (3) if the vehicle does travel through a jam at location three then it does so at or before state $|w| - \delta_1(j_3)$ (which is $|w| - 3$).

If a formula φ is true (or false) for a finite word w, i.e., $[w \models \varphi] = t$ (or $= f$), then the maximum delay of a decentralized delay monitor to detect this is equal to the number of local delay monitors, assuming that the set of propositional atoms is fully covered by non-infinite delays for the main monitor. Consider a decentralized delay monitor with k delay monitors. In the worst case, if an atom p holds in a given

state at moment i in a word w, i.e. $w, i \models p$, then a delay monitor discovers this after maximally k steps. Because the shortest path between any two monitors would be maximally k long. This means that given a word w of length $l > k$, the monitor can fully reconstruct the system behavior $w_{..l-k}$. I.e. for each word $w' \sim_\delta w$, we have that $w'_{..l-k} = w_{..l-k}$. So, if $w_{..l-k}$ is the shortest prefix $w_{..j}$ of word w such that $[w_{..j} \models \varphi] \neq u$, for some φ, then there is a maximum delay of k before a delay monitor, and by extension a decentralized delay monitor, determines the evaluation of φ. This is formalized by the following proposition.

Proposition 3.4: *Let $w \in S^*$ be a word and $D^d = (M, m_\delta, \varphi, E)$ be a decentralized delay monitor such that $|M| = k$ and for each $p \in A$ it holds that $\delta(p) \neq \infty$ and $[w_{..|w|-k} \models \varphi] \neq u$. $[w_{..|w|-k} \models \varphi] = t$ (resp f) iff $[D^d, w] = t$ (resp f).*

Proof. For an arbitrary $p \in A$ the shortest path between m_δ and another monitor $m_\delta{}'$ such that $\delta'(p) = 0$ is maximally k, which is the case if the monitors are linearly connected. From Definition 3.11 it follows that, given the maximum delay k and a word w' where $w \sim_\delta w'$, $w[i] = w'[i]$ for any $i \in [1, |w|-k]$. Hence $w_{..|w|-k} = w'_{..|w|-k}$. Therefore, if $[w_{..|w|-k} \models \varphi] \neq u$, then for each indistinguishable word w' it must hold that $[w'_{..|w|-k} \models \varphi] = [w_{..|w|-k} \models \varphi]$. By definition; $[D^d, w] = m_\delta(w, \varphi)$, and $m_\delta(w, \varphi) = t$ (or f) iff for each indistinguishable word from w it holds that φ is evaluated to t (or f). This is the case if $[w_{..|w|-k} \models \varphi] \neq u$. Hence, under the assumptions: $[w_{..|w|-k} \models \varphi] = t$ (resp f) iff $[D^d, w] = t$ (resp f). □

3.6 Discussion wrt. Norms

We have discussed two different approaches to monitoring system behavior. Both approaches relied on LTL and a network of local monitors. We shall now give a brief overview of how we may use these approaches for monitoring norm violations and how the different approaches differ in terms of their technical aspects. As argued earlier, we focus in this thesis on legalistic norms that can be explicitly represented. A norm is a specification of how a target system ought to behave. If the target system is a multi-agent system then typically norms represent how agents ought to behave. Given that we model system behavior as words of states, and that LTL formulas describe such behavior, we may adopt the approach of stating that any LTL formula models a norm. The violations of such norms can be detected by monitor frameworks such as the ones described in this chapter. However, to say that each LTL formula models a norm would make norms lose their conceptual meaning. In the normative systems literature we can find various formalizations of norms. In this section we will look at two common formalizations, those of the counts-as rules and conditional obligations/prohibitions, and how they correspond to temporal logic. If we translate a norm representation to LTL, then we can use our monitoring models to design monitors for applications that use those norm representations.

3.6.1 Counts-As and Sanction Rules

Some of the most basic norm-oriented programming languages rely on the use of counts-as and/or sanction rules to represent norms [46, 60]. Recall from the previous

chapter that a counts-as rule relates a state description to a violation atom, and a sanction rule relates a violation atom to a sanction. A state description given the model of this chapter is a propositional formula φ. As a temporary notation, consider v to be a violation atom and s to be a sanction atom. Given a set of counts-as rules of the form $\varphi \Rightarrow v$ and a sanction rule $v \Rightarrow s$, there are two interesting questions to consider for monitoring. For a given word, is there a moment where a counts-as rule for violation v fires? Another is, for a given word, is there a moment where a sanction rule for sanction s fires? Let w be an infinite word. If all the counts-as rules for atom v are given by $\varphi_1 \Rightarrow v, ..., \varphi_k \Rightarrow v$, and $\varphi_v = \varphi_1 \vee ... \vee \varphi_k$, then one of those rules fires somewhere in w iff $w \models \Diamond\varphi_v$. If all the sanction rules for atom s are given by $v_1 \Rightarrow s, ..., v_k \Rightarrow s$, and $\varphi_s = v_1 \vee ... \vee v_k$, then one of those rules fires somewhere in w iff $w \models \Diamond\varphi_s$. Note that the LTL formula that represents whether the target system's behavior calls for a sanction s ($\Diamond\varphi_s$) is only monitorable given the target system if it is possible for the target system to violate those norms such that s must be applied.

If we want to implement a decentralized monitor for monitoring some φ_v where A_{φ_v} are the atoms that occur in φ_v, then there are a few things to consider (similarly for some φ_s and A_{φ_s}). Programming languages that use the counts-as and sanction rule structures assume the fact that the sanction is executed immediately when the conditions of the sanction rule hold. Hence a delay in detecting whether the condition hold might be problematic. Therefore, strictly speaking, a decentralized progression or delay monitor would only be appropriate if the local view of the main monitor contains A_{φ_v}, or all delays of atoms in A_{φ_s} is zero. If a delay is not per se problematic then in a decentralized progression monitor the union of all local views should equal A_{φ_v}, and in a decentralized delay monitor the delay of atoms in A_{φ_v} should not be infinite. If these conditions are fulfilled, then the decentralized monitor has adequate observation capabilities to determine whether a norm is violated.

Conditional Obligations/Prohibitions

A common alternative to counts-as and sanction rules is the use of detachable obligations/prohibitions with deadlines (cf. [73] and [130], Chapter 4). These programming constructs consist of a precondition, deontic content and deadline, which are all state formulas. For example a conditional prohibition might be defined such that the condition is defined as 'the vehicle enters an urban area', the deontic content as 'the vehicle speed is over 50 km/h' and the deadline as 'the vehicle exits the urban area'. Such a norm becomes detached if the condition holds. A detachment can be seen as the obligation or prohibition being in effect. In the example prohibition, if the vehicle enters the urban area, then it is prohibited to exceed the maximum speed of 50 km/h. The detachment is removed if the deontic content holds or the deadline holds. In case of obligation, if the deadline holds whilst the norm is detached then this constitutes a violation. In case of prohibition, if the deontic content holds whilst the norm is detached then this constitutes a violation. For instance, if the vehicle enters an urban area and exceeds the speed limit before it exits the area, then this constitutes a violation.

We may represent such a norm as a triple $(\varphi_c, \varphi_x, \varphi_d)$ of state formulas, where φ_c is the condition, φ_x is the deontic content and φ_d is the deadline. To get a

feel of the intended semantics of such a norm we shall discuss three example words $w_1, w_2, w_3 \in S^*$. We use a notation where if a state s_i falls under the scope of a brace which is labeled with φ, then $[w, i \models \varphi] = t$. First, let us consider the word:

$$w_1 = \overbrace{s_1 ... s_k}^{\neg\varphi_c}$$

In this word the condition of the norm is never true, and hence the norm is not violated anywhere in the word. Next consider the following word:

$$w_2 = \overbrace{s_1 ... s_{i-1}}^{\neg\varphi_c} \underbrace{s_i}_{\varphi_c \wedge \neg\varphi_x \wedge \neg\varphi_d} \overbrace{s_{i+1} ... s_{k-1}}^{\neg\varphi_x \wedge \neg\varphi_d} \underbrace{s_k}_{\varphi_x \wedge \neg\varphi_d}$$

In this word the condition of the norm is true at index i. From then on, the deontic content and deadline do not hold until moment k when the deontic content holds and the deadline still does not hold. If the norm models an obligation, then the obligation is fulfilled at moment k and no violation has occurred. If the norm models a prohibition then a prohibited state (given that in the past the condition held and the deadline has not held since) held and hence the norm is violated at moment k. Lastly, consider the word:

$$w_3 = \overbrace{s_1 ... s_{i-1}}^{\neg\varphi_c} \underbrace{s_i}_{\varphi_c \wedge \neg\varphi_x \wedge \neg\varphi_d} \overbrace{s_{i+1} ... s_{k-1}}^{\neg\varphi_x \wedge \neg\varphi_d} \underbrace{s_k}_{\neg\varphi_x \wedge \varphi_d}$$

This word is similar to w_2, except that at moment k the deadline holds instead of the deontic content. This means that if the norm models an obligation that then the norm is violated at moment k because the obligation was not fulfilled on time. If the norm modeled a prohibition then the deadline passed for that prohibition and no prohibited state occurred since the condition held true. There is no violation in that case.

To capture these intended semantics we may use LTL to describe when a word contains a violation of the norm. For instance in [7] a violation occurs in a word w at index i given a norm $n = (\varphi_c, \varphi_x, \varphi_d)$ if either

$$w, i \models \varphi_d \wedge \neg\varphi_x \wedge ((\mathcal{Y}(\neg\varphi_x \wedge \neg\varphi_d)\mathcal{S}(\varphi_c \wedge \neg\varphi_x \wedge \neg\varphi_d)) \vee \varphi_c)$$

in case the norm is a conditional obligation, or

$$w, i \models \varphi_x \wedge \neg\varphi_d \wedge ((\mathcal{Y}(\neg\varphi_x \wedge \neg\varphi_d)\mathcal{S}(\varphi_c \wedge \neg\varphi_x \wedge \neg\varphi_d)) \vee \varphi_c)$$

in case the norm is a conditional prohibition.

However, these formulas do not help us in this chapter as we considered only formulas from LTL_f, which do not contain $\mathcal{Y}$ and $\mathcal{S}$ operators. We observe that if there is an index in a word such that the above conditions hold, then for obligations it must the case that given the word and the initial state, somewhere in the future the condition holds, and from then on the deadline held before the deontic content held [128]. This is captured by the formula $\varphi_n = \Diamond(\varphi_c \wedge \neg(\neg\varphi_d \mathcal{U} \varphi_x))$. For prohibitions the same holds, except that the deontic content should not hold before the deadline holds. This is captured by the formula $\varphi_n = \Diamond(\varphi_c \wedge \neg(\neg\varphi_x \mathcal{U} \varphi_d))$. We note that φ_n is only monitorable for a given target system if the target system may violate the norm.

Consider we have some norm n and φ_n is either the formula for obligation or prohibition. If we use a decentralized progression monitor, then we only have to make sure that the local views of the progression monitors cover the atoms that occur in φ_n. Similarly for delay monitors; the main monitor has to have a non-infinite delay for each of the atoms in φ_n.

3.7 Conclusion

In this chapter we investigated models for decentralized runtime monitoring. As a basis for system monitoring we used linear temporal logic (LTL [107]). The task of a monitor is to determine whether the behavior of a target system, modeled by a word of states, satisfies some LTL formula. In open multi-agent systems we often do not have a model available of the target system. In such cases we must deploy runtime monitors. These monitors can only observe finite prefixes of the possibly infinite behavior of a target system. We discussed the foundations of LTL monitoring processes. There we concluded that it is not always possible to determine whether a process' behavior satisfies some LTL formula, if the behavior is infinite. We then discussed two approaches, progression and delay monitoring, for decentralized monitoring. These models relied on a network of local monitors where each local monitor has a local view on the target system. The differences between the monitor approaches were mainly the connectiveness and the type of information that they share (progressed formulas for progression monitoring or observations for delay monitoring). We finally linked the theory back to our original goal; how we can decentrally detect norm violations at runtime. For this we modeled norms also as LTL formulas and suggested the use of the aforementioned decentralized LTL monitors to detect norm violations.

4 Robustness and Security for Monitoring

We continue our work on runtime norm enforcement by expanding upon the robustness and security aspects of decentralized monitoring. A decentralized monitor consists of multiple local monitors that work together. This is very similar to sensor network technology where robustness and security have been investigated thoroughly. The first contribution of this chapter is a proposal for a decentralized monitoring framework that is inspired by sensor networks and allows us to highlight different robustness and security challenges. We compare this framework to the decentralized progression and delay monitors of the previous chapter. The second contribution is the investigation of robustness and security. The main conclusions of this chapter are that we cannot always have optimal robustness and security by the design of the decentralized monitor topology, but we can identify critical local monitors that require extra attention.

4.1 Introduction

A major concern of many decentralized runtime monitoring applications is their robustness and security. The data that is gathered from a multi-agent system can severely compromise the agents' privacy if leaked. Adversaries can also try to take down parts of the networks to impede its functioning. Formal models of decentralized monitors can identify the critical parts in decentralized monitors in terms of robustness and security. This helps in their design and implementation as extra resources can be invested in critical parts. In this chapter we present an abstract model for decentralized runtime monitors and investigate the properties of robustness and security issues that have to be faced when designing a decentralized monitor. Smart roads applications also face many challenges, including physical attacks on the monitor infrastructure and the privacy of individuals (cf. [72]). Our main contribution is a formal framework for specifying decentralized monitors with which we analyze how critical local monitors are with respect to the security and robustness of the decentralized monitor. We formally evaluate the framework. It is our belief that the presented framework is not only beneficial to the design and development of decentralized runtime monitors for LTL verification, but can also provide insight on design-time and/or runtime analysis of robustness and security risks for other decentralized verification technologies.

We draw inspiration from wireless sensor networks (WSNs). WSN research and practice contain a vast number of identified robustness and security risks as well as countermeasures (cf. [105, 104]). Example risks include the malfunctioning of hardware and software and attempts by an adversary to eavesdrop on communication. Countermeasures include various techniques such as routing protocols and encryption. The aim of countermeasures is to keep the monitoring service online if local monitors malfunction and prevent sensitive information from being obtained by an adversary. The requirements of WSNs are commonly organized by: 1) data confidentiality (only intended receivers can see sensitive data), 2) data integrity and freshness (data is correct and new), 3) availability (continued operation of monitors). The structure of a WSN is that a collection of information gathering nodes routes sensor data towards a sink, which acts as the central data gathering point. Intermediate data aggregation is often used to increase security and save energy. The decentralized runtime monitoring method of this chapter also makes use of intermediate aggregation by the local monitors. Hence we call such local monitors aggregation monitors, and a network of such monitors a decentralized aggregation monitor. Each local monitor has observation capabilities of its own and can query other local monitors to verify other properties. This is similar to the decentralized progression and delay frameworks proposed in [19] and [128] (see Chapter 3). For decentralized aggregation monitors we assume that information flows through the network without delay. Also, instead of sharing observations as in [128] or passing around progressed formulas as in [19], local monitors in the framework of this chapter combine their input into a Boolean evaluation and share that with whichever other local monitors need it. This represents the aggregation of input. No particular topology is assumed, other than that there is no cyclic communication.

Protection against confidentiality and availability attacks will be the main focus in this chapter. Data integrity and freshness is assumed. We assume that monitors either work correctly or they are unavailable, but cannot for instance send false information. Different kinds of attacks can be categorized between attacks that change the network topology (e.g. physically compromising a node or communication line, or a wormhole attack that connects two nodes) and those that extract information from the network (capture and/or imitation of nodes). We shall address the case that a local monitor can malfunction. This encapsulates both aggressive and non-aggressive failures of local monitors. We shall also discuss the case where an attacker can query a local monitor without proper authorization.

We note that in WSN literature privacy is often intended as the privacy of which sensor provided what contribution to the aggregated information. For instance, if a sensor computes the average velocity of all vehicles, then it may receive from other sensors the average velocities of partitions of the set of vehicles. Privacy in related literature would mean that it is not known which sensors provided what data to compute the total average velocity. In this chapter we consider only privacy in the form of the privacy of agents. For instance, in our example scenario from the previous chapter (Example 3.1, which we will also use in this chapter) the location of a specific vehicle can be seen as privacy sensitive information. Therefore, instead of a monitor sharing separately the velocity of the vehicle and whether location two is jammed, it may share the evaluation of the conjunction of these two facts. If a receiver obtains

this evaluation and it is false, then the receiver cannot derive whether the vehicle was not at location two or whether there was a traffic jam. Note however that if the evaluation is true, that then it is known that the vehicle was at location two.

Security related papers on wireless networks for monitoring tend to focus on how to prevent security risks by using cryptography (e.g. [106]) and/or special routing protocols (e.g. [78]). Our approach is complementary to this. We do not look at runtime implementation techniques for preventing risks, but address design time questions and analysis to see where potential robustness and security risks lie. We believe design based analysis can help in further improving decentralized monitors. Depending on the practical limitations of an application it might not be possible to always make a perfect design. But our work can help in determining which parts of a network require more advanced/expensive hardware to increase safety.

The rest of this chapter is structured as follows. In Section 4.2 we provide our aggregation based runtime monitoring model. In Section 4.3 we discuss robustness and in Section 4.4 we discuss security. Finally, we conclude this chapter in Section 4.5.

4.2 Monitoring with Aggregation

In this section we present the framework that we earlier published in [125] and [124]. We refer to this method as monitoring with aggregation, which is an important concept to the framework. Again, a decentralized monitor consists of multiple local monitors. These local monitors are referred to as aggregation monitors. For the structure of decentralized aggregation monitors we draw inspiration from decentralized monitoring techniques such as wireless sensor networks (WSNs)[86].

Like sensor nodes in WSNs, aggregation monitors in our model have a local view. This view allows a monitor to evaluate a local property. As in the previous chapter we aim at verifying future LTL formulas[1]. The observation capabilities of an aggregation monitor are captured by a future LTL formula. The intuition is that the monitor can directly observe the three-valued Boolean evaluation of the formula for each finite word of the target system. This differs from the progression and delay monitors where local views were modeled by a set of observable propositional atoms. There is also communication among aggregation monitors, just like the other frameworks. An important difference to the frameworks of the previous chapter is the kind of information that is communicated. Progression monitors in decentralized progression monitors share progressed formulas and delay monitors share observations. Aggregation monitors share property evaluations, which are constructed from the evaluation of the locally observable property and the properties that are received. As communication is hierarchical we can view the recipient of a message to be the parent of the monitor that sent the message. The parent monitor collects from all its children the communicated evaluations and evaluates its own observable property. The list of evaluations from its children and its own observation property are aggregated into a single three-valued Boolean using an aggregation function and is shared with its parent. As a consequence, it is possible for an aggregation monitor to generate its output without knowing what its children are monitoring.

[1]For an explanation of LTL see Section 3.2.

a	$\neg a$
f	t
u	u
t	f

$a \wedge b$		b		
		f	u	t
a	f	f	f	f
	u	f	u	u
	t	f	u	t

$a \vee b$		b		
		f	u	t
a	f	f	u	t
	u	u	u	t
	t	t	t	t

Figure 4.1: *Truth definition of Kleene logic.*

4.2.1 Monitor Model

As technical preliminaries we continue from the previous chapter. Hence, we assume a given global set of propositional atoms A and set of environment states $S = 2^A$. Before we define aggregation monitors, we first discuss their capability of aggregating (ternary) evaluations of formulae into a single evaluation. A monitor aggregates evaluations using an aggregation function. The definition of an aggregation function depends on aggregation formulas, which are propositional formulas that we evaluate using Kleene's ternary semantics [81]. One can see these formulas as a variation upon classic two-valued Boolean propositional logic. The difference with Kleene's semantics is that a variable can take three values (true, false, unknown) and hence the evaluation of a formula has to be adapted accordingly. The reason that we use Kleene's semantics is that it captures the intuition of a monitor that makes a verdict about a Boolean combination of unknown properties. For instance, if a monitor wants to output the disjunction of two input values, and one of them is false and the other is unknown, then it outputs unknown since somewhere in the future the unknown input value may switch to either false or true.

Definition 4.1 (Aggregation Formula, α): *An aggregation formula with k variables is a propositional formula α with k propositional symbols $x_1, \ldots, x_k$ which can take on the values in $\{t, f, u\}$. Aggregation formulas are evaluated using Kleene's ternary semantics shown in Figure 4.1. Given truth values $v_1, \ldots, v_k \in \{t, f, u\}$ we write $\alpha(v_1, \ldots v_k)$ to refer to the evaluation of α if truth value v_i is assigned to variable x_i.*

> **Example 4.1 (Ex. 3.5 Cont.: Aggregation Formula):** We continue the example scenario from the previous chapter. Consider two variables a and b and the formula $\alpha = \neg a \vee b$. If a is set to t then α can only be evaluated to t if b is also set to t, just like classic propositional logic with two-valued Booleans. If a to u, then α will be evaluated to u unless b is set to t, in which case α is evaluated to t. We can see that the formula α specifies a function from the Boolean values of the variables a and b to a new Boolean value (the evaluation of α given Kleene's ternary semantics). We use this observation to define aggregation functions.

An aggregation function takes an arbitrary number of input variables over the domain $\{t, f, u\}$ and returns a new value from $\{t, f, u\}$. However, we do not allow arbitrary aggregation functions. The application of aggregation is to combine the

evaluation of any number of k LTL formulas $\varphi_1, ..., \varphi_k$ given some finite word into the evaluation of another LTL formula that is a Boolean combination of the formulas $\varphi_1 ... \varphi_k$. We achieve this intuition by defining that the aggregation function output given k input variables corresponds to the evaluation of an aggregation formula over maximally k variables.

Definition 4.2 (Aggregation Function, a): *An aggregation function* $\mathsf{a} : \bigcup_{k \in \mathbb{N}^+} \mathbb{B}_3^k \to \mathbb{B}_3$ *returns given x three valued Boolean variables, $x \in \mathbb{N}^+$, a three valued Boolean such that $\mathsf{a}(v_1, ..., v_x) = \alpha(v_1, ..., v_x)$ for some aggregation formula α with x variables.*

> **Example 4.2 (Ex. 4.1 Cont.: Aggregation Function):** Consider an aggregation function that for any input values returns the evaluation of their conjunction, or the input value itself if there is only one input variable. Hence if a is such a function then for instance $\mathsf{a}(t, t) = t$, $\mathsf{a}(t, f) = f$ and $\mathsf{a}(t, u) = u$ (the other cases can be seen in Figure 4.1 in the middle table). Note that it is impossible to write another aggregation function a' such that for instance $\mathsf{a}'(t) = u$, because there is no aggregation formula α with a single variable where $\alpha(t) = u$. Also, there is no aggregation function such that $\mathsf{a}''(u) = t$. A formula such as $\alpha' = a \vee \neg a$ still evaluates to u ($\mathsf{a}''(u) = u$) if $a = u$.

Like progression and delay monitors, aggregation monitors also model tick-based systems where each tick corresponds with a tick in the target system. An aggregation monitor maintains the revealed states of the target system. If a new state is revealed then this state is appended to the previous revealed states and the observable property of the monitor is evaluated upon the resulting sequence. In an implementation of the monitor we do not need to explicitly store the system states but may use a much more efficient program if possible, such as a Büchi automaton. Next, the monitor remains idle until it has received input (three-valued Booleans) from all its children. Due to an acyclic topology in decentralized aggregation monitors (defined later in Definition 4.4) there will always be aggregation monitors without children. Finally the aggregation monitor sends to its parents the result of applying an aggregation function upon the value of its observable property and the inputs, when all the inputs are received.

An aggregation monitor is modeled by its observation and aggregation capabilities. The first is modeled by a formula that the monitor can locally observe, and the second is modeled by an aggregation function.

Definition 4.3 (Aggregation Monitor, m): *An aggregation monitor m is specified by (φ, a), where $\varphi \in LTL_f$ is called m's observable property and a is an aggregation function.*

> **Example 4.3 (Ex. 4.2 Cont.: Aggregation Monitor):** Consider the four monitors m_1, m_2, m_3 and m_4, where $m_i = (\varphi_i, \mathsf{a}_i)$. We assume that the first monitor can observe whether at some point the vehicle passes location one, i.e. $\varphi_1 = \Diamond v_1$. Monitor two can observe whether the vehicle ever gets stuck in jam at location two, hence $\varphi_2 = \Box(\neg v_2 \vee \neg j_2)$. Monitor three can observe the same for

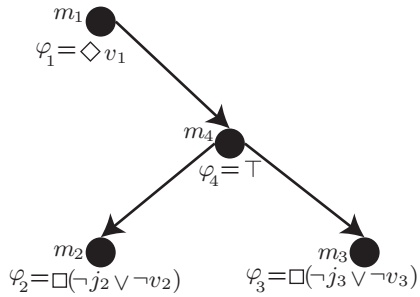

Figure 4.2: *Example decentralized aggregation monitor. Below monitors are their observation formulas and arrows represent the query function.*

location three, hence $\varphi_3 = \Box(\neg v_3 \vee \neg j_3)$. Finally, monitor four does not observe anything, which is modeled by $\varphi_4 = \top$. We use the same aggregation function for each monitor, which is a from Example 4.2, which returns the evaluation of the conjunction of the arguments. Note that even though each monitor has an aggregation function that can aggregate input from other monitors, we have not specified which monitor provides input to what other monitor. This is part of the decentralized aggregation monitor specification. As we will see, for each word that represents revealed behavior, the arguments of the aggregation function for each monitor will be the evaluation of its local observable property based on the revealed behavior and the input that it received from other monitors.

A decentralized aggregation monitor is specified by a set of aggregation monitors and an acyclic query function. The set of aggregation monitors are the components of the decentralized aggregation monitor. The query function specifies given an aggregation monitor which other monitors are connected to this monitor. The query function returns given a local monitor an ordered tuple of monitors, since they represent a Boolean variable that matches an argument of the aggregation function of the local monitor. The main reason for acyclicity of the query relation is to avoid unnecessary complexities. Note that unlike decentralized progression monitors not each aggregation monitor is dedicated to verifying the same global property. Also, there is not full connectedness between aggregation monitors.

Given the specification of a decentralized monitor we can interpret the decentralized aggregation monitor as a function that given an aggregation monitor and a finite word returns the output of that monitor. This output is determined by applying the monitor's aggregation function over the evaluation of its observable property and all the input from other aggregation monitors. Note that due to acyclicity there must be aggregation monitors without connected peers. Hence we can recursively define the output of an aggregation monitor within a decentralized aggregation monitor.

Definition 4.4 (Decentralized Aggregation Monitor, D^a): *A decentralized aggregation monitor D^a is specified by (M, Q), where M is a set of aggregation monitors and $Q : M \to \bigcup_{i=0}^{|M|} M^i$ is a function called the query function. For an aggregation monitor $m = (\varphi, \mathsf{a}) \in M$ and word $w \in S^*$ we use $D^a(m, w)$ as shorthand for $\mathsf{a}(v_0, ..., v_k)$ where $v_0 = [w \models \varphi]$, and $v_1, ..., v_k = D^a(m_1, w), ..., D^a(m_k, w))$, such*

that $Q(m) = (m_1, ..., m_k)$.

Furthermore, there is no sequence of monitors $m_1 ... m_k \in M^k$, $k \in [1, |M|]$, *such that:* $m_1 = m_k$ *and for each* $i \in [1, k-1]$ *it holds that* $Q(m_i)$ *contains* m_{i+1}.

Example 4.4 (Ex. 4.3 Cont.: Decentralized Aggregation Monitor): As an example decentralized aggregation monitor consider $D^a = (M, Q)$, which is depicted is Figure 4.2. The set of monitors $M = \{m_1, m_2, m_3, m_4\}$ are the aggregation monitors from Example 4.3. In our example monitor we allow monitor four to query monitors two and three, and monitor one can query monitor four. The query function is specified by $Q(m_1) = (m_4)$, $Q(m_4) = (m_2, m_3)$ and for m_2 and m_3 the query function returns the empty tuple.

All aggregation functions are Boolean functions over Boolean input, and these inputs are ultimately the evaluation of LTL formulas on a given word. Hence, a monitor aims to verify some Boolean combination of observable formulas. We call this Boolean combination the aggregate of an aggregation monitor. Consequently, if one aggregation monitor is directly connected to another aggregation monitor, then it means that the first monitor can query the evaluation of the aggregate of the second monitor. Given a decentralized aggregation monitor all the aggregates can be determined before runtime, because for each aggregation monitor the aggregation function is fixed. Also note that due to acyclicity there are aggregation monitors m without neighbors and hence for such aggregation monitors their aggregate is equivalent to φ, $\neg\varphi$, $\varphi \wedge \neg\varphi$ or $\varphi \vee \neg\varphi$. Finally, if in a decentralized aggregation monitor the query relation, an aggregation function or a locally observable property is updated, then this may effect the aggregate of monitors that were not directly involved in the update.

Definition 4.5 (Aggregate): *Let* $D^a = (M, Q)$ *be a decentralized aggregation monitor and* $m = (\varphi, \mathsf{a}) \in M$ *be an aggregation monitor. We say a formula* $\varphi' \in LTL_f$ *is an aggregate of* m *iff it can be constructed by the following procedure:*

1. *If* a *is a formula over* k *variables, then let* α *be an aggregation formula such that* $\mathsf{a}(v_1, ..., v_k) = \alpha(v_1, ..., v_k)$.
2. *Every occurrence of* v_1 *in* α *is replaced with* φ.
3. *If* $Q(m)$ *is non-empty then every occurrence of* v_i, $i \in [2, k]$, *is replaced by an aggregate of* m_{i-1}, *where* $Q(m) = (m_1, ...m_{k-1})$

Example 4.5 (Ex. 4.4 Cont.: Aggregate): The aggregates of monitors two and three in our example are φ_2 and φ_3. These are aggregated by monitor four through conjunction, which makes monitor four's aggregate equal $\varphi_2 \wedge \varphi_3$. Finally, monitor one aggregates φ_1 together with the aggregate of monitor four. This means that the aggregate of monitor one is equal to $\Diamond v_1 \wedge \Box(\neg v_2 \vee \neg j_2) \wedge \Box(\neg v_3 \vee \neg j_3)$.

A monitor can have multiple syntactically different aggregates because an aggregation formula such as $a \wedge b$ is equivalent to $b \wedge a$. However, these aggregates are

necessarily all semantically equal. Therefore we shall refer to aggregates of monitors throughout this section as 'the aggregate of m' when we talk about a specific aggregation monitor m.

We observe a constraint on the aggregation function of a monitor if we know its aggregate. Assume that for some aggregation monitor $m = (\varphi, \mathsf{a})$ in a decentralized aggregation monitor $D^a = (M, Q)$ its aggregate is $\psi = p\mathcal{U}q$. Assume further that $Q(m) = (m')$ where $m' = (\varphi', \mathsf{a}')$. We cannot rewrite ψ as a Boolean combination of different LTL formulas. Therefore, either $\psi = \varphi$ or ψ equals the aggregate of m', or the negation of either. In general it holds that if the aggregate of a monitor cannot be rewritten as a conjunction/disjunction of separate formulas, then the input variables of the aggregation function for that monitor are either ignored, negated and/or equal to the other inputs of the function. We shall see in Section 4.3 that having monitors with the same aggregates can in fact be useful for robustness purposes.

4.2.2 Aggregation Based Monitoring and Norms

Ultimately we aim at monitoring for norm violations. Recall that aggregates are always a Boolean combination of LTL formulas. Hence a formula $\Diamond(\varphi_1 \wedge \varphi_2)$ cannot be decentrally monitored by two or more aggregation monitors if φ_1 and φ_2 are propositional formulas. A formula $\Diamond(\varphi_1 \vee \varphi_2)$ can be decentrally monitored by two or more aggregation monitors. For instance, one monitor may monitor $\Diamond\varphi_1$ and another $\Diamond\varphi_2$ and a third aggregates the result with disjunction (because $\Diamond(\varphi_1 \vee \varphi_2) \equiv \Diamond\varphi_1 \vee \Diamond\varphi_2$). The counts-as rule representation of norms (Section 3.6.1) models a norm as one or more counts-as rules of the form $\varphi_1 \Rightarrow v, ..., \varphi_k \Rightarrow v$, where φ_i is a propositional formula and v is a violation atom. If $\varphi_1 \Rightarrow v, ..., \varphi_k \Rightarrow v$ are counts-as rules then the formula $\varphi_v = \bigvee_{i\in[0,k]} \varphi_i$ is a propositional formula that represents that a violation has occurred in a state. The LTL formula $\Diamond\varphi_v$ represents that a violation occurs at runtime and can be decentrally monitored by an aggregation monitor where potentially for each formula φ_i in φ_v there is an aggregation monitor where $\Diamond\varphi_i$ is that monitor's observation formula. Norm violations are sanctioned through sanction rules. Recall from Chapter 2 that sanction rules have the form $v \Rightarrow \psi$, where v is a violation atom and ψ is a conjunction of literals that is to be made true when the violation has occurred. For decoupled enforcement in a decentralized setting it is possible to for instance create a monitor for every violation atom. Such a monitor would observe the violation atom to be true if it is communicated as such through the institutional communication relation that we used in the model of Chapter 2.

Monitoring conditional obligations/prohibitions with deadlines is also different for aggregation monitors than for progression and delay monitor. Recall from Section 3.6.1 that we may represent a conditional obligation/prohibition as a triple $(\varphi_c, \varphi_x, \varphi_d)$ of propositional state formulas, where φ_c is the condition, φ_x is the deontic content and φ_d is the deadline. We also noted that the property whether a violation occurs in the target system of a conditional obligation can be captured by the LTL formula $\varphi_n = \Diamond(\varphi_c \wedge \neg(\neg\varphi_d \mathcal{U} \varphi_x))$. In case of a conditional prohibition we have that $\varphi_n = \Diamond(\varphi_c \wedge \neg(\neg\varphi_x \mathcal{U} \varphi_d))$. In either case, for decentralized aggregation monitors we cannot distribute φ_n over multiple aggregation monitors. Hence, the only way in which decentralized monitoring may occur is if we have multiple norms

Framework	Topology	Max. Delay	Local View	Message Content	Messages per Tick
progression	fully connected	k	set of atoms	formula	k
aggregation	directed acyclic	0	formula	boolean value	$(k^2-k)/2$
delay	any topology	k	0-delay atoms	sets of formulas	k^2-k

Table 4.1: *Main differences per approach, see Section 4.2.3. k is the number of local monitors in a given decentralized monitor.*

and we assign an aggregation monitor per norm. That way we can monitor decentrally whether a norm violation has occurred for any norm, rather than monitoring decentrally whether a specific norm is violated.

4.2.3 Comparison to Progression/Delay Monitoring

We now turn to the more technical differences between the different monitoring approaches. The overview is summarized in Table 4.1.

Topology The network of local monitors in a decentralized progression monitor is fully connected. Each progression monitor can send a message to any other progression monitor. In a decentralized aggregation monitor the topology is any acyclic directed graph. In a decentralized delay monitor the topology is any undirected graph.

Maximum Delay Let T be a target system and φ be a formula that is strictly monitorable with respect to T and ε. We know that a decentralized monitor that evaluates φ will eventually, given the infinite behavior of the target system and a local monitor, output whether the property holds or not. A decentralized aggregation monitor D^a does this with a maximum delay of zero, meaning that given the system's behavior $w \in S^\omega$, we have that $D^a(m, w') \neq u$ for the shortest prefix w' of w such that $[w' \models \varphi] \neq u$ and an aggregation monitor m such that φ is the aggregate of m. For the decentralized progression and delay monitors this delay is maximally k, where k is the number of local monitors. In the worst case scenario for a decentralized progression or delay monitor D there is another prefix $w'' \preceq w$ of length $|w'| + k$ such that given w'' the conclusion is conclusive but for each prefix it is inconclusive.

Local View The local view of monitors in a decentralized progression is modeled by a set of atoms. In a decentralized aggregation monitor the local views are modeled by the 'observable' formulas of aggregation monitors. In a decentralized delay monitor the locally observable atoms are represented by a delay of zero in the delay function of a delay monitor.

Message Content The communication between progression monitors consists of progressed formulas. This formula may grow indefinitely if for instance one occurring atom in the formula cannot be observed by any of the local monitors. The communication between aggregation monitors consists of Boolean values. The size of communication messages is hence always the same (and small). The communication between delay monitors consists of sets of formulas of the form p or $\mathcal{X}^k p$. Strictly speaking this set may grow indefinitely, as past observations might be circulated through a cycle of communication steps. However, from the results we know that an observations of more than k steps ago, where k is the number of delay monitors, can

be ignored. Hence the maximum size of the set would be $k \cdot |A|$.

Messages per Tick At any tick, for any monitoring approach, a monitor may receive up to k messages, where k is the number of local monitors. However, for decentralized progression monitors the number of messages that is allowed to be send away per progression monitor is exactly one. Hence per tick, a decentralized progression monitor requires a maximum of k messages. For decentralized aggregation monitors the maximum number of messages per tick is the number of edges in the graph. Given that the graph must be directed and acyclic, this is maximally $(k^2-k)/2$. For a decentralized delay monitor each edge gives rise to two messages, and the graph need not be acyclic. Hence in a decentralized delay monitor the maximum number of messages per tick is $k^2 - k$.

4.3 Robustness

Suppose we are designing a decentralized runtime monitor that we want to deploy in practice. For various reasons, among which are physical sabotaging attacks, local monitors and/or communication links between them can malfunction. This is quite common in for instance wireless sensor networks where the hardware tends to be cheap, or in traffic monitoring where people physically sabotage equipment. From a system designer's perspective it can then make sense to construct a decentralized monitor with some redundancy such that the malfunction of some local monitors will not impede the functioning of the decentralized monitor. In this section we provide some simple metrics to analyze robustness given a model of a decentralized aggregation monitor. We do note that because we discuss theoretical models that the metrics only cover very basic notions of robustness (mainly redundancy). Issues such as robust communication protocols are not covered.

4.3.1 Reconfiguration

In [125] we defined an update over decentralized aggregation monitors for when a set of local monitors fails. In our original proposal it was not possible for aggregation monitors to query different other monitors when an aggregation monitor failed. In this chapter we take a more generalized approach where this is possible by defining the concept of reconfiguration. We model reconfiguration as a function that given a decentralized aggregation monitor and a set of failing monitors returns a new decentralized aggregation monitor where the failing monitors are removed and the query function and aggregation formulas are updated. The observation formulas do not depend on other aggregation monitors and hence they do not change. For any particular application a reconfiguration function is given by the system designer.

Definition 4.6 (Reconfiguration Function, $\mathcal{R}$): *Let $D^a = (M, Q)$ be a decentralized aggregation monitor. A reconfiguration function $\mathcal{R}$ for D^a returns given a set of aggregation monitors $M' \subseteq M$ a new decentralized aggregation monitor $D^{a\prime} = (M'', Q')$, such that for each $(\varphi, \mathsf{a}) \in M \setminus M'$ there is a corresponding $(\varphi, \mathsf{a}') \in M''$.*

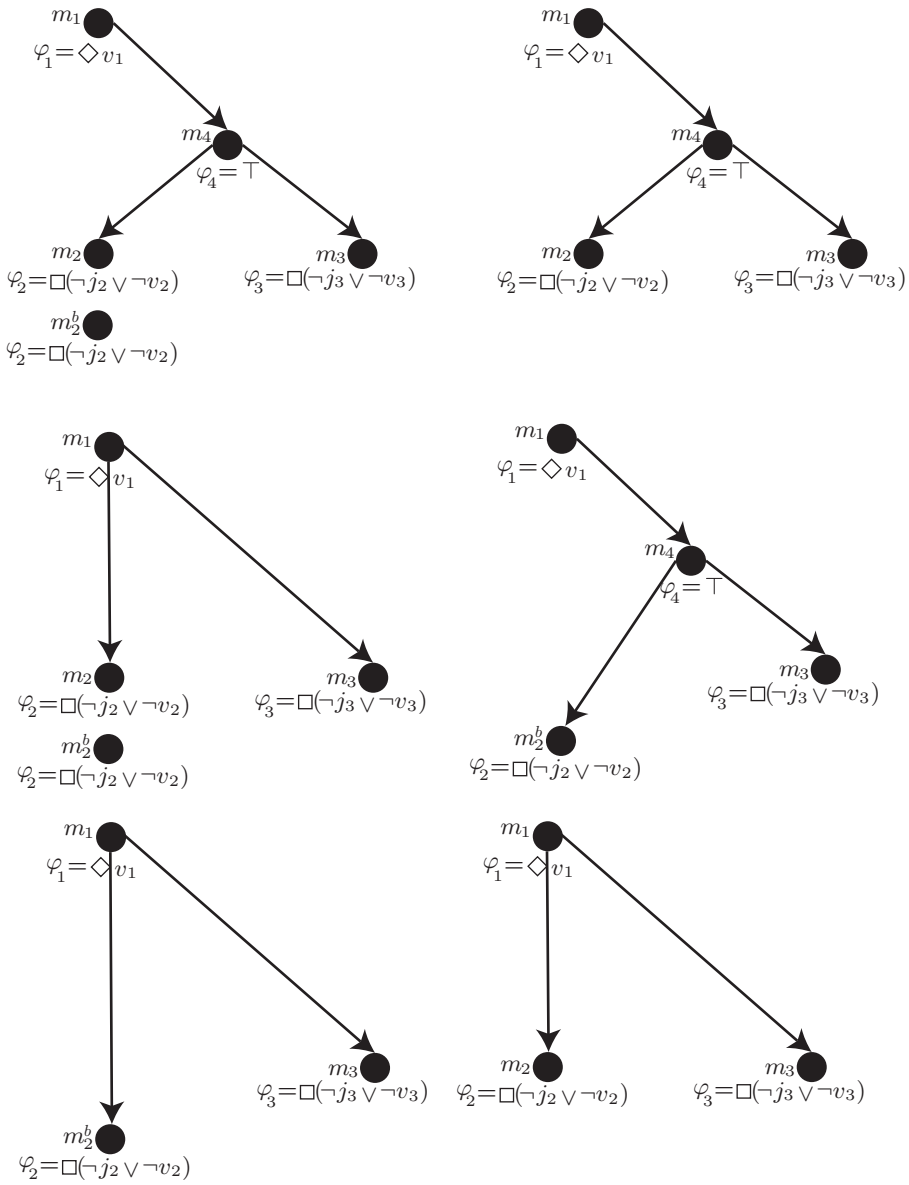

Figure 4.3: *Example reconfigurations, see Example 4.6.*

Example 4.6 (Ex. 4.5 Cont.: Reconfiguration): Assume we expand our example scenario with a backup monitor for m_2, denoted with m_2^b (Figure 4.3, top-left). Further assume that monitors m_2, m_2^b and m_3 may switch to long range communication in order to communicate to aggregation monitor m_1. Figure 4.3 depicts the reconfigurations by our example reconfiguration function $\mathcal{R}$ in case m_2 (middle-right), m_2^b (top-right), m_4 (middle-left), the set $\{m_2, m_4\}$ (bottom-left) or the set $\{m_2^b, m_4\}$ (bottom-right) fails. The query functions are given by the outgoing edges and in the order from left to right. For example, in the reconfiguration of $\mathcal{R}(\{m_4\}) = (\{m_1', m_2^{b\prime}, m_2', m_3'\}, Q')$ we have that $Q'(m_1') = (m_2', m_3')$ and for the other aggregation monitors the query function returns the empty tuple. Furthermore, each reconfigured aggregation monitor has a new aggregation function such that its aggregate remains the same.

For the other sets of failing monitors, the reconfiguration consists of the remaining monitors after the failure of the aggregation monitors in F, and the query relation returns the empty tuple for every aggregation monitor. The aggregates of the remaining monitors are then equal to their observation formula.

Alongside the reconfiguration function we require a definition to determine whether a decentralized aggregation monitor is 'broken' after a failure. As in [125] we define a concept called expressiveness constraint which is a relation between aggregation monitors and LTL formulas. The intuition is that the aggregate of a monitor must be similar to the LTL formula to which it is paired. Similarity among aggregates is conditioned on the monitored language (see Definition 3.6) of the target system. Given the monitored language, if all evaluations of two formulas are equal, then they are similar. As a short hand we use the following notation $[w \models \varphi]_{\mathcal{L}_T}$ for the evaluation of φ on w given a monitored language $\mathcal{L}_T$. The following definition specifies this shorthand.

Definition 4.7 (Target System Entailment): *Let T be a target system, and $\mathcal{L}_T$ be its monitored language. For a word $w \in \mathcal{L}_T \cap S^*$ and LTL formula $\varphi \in LTL_f$:*

$$[w \models \varphi]_{\mathcal{L}_T} = \begin{cases} t & \text{if } \forall w' \in \{w'' \in S^\omega \cap \mathcal{L}_T | w \preceq w''\} : w'' \models \varphi \\ f & \text{if } \forall w' \in \{w'' \in S^\omega \cap \mathcal{L}_T | w \preceq w''\} : w'' \not\models \varphi \\ u & \text{otherwise} \end{cases}$$

The expressiveness constraint of a decentralized aggregation monitor is given by pairs of aggregation monitors with formulas. An aggregation monitor works correctly if its aggregate given the decentralized monitor equals (using target system entailment) the formulas to which the monitor is paired. The idea is that a local aggregation monitor might be monitoring the formula φ which is true if and only if some norm is violated. So long as the evaluation of the aggregate of the monitor is equal to that of φ (using target system entailment) we may consider the monitor to work correctly. This would be represented in the expressiveness constraint by pairing the monitor with φ. Hence, if for each monitor-formula pair in the expressiveness constraint the monitor's aggregate evaluations equal that of the formula, then we say that the decentralized monitor satisfies its expressiveness constraint.

Definition 4.8 (Expressiveness Constraint, E): *Let $D^a = (M, Q)$ be a decentralized aggregation monitor. An expressiveness constraint $E \subseteq M \times LTL_f$ for D^a pairs aggregation monitors to LTL formulas. For a target system T and its monitored language $\mathcal{L}_T$ we say that D^a T-satisfies E iff for each finite word $w \in \mathcal{L}_T \cap S^*$, $(m, \varphi) \in E$ and aggregate φ' of m it holds that $[w \models \varphi]_{\mathcal{L}_T} = [w \models \varphi']_{\mathcal{L}_T}$.*

> **Example 4.7 (Ex. 4.6 Cont.: Expressiveness Constraint):** For our example we define the expressiveness constraint to be (m_1, φ), where $\varphi = \Diamond v_1 \wedge \Box(\neg v_2 \vee \neg j_2) \wedge \Box(\neg v_3 \vee \neg j_3)$. For our example decentralized aggregation monitor D^a this constraint is satisfied as this formula is equal to the aggregate of m_1 given D^a.

4.3.2 Robustness Metrics

For the rest of this subsection we assume, unless specified otherwise, a given target system T, a decentralized aggregation monitor $D^a = (M, Q)$ a reconfiguration function $\mathcal{R}$ for D^a and an expressiveness constraint E for D^a such that D^a T-satisfies E. By slight abuse of notation we say given a set of failing monitors $F \subseteq M$ that $\mathcal{R}(F) = (M', Q')$ T-satisfies E if each occurrence of an aggregation monitor in M would be replaced in E by its corresponding aggregation monitor in $\mathcal{R}(F)$ and $\mathcal{R}$ T-satisfies the updated version of E.

We aim at quantifying robustness in terms of how much damage a decentralized aggregation monitor can take before its expressiveness constraint is not satisfied anymore. This damage can be expressed as set of failing aggregation monitors. We consider aggregation monitors that do not occur in an expressiveness constraint to be supporting monitors. F-robustness for a set of aggregation monitors F captures whether the decentralized aggregation monitor can withstand (i.e. still satisfy its expressiveness constraint) if the monitors in F fail.

Definition 4.9 (F-robustness): *Let $F \subseteq M \backslash \{m | (m, \varphi) \in E\}$ be a set of aggregation monitors. We say that D^a is F-robust with respect to T, $\mathcal{R}$ and E iff $\mathcal{R}(F)$ T-satisfies E.*

> **Example 4.8 (Ex. 4.7 Cont.: F-robustness):** Our example decentralized aggregation monitor is F-robust with respect to our example T, $\mathcal{R}$ and E, where $F = \{m_2\}$, $F = \{m_2^b\}$, $F = \{m_4\}$, $F = \{m_2, m_4\}$ or $F = \{m_2^b, m_4\}$. For instance in the case of $F = \{m_2, m_4\}$ (monitors two and four fail) we have that due to $\mathcal{R}$ the decentralized aggregation monitor will reconfigure to the bottom-left monitor of Figure 4.3. In that decentralized monitor m_1 receives not input from m_4, but from m_2^b and m_3 directly. The input that it receives are the evaluations of φ_2 and φ_3. Its aggregation function combines these evaluations with its own observable property (φ_1) using conjunction. Hence the resulting evaluation always corresponds to the evaluation of $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$, which is now m_1's aggregate after reconfiguration. This aggregate is equal to the formula to which the monitored is paired in the example expressiveness constraint.

Aside from a specific attack we might wonder how much damage a decentralized aggregation monitor can take in general before it fails, whilst taking into consideration that it may reconfigure in order to compensate for damage. This is especially useful in scenarios with many homogeneous local monitors such as botnets where attacks can be widespread and targeting any point in the network. This notion is captured by k-robustness. To determine the k of k-robustness, one has to consider the potential set of monitors which might fail and then check for each set of monitors of size k whether the decentralized aggregation monitor is robust with respect to those subsets and its expressiveness constraint. We do not consider the failure of monitors that occur in the expressiveness constraints when determining k. The reason is that if such a monitor fails we trivially have that the expressiveness constraints are not satisfied anymore.

Definition 4.10 (k-robustness): *We say that D^a is k-robust for $\mathcal{R}$, E and T, $k \in \mathbb{N}^0$ if D^a is F-robust with respect to T, $\mathcal{R}$ and E for any $F \subseteq M \setminus \{m | (m, \varphi) \in E\}$ such that $|F| \leq k$.*

Example 4.9 (Ex. 4.8 Cont.: k-robustness): Our example decentralized aggregation monitor is 0-robust for $\mathcal{R}$, E and T. This is the case because if m_3 alone fails ($F = \{m_3\}$, $|F| = 1$) then the expressiveness constraints cannot be satisfied anymore. If we would have a backup version of m_3 and could reconfigure accordingly using some $\mathcal{R}'$, then we would have a 1-robust decentralized aggregation monitor for $\mathcal{R}'$, E and T. This is the case because in that scenario if only one monitor fails (except for m_1, which occurs in the example expressiveness constraints) we can always reconfigure such that the expressiveness constraints are still satisfied.

If the decentralized aggregation monitor D^a is k-robust, then it means that there are at least k aggregation monitors who's aggregate equals the aggregate of another aggregation monitor. We note that 0-robustness with respect to T, $\mathcal{R}$ and E is equivalent to $\emptyset$-robustness with respect to T, $\mathcal{R}$ and E which simply means that D^a T-satisfies E. If every local monitor occurs in E then only 0-robustness can be obtained.

Though k-robustness provides us with a basic metric for the robustness of a decentralized aggregation monitor, we also want to know how important a local aggregation monitor is. We call this notion k-fail criticality, where $k = 1$ means that the local aggregation monitor is maximally important to the decentralized aggregation monitor and $k = \infty$ mean that the monitor is not important. If a local aggregation monitor has a criticality of $k = 1$, then its failure will cause the decentralized aggregation monitor to not function properly and no reconfiguration is possible to repair the defect. This is the case if the local aggregation monitor occurs in the expressiveness constraint of the decentralized monitor. It could also be that a monitor is not contributing at all to the fulfilment of the expressiveness constraint. In that case we define its criticality to be ∞. Finally, if a specific local aggregation monitor fails, then it could be that the decentralized aggregation may still reconfigure in order to work properly (i.e. satisfy its expressiveness constraint). However, after the reconfiguration there are now less other local aggregation monitors that may fail until the decentralized

aggregation monitor cannot reconfigure any more such that it satisfies its expressiveness constraint. In those cases, k-fail criticality for a local aggregation monitor means that k is the minimum number of other local aggregation monitors that have to fail before the decentralized aggregation monitor is not functioning properly, and which would not be an issue if the local aggregation monitor did not fail.

Definition 4.11 (k-fail criticality): *We say that an aggregation monitor $m \in M \setminus \{m' | (m', \varphi) \in E\}$ is k-fail critical, $k \in \mathbb{N}^0$, for D^a, $\mathcal{R}$, T and E iff there is a set $F \subseteq M \setminus \{m' | (m', \varphi) \in E\}$ such that:*

1. *$|F| = k$, $m \in F$, D^a is not F-robust with respect to T, $\mathcal{R}$ and E, and for each subset $F' \subseteq F \setminus \{m\}$ D^a is F'-robust with respect to T, $\mathcal{R}$ and E.*
2. *There is no $F' \subseteq M$ such that (1) holds for F' and $|F'| < |F|$.*

If there does not exist a $k < \infty$ such that m is k-fail tolerant for D^a, $\mathcal{R}$, T and E then we say that m is ∞-fail tolerant for D^a, $\mathcal{R}$, T and E. We say that each aggregation monitor $m'' \in \setminus\{m' | (m', \varphi) \in E\}$ is 1-fail tolerant for D^a, $\mathcal{R}$, T and E.

> **Example 4.10 (Ex. 4.9 Cont.: k-fail criticality):** In our example we have that monitors m_1 and m_3 are 1-fail critical since their failure will immediately result in the expressiveness constraint not being satisfied. m_2 and m_2^b are 2-fail critical. For either local aggregation monitor the minimal set such that the decentralized aggregation monitor fails, but for none of the subsets, is $\{m_2, m_2^b\}$. Monitor m_4 is ∞-fail critical. For each set of failing monitors that cause the decentralized aggregation monitor to not satisfy the expressiveness constraint and which includes m_4, there is a subset that does not contain m_4 and also causes the expressiveness constraint to be unsatisfied.

There is a connection between k-robustness and k-fail criticality. If a decentralized aggregation monitor is k-robust, then necessarily the fail criticality of a local aggregation monitor in that decentralized aggregation monitor must be 1 or higher than k. And if a local aggregation monitor is k-fail critical and $k \neq \infty$ and $k \neq 1$, then the minimum k-robustness of the decentralized aggregation monitor that it belongs to must be k.

Proposition 4.1: *Let $m \in \setminus\{m' | (m', \varphi) \in E\}$ be an aggregation monitor. If D^a is k-robust for $\mathcal{R}$, E and T then m cannot be j-fail critical, $j \in [1, k]$, for D^a, $\mathcal{R}$, T and E.*

Proof. From the definition of k robustness it follows that for m $k-1$ monitors in $\{m' | (m', \varphi) \in E\}$ can fail alongside m without causing the expressivity constraint to be unsatisfied, since D^a is F-robust for each set $F \subseteq M$ of aggregation monitors of size k. Hence the minimum k-fail criticality of D^a must be $k+1$. □

Proposition 4.2: *Let $m \in \setminus\{m' | (m', \varphi) \in E\}$ be an aggregation monitor. If m is k-fail critical for D^a, $\mathcal{R}$, T and E then D^a is maximally $(k-1)$-robust for $\mathcal{R}$, E and T.*

Proof. From the definition of k-fail criticality if follows that there is a set $F \subseteq M \setminus \{m|(m,\varphi) \in E\}$ of size k such that D^a is not F-robust with respect to T, $\mathcal{R}$ and E. Hence the maximum number j such that D^a is robust against failures of all subsets of $\{m|(m,\varphi) \in E\}$ of size j must be below k. □

There are a few ways in which the core metrics from this section can be expanded. Given a monitor design we can now see how critical a local aggregation monitor is in terms of its functioning. A straightforward expansion of this fail tolerance analysis is to not only check for F-robustness, but also look at which expressiveness pairs (m,φ) are not satisfied anymore in case the decentralized aggregation monitor is not F-robust. If not all expressiveness constraints are equally important, then for tolerance analysis this can be taken into account. Also looking at how many pairs are not satisfied is an important ingredient should one want to specify graceful degradation for decentralized aggregation monitors. At runtime the decentralized aggregation monitor can benefit from fail tolerance analysis by assigning higher repair priorities to critical monitors, if possible. We can also expand the work by looking at communication lines. Communication failures can be modeled quite similarly to local aggregation monitor failures.

4.4 Security

A leading reason to reduce security risks in monitoring applications is to prevent an attacker from gaining sensitive information. As mentioned before, there are various ways to increase the security of an implemented monitor by for instance using encrypted communication. Some of these techniques have drawbacks such as worse performance or more energy consumption. We, however, do not model in detail the communication that takes place among monitors, but we do model the kind of information that is shared. As with robustness in the previous section we shall provide core metrics for determining how vulnerable a decentralized aggregation monitor is and how critical a specific local aggregation monitor is. These metrics might help a designer of decentralized aggregation monitors to determine which monitors require extra attention in regards to security when they are implemented.

4.4.1 Attacks

The kind of attacks that we model in this section are those where the attacker may obtain information from monitors. This can in practice either take the form of the attacker obtaining the possibility of querying a monitor directly (e.g., if the attacker can falsely authorize itself as a parent of the monitor) or intercepting (and possibly decrypting) communication between two monitors. In either case, what the attacker obtains is the output of the monitor. If the attacker does not know what the aggregate of the monitor is, then the obtained output is quite meaningless. This is a security benefit that we automatically gain from aggregation where monitors have no further information about the input that they receive. In this section we will assume the worst-case where the attacker does know the aggregates of monitors, for instance because he/she has access to the design of the decentralized aggregation monitor.

In that case, we still gain some inherent protection from aggregation. If a local aggregation monitor is known to be emitting 't' and its aggregate is known to be $\Diamond p \vee \Diamond q$ then we still do not know whether p or q occurred. However, by combining information that is gained from multiple monitors, the attacker may still be able to obtain sensitive information. The security metrics in this section are based on how many monitors the attacker has to successfully attack in order to obtain sensitive information.

We model the attack itself as simply the subset of local monitors of a decentralized aggregation monitor which are considered to be attacked by the attacker. The kind of information for which the attacker gains access is the kind of information that these attacked local aggregation monitors share with other monitors. We do not have an explicit model of sensitive information. Instead we determine whether an attack reveals a property, meaning that the attacker may get the evaluation of that property. We assume that, like monitors, the attacker can aggregate extracted information by using some aggregation formula. However, for the attacker this is not a fixed formula. Hence, given the aggregates that it obtains from the decentralized aggregation monitor it can combine them by standard Boolean connectives. In the following $\mathsf{PL}(X)$ denotes all possible formulas that can obtained by applying Boolean connectives to elements in X.

Definition 4.12 (Monitor Attack, att): *Let $D^a = (M, Q)$ be a decentralized aggregation monitor. An attack $\mathsf{att} \subseteq M$ on D^a is a set of aggregation monitors. Let Φ contain all aggregates of aggregation monitors in att. The set of extracted properties $\mathcal{L}^{D^a}_{\mathsf{att}} = \mathsf{PL}(\Phi)$.*

> **Example 4.11 (Ex. 4.10 Cont.: Monitor Attack):** If an attacker can imitate m_4 then it will be able to query m_2 and m_3. In that case the attack is specified by $\mathsf{att} = \{m_2, m_3\}$. The aggregates of m_2 and m_3 are $\Box(\neg j_2 \vee \neg l_2)$ and $\Box(\neg j_3 \vee \neg l_3)$. Therefore, one of the extracted properties in $\mathcal{L}^{D^a}_{\mathsf{att}}$ is $\neg\Box(\neg j_2 \vee \neg l_2) \vee \neg\Box(\neg j_3 \vee \neg l_3)$, which is read as "somewhere in the future there was a jam at either location and/or the vehicle as at either location".

The framework can be expanded with different types of attacks. For instance, one might include attacks where the observation capabilities of local monitors might be hijacked. This type of expansion can be incorporated in the above definition. For aggregation monitors the evaluation of the observation formula would become available to the attacker, along side a monitor's aggregate.

For monitor safety we look at whether a specific attack can be used to observe some given property from the decentralized aggregation monitor. We assume the attacker knows what an aggregate represents.

Definition 4.13 (T-safe): *Let T be a target system, D^a be a decentralized aggregation monitor, att be an attack on D^a and $\varphi \in LTL_f$ be an LTL formula. We say that D^a is T-safe for φ and att iff there is no $\psi \in \mathcal{L}^{D^a}_{\mathsf{att}}$ such that $[w \models \varphi]_{\mathcal{L}_T} = [w \models \psi]_{\mathcal{L}_T}$ for each $w \in \mathcal{L}_T \cap S^*$.*

Example 4.12 (Ex. 4.11 Cont.: T-safe): Let $\mathsf{att} = M$ be an attack in which the attacker can query any of the aggregation monitors. Our example decentralized aggregation monitor is T-safe for $\Diamond(l_1 \wedge \Diamond l_2)$ and att. It is also T-safe for $\Diamond(l_1 \wedge \Diamond l_3)$ and att. This means that even if the attacker can obtain all available aggregates, it still cannot determine for a trace whether the vehicle used or may use in the future the route through locations one and two, or through locations one and three, respectively.

The space of potential attacks is heavily restricted by practical details that are not covered by our model. For instance, if in a network some local aggregation monitor only has wired connections to other local aggregation monitors in a safe environment, then it might be that the local aggregation monitor cannot be targeted for an attack. Therefore we focus on analyzing security risks wrt. potentially attacked local aggregation monitors and with the assumption that the attack is practically feasible. The security constraint of a decentralized aggregation monitor consists of a set of aggregation monitors that can potentially be attacked and a set of properties that represent sensitive information. The analysis of what properties count as sensitive should be part of the system's design methodology. These will differ per practical real-world scenario. A decentralized aggregation monitor satisfies its security constraint if none of the considered attacks allows the attacker to monitor a sensitive property.

Definition 4.14 (Security Constraint, S): *Let T be a target system and $D^a = (M, Q)$ be a decentralized aggregation monitor. A security constraint S for D^a is specified by (A, P) where $A \subseteq M$ is a set of local aggregation monitors and $P \subseteq LTL_f$ is a set of sensitive properties. D^a T-satisfies its security constraint iff for each and $\varphi \in P$, D^a is T-safe for φ and A.*

Example 4.13 (Ex. 4.12 Cont.: Security Constraint): Given our set of example aggregation monitors M and $\varphi = \Diamond(l_1 \wedge \Diamond l_2)$ and $\psi = \Diamond(l_1 \wedge \Diamond l_3)$, let our example security constraint be $S = (M, \{\varphi, \psi\})$. The example decentralized aggregation monitor T-satisfies this security constraint, following the previous example. This means that no matter which local aggregation monitors are attacked, the route of the vehicle remains private.

It should be noted that for some constraint (A, P) a not monitorable property $\varphi \in P$ might still be reasonable to consider as sensitive information. There is the possibility that given the behavior of the monitored system at runtime such a formula will always be evaluated to 'u' (unknown), but this is not guaranteed to be the case, unless φ is nonmonitorable after the empty trace.

4.4.2 Attack Tolerance

For security we also want to know how critical an attack on a monitor can be. That is, if a certain monitor's aggregate can be obtained, how bad is that? This is only interesting if the decentralized aggregation monitor *does not* satisfy its security constraint, because that would indicate that there are combinations of monitors such that the

attack on those monitors would reveal sensitive information. The attack tolerance of a local aggregation monitor m indicates how many other monitors need to be attacked in addition to m before sensitive information is leaked. A local aggregation monitor is maximally attack tolerant if it cannot contribute to any security leakage at all. For an aggregation monitor to contribute to a security leakage means that more sensitive properties were revealed due to the monitor being included in the set of attacked monitors.

Definition 4.15 (k-attack tolerant): *Let $S = (A, P)$ be a security constraint for a decentralized aggregation monitor $D^a = (M, Q)$. For an attack $\mathsf{att} \subseteq A$ and an aggregation monitor $m \in \mathsf{att}$ we say that m is contributing to att iff $P \cap \mathcal{L}^{D^a}_{\mathsf{att}} \neq \emptyset$ and $P \cap \mathcal{L}^{D^a}_{\mathsf{att}'} \subset P \cap \mathcal{L}^{D^a}_{\mathsf{att}}$, where and $\mathsf{att}' = \mathsf{att} \setminus \{m\}$. For an aggregation monitor $m \in M$ we define:*

- *m is k-attack tolerant for D^a and S iff $\mathsf{att} \subseteq A$ is the smallest attack such that m is contributing to att and $k = |\mathsf{att}|$.*
- *m is ∞-attack tolerant for D^a and S iff there is no $\mathsf{att} \subseteq A$ such that $m \in \mathsf{att}$ and m is contributing to att.*

> **Example 4.14 (Ex. 4.13 Cont.: k-attack tolerant):** For our example security constraint we showed that the decentralized aggregation monitor T-satisfies the constraint, meaning that every local aggregation monitor is ∞-attack tolerant for our example D^a and S. If our example security constraint was to be $S' = (M, \{\neg\Box(\neg j_2 \vee \neg l_2) \vee \neg\Box(\neg j_3 \vee \neg l_3)\})$, then m_2 and m_3 would have been 2-attack tolerant for D^a and S'. This follows from the fact that both monitors have to be attacked before the property is revealed.

If $D^a = (M, Q)$ T-satisfies a security constraint then all local monitors are ∞-attack tolerant. If a local monitor $m \in M$ is 1-attack tolerant then the monitor's aggregate is equivalent to a sensitive property, or the aggregate's negation is. Also, if a local aggregation monitor is k-attack tolerant for a decentralized aggregation monitor and a security constraint, then k other monitors are maximally k attack tolerant. This is explained in the proposition below. This also means that a high attack tolerance (other than infinity) can be good since a large attack is required to reveal a property, but it also indicates that many monitors require security measures. In a design one may want to balance between low attack tolerance of a few local aggregation monitors versus high attack tolerance of many local aggregation monitors.

Proposition 4.3: *Let $D^a = (M, Q)$ be a decentralized aggregation monitor and $S = (A, P)$ be a security constraint for D^a and $m \in M$ be an aggregation monitor. If m is k-attack tolerant for D^a and S then k aggregation monitors are maximally k-attack tolerant for D^a and S.*

Proof. If m is k-attack tolerant for D^a and S then there is an attack $\mathsf{att} \subseteq A$ of size k such that a property in P is revealed. Since att is minimal, every monitor in att contributes to the reveal of a property. Hence for each of the k aggregation monitors in att either att or a smaller attack reveals a property from P, meaning that their maximal attack tolerance is k. □

Now we can determine how attack tolerant a local aggregation monitor is, which is an indicator for how critical the local aggregation monitor is for the security of the decentralized aggregation monitor. Based on this basic framework several extensions are possible. For instance if a monitor is easier to attack than another (i.e. a simple sensor in a WSN versus a sophisticated sink), then the tolerance of the easier target should be decreased relatively to the harder target. In a runtime environment if attacks are detected then new attack tolerance values can be computed with the knowledge that some local monitors are already attacked. This could for instance be countered with low attack tolerance local monitors switching to more secure, albeit energy and/or more overhead cost expensive, communication protocols.

4.5 Conclusion

We looked at the robustness and security aspects of a decentralized monitoring. For this we drew inspiration from wireless sensor networks in which information is intermediately aggregated for energy and security reasons. We proposed and discussed an aggregation based monitoring framework and provided basic metrics for robustness and security. With these metrics we can prioritize the investment of resources regarding robustness and security for those monitors that are most important for the continued operation of a decentralized monitor or reveal sensitive information when captured.

5 Control

For norm enforcement we require both monitoring for detecting norm violations and control to modify the behavior of the target system that is controlled. The previous two chapters discussed runtime monitoring. This chapter builds on existing approaches from runtime verification and control automata to define controllers that enforce norms at runtime. We propose norm-based control mechanisms that modify system executions to make them norm compliant. A centralized control mechanism does not suffice for many multi-agent systems as it may pose scalability issues, a single point of failure and requires central data gathering which is not always allowed. We present a novel approach for combining individual norm-based controllers into a collaborative controller and investigate the norms that they collaboratively can enforce in terms of properties of the individual controllers. Our approach promotes scalable and modular development of controllers for norm enforcement.

5.1 Introduction

In the previous chapters we discussed how we can monitor a target system in order to see whether it behaves 'correctly'. In this chapter we focus on how we can change the behavior of the target system so that it becomes 'correct'. We refer to this process as enforcement. To specify desirable behavior we turned to linear temporal logic and the notion of norms. Norms are a popular candidate for the specification of system level properties and can be seen as standards of behavior that distinguish good and bad behavior (see e.g., [120, 52, 26]). Various languages have been proposed to represent different types of norms such as state-based norms, action-based norms, temporal norms, and combinations thereof (e.g., [33, 4, 48, 8]). For each class of norms, monitoring and enforcement models have been proposed to detect and control norm violations, respectively.

Existing proposals for norm monitoring and enforcement are either based on logical models where norm violations are explained in terms of the satisfiability of a violation formula (i.e. a formula that characterizes violated states or executions like we discussed in the previous chapters) and norm enforcement is explained in terms of model updates [4, 7], or are concerned with practical frameworks for building normative multi-agent systems (e.g., [57, 50, 75]). Most logical approaches to norm enforcement focus on infinite executions and are not concerned with runtime norm

enforcement. An exception is the runtime model for norm enforcement proposed by Alechina et al. [5]. However, in this work norms are only enforced by halting the system execution before a norm violation occurs.

In other branches of computer science, such as runtime verification and control automata, the idea of monitoring and control at runtime have been extensively studied, albeit from a different perspective. The main research problem in these areas is to ensure system level properties when an untrusted target system is expected to produce unwanted behavior. A runtime controller can be used to revise the behavior of an untrusted system to enforce system level properties. Such a controller reviews the actions that the system produces and may decide per action whether to allow the action, suppress the action or execute extra actions. These controllers have been studied in terms of their formal properties, for example in [115, 90, 59, 37].

A characteristic feature of norms is that they may be enforced by means of regimentation and sanctioning. A norm can be either regimented in the sense that norm violations are prevented, or sanctioned in the sense that norm violations incur sanctions. This chapter builds on runtime controllers from related work and applies them to norm enforcement in multi-agent systems. We call the resulting controller either a regimenting or sanctioning controller for a norm, depending on whether regimentation or sanctioning is applied. We have to redefine the notions of norms, regimentation and sanctioning in order to bridge the gap between runtime control theories and normative systems. Without loss of generality, we consider a norm as a set of violating behaviors. A norm-based controller reviews the actions that are performed by a target system. If there is a violation about to happen, then the execution is halted in case of regimentation, or the action is allowed and followed by a sanctioning action in case of sanctioning. Our first contribution in this chapter is to model norm regimentation and sanctioning in a consistent manner with respect to the aforementioned controller models of Ligatti et al. [90] and Falcone et al. [59]. These models allow us to analyze regimenting and sanctioning controllers and investigate the enforcement of norms.

A target system should be compliant with a set of norms for many applications. Centralizing the task of enforcing all norms can cause issues regarding scalability, security and robustness. This has led to various proposals for distributed architectures for norm enforcement (cf.[96, 75, 60, 100, 126]). In these architectures the general setup is that multiple norm-based control mechanisms are applied concurrently on a target system. These control mechanisms independently process local observations and communicate them in order to collaborate on the task of enforcing norms. We follow this general architectural setup and use the term distributed norm-based controller to refer to a set of norm-based controllers. The main problem to address for distributed norm-based controllers is the issue that multiple controllers may try to realize conflicting changes in the target system's behavior. Our second contribution is the introduction of a framework that allows us to formally analyze a set of norm-based controllers that enforce their norms concurrently on one and the same target system. We propose the construction of collaborative automata to show that a set of norm-based controllers is able to modify the behavior of a target system to comply with the entire set of norms without concurrent conflicting changes.

The rest of this chapter is structured as follows. In Section 5.2 we explain some basic notation that is used in this chapter. This is slightly different from the previous

chapters as we now work with target system actions/events rather than target system states. In Section 5.3 we discuss the theoretical background and foundation of properties, which are a means to specify desirable behavior. In Section 5.4 we discuss the background literature on the enforcement of properties. In Section 5.5 we relate norms and norm enforcement to properties and property enforcement. In Section 5.6 we discuss a formal tool called controller automata for modeling property (and norm) enforcers, which in Section 5.7 we combine in collaborative automata.

5.2 Formal Setting

In this chapter we focus on sequences of actions, rather than sequences of states as we did in Chapters 3 and 4. There is overlap in terms of notation, but for completeness we list all notation here. The notation in this chapter is independent from the previous chapter. For instance a word in this chapter is denoted with α, which has no connection to aggregation formulas from Chapter 4 for which we used the same symbol.

The focus of control lies upon the actions of a target system. These can be modeled in different ways. Bauer and Falcone [19] model a target system action as a set of subactions that co-occurred at a certain time interval. This is suitable if one wants to model a partial view of the target system, because one can let a component only see a subset of the actions at any given time. However, we are mainly interested in the conflicts that arise when concurrent controllers try to modify the target system behavior at the same time. For this reason we take a simple approach in order to keep the model more concise. We take only single actions for any time as in the work by Falcone et al. [59] and Ligatti et al. [90]. For the remainder of this chapter, let $\mathcal{A}$ be a finite non-empty set of actions. We use $a \in \mathcal{A}$ for an arbitrary action.

A target system produces sequences of actions from $\mathcal{A}$, which we refer to as words. We denote with $\mathcal{A}^*$ and $\mathcal{A}^\omega$ the finite and infinite words of elements of $\mathcal{A}$ and $\mathcal{A}^\infty = \mathcal{A}^* \cup \mathcal{A}^\omega$. The empty word is denoted by ε and is contained in $\mathcal{A}^*$. We use $\alpha \in \mathcal{A}^\infty$ for an arbitrary word. For a word $\alpha \in \mathcal{A}^\infty$ we use $|\alpha|$ to indicate the length of the word, which is ∞ in case $\alpha \in \mathcal{A}^\omega$. For a word $\alpha = a_1 a_2 ...$ we use $\alpha[i]$, $i \in [1, |\alpha|]$ (or $i \in [1, \infty)$ in case α is infinite) to indicate the ith action of α. For a word $\alpha = a_1 a_2 ...$ and two indices $i \in [1, |\alpha|], j \in [i, |\alpha|]$ (or $i \in [1, \infty), j \in [i, \infty)$ in case α is infinite) we use $\alpha_{..i}$, $\alpha_{i...j}$ and $\alpha_{j..}$ to indicate the words $a_1 ... a_i$ (called prefix up to i), $a_i ... a_j$ (called the subword between i and j) and $a_j ... a_k$ (called the suffix from j). For two words $\alpha \in \mathcal{A}^*$, $\alpha' \in \mathcal{A}^\infty$ we use $\alpha \preceq \alpha'$ and $\alpha \prec \alpha'$ to indicate that α is a prefix or strict prefix of α' respectively. For two words $\alpha \in \mathcal{A}^*$, $\alpha' \in \mathcal{A}^\infty$ we notate their concatenation simply by $\alpha\alpha'$.

> **Example 5.1 (Scenario):** Consider a file system in which an unknown agent can manipulate a file. The possible actions are r (read), w (write), s (save) and b (backup), i.e. $\mathcal{A} = \{r, w, s, b\}$. The agent can use an interface to the system in order to attempt to execute these actions in any order and as many times as it desires. A session of the agent with the file system is modeled by a word. We cannot assume anything about the possible word of actions that can be produced

during a session because we do not have a model of the agent.

The system designer may want to exclude certain words from happening. For instance, it might be prescribed that work must be saved in-between any two write actions. That would mean that $rwrsw$ is a good word but $rwrws$ is a bad word. In this chapter we will discuss how we can model controllers that upon reading a bad word can transform it into a good word.

Finally, we use a given target system specific equivalence relation on words, $\sim \subseteq \mathcal{A}^\infty \times \mathcal{A}^\infty$. I.e. $\sim$ is transitive, symmetric and reflexive. If $\alpha \sim \alpha'$ for two words $\alpha, \alpha' \in \mathcal{A}^\infty$, then we say α and α' are similar. This relation allows us to model semantical equivalence between syntactically different words. Ligatti et al. describe the following common reasons for word similarity to occur in an application [90]:

- A word may contain a sequence of idempotent actions or 'void' actions, which - if removed or replaced by another action - still amounts to the same system behavior. For instance saving a file twice in a row is similar to saving it once. Hence for our example we might have that $rwsb \sim rwssb$.

- One of the actions in $\mathcal{A}$ may model a macro that is a sequence of other actions in $\mathcal{A}$. Hence, a word that has the macro, or one where the macro is replaced by the actions that it abstracts from are similar. In our example scenario we have no such macro, but for instance one could be defined as: action x stands for 'save and back up', which is similar to first performing a 'save' action and then performing a 'back up' action. In such a case we could have that $rwsb \sim rwx$.

- Action sequences might be permuted without consequence. For instance our example can be expanded such that there might be multiple interleaved and independent sessions regarding different files. Let us assume we have then a set of actions $\mathcal{A} = \{r^1, w^1, s^1, b^1, r^2, w^2, s^2, b^2\}$, with the same reading as in Example 5.1, except that superscripts indicates a session identifier. We could have that $r^1w^1s^1r^2w^2s^2 \sim r^1r^2w^1w^2s^1s^2$.

5.3 Policies and Properties

Recall that we assume that there is some target system, which may or may not exhibit unwanted behavior. We can model that target system in different ways. In this chapter we model a target system as a set of words, which are exactly those words that the target system may produce and are hence the words that a controller has to deal with. Note that this model encapsulates models where the target system is represented by a labeled transition system where transition labels are actions, and the possible words of the system are the labels of all possible consecutive transitions. More precisely, we model a target system T as $T \subseteq \mathcal{A}^\infty$, where T is prefix closed. Such a target system is uniform if $T = \mathcal{A}^\infty$ and otherwise is nonuniform [90]. The target system from Example 5.1 is for instance uniform as the agent is inherently unconstrained in the actions that it can execute. For the remainder of this chapter we only consider uniform target systems and therefore omit the mentioning of a target system where possible.

A security policy is a predicate over sets of words [115]. Let $P : 2^{\mathcal{A}^\infty} \to \{\top, \bot\}$ be a policy. A target system $T \subseteq \mathcal{A}^\infty$ obeys P if, and only if, $P(T) = \top$. I.e. one can say that the behavior of the target system is correct given a policy P if, and only if, all its possible behavior is conform the policy. As an example, a policy P may be true for T if every word in T starts with a specific initialize action. In the previous chapters we mainly concerned ourselves with the question whether the target system obeyed some specification of desired behavior. In the setting of this chapter this is equal to determining the target system obeys a policy. However, in this chapter we are mainly concerned with guaranteeing that the behavior of the target system is altered in such a way that it cannot but obey a given policy. Furthermore, we only want to change the behavior of the target system if it does not obey the policy. When we talk about control in this chapter we refer to the process that alters the target system behavior if necessary. Control is performed by a controller which in turn encapsulates the target system. It does not change the target system itself.

If we have a model of the target system, then we may use offline/design time techniques (as opposed to runtime) to determine whether the behavior of the target system is correct given some policy. If we do not have model of the target system available, or if we know that it may violate the policy, then we may resort to runtime control techniques (e.g. [90, 59, 110, 115]). With these techniques a target system's behavior is incrementally revealed to the controller. A controller may decide upon altering the behavior upon the reveal of a new action. Hence to do this, it must determine whether or not the target system is violating the given policy. But this is not always possible for each policy. Consider for instance the policy P that is true for a target system $T \subseteq \mathcal{A}^\infty$ if, and only if, there is a word $\alpha \in T$ such that $\alpha[1] = a$, for some action $a \in \mathcal{A}$. If we do not know T, but it is revealed at runtime that a word with prefix $a_1...a_k$ is in T such that $a_1 \neq a$, then we cannot determine whether it could have been the case that the word started with a. Therefore, policies which are meaningful to runtime control are those policies where a breach of the policy can be determined from individual words. We call such policies properties [10]. A policy P is a property if, and only if, there exists a predicate $\hat{P}$ over individual words such that a given target system $T \subseteq \mathcal{A}^\infty$ obeys P if, and only if, $\forall \alpha \in T : \hat{P}(\alpha)$.

Given that we can tell for a given word and property whether or not the word is in accordance with the property, we may model a property as a set of words that obeys the property. Also recall that we have a system-specific equivalence relation of words; $\sim$. Given that the relation models when two words are deemed similar, it should not be the case that a property can distinguish between words that are similar given $\sim$. Finally, we take the position that only execution of actions can cause the violation of a property. Therefore the empty word must be in a property.

Definition 5.1 (Property, P): *A property is given by $P \subseteq \mathcal{A}^\infty$ such that $\varepsilon \in P$ and if $\alpha \in P$ and $\alpha \sim \alpha'$ then also $\alpha' \in P$ for all $\alpha, \alpha' \in \mathcal{A}^\infty$.*

Example 5.2 (Ex. 5.1 Cont.: Property): Let property P contain all words where in between every two write actions w there is a save action s. Example words in this property are rwr, $rwswr$ and $rwsbrwsb$. Example words that are not in this property are $rwrw$ and $rwbrwsb$. As mentioned before, our example target system is uniform, meaning that if $T \subseteq \mathcal{A}^\infty$ are the possible words that might be produced, that $T = \mathcal{A}^\infty$. Hence, we assume there are words in T which are not in P.

The set of all possible properties has some interesting subsets. Two of those subsets are the safety properties [87] and liveness properties [11]. Any property is a safety or liveness property (or a combination thereof) [10]. If a property is a safety property or a liveness property then different techniques can be used to prove whether some target system obeys that property. It also helps in designing the requirements of the target system. For our purposes we shall see that formal properties of behavior properties depict what kind of capabilities a controller requires [59].

Safety properties are generally regarded as those properties that state that nothing bad ever happens. An informal example safety property would be 'the agent should always back up immediately after a save'. Hence if such a property is violated in a word $\alpha \in \mathcal{A}^*$, then for an extended word $\alpha\alpha'$, $\alpha' \in \mathcal{A}^\infty$, it must be the case that the property is also violated for that word. This precisely classifies safety properties [87]: A property P is a safety property if, and only if, for each word $\alpha \in \mathcal{A}^*$ if $\alpha \notin P$ then for each other word $\alpha' \in \mathcal{A}^\infty : \alpha\alpha' \notin P$.

Liveness properties are generally regarded as those properties that state that something good eventually happens. For instance, 'the agent should eventually write' is an informal liveness property. However, regarding infinite words it might also include properties such as 'the agent will always eventually write'. If a word is in a liveness property, then each of its extensions must be in the property. Also, every word that is not in a liveness property can be extended such that it does obey the property (by adding the 'good thing' to the word). This precisely classifies liveness properties [11]: A property P is a liveness property if, and only if, for each word $\alpha \in \mathcal{A}^*$ there is a word $\alpha' \in \mathcal{A}^\infty$ such that $\alpha\alpha' \in P$.

Hence with safety and liveness one usually describes the constraints (no illegal sequences of actions) and the goals (some particular goal sequence of actions must be executed at some point) of a system. Agents typically are designed to run indefinitely, and multi-agent systems are the kind of target systems that we want to control. Therefore, we assume that a target system may run indefinitely. Hence for a runtime controller it is unknown when the word that describes the target system's behavior ends. To determine whether the target system is producing a correct word given a safety property is relatively straightforward in such a setting. The controller can monitor whether something illegal occurred and then perform some measure to 'fix' the target system's behavior. We will go more into such measures in the rest of this chapter. However, for liveness properties this is not so clear. It might be the case that the 'good thing' never happens, in which case the controller indefinitely does not know whether the target behavior is in the property.

Falcone et al. [59] utilize a different classification of properties in order to get a firmer grasp on what is, and what is not, controllable behavior for different capa-

bilities of controllers. They turn to the safety-progress classification as proposed by Chang et al. [38]. In the next section we refer back to the safety-progress classification when we talk about different definitions that specify that a target system's behavior is altered in such a way that it satisfies some given property. This classification describes a hierarchy of property sets. These sets are the safety, guarantee, persistence, response, obligation and reactivity properties. Reactivity strictly contains persistence and response. Persistence and response strictly contain obligation. Obligation strictly contains safety and guarantee. Originally this classification is over infinitary properties, but it is straightforwardly extensible to finitary properties. The informal reading of these sets is as follows:

- A safety property states that something bad never happens. (e.g., the agent never writes)
- A guarantee property states that something good happens at least once. (e.g., the agent will write at some point)
- A persistence property states that eventually something bad never happens (again). (e.g., the agent will at some point never write)
- A response property states that something good happens infinitely often. (e.g., the agent will always write sometime in the future)
- An obligation property is a Boolean combination of safety and guarantee properties. (e.g., the agent will never write and it will make a backup at some point)
- A reactivity property is a Boolean combination of persistence and response properties. (e.g., the agent will at some point never write or it will always make a backup at some point in the future)

This classification is depicted in Figure 5.1. In the next section we discuss the enforcement of properties. The figure shows already which classes of properties can and cannot be enforced by the techniques that are discussed in this chapter.

5.4 Enforcement of Properties

So far we only mentioned what a property is and that a controller should alter the behavior of the target system such that it fulfills some property. The process of altering behavior to ensure property fulfillment is called enforcement. Given a word from the target system, a controller enforces a property by reviewing the word from left to right and producing an output word that satisfies the property. This corresponds with the incremental reveal of the word at runtime. It can be the case that the controller introduces new actions into a word that a target system would not introduce by itself. We assume that a controller works deterministically, i.e. it is a mapping from words to words. We further assume that a controller never infinitely inserts new actions between revising two actions. The intuition behind this is that a controller

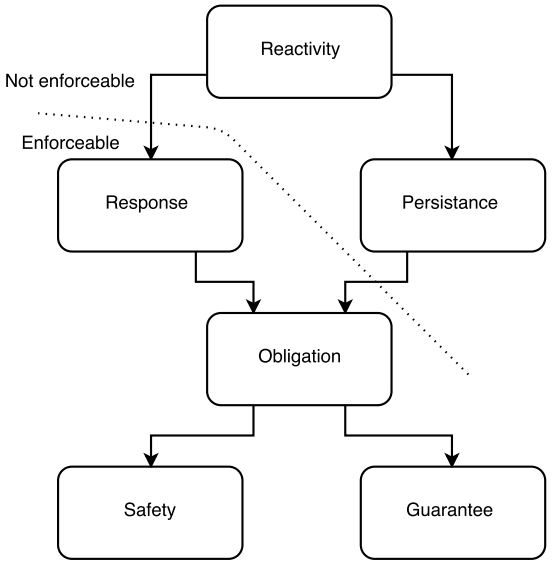

Figure 5.1: *The safety-progress classification. Arrows indicate superset relations, e.g., the class of obligation properties is a superset of the class of safety properties. The reactivity and persistance class are not enforceable. Also note that all non-safety properties are considered to be progress properties.*

should process a newly revealed action in a finite amount of time, so that the target system may produce the next action.

Our model is for runtime controllers without look-ahead or knowledge of the target system. As such, a controller must decide upon the reveal of an action what action or actions it appends to the output. It is therefore not possible that upon reading some word $\alpha \in \mathcal{A}^*$ a controller outputs a word $\alpha' \in \mathcal{A}^*$, and that upon reading a word $\alpha\alpha'' \in \mathcal{A}^\infty$ that the output does not have α' as a prefix. This is captured by the formal definition below.

Definition 5.2 (Controller, c): *A controller is given by a function $c : \mathcal{A}^\infty \to \mathcal{A}^\infty$ such that:*

1. *if $\alpha \in \mathcal{A}^*$ then $c(\alpha) \in \mathcal{A}^*$, and*
2. *if $\alpha' \preceq \alpha$ then $c(\alpha') \preceq c(\alpha)$ for all $\alpha \in \mathcal{A}^\infty$ and $\alpha' \in \mathcal{A}^*$.*

Example 5.3 (Ex. 5.2 Cont.: Controller): Let c be a controller over $\mathcal{A}$. Furthermore, among possible other rules, the controller c inserts a save action after each write action. E.g. consider the word $\alpha = rwr$. This word is revised as $c(\alpha) = \alpha' = rwsr$. This implies that given $\alpha'' = rwrs$ it cannot be the case that $c(\alpha'') = \alpha''$, because $\alpha \preceq \alpha''$ but $c(\alpha) \not\preceq c(\alpha'')$ (i.e. $rwsr \not\preceq rwrs$). Intuitively, when c reviews the word rwr it enforces a save action. Then, it proceeds to read s and cannot retract the inserted save action. The action has already been executed.

To grasp the intuition behind a controller, it may help to consider the word that the controller is reviewing as a stream of action requests for which it must decide whether the requests are granted, suppressed or expanded with another action. Once the controller makes a decision (and it must do so immediately upon receive a request), it cannot revisit that decision later on.

In the following, we assume, unless stated otherwise, that a property $P \subseteq \mathcal{A}^\infty$, a controller c, and a word $\alpha \in \mathcal{A}^\infty$ are given.

We did not specify how a controller should rewrite functions given a property. The reason is that we can distinguish different intuitions of what the enforcement of a property might mean. We follow the line of Ligatti et al. [90] and concern ourselves with three notions of property enforcement. The most basic type of enforcement only concerns the intended purpose of a controller, which is that it revises all words from $\mathcal{A}^\infty$ such that the revisions satisfy some desired property P (i.e. $c(\alpha) \in P$ for all $\alpha \in \mathcal{A}^\infty$). This intuition is called soundness and enforcement that is only constrained by soundness is called conservative enforcement.

Definition 5.3 (Conservative Enforcement [90]): *A controller c conservatively enforces a property $P \subseteq \mathcal{A}^\infty$ iff $\forall \alpha \in \mathcal{A}^\infty : c(\alpha) \in P$.*

Example 5.4 (Ex. 5.3 Cont.: Conservative Enforcement): Let c be a controller that conservatively enforces P from Example 5.1, which is the property where a save action is executed between any two write actions. One way to achieve this revision is to simply revise any occurrence of the write action w in a word $\alpha \in \mathcal{A}^\infty$ into the word sw. I.e. the controller might be defined as: $c(\alpha) = \alpha_{..i} \cdot s \cdot \alpha_{i+1..}$ for each $i \in [1, |\alpha|]$ (or $i \in [1, \infty)$ if $\alpha \in \mathcal{A}^\omega$) where $\alpha[i] = w$.

If a controller is conservatively enforcing some property, then it still might be not quite what we have in mind. In particular, a trivial way to achieve soundness is to revise any word to the empty word. Because the empty word is contained in every property. But also Example 5.4 showed a controller that revises correct words to syntactically different words (e.g., it rewrites $wsrw$ to $swsrsw$). This is something that is often frowned upon; a controller should not interfere with a word if it is correct. Hence, we want a controller to not only ensure correct behavior, but also to be transparent in the sense that the target system's behavior is not meaningfully altered if there is no violation of the property [59]. In other words, if a word already satisfies the property, then it should remain unchanged or at least be revised to a similar word with respect to $\sim$. Otherwise the word is mapped to its longest prefix satisfying the property.

Definition 5.4 (Longest Correct Prefix): *The longest correct prefix of $\alpha \in \mathcal{A}^\infty$ with respect to a property P is $\alpha' \in P$ such that $\alpha' \preceq \alpha$ and for all $\alpha'' \in \mathcal{A}^\infty$ with $\alpha' \prec \alpha'' \preceq \alpha$ it holds that $\alpha'' \notin P$.*

A word may not have a longest correct prefix given a property. Consider an infinite word α in which some action a does not occur. Next, let P be a property such that every finite word occurs in P and those infinite words where at some point a always

holds (note that this is a persistence property). There is no finite prefix of α that does not satisfy the property P. Hence, there is also no longest prefix such that all longer prefixes do not satisfy P. Even though α itself violates the property.

Next we recall the definition of precise enforcement from Ligatti et al. [90]. Controllers that precisely enforce a property are sound, plus they output revealed actions, without alteration, until the property is violated. This type of enforcement is ideal for situations where the controller must decide upon reading an action whether to halt the system or to allow the action. Such controllers have been extensively analyzed by Schneider [115].

Definition 5.5 (Precise Enforcement [90]): *A controller c precisely enforces P if, and only if, for all $\alpha \in \mathcal{A}^\infty$ the following holds:*

1. *$c(\alpha) \in P$ and*
2. *if $\alpha \in P$ then for all $\alpha' \in \mathcal{A}^\infty$ with $\alpha' \preceq \alpha$ it holds that $c(\alpha') = \alpha'$.*

> **Example 5.5 (Ex. 5.3 Cont.: Precise Enforcement):** Let P be again the property that contains all words where between each two write actions there is a save action. Let c be the controller that revises a word $\alpha \in \mathcal{A}^\infty$ as follows: let $i \in [1, |\alpha|]$ (or $i \in [1, \infty)$ if $\alpha \in \mathcal{A}^\omega$) be the lowest index such that $\alpha[i] = w$ and there is an index $j \in [1, i-1]$ such that $\alpha[j] = w$ and for all $k \in [j, i] : \alpha[k] \neq s$, $c(\alpha) = \alpha_{..i-1}$ if such an i exists and $c(\alpha) = \alpha$ otherwise. That is, if the agent tries to write without performing a save since the last write action, then c will suppress the write action as well as all subsequent actions. c precisely enforces P.

Not each property can be precisely enforced. It is well known that exactly the class of safety properties from the safety-progress classification from Chang et al. [38] can be precisely enforced [90]. We observe that if a controller c precisely enforces a property P then it will revise any word $\alpha \in \mathcal{A}^\infty$ to its longest correct prefix with respect to P. Note that this type of enforcement is transparent on a syntactical level (i.e., no syntactical change is made to the word if it is not violating the property).

It might be the case, however, that a controller withholds or inserts some actions upon revising a word without changing its semantic meaning. For those types of controllers the notion of effective enforcement has been introduced [90]. A controller that effectively enforces some property will rewrite any correct word to a similar word according to $\sim$.

Definition 5.6 (Effective Enforcement, [90]): *A controller c effectively enforces a property P with respect to $\sim$ iff for all $\alpha \in \mathcal{A}^\infty$ it holds that:*

1. *$c(\alpha) \in P$ and*
2. *if $\alpha \in P$ then $\alpha \sim c(\alpha)$.*

> **Example 5.6 (Ex. 5.3 Cont.: Effective Enforcement):** Let P again be the property where in between each two write actions a save action occurs. Furthermore, let us assume that saving more than twice between write actions does not semantically alter a word according to $\sim$. Let c be a controller such that for a given word any second or later occurrence of w is revised to sw (i.e. a save is

forced before each write action after the first). Then, c effectively enforces P with respect to $\sim$. Note that if we would take equality $=$ as the similarity relation, then c would neither precisely enforce P nor would it effectively enforce P with respect to $=$, though c would be sound.

Note that effective enforcement with respect to some property by a controller c does not require that all prefixes of correct words are rewritten by c to themselves as it was the case for precise enforcement. Note that if c precisely enforces P, then it also effectively enforces P up to any similarity relation. If c effectively enforces a safety property P with respect to $=$, then it also precisely enforces P. It is well known that only the class of response properties from the safety-progress classification from Chang et al. [38] can be effectively enforced [59]. These are the properties where each word that does not satisfy the property has a longest correct prefix for that property. To get an intuition about this, consider the property that states that 'the agent will eventually only read, or the system halts'. Given an infinite word α where the agent never stops writing, saving or backing up, all prefixes are correct prefixes, and hence no longest correct prefix exists. Therefore, the controller cannot but allow all the actions that it is reviewing, which means that α will be rewritten to itself, which makes such a controller not sound.

5.5 Norms

In Section 3.6 we discussed norms and how their violations might be detected through runtime (decentralized) monitoring. To summarize, norms are a means to specify desirable behavior. We can see that for different norm models, such as counts-as and sanction rule structures and temporal conditionalized obligations and prohibitions, a norm can be violated multiple times in a word. For instance norms that are represented as conditional obligations with deadlines (cf. [33, 7]) are violated each time that the deadline occurs and the obligation has not been satisfied since the last time the condition held. We can imagine other ways to model norms, but whichever model we choose, it must be possible to detect a norm violation in a word after an arbitrary but finite number of steps. Otherwise a violation cannot be detected by a runtime mechanism. Hence it must be possible for a norm model to capture it using a set of finite words, such that these words contain the norm's violation at the final action (i.e., the final action is the deadline of a conditionalized obligation which has been triggered earlier, or a prohibited action). Just as with properties, this set must not be able to distinguish between similar words according to the similarity relation $\sim$.

Properties as discussed in this chapter also specify desirable behavior. For an arbitrary norm, let $V \subseteq \mathcal{A}^*$ be the set of finite words, which is closed under $\sim$, such that the norm is considered to be violated because of the final action. As with properties we assume that a norm cannot be violated if no action has been executed, hence $\varepsilon \notin V$. We know that $\mathcal{A}^\infty \setminus V$ can specify a property because it is closed under $\sim$ (in turn, because V is closed under $\sim$) and we know that $\varepsilon \in P$. One way to prevent violations is to make a controller that soundly enforces the property $P = \mathcal{A}^\infty \setminus V$. This particular manner of preventing norm violations is related to regimentation in normative systems literature. When a norm is about to be violated,

then the controller halts the system if the norm is regimented. Example programming frameworks that support regimentation are $\mathcal{S}$-Moise$^+$ [75], 2OPL [46] and NPL [73].

However, only considering regimentation would not do the field of normative systems justice. A key concept alongside regimentation is that of sanctioning. With sanctioning a sanction is executed whenever a norm is violated. Example proposals for sanctioning controllers can be found in for instance [46, 73]. This type of norm enforcement is not supported by the property enforcement concepts that we see in property enforcement literature. Because with sanctioning we do not want to exclude any words that contain norm violations, but exclude any words where norm violations are not sanctioned. In particular, if a norm is violated, then the word is revised to include the sanction right after the violation. Hence, in contrast to the transparency principle of property enforcement, we *do* want to change the semantical meaning of words, other than output a longest correct prefix. We assume a global and fixed set of actions $\mathcal{S} \subseteq \mathcal{A}$ that can be used as sanctions. For clarity throughout the formal results in this chapter we define and assume normative similarity. Normative similarity is the same as identity, except that the consecutive execution of the same sanction is similar to having the sanction executed once.

Definition 5.7 (Normative Similarity, $\sim_{\mathcal{S}}$): *Let $\alpha_1, \alpha_2 \in \mathcal{A}^\infty$ be two words. α_1 is similar to α_2 with respect to the the set of sanction actions $\mathcal{S}$, notated as $\alpha \sim_{\mathcal{S}} \alpha'$ iff $\alpha'_1 = \alpha'_2$, where α'_i is α_i such that any subwords that consist solely of the same sanction are replaced by a single occurrence of that sanction.*

As mentioned earlier, for runtime control it is required that the violation of a norm should be detectable after a finite number of actions. We therefore represent all violations of a norm as a set of words such that a violation occurs necessarily at the last action of those words, but possibly also earlier. A norm itself is represented as a tuple that contains the violations of the norm and the sanction that should be applied after a violation occurs, in case the norm's violations should be sanctioned. A sanction cannot take away a violation of the norm, or cause it. Therefore the following constraints hold for the set of violations of a norm (which correspond to the bullets in the definition of norms):

- Violations are closed under normative similarity.
- An action must be executed to cause violation.
- A sanction cannot cause a violation, i.e. a sanction cannot be the last action of a violation word.
- A violation word that contains a sanction is also a violation word if that sanction is removed.
- A violation that contains no sanction, is still a violation if a sanction is added somewhere other than the end.

Note that the definition of norms does not specify whether a norm should be enforced through regimentation or sanctioning. If it is decided that a norm is to be enforced through regimentation, then the sanction will be ignored by the regimenting controller, as formally defined in the next section.

Definition 5.8 (Norm, n): *A norm is represented as a tuple $n = (V, \sigma)$ where $V \subseteq \mathcal{A}^*$ is the set of violations and $\sigma \in \mathcal{S}$ is a sanction. A word $\alpha \in \mathcal{A}^*$ is a violation of n iff $\alpha \in V$. Moreover, for each $\alpha \in \mathcal{A}^*$, α is said to violate n if there is a prefix $\alpha' \preceq \alpha$ that is a violation of n. Furthermore:*

- *V is closed under $\sim_{\mathcal{S}}$, and*
- *$\varepsilon \notin V$, and*
- *$\alpha[|\alpha|] \notin \mathcal{S}$, for all $\alpha \in V$, and*
- *If $\alpha = \alpha_{..i-1} s' \alpha_{i+1..} \in V$ for some $s' \in \mathcal{S}$ then $\alpha_{..i-1}\alpha_{i+1..} \in V$, and*
- *If $\alpha \in V$ then $\alpha_{..i} s' \alpha_{i+1..} \in V$ for each $i \in [1, |\alpha| - 1]$, $s' \in \mathcal{S}$.*

Example 5.7 (Ex. 5.6 Cont.: Norm): We expand our scenario with an extra action u (undo) which reverts the last action only. If multiple undo actions are performed in sequence, then this is similar to one undo action according to $\sim$. E.g. $rwbuus \sim rwbus$. The set of sanctions for our scenario is $\mathcal{S} = \{u\}$. We assume that the agent will not by itself execute u. Consider a norm that says that the agent has to save between writes, and the sanction of which is the undo action. The set $V \subseteq \mathcal{A}^*$ contains all words $\alpha \in \mathcal{A}^*$ such that the final action is a write action and no save action has occurred since the last write action. By this definition $wswrw$ is a norm violation since it ends with a write action that was not preceeded by a save since the last write action. The norm is specified by $n = (V, u)$. In Definition 5.8 we defined that a word violates a norm if it is a violation or if it contains a violation. The word $\alpha = wswrws$ violates n, because the prefix $wswrw$ of α is a violation of n. However, α itself is not a violation of n because $w \notin V$. Therefore a word does not itself need to be a violation in order to violate a norm, but it must contain a violation as a prefix in order to violate a norm.

We stress that given our representation of norm violations, and as highlighted in the last example, each norm violation is a word that violates the norm, but not every violating word is a norm violation. Next we need to define what it means for a word to be compliant with a norm, i.e. whether the norm is properly enforced through either regimentation or sanctioning. This is the case if the word does not violate the norm, or if each norm violation is immediately followed by the norm's sanction.

Definition 5.9 (Compliant Words, P_n): *Let $n = (V, \sigma)$ be a norm. The set of n-compliant words $P_n \subseteq \mathcal{A}^\infty$ is the set of words $\alpha \in \mathcal{A}^\infty$ such that $\alpha \notin V$ or $\alpha[i+1] = \sigma$ for all $i \in [1, |\alpha| - 1]$ (or $i \in [1, \infty)$ if $\alpha \in \mathcal{A}^\omega$) where $\alpha_{..i} \in V$.*

Example 5.8 (Ex. 5.7 Cont.: Compliant Words): We continue with the norm from the previous example. The n-compliant words P_n are those that do not violate n, such as $wswrsws$, or those where the sanction is applied after each

violation, such as $wswrwus$. Note that the word $wswrwus$ still violates n, but afterwards the violating write action was made undone.

Note that aside from words with correctly sanctioned violations, P_n also includes all words $\alpha \in \mathcal{A}^\infty$ that do not violate a norm n. For a norm $n = (V, \sigma)$ the set of compliant words P_n is a property. Because $\varepsilon \notin V$ it must be that $\varepsilon \in P_n$. Also, P_n is closed under $\sim_\mathcal{S}$. If this was not the case, then there would be two words $\alpha, \alpha' \in \mathcal{A}^\infty$ such that $\alpha \sim_\mathcal{S} \alpha'$, $\alpha \in P_n$ and $\alpha' \notin P_n$. That implies that α can become compliant (or α' non compliant) by only duplicating sanctions, or removing duplicate sanctions, which in either case is not possible.

5.5.1 Regimenting Controller

We shall now look at what it means for a controller to be regimenting or sanctioning for a norm. A regimenting controller for a norm prevents norm violations. Such a controller halts the system execution if it is about to violate a norm.

Definition 5.10 (Regimenting Controller): *Let n be a norm and c be a controller. c is a regimenting controller for n iff for all $\alpha \in \mathcal{A}^\infty$ it holds that $c(\alpha) = \alpha'$ where α' is the longest prefix of α that does not violate n.*

Example 5.9 (Ex. 5.7 Cont.: Regimenting Controller): Let c_r be a regimenting controller for the norm n from Example 5.7. We can have for example that $c_r(wswrws) = wswr$. The norm's violation is prevented by blocking further execution when the norm is about to be violated by the third write action. Recall that we assumed that the agent would not execute a sanction by itself, but hypothetically the word $wswrwus$ would be rewritten as $c(wswrwus) = wswr$. I.e. though the word is compliant, it still is truncated to the point where no violation occurs. Because at runtime the controller, upon reading the violating write action, does not at that point see that u follows, hence it stops execution to prevent the norm violation from occurring.

We shall now establish a connection between regimenting controllers and precise enforcement. Recall that for a norm n the set of n-compliant words P_n is not in general a safety property. Hence precisely enforcing P_n is not in general possible. Consider the norm $n = (V, u)$ from Example 5.7 and the word ww. As ww contains an unsanctioned violation it is not in P_n. However, wwu is in P_n, as all violations are properly sanctioned. Hence, an incorrect word might be extended to a correct one showing that P_n is not a safety property. The subset of P_n that contains no norm violations is however always a safety property.

Proposition 5.1: *Let n be a norm and $P \subseteq P_n$ be the set of all words not violating n. Then, P is a safety property.*

Proof. If a word $\alpha \in \mathcal{A}^*$ violates n, then there is a prefix $\alpha' \preceq \alpha$ such that $\alpha' \in V$. It is impossible to created an extension α'' of α such that α' is not a prefix of α''. Therefore, any extension of α would be violating the norm n as well and hence not be in P. □

A controller is a regimenting controller for a norm if, and only if, it precisely enforces the property that contains all words without norm violations.

Proposition 5.2: *Let c be a controller, $n = (V, \sigma)$ be a norm and $P \subseteq P_n$ be the set of all words not violating n. Then, c is a regimenting controller for n iff c precisely enforces P.*

Proof. Let $\alpha \in \mathcal{A}^\infty$ be an arbitrary word. By definition c can only revise a word α to its longest prefix $\alpha' \preceq \alpha$ that contains no violation of n. As α' has no violations of n we have that $\alpha' \in P$, hence $\forall \alpha \in \mathcal{A}^\infty : c(\alpha) \in P$, which is required for precise enforcement. If $\alpha \in P$ then all prefixes of α are in P. If a word contains no norm violations then it is its own longest correct prefix given P, and hence is mapped to itself by c. This holds for all prefixes of α, therefore: for all $\alpha' \in \mathcal{A}^\infty$ with $\alpha' \preceq \alpha$ it holds that $c(\alpha') = \alpha'$.

For the other direction, if c precisely enforces P then each word in P is mapped to itself, and hence is its own longest correct prefix. If a word is not in P then it is rewritten to its longest correct prefix, which in this case is the longest prefix such that n is not violated. This matches the definition of a regimenting controller. □

5.5.2 Sanctioning Controller

A sanctioning controller for a norm revises a word by inserting a sanction after each violation of the norm. Hence it will make any word a norm compliant one. The following definition defines this. Note that inserting the sanction causes the revised word to be longer and we therefore have to consider shifted indices in the revised word. For example, assume we have the word $\alpha = a_1a_2a_3$, the violations a_1 and a_1a_2 and the sanction σ. The sanction is inserted after a_1 and a_2 in α by a sanctioning controller. Hence then the result of revising α would be $\alpha' = a_1, \sigma, a_2, \sigma, a_3$. The index of a_3 in the revised word is the index of a_3 in α plus the number of violations that occurred before a_3, since that is how many sanctions were inserted and hence how many indices a_3 got shifted. Similarly, the sanction of violation a_1a_2 is at the index of a_2 plus one (since the sanction is inserted after the violation) plus the number of violations that occurred before a_1a_2 (in this case one, since a_1 is a violation on its own). The following definition defines these revisions for the general case.

Definition 5.11 (Sanctioning Controller): *Let $n = (V, \sigma)$ be a norm and c be a controller. c is a sanctioning controller for n iff for all $\alpha \in \mathcal{A}^\infty$ it holds that $c(\alpha)[i + 1 + x] = \sigma$ for each $i \in [1, |\alpha|]$ (or $i \in [1, \infty)$ if $\alpha \in \mathcal{A}^\omega$) such that $\alpha_{..i} \in V$ and x is the number of violations before i, and $c(\alpha)[j + x] = \alpha[j]$ for each $j \in [1, |\alpha|]$ (or $j \in [1, \infty)$ if $\alpha \in \mathcal{A}^\omega$) such that $\alpha_{..j} \notin V$ and x is the number of violations before j.*

> **Example 5.10 (Ex. 5.7 Cont.: Sanctioning Controller):** Let n be the norm from Example 5.7 and c_s be a sanctioning controller for n. A revision of c_s is $c_s(wswrws) = wswrwus$. The norm's violation is sanctioned by undoing the last write action when the violation occurred.

We shall now establish the connection between sanctioning controllers and effective enforcement. Note first that a controller that effectively enforces P_n for some norm n is not necessarily a sanctioning controller. If c effectively enforces P_n then it is allowed that for a word $\alpha \notin P_n$ it holds that $c(\alpha) = \alpha'$ where α' is the longest correct prefix of α in P_n. However, the definition of sanctioning controllers requires that the controller injects sanctions which c does not do.

A sanctioning controller can possibly duplicate sanctions if they already occur in an input word. If a norm n has a sanction σ which may be duplicated in any word in which σ occurs without changing the word in a meaningful way with respect to $\sim_{\mathcal{S}}$, then a sanctioning controller for a norm n effectively enforces P_n.

Proposition 5.3: *If c is a sanctioning controller for a norm n, then c effectively enforces P_n.*

Proof. Let $\alpha \in \mathcal{A}^\infty$ be an arbitrary word and $c(\alpha) = \alpha'$. The definition of a sanctioning controller ensures that each violation in α is followed by σ. Hence, for all $\alpha \in \mathcal{A}^\infty$ it holds that $c(\alpha) \in P_n$; the first constraint of effective enforcement. Second, if $\alpha \in P_n$ then any occurring violation of n in α is already followed by a sanction. This sanction may be duplicated by c. But the duplication of a sanction was assumed to not change the word in a meaningful way with respect to $\sim_{\mathcal{S}}$. Hence for all $\alpha \in P_n$ we have that $c(\alpha) \sim_{\mathcal{S}} \alpha$; the second constraint of effective enforcement. □

5.6 Controller Automata

We modeled controllers as mappings from words to words. We now turn to automaton-based models of controllers, called controller automata. These automata are a more detailed model of controllers that allows us in Section 5.7 to specify how controllers can be combined into collaborative controllers. There have been several works on automata based controllers for discrete event systems [110, 91, 115, 65, 59]. In particular of interest to us are the automata that are described by Ligatti et al. [90], which in turn are inspired by work initiated by Schneider in [115] and have been an inspiration for the 'enforcement monitors' from Falcone et al. [59]. These automata have been investigated for their connection to the different property enforcement types that we mentioned in Section 5.4.

5.6.1 Automaton-Based Controllers

Automaton-based controllers are labeled transitions systems where upon reading an action the automaton can make a transition to a new state whilst performing some revision of the action. These revisions are either to allow the action, suppress it, or insert a sequence of actions before it. Depending on the type of revisions that the automaton can make it can manage the enforcement of different properties [90, 22, 59]. Ligatti et al. [90] identify the following types of controllers:

- Truncation automaton. These enforcers can halt a system at any time. These automata are similar to security automata that Schneider discusses in [115].

- Suppression automaton. These may halt a system at any time, or suppress actions.
- Insertion automaton. These automata can halt the system or insert actions.
- Edit automata. These automata can halt the system, suppress actions, and insert actions.

Regarding norms we may use a truncation or suppression automaton to implement regimentation. Intuitively, if an action will result in the violation of a norm, then we can suppress it and all subsequent actions (or simply halt the system) to achieve regimentation. For sanctioning we require an insertion automaton in terms of revision capabilities. Intuitively, when a sanctioning controller sees an action that causes a violation, it inserts the sanction after the action. A technical mismatch occurs however, because insertion can only be done before the action that an insertion automaton is reading. With an edit automaton one might still achieve the intended result. Assume we want to replace any occurrence of an action a by ab. Upon reading a, but before its execution, we first insert a 'copy' of a, then we insert b, and finally suppress the a that was read.

5.6.2 Model and Operational Semantics

In the following we introduce controller automata which offer a formal tool to implement controllers. A controller automaton is essentially the same as an edit automaton introduced by Ligatti et al. [90]. The difference is of a syntactic rather than conceptual nature: they are equally expressive. Following the spirit of edit automata, controller automata are also labeled transition systems where the input alphabet is the set of target system actions $\mathcal{A}$. Such an automaton makes a transition from one state to another state by reviewing an action and performing a revision. A *revision* is given by a/X where $a \in \mathcal{A} \cup \{\varepsilon\}$ is an output action (or the empty word) to which an input action is revised and $X \in \{A, I, S, L\}$ is the name of the revision operation, where A stands for allow, I for insert, S for suppress and L for loop.

Suppose some controller automaton is given the input word $\alpha = a_1 a_2 \dots a_k$. It reviews the input from left to right starting at a_1. Then upon reviewing a_i, $1 \in \{1, ..., k\}$ if the selected revision is a_i/A, then this is read as "a_i is allowed, continue with $\alpha_{i+1..}$", ε/S is read as "a_i is suppressed, continue with $\alpha_{i+1..}$", a'/I is read as "a' is inserted in the output word, keep reviewing $\alpha_{i..}$", and ε/L is read as "do nothing, keep reviewing $\alpha_{i..}$". The minor difference to edit automata is that we allow the controller automata to make a loop (as first class citizen in the revision set). Such a loop causes the controller to do nothing: it does not allow/suppress, nor insert an action, nor move on to the next action. Clearly the loop revision does not alter in any way the rewriting capabilities of the automaton, and hence it is equally expressive as an edit automaton. We require the loop revision for the collaborative setting in the next section. We also omit the "halt" revision present in edit automata. A halt operation can be modeled by a special sink state that suppresses any action with a reflexive transition. We recall the definition of edit automata [90] with minor adjustments required for our setting. We require that upon reading an action that the

controller automaton makes a finite number of transitions before the action is either suppressed or allowed. This is a realistic assumption as a controller is supposed to be a corrective force, rather than taking over the full execution of a system.

Definition 5.12 (Controller Automaton, l): *A controller automaton is a tuple $l = (\mathcal{A}, Q, q_0, \delta)$ consisting of an input alphabet $\mathcal{A}$, a countable set of (control) states Q, an initial state $q_0 \in Q$ and a transition function $\delta : Q \times \mathcal{A} \to E \times Q$ where $E = \{a/X | a \in \mathcal{A}, X \in \{A, I\}\} \cup \{\varepsilon/S\}$ is the set of revisions. For every $a \in \mathcal{A}$ and $q \in Q$: if $\delta(q, a) = (a'/A, q')$, then $a = a'$. Moreover, we require that there is no infinite sequence of control states $q_1 q_2, \ldots$ such that $\delta(q_1, a) = (\alpha/I, q_2)$, $\delta(q_2, a) = (\alpha'/I, q_3)$, etc.*

In the following we assume, unless stated otherwise, that a controller $l = (\mathcal{A}, Q, q_0, \delta)$ is given. The operational semantics describes how the automaton behaves on an input word.

Definition 5.13 (Operational Semantics): *A configuration of l is a tuple $(\alpha, q) \in \mathcal{A}^\infty \times Q$ where α represents the input word that remains to be reviewed and q is the current control state. The operational semantics is defined by the following transition rule:*

$$\frac{\delta(q, a) = (r, q')}{(a\alpha, q) \xrightarrow{r}_l (\alpha', q')} \qquad (\textit{Controller Transition})$$

where $\alpha' = \alpha$ if $r = a/A$ or $r = \varepsilon/S$, otherwise $\alpha' = a\alpha$. We write $\mathsf{ctrl}_l(\alpha) = \alpha_1 \alpha_2 \ldots \alpha_k$ iff l can make the transitions $(\alpha, q_0) \xrightarrow{\alpha_1/X_1}_l (\alpha', q_1) \xrightarrow{\alpha_2/X_2}_l \ldots \xrightarrow{\alpha_k/X_k}_l (\varepsilon, q_k)$. Often, we also identify ctrl_l with l.

We note that the operational semantics of a controller automaton is easily extensible to the input of infinite words. If the input word is infinite then ε will never be reached as input. In that case $\mathsf{ctrl}_l(\alpha) = \alpha_1 \alpha_2 \ldots$ is the infinite concatenation of the words in the labels of transitions that are made by the automaton.

A transition $(a\alpha, q) \xrightarrow{r}_l (\alpha', q')$ represents that in state q, when action a is being reviewed, a transition to state q' takes place whilst revision r is executed, and the automaton continues reviewing the input α'. Note that for each $\alpha \in \mathcal{A}^\infty$ there is exactly one possible output word provided by $\mathsf{ctrl}_l(\alpha)$.

Proposition 5.4: *For every controller automaton l, we have that $\mathsf{ctrl}_l : \mathcal{A}^\infty \to \mathcal{A}^\infty$ is a controller.*

Proof. Recall from Definition 5.2 that "a controller is given by a function $c : \mathcal{A}^\infty \to \mathcal{A}^\infty$ such that (1) if $\alpha \in \mathcal{A}^*$ then $c(\alpha) \in \mathcal{A}^*$ and (2) if $\alpha' \preceq \alpha$ then $c(\alpha') \preceq c(\alpha)$ for all $\alpha \in \mathcal{A}^\infty, \alpha' \in \mathcal{A}^*$." From the definition and operational semantics of controller automata these conditions are fulfilled straightforwardly. We demanded that no infinite number of actions can be inserted after reading an action. This ensures that given a finite input word the output is also finite, hence fulfilling condition (1). Second, due to the deterministic nature of the automaton it must be the case that if after reading a word w the word w' is the output, that then w' is a prefix of the output word given an extension of w. □

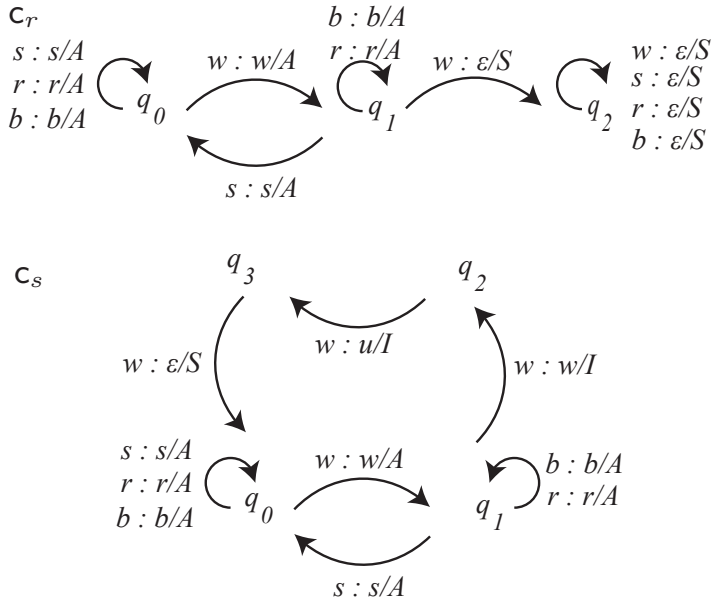

Figure 5.2: *Top: controller automaton implementing c_r from Example 5.9, bottom: controller automaton implementing c_s from Example 5.10. A transition from $q \xrightarrow{a:\alpha/X} q'$ indicates that upon reviewing action a in state q the controller transitions to q' whilst executing revision α/X.*

Given this result we will also say that "l enforces..." when meaning that "ctrl_l enforces...", etc.

Example 5.11 (Ex. 5.9, 5.10 Cont.: Controller Automata): In Figure 5.2 two example controller automata l_r (top) l_s (bottom) are shown. We have that ctrl_{l_r} and ctrl_{l_s} are the controllers c_r and c_s from Examples 5.9 and 5.10, respectively. According to definition 5.12 states q_2 and q_3 in the automaton l_s should have transitions for each action. We omitted some because a transition that inserts an action upon reviewing w will not consume w, hence the next transition must be triggered by w again. We also omitted transitions for u as we assume that the target system will not produce this action.

Consider l_r and the word ww. We defined c_r and c_s such that they enforce the norm that a save has to occur between any two writes. Hence the word ww is a violation of this norm, but w is not. Regimentation truncates a word such that given a word that violates a norm the output is the longest prefix that does not violate the norm. Hence, c_r, which is a regimenting controller, rewrites ww as follows: $c_r(ww) = w$. Given the word ww, l_r has as an initial runtime configuration (q_0, ww). The first action results in a transition towards q_1 whilst the action is allowed. Hence the second configuration is (q_1, w). The second transition goes towards q_2 and suppresses the second write. Hence the transitions are

$(q_0, ww) \xrightarrow{w/A}_{l_r} (q_1, w) \xrightarrow{\varepsilon/S}_{l_r} (q_2, \varepsilon)$, and the output is w. Note that suppressing an action and going to state q_2 equals halting the target system.

Consider l_s and the word ww. We already established that $c_s(ww) = wwu$, and shall show that this is also what l_s does. Upon initialization the controller will have the configuration (q_0, ww). Then, upon reading w it makes a transition to q_1 whilst execution the revision w/A, meaning that w is allowed as output. The next configuration then becomes (q_1, w). The next transition that l_s can make upon reading w is to q_2 whilst inserting w. This means that w is not consumed from the input word. I.e. the next configuration becomes (q_2, w). The controller automaton does not output ww yet as it is not yet finished with reading w. The next transition insert u and still does not consume w, which makes the next configuration (q_3, w). Then, the transition that follows is back to q_0, whilst performing ϵ/S, which suppresses the w action, and the new configuration becomes (q_0, ϵ). Also, because w is now fully read, the output is wu from reading w, which appended to the earlier allow revision of the first write action makes the final output wwu, as intended. In summary, upon reading ww, the transitions are $(q_0, ww) \xrightarrow{w/A}_{l_s} (q_1, w) \xrightarrow{w/I}_{l_s} (q_2, w) \xrightarrow{u/I}_{l_s} (q_3, w) \xrightarrow{\varepsilon/S}_{l_s} (q_0, \varepsilon)$, and the output is wwu.

The connection between regimenting and sanctioning controllers for norms and controller automata is that any regimenting or sanctioning controller can be implemented by a controller automaton.

Proposition 5.5: *Let n be a norm and controller c be a regimenting or sanctioning controller for n. Then, there is a controller automaton l such that* $\mathsf{ctrl}_l = c$.

Proof. Recall from Proposition 5.2 that a regimenting controller for a norm n is also a controller that precisely enforces the set of words $P \subseteq P_n$ such that P are all words that do not violate n. A controller automaton is as expressive as an edit automaton. For edit automata it is shown that they can specify precisely enforcing controllers [90]. Therefore a regimenting controller for P can be implemented by an edit automaton and hence by a controller automaton.

A controller automaton $l = (\mathcal{A}, Q, q_0, \delta)$ such that $\mathsf{ctrl}_l = c$ for a sanctioning controller c for $n = (V, \sigma)$ can be constructed as follows: (1) for each word $\alpha \in \mathcal{A}^*$ assign a new state q in Q and for ε that state is q_0, (2) for each action $a \in \mathcal{S}$ and word $\alpha \in \mathcal{A}^*$ let q and q' be the states belonging to α and αa respectively, define $\delta(q, a) = (a/A, q')$ if $\alpha a \notin V$, otherwise make two new states q_1, q_2, add them to Q, and define: $\delta(q, a) = (a/I, q_1)$, $\delta(q_1, a) = (\sigma/I, q_2)$, $\delta(q_2, a) = (\varepsilon/S, q')$. Note that by this construction Q is countably infinite. □

5.6.3 Controllers for Multiple Norms

In most normative systems there are multiple norms that the designer wants to enforce. Often norms are grouped together and put in to controlling entities such as different institutions, (e.g. [96, 46, 74, 53, 57]). It is possible with our approach to model a controller that enforces multiple norms. First we note that the sanction of the norm is an abstraction of a sanction procedure when that norm is violated. For

instance for a traffic scenario we may have a norm where all violating traces are those where an agent is speeding, and the sanction action is an abstract representation of the procedure that determines a fine for the agent. Hence if we have for instance two norms (V_1, σ_1) and (V_2, σ_2) then a new norm $(V_1 \cup V_2, \sigma_1 \cdot \sigma_2)$ can be made, where $\sigma_1 \cdot \sigma_2$ is a new sanction symbol which models sanction procedure such that if a violation occurs from V_1 then σ_1 is executed and if a violation occurs from V_2 then σ_2 is executed. The two norms can be two different speeding laws for two different velocity values where each value has a different fine attached to it. The combined sanction procedure can then pick the appropriate sanction given the velocity of a vehicle. We may then proceed to define a controller that enforces this new norm. What is not possible however, is to construct one controller that is a sanctioning controller for some norms and a regimenting controllers for others. If a controller is a regimenting controller for some norm, then by definition it will not insert actions during the revision of a word. Hence, if the same controller is also a sanctioning controller for another norm, then the insertion of the sanction would make the definition of a regimenting controller not apply to the controller. We can, however, still construct a meaningful combined controller that is the result of a combination of regimentation and sanctioning. In that case we want a controller such that all the regimented norms are not violated, and all the occurring violations of sanctioned norms are followed by the appropriate sanctions. Such a combined controller may achieve the intended result of norm enforcement of a set of norms, even though the combined controller may strictly speaking not be a regimenting or sanctioning controller for any of those norms. The rest of this section is about such combined controllers.

Let c_r be a regimenting controller[1] for a norm $n_1 = (V_1, \sigma_1)$ and c_s be a sanctioning controller for a norm $n_2 = (V_2, \sigma_2)$. If we apply these controllers consecutively on a word then no violations of n_1 will be part of the output and any violations of n_2 are sanctioned, i.e. for each word $\alpha \in \mathcal{A}^*$ we have that $c_s(c_r(\alpha)) \in P \cap P_{n_2}$, where P are all words where n_1 is not violated. Intuitively, any input that c_r gives to c_s contains no norm violations of n_1, and the only thing that c_s may do is insert sanctions after any remaining violations of n_2, which may not cause a violation of n_1 (due to our constraints on the definitions of norms).

We can combine automata models of the regimenting and sanctioning controller into a new controller automaton that models the consecutive application of the automata. Let controller automata l_r and l_s be the controller automata for c_r and c_s, respectively. We call the controller automaton model of the consecutive application of the two automata l_{rs}. As the state space for c_{rs} we take the Cartesian product of the states from c_r and c_s. We assume that a controller automaton for a regimenting controller only uses allow and suppress transitions and a controller automaton that inserts a sanction does this by first inserting the next action, then the sanction and finally suppresses the next action. Then, upon suppressing one action, l_r will remain suppressing all the other actions. If it does not suppress the action, i.e. allow it, then controller automaton l_s sees that action. If that action causes a violation for norm

[1] Recall that a controller can be regimenting or sanctioning for a norm. The specification of a sanction is part of a norm specification since whether it is regimented or sanctioned is independent of the norm itself. A regimenting controller ignores the sanction part of the norm's specification, though.

n_2, then σ_2 will be inserted after the next action. Otherwise, l_s will also allow the action.

Given a state $q_{rs} = (q_r, q_s)$ of l_{rs} and an action a we can have one of four cases. (1) l_r allows a in q_r, and l_s also allows a in q_s. In that case the next state of q_{rs} is the state where both controllers can transition to given a. l_{rs} will allow the action. (2) l_r allows a in q_r, but l_s determines that the norm is violated if a is the next action in state q_s, and is hence inserting actions. In this case l_r remains in state q_r and l_s inserts the action that it wants to insert and makes a transition. l_{rs} will insert the actions that l_s inserts. (3) l_s is finished with its insertions and will now suppress a. Note that l_s can only reach a state where it suppresses an action if l_r allows the action. Because we model the consecutive application of the controllers, l_r will make now a transition where it allowed the action, whereas l_s makes the transition where it suppresses the action. l_{rs} will suppress the action. (4) Finally, if l_r suppresses an action, then the action should be suppressed in order to avoid a norm violation. l_s does not 'see' the action so in the next state of l_{rs} l_r makes a transition and l_s does not. l_{rs} will also suppress the action.

Hence we can use the following procedure to construct a controller automaton l_{sr} that implements the controller which is the result of consecutively applying c_r and c_s on words:

1. Let $l_r = (\mathcal{A}, Q_r, q_0^r, \delta_r)$ and $l_s = (\mathcal{A}, Q_s, q_0^s, \delta_s)$.
2. Construct a new state space $Q_{rs} = Q_r \times Q_s$, and let $q_0^{rs} = (q_0^r, q_0^s)$.
3. Let δ_{rs} be defined as follows for each $(q_r, q_s) \in Q_{rs}$ and $a \in \mathcal{A}$:
   $\delta_{rs}((q_r, q_s), a) =$
   $$\begin{cases} (a/A, (q_r', q_s')) & \text{if } \delta_r(q_r, a) = (a'/A, q_r') \text{ and } \delta_s(q_s, a) = (a/A, q_s') \\ (a'/I, (q_r, q_s')) & \text{if } \delta_r(q_r, a) = (a/A, q_r') \text{ and } \delta_s(q_s, a) = (a'/I, q_s') \\ (\varepsilon/S, (q_r', q_s')) & \text{if } \delta_r(q_r, a) = (a/A, q_r') \text{ and } \delta_s(q_s, a) = (\varepsilon/S, q_s') \\ (\varepsilon/S, (q_r', q_s)) & \text{if } \delta_r(q_r, a) = (\varepsilon/S, q_r') \end{cases}$$
4. $l_{rs} = (\mathcal{A}, Q_{rs}, q_0^{rs}, \delta_{rs})$.

Example 5.12 (Ex. 5.11 Cont.: Multiple Norms): Consider the regimenting controller automaton l_r from Figure 5.2 a new sanctioning controller automaton l_s' from Figure 5.3. l_s' enforces a norm that that is violated when between a save and a write action no backup is performed, and the sanction is again the undo action. E.g. $\mathsf{ctrl}_s'(wsw) = wswu$ and $\mathsf{ctrl}_s'(wsbw) = wsbw$. Below l_s' the controller automaton $l_{rs'}$ is depicted, which is the consecutive application of l_r and l_s'. A state of $l_{rs'}$ is written as (q_i, q_j) where q_i is a state from l_r and q_j is a state from l_s'. For brevity we have omitted all states and transitions that cannot be visited from (q_0, q_0).

The word ww should be rewritten to w by $l_{rs'}$, as the first w action is allowed by both controllers, and the second will be suppressed by l_r, hence l_s' will not even 'see' the second w. In $l_{rs'}$ this is reflected by the fact that the transitions given

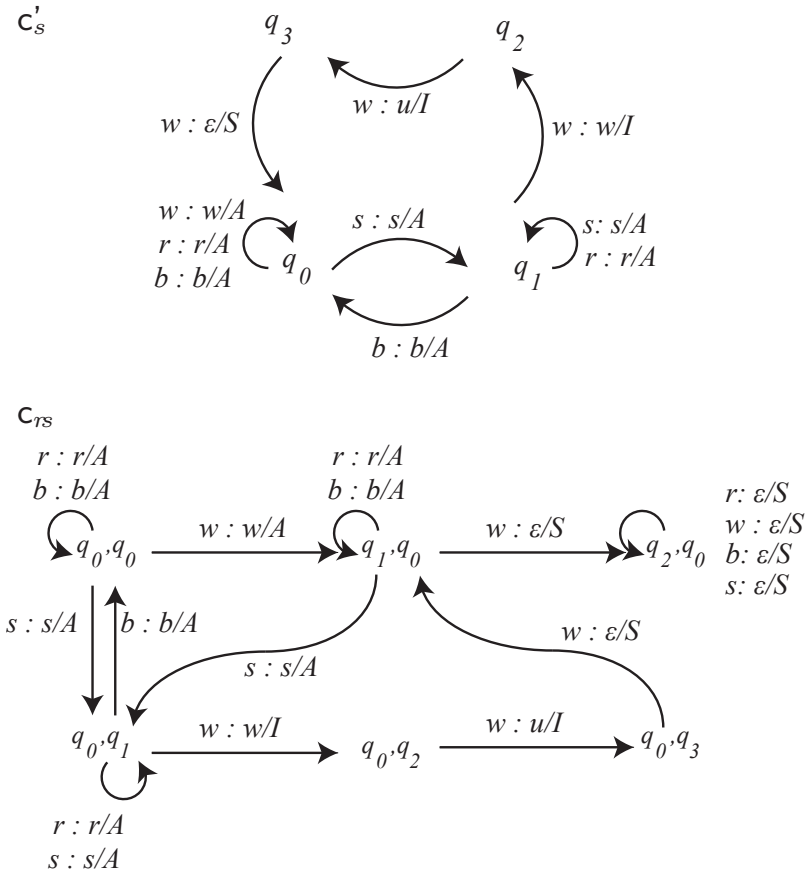

Figure 5.3: *Example sanctioning controller automaton and consecutive automaton. Description in Example 5.12.*

ww will go first from state (q_0, q_0) to (q_1, q_0) whilst allowing w, and then go to (q_2, q_0). Any action from that is read whilst in (q_2, q_0) will be suppressed, as l_r suppresses all actions when it is in state q_2.

The word wsw is rewritten by l_r to wsw, i.e. no change is made since the word fulfills the norm that l_r is enforcing through regimentation. l_r is in state q_1 after wsw is fully read. The word wsw *is* changed by l'_s, namely, it is rewritten to $wswu$, since no backup occurred between the save action and the second write action. After reading wsw the state of l'_s is q_0. Hence, the output of $l_{rs'}$ given wsw should be $wswu$ and the state that it should be in is (q_1, q_0), which is the case.

5.7 Collaborative Automata

If multiple controllers are applied concurrently on the same target system, then compatibility of controllers must be ensured. For instance, when reviewing some action, some controllers may propose to suppress that action whilst other controllers propose to allow that action. A conflict resolution mechanism should in those cases decide which of the proposed revisions is applied. Such mechanisms may take shape as design constraints on controllers or communication protocols. Each controller has its own norm for which it is a regimenting/sanctioning controller. It must be ensured that the output of the control mechanism contains only norm-compliant words, or restricts input words to words without any norm violations for any norm.

Consider again the controller automata from Example 5.11 and Figure 5.2. If they simultaneously review ww then the revisions that the top controller automaton l_r executes are first w/A and then ε/S, whereas the bottom controller automaton l_s executes w/A, w/I, u/I and then ε/S. Both controller automata agree on the initial allow revision (w/A) but then execute different revisions. For this purpose, we introduce a *selection function* that decides which controller automaton's revisions are performed. Also note that the number of revisions for controller automata may not be equal as in the example above. This happens when some of the controllers perform insert revisions and thus do not move to the next input action, whilst other automata may perform allow or suppress revisions so that they do move on to next action. We make use of loop revisions to maintain synchronization. If some automata can perform an insert revision whilst others do not, then those others have to loop until all automata are ready with insertions and can allow or suppress the action that is being reviewed. In our example, the top controller automaton l_r should be forced to loop when the second action w is under review until l_s is at state q_3 and, just like l_r, is ready to process w by an allow or suppress revision.

We approach the problem by combining concurrent controller automata to a collaborative (controller) automaton. The automaton captures all possible concurrent revision proposals for an input action. A selection function decides which of the proposed revisions to execute. The challenge is to define an appropriate selection function. It is important to keep in mind that a collaborative automaton is a model of concurrent controllers and how they might synchronize using for instance communication. A collaborative automaton is not meant as an extra entity that has to be implemented along side the individual controllers in an application.

5.7.1 Collaborative Automaton

A collaborative automaton models the collaboration between a set of concurrent controller automata. A collaborative automaton is very similar to a controller automaton. It too has a state space and transition function that given a state and action returns a label and next state. The key difference is that a transition is labeled by a vector of revisions, one for each controller automaton. In order to select a revision a selection function is required. This selection can be interpreted as the result of coordination between controllers to decide upon a revision. Hence, the mechanism is not necessarily implemented as a centralized entity alongside the controller automata with which they need to fully synchronize. Neither will it be always required that the controller automata fully synchronize with each other. For example two controller automata may never propose two conflicting revisions at the same time. In such a case no synchronization between the two controllers is required. Our framework assumes full synchronization in order to also encapsulate the worst cases where controllers always propose conflicting revisions. Section 5.7.2 discusses an example with a collaborative automaton that consists of two controllers where there are moments that the controllers need to synchronize with each other, but otherwise can run independently.

The state space and transition function of a collaborative automaton are constructed from the individual controller automata. The state space is essentially the Cartesian product of the state spaces of the controller automata where we need to duplicate the local states of each controller automaton to allow them to loop if necessary. The duplicate of state a q is denoted by $\widehat{q}$. For a set of controller automata, the collaborative state is hence a snapshot of the states in which each controller automaton is at a certain moment in time. If a controller automaton's state in a collaborative state is $\widehat{q}$ then this can be interpreted as that the controller automaton is 'on hold'. The combination of all initial states of the controller automata is the initial state of the collaborative automaton.

We say a controller automaton proposes a certain revision in the context of a collaborative state and action when given that automaton's state in the collaborative state and the action the controller automaton can make a transition with that revision as a label. We say that the revisions in a label from a collaborative transition are assigned to the controller automata. A transition label from a one state of the collaborative automaton to the next for a given action is constructed as follows:

1. If there is a controller automaton that proposes an insert, then the collaborative controller assigns to each controller automaton that proposes an insert their proposal. The other controller automata in that case must propose either an allow or suppress revision. Those controller automata are assigned loop revisions by the collaborative controller.

2. If no controller automaton proposes an insert, then all controller automata propose either an allow or suppress revision. The collaborative controller assigns to each controller automaton the revisions that they themselves propose.

The next state after a transition is determined by the label of the transition. Assume some given action. If a controller automaton is in state q in some collaborative

state, and its assigned revisions given the collaborative state and action is the loop revision, then its next state becomes $\widehat{q}$ in the next collaborative state. If, however, another revision was assigned to the controller automaton, then it must be a label of a transition that the controller automaton can make upon reading the action. The resulting state of that transition becomes the state of the controller automaton in the next collaborative state.

In the following, for a state $x \in \{q, \widehat{q}\}$ we write $\overline{x}$ to refer to q. I.e. the overline removes the hat annotation. Also recall that E is the set of possible revisions. We give the formal definition of a collaborative automaton.

Definition 5.14 (Collaborative Automaton, C): *Let $M = \{l_1, ..., l_k\}$ be a set of controller automata such that $l_i = (\mathcal{A}, Q^i, q_0^i, \delta^i)$ and $\widehat{Q}^i = Q^i \cup \{\widehat{q} \mid q \in Q^i\}$. A collaborative automaton C over M is a labeled transition system $C = (\mathcal{A}, \mathsf{Q}, \mathsf{cq}_0, \Delta, \pi)$, where $\mathsf{Q} = \widehat{Q}^1 \times ... \times \widehat{Q}^k$ is the set of collaborative states, $\mathsf{cq}_0 = (q_0^1, ..., q_0^k)$ is the initial collaborative state, and $\Delta : \mathsf{Q} \times \mathcal{A} \to E^k \times \mathsf{Q}$ is the transition function defined as follows. $\Delta((x_1, ..., x_k), a) = ((r_1, ..., r_k), (y_1, ..., y_k))$ if, and only if, it holds that:*

1. *If there is an $i \in \{1, ..., k\}$ such that $\delta^i(\overline{x_i}, a) = (a'/I, q')$ then for all $j \in \{1, ..., k\}$ it holds that:*

$$\begin{cases} r_j = a'/I, y_j = q' & \text{if } \delta^j(\overline{x_j}, a) = (a'/I, q') \\ r_j = \varepsilon/L, y_j = \widehat{\overline{x_j}} & \text{otherwise;} \end{cases}$$

2. *otherwise, $\delta^j(\overline{x_j}, a) = (r_j, y_j)$, for all $j \in \{1, ..., k\}$.*

Finally, π is a function $\mathsf{Q} \times \mathcal{A} \to E$ such that $\pi(\mathsf{cq}, a) \in \{r_1, ..., r_k\}$ where $\Delta(\mathsf{cq}, a) = ((r_1, ..., r_k), \mathsf{cq}')$. The function is called the selection function *of C.*

We note that a collaborative automaton is completely specified by a set of controllers M apart from the selection function. In the next section we shall look at different possible constructions of the selection function in case M consists of regimenting/sanctioning controllers. Next we specify how a collaborative automaton processes a word. In the following we assume that $C = (\mathcal{A}, \mathsf{Q}, \mathsf{cq}_0, \Delta, \pi)$ is given as in the definition above. As with controller automata, a collaborative automaton reads a word from left to right. A transition either moves on to the next action (in case an allow or suppress revision is selected) or keeps reading the current action (in case an insert or loop revision is selected).

Definition 5.15 (Operational Semantics): *A configuration of C is a tuple $(\alpha, \mathsf{cq}) \in \mathcal{A}^\infty \times \mathsf{Q}$ where $\alpha \in \mathcal{A}^\infty$ is the input word that remains to be reviewed and cq is the current state of C. The operational semantics of a collaborative automaton is defined by the following transition rule:*

$$\frac{\Delta(\mathsf{cq}, a) = ((r_1, \ldots, r_k), \mathsf{cq}')}{(a\alpha, \mathsf{cq}) \xrightarrow{\pi(\mathsf{cq}, a)}_C (\alpha', \mathsf{cq}')} \qquad (\textit{Col. Transition})$$

where $\alpha' = a\alpha$ if $\pi(\mathsf{cq}, a) \in \{a'/I \mid a' \in \mathcal{A}\} \cup \{\varepsilon/L\}$, and $\alpha' = \alpha$ otherwise. As before, we write $\mathsf{ctrl}_C(\alpha) = \alpha_1\alpha_2 \ldots \alpha_n$ iff C can make the transitions $(\alpha, q_0) \xrightarrow{\alpha_1/X_1}_C (\alpha', q_1) \xrightarrow{\alpha_2/X_2}_C \ldots \xrightarrow{\alpha_n/X_n}_C (\varepsilon, q_n)$. Again, we often identify ctrl_C with C.

As with controller automata we note that the definition is easily extensible to the case of infinite input words. If the input word is infinite then ε will never be reached as input. In that case $\mathsf{ctrl}_C(\alpha) = \alpha_1\alpha_2\ldots$ is the infinite concatenation of the words in the labels of transitions that are made by the automaton.

Proposition 5.6: *For every collaborative automaton C, we have that $\mathsf{ctrl}_C : \mathcal{A}^\infty \to \mathcal{A}^\infty$ is a controller.*

Proof. This follows the same line as the proof that controller automata specify controllers. Each controller automaton rewrites any finite word to another finite word. Thus, the sequence of revisions given an action and collaborative state must be finite, and hence C rewrites finite words to finite words. The collaborative automaton is deterministic, hence if w is the output given w', then necessarily w is a prefix of the output for any extension of w'. □

Example 5.13 (Ex. 5.11 Cont., Collaborative Automaton): In Figure 5.4 two controller automata l_1 and l_2 with $l_i = (\mathcal{A}, Q^i, q_0^i, \delta^i)$, are shown. Controller l_1 is controller l_s from Example 5.11 which is a sanctioning controller for $n_1 = (V_1, u)$. Controller automaton l_2 is an implementation of a sanctioning controller for the norm $n_2 = (V_2, u)$ where V_2 contains all words where a write action is not immediately preceded by a backup action. In Figure 5.4 there is also an example collaborative automaton $C = (\mathcal{A}, \mathsf{Q}, \mathsf{cq}_0, \Delta, \pi)$ over $\{l_1, l_2\}$. For a collaborative transition we use $(x_1, x_2) \xrightarrow{a:r_1r_2} (y_1, y_2)$ to indicate that a controller l_i can make a transition from x_i to y_i when reviewing an action a whilst executing r_i. It also indicates that $\Delta((x_1, x_2), a) = ((r_1, r_2), (y_1, y_2))$. The underlined revision indicates the selection made by the selection function π. From the figure we can for instance determine that $\pi((q_0, q_0), w) = w/I$. We again omit some transitions like in Example 5.11.

Observe that for any input word the output is both n_1 compliant and n_2 compliant. n_1 requires that a write action is followed by the undo action if no save occurred since the last write action. n_2 requires that each write action which was not immediately preceded by a backup is followed by the undo action. So for instance if $\alpha = w$ then immediately n_2 is violated. In C we can see that reviewing α will immediately go from the initial state through three other states such that during those transitions l_2 sanctions the violation and l_1 does nothing, but ends in a state as if it has allowed w. It can also be seen that loops and duplicate states are required for synchronization. We can see that $\mathsf{ctrl}_C(w) = wu$. If $\alpha = ww$, then after the second w both controllers detect a norm violation and want to insert the sanction. But it suffices to only once insert the sanction, which is what happens in the collaborative automaton, i.e. $\mathsf{ctrl}_C(ww) = wuwu$. Note that if the sanctions of the controllers would differ, then in this particular case we had to choose between the sanctions in the transition between state (q_1, q_0) and (q_2, q_2), which may have resulted in non-compliance of the output given the input ww.

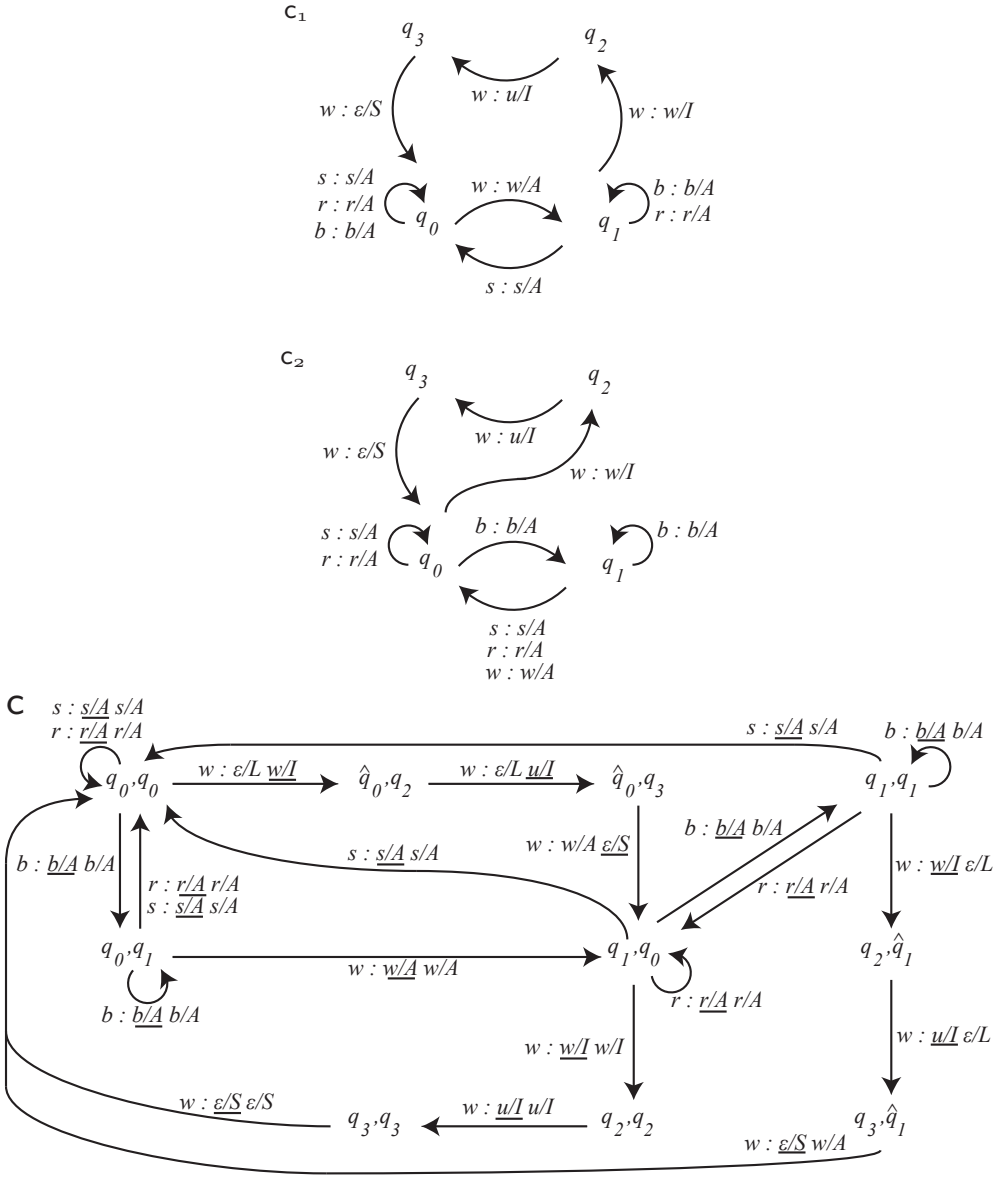

Figure 5.4: *Example controller automata and collaborative automaton. Description in Example 5.13.*

5.7.2 Collaborative Automata for Norms

If we have multiple regimenting controllers for a set of norms, then we can combine their controller automata such that the collaborative automaton will prevent any norm violation. For this purpose we first provide a proposition that explains the connection between collaborative automata and precise enforcement.

We follow Falcone et al. [59] and consider only controllers that rewrite an incorrect word to its longest correct prefix wrt. the property it enforces. Ligatti et al. [90] showed that precisely enforcing controllers can be modeled with automata that do not use insert revisions. Thus, we can assume that controller automata that specify precisely enforcing controllers do not use insert revisions. Under these constraints, we can determine a collaborative automaton, by finding an appropriate selection function, that precisely enforces the intersection of a set of precisely enforced properties.

Proposition 5.7: *Let $M = \{l_1, ..., l_k\}$ be a set of controller automata that do not perform insert revisions where l_i precisely enforces property P_i such that if $\alpha \notin P_i$ then l_i rewrites α to its longest correct prefix with respect to P_i. Then, there exists a collaborative automaton C over M such that ctrl_C precisely enforces $P_\cap = \bigcap_{i\in\{1,...,k\}} P_i$.*

Proof. Let $C = (\mathcal{A}, \mathsf{Q}, \mathsf{cq}_0, \Delta, \pi)$ be the collaborative automaton. We define the following selection function π: $\pi(\mathsf{cq}, a) = \begin{cases} \varepsilon/S & \text{if } \exists i \in \{1, \ldots, k\} : r_i = \varepsilon/S \\ a/A & \text{otherwise} \end{cases}$ where $\Delta(\mathsf{cq}, a) = ((r_1, \ldots, r_k), \mathsf{cq}')$.

For precise enforcement we have to show that for each $\alpha \in \mathcal{A}^\infty$ we have that $\mathsf{ctrl}_C(\alpha) \in P_\cap$ and if $\alpha \in P_\cap$ then we have for each $\alpha' \preceq \alpha$ that it holds that $\mathsf{ctrl}_C(\alpha') = \alpha'$.

Now, suppose $\alpha \in \mathcal{A}^\infty$. We have that $l_i(\alpha) = \alpha^i \in P_i$ for all i. Let $\alpha' = a'_1 \ldots a'_j$ be the longest common prefix of all the $l_i(\alpha)$. By precise enforcement, $\alpha' \in P_\cap$. On input α no controller automaton can suppress for the first j transitions. If $\alpha = \alpha'$ then also $\mathsf{ctrl}_C(\alpha) = \alpha' \in P_\cap$, because no suppression occurs. If $\alpha' \prec \alpha$ then some controller automaton must suppress the next input action a_{j+1}. Moreover, this automaton will keep on suppressing actions from that moment on, and therefore so will C, showing that $\mathsf{ctrl}_C(\alpha) = \alpha' \in P_\cap$. Secondly, suppose that $\alpha \in P_\cap$. Then for each i, also $\alpha \in P_i$ and thus for all $\alpha' \preceq \alpha$ it holds that $l_i(\alpha') = \alpha'$; in particular, l_i allows all actions of α'. Thus, by definition of the selection function, $\mathsf{ctrl}_C(\alpha') = \alpha'$. □

From Proposition 5.7 in combination with Proposition 5.2 it follows that a set of regimenting controllers $\{l_1, ..., l_k\}$ for the norms $n_1 = (V_1, \sigma_1), \ldots, n_k = (V_k, \sigma_k)$ can be combined to a collaborative automaton that prevents any violation of a norm n_i, $i \in \{1, ..., k\}$. Note that this is also the same as stating that the collaborative automaton specifies a regimenting controller for a norm $(V_\cup, \sigma)$, where $V_\cup = \bigcup_{i\in\{1,...,k\}} V_i$ and $\sigma \in \mathcal{S}$ is an arbitrary sanction (because sanctions play no role in regimentation). For simplicity we again assume that the controllers do not use insert revisions.

Theorem 5.1: *Let $M = \{l_1, ..., l_k\}$ be a set of controller automata which implement regimenting controllers for norms $n_1 = (V_1, \sigma_1), ..., n_k = (V_k, \sigma_k)$ and do not use insert actions, and $P_\cap = \bigcap_{i\in\{1,...,k\}} P_i$, where $P_i \subseteq P_{n_i}$ are all words that do not violate n_i.*

Then, there exists a collaborative automaton C over M such that $\mathsf{ctrl}_C(\alpha) \in P_\cap$ for each $\alpha \in \mathcal{A}^\infty$.

Proof. Recall from Proposition 5.7 that each controller l_i is precisely enforcing the property P_i. A regimenting controller l_i also rewrites a word $\alpha \in \mathcal{A}^\infty$ to its longest correct prefix given P_i.

Therefore $M = \{l_1, ..., l_k\}$ is a set of controller automata that do not perform insert revisions where l_i precisely enforces property P_i such that if $\alpha \notin P_i$ then l_i rewrites α to its longest correct prefix with respect to P_i. Following Proposition 5.7 there exists a collaborative automaton C over M such that ctrl_C precisely enforces $P_\cap$. Hence for that controller C we have $\mathsf{ctrl}_C(\alpha) \in P_\cap$ for each $\alpha \in \mathcal{A}^\infty$. □

As mentioned before, sanction actions represent sanction procedures to handle norm violations. We assume that if two procedures are not compatible for concurrent execution that then their symbols are different. For a set of sanctioning controllers we may run into a conflict if two controllers propose to insert a different sanction action at the same time. The selection function can only select one of those sanctions. This issue will not occur if for any two sanctioning controllers for norms $n_1 = (V_1, \sigma_1)$ and $n_2 = (V_2, \sigma_2)$ there is no situation where they both propose to insert a sanction (i.e. $V_1 \cap V_2 = \emptyset$) or if the sanctions are the same (i.e. are compatible procedures). If those conditions hold, then a collaborative automaton can be constructed such that each input word is revised to a n_i compliant word for each $i \in \{1, ..., k\}$.

Theorem 5.2: *Let $M = \{l_1, ..., l_k\}$ be a set of controller automata which implement sanctioning controllers for norms $n_1 = (V_1, \sigma_1), ..., n_k = (V_k, \sigma_k)$ and $P_\cap = \bigcap_{i \in \{1,...,k\}} P_{n_i}$. Then, there exists a collaborative automaton C over M such that $\mathsf{ctr}_C(\alpha) \in P_\cap$ for each $\alpha \in \mathcal{A}^\infty$ iff for each $i, j \in \{1, ..., k\}$ if $V_i \cap V_j \neq \emptyset$ then $\sigma_i = \sigma_j$.*

Proof. For simplicity we assume that a controller automaton uses a/A if possible and not the equivalent revisions a/I followed by ε/S. In that case, note that the controller automata will only propose allow revisions, unless a violation is detected upon reviewing an action a, in which case a word of revisions equivalent to a/I, σ/I, ε/S is executed, where $\sigma \in \mathcal{S}$ is a sanction. Let $C = (\mathcal{A}, \mathsf{Q}, \mathsf{cq}_0, \Delta, \pi)$ be the collaborative automaton. We define the following selection function π:

$$\pi(\mathsf{cq}, a) = \begin{cases} \varepsilon/S & \text{if } \exists i \in \{1, ..., k\} : r_i = \varepsilon/S \\ a'/I & \text{if } \exists i \in \{1, ..., k\} : r_i = a'/I \\ a/A & \textit{otherwise} \end{cases}$$

where $\Delta(a, \mathsf{cq}) = ((r_1, ..., r_k), \mathsf{cq}')$.

Consider an arbitrary word $\alpha \in P_\cap$ (such as ε), $\mathsf{cq} \in \mathsf{Q}$ such that cq is reached after reviewing α, and action $a \in \mathcal{A}$. If $\alpha a \in P_\cap$ then $\alpha a \in P_{n_i}$ for each $i \in \{1, ..., k\}$, and hence each controller will allow the action. Therefore $\pi(\alpha, a) = a/A$ and $\mathsf{ctrl}_C(\alpha a) = \alpha a \in P_\cap$. If $\alpha a \notin P_\cap$ then, given our assumptions, for each $i \in \{1, ..., k\}$ such that $\alpha a \notin P_{n_i}$ we know that l_i will sanction the violation with the same sanction $\sigma \in \mathcal{S}$. Therefore $\alpha a \sigma \in P_\cap$. These controllers will all first insert a, the others will have to loop. Hence $\pi(\mathsf{cq}, a) = a/I$. Then, all these controllers will insert σ and the others will

loop, so $\pi(\mathsf{cq}', a) = \sigma/I$, finally, all the controllers that proposed inserting the action will now suppress a and the others will propose to allow a. Therefore $\pi(\mathsf{cq}'', a) = \varepsilon/S$ and $\mathsf{ctrl}_C(\alpha a) = \alpha a \sigma \in P_\cap$. Hence for any word $\alpha \in \mathcal{A}^\infty$ if α violates some n_i then in $\mathsf{ctrl}_C(\alpha)$ the violation will be followed by σ_i. Therefore, $\mathsf{ctrl}_C(\alpha) \in P_\cap$ for any word $\alpha \in \mathcal{A}^\infty$. □

We discussed in Section 5.6.3 how controller automata could be constructed for multiple norms (by merging norms) and how automata that model regimenting and sanctioning controllers could be combined (by applying them consecutively). We can do something similar for collaborative controllers. If we have a set of regimenting and sanctioning controllers of which we want to model their concurrent application, then we may proceed by first defining a collaborative automaton for the regimenting controllers, and then by constructing a collaborative automaton for the sanctioning controllers. The resulting two automata can be applied consecutively (first the regimenting and then the sanction collaborative controller) in order to obtain both the regimenting and sanctioning aspects. The consecutive application of collaborative automata can also be modeled as a single collaborative automaton since ultimately a collaborative automaton can be represented as a controller automaton. Note that concurrently combining regimentation and sanctioning is not supported by the definition of collaborative automata. When an action is read in collaborative automata, all controllers concurrently make one or more transitions. If regimenting controllers suppress the action, then the sanctioning controllers should not make transitions as the action did not happen. However, in a collaborative automaton they do make transitions, hence it is not suitable. Therefore, upon the reveal of a new action, the regimenting controllers should be applied concurrently first and then consecutively the sanctioning controllers should be applied concurrently on the regimenting controller's output.

Finally, we note that the formal models of collaborative automata might be quite involved in some cases, but can be quite straightforwardly realized through practical communication protocols. Consider for instance the collaborative controller from Figure 5.4. The selection function shows that l_2's revisions are only selected when both controller automata are in state q_0 and the next action is w, and until l_2 reaches state q_0 again. In that same period, l_1 makes loop revisions, i.e. does nothing. This means that when reading a word, the two controllers only need to communicate with each other when they are in state q_0 and the next action is w. Other than that, l_1 can simply proceed as if l_2 does not exist, and l_2 can proceed normally except that it does not execute its revisions.

5.8 Related Approaches

Team automata [56, 121] are a related formalism to model synchronized transitions between labeled transition systems. There exists work in the field of security property enforcement that use team automata to enforce properties distributively [122]. In particular Yang et al. [144] model edit automata as component automata that can be combined into a team edit automaton. During the transition of a team edit automaton, all component edit automata's revisions (in that work these are insert, suppress

or warning) are gathered and through the team edit automaton's transition function combined into a single revision. This transition function resembles our approach in that proposed revisions have to be coordinated. How the team edit automaton's transition function can be constructed is not specified in [144]. A formal analysis of team edit automata is also not given.

Gay et al. [62] describe a monitoring system, called a service automata framework, where separate controllers called service automata work together. They are deployed to monitor decentralized systems. Their specification is given in Hoare's CSP language [71]. The authors indicate that service automata are as expressive as edit automata. A service automata framework considers networks of service automata that do not have to be fully connected. Each service automaton has a set of critical actions that it can see. A service automaton synchronizes on the target system with its critical actions. Note that in our work controller automata synchronize with all actions of the target system. If multiple service automata react to the same event, then they are required to all unanimously react to it through allow/suppress/insert. Interaction between service automata consists of sharing observations and delegating revision decisions. It is not analyzed how separately developed service automata can be combined into a service automata framework. Hence we view this type of work as a step between our models and an application, as it can model communication protocols on a more detailed level.

5.9 Conclusion

In this chapter we discussed how we can model the synchronized and concurrent application of controllers that enforce norms. For this we first turned to a related field in computer science that concerns security policies and property enforcement. A property is a special type of policy which allows us to determine for a word of actions whether or not that word violates the property. In the field of property enforcement there are different concepts which are used to define what correct enforcement of properties is (soundness and transparency constraints, captured by conservative, precise and effective enforcement). We then turned back to norms and discussed what an appropriate model for norms is in this context and how this relates to properties. We observed that norm enforcement is not the same concept as property enforcement. In particular, with norm enforcement we have the option to use regimentation, which prevents norm violations, and sanctioning, which ensures that norm violations are followed by a sanction. Hence, in the case of sanctioning, to enforce a norm does not mean that the norm cannot be violated, which differs from property enforcement. Edit automata and related approaches have been investigated as enforcement mechanisms for property enforcement. We have done the same for norm enforcement, with a syntactically different type of mechanism called controller automata. We then discussed how sets of controller automata can be combined in collaborative controllers, and how these can enforce norms through regimentation and sanctioning.

6 Object-Oriented Normative MAS

One of the main objectives in this thesis is to investigate how we may develop multi-agent systems with decentralized runtime norm enforcement to control agent behavior. In this chapter we discuss how we may develop these systems using the object-oriented programming paradigm. Various agent-based and organization-based programming languages and frameworks have been proposed to support the development of autonomous agents and norm-enforcing controllers. They have provided a valuable contribution to the identification and operationalisation of agent and organization concepts and abstractions by proposing specific programming constructs. Unfortunately, these contributions have not yet been widely adopted by industry. For logic programming-oriented norm controlled MAS we recommend using established logic-oriented agent programming languages such as 2APL [43], GOAL [70] and Jason [31] for the development of agents. However, for object-oriented programming there are limited options to implement the concepts and abstractions of these languages. Hence, unlike other chapters, we also focus here on agents alongside norms.

In this chapter, we follow the argument that multi-agent programming technology can find its way to industry by providing a methodology that guides the development of autonomous agents and norm-controlled multi-agent systems with standard programming technology. The proposed methodology explains how some characteristic concepts and abstractions related to autonomous agents and normative multi-agent systems can be implemented with an object-oriented approach. This is done by initiating a Java library of object-oriented design patterns for some characteristic but established programming constructs that have been developed in some agent-based and organization-based programming languages.

6.1 Introduction

Multi-agent system (MAS) technology aims at improving solutions for industry problems related to distributed autonomous systems. The MAS community, in particular the agent-oriented software engineering community, provides high-level (social/cognitive) concepts and abstractions to conceptualize, model, analyze, implement, and test distributed autonomous systems. The development of a multi-agent system boils down to the development of a set of autonomous agents, their organization, and the

environment with which they interact. Autonomous agents are required to make their own decisions to either achieve their objectives (proactive behavior) or to respond to their received events (reactive behavior). The agents' organization is supposed to coordinate the agents' behavior in order to ensure the overall objectives of the multi-agent system. Such an agents' organization can be developed either endogenously in the sense that agents are built to follow specification rules or protocol when they are executed, or exogenously in the sense that a third software artifact is built to orchestrate the behavior of individual agents through, for example, synchronization or by determining the effects of their external behavior. Finally, the environment encapsulates shared resources and services that can be used by the agents. These include databases, sensors and actuators, or any interface to the external world.

In the past decades, various programming languages and frameworks have been proposed to support the development of multi-agent systems [29, 30]. These programming languages have provided dedicated programming constructs (either in a declarative, imperative, or a hybrid style) to support the development of specific features of multi-agent systems. While some programming languages involve programming structures based on standard programming technologies such as Java (e.g. Jade [23], Jack™ [35], and Jadex [109]), other agent-based programming languages are specified by a newly created syntax (e.g. 2APL [43], GOAL [70] and Jason [31]). These programming languages and frameworks focus on specific sets of concepts and abstractions for which operational semantics and execution platforms are provided in some cases.

Without doubt a merit of these programming languages is the identification and operationalization of a plethora of concepts and abstractions that are required for the development of autonomous agents and multi-agent systems. For example, BDI-based agent-oriented programming languages such as 2APL, GOAL and Jason can be seen as technologies that demonstrate how an autonomous agent can be developed by means of a set of conditional plans and a decision procedure that continuously senses the environment to update its state, reasons about its state to select conditional plans, and executes the selected plans. Other programming proposals focus on the implementation of specific features concerning organizations or environments of multi-agent systems by proposing programming constructs to implement norms and sanctions, mobility, services and resources.

Although these programming languages and frameworks have contributed to the identification and operationalization of multi-agent systems, concepts and abstractions, they have not been widely adopted as standard technologies to develop large-scale industry applications. This may sound disappointing, in particular because technology transfer has been identified as a main challenge and a milestone for the multi-agent programming community. There are various reasons why these programming languages and frameworks fell short of expectations. First of all, the adoption of new technologies by industry is generally assumed to be a slow process as industry often tends to be conservative, employing known and proven technologies. Moreover, industry adopts new technologies when they can be integrated in their existing technologies, and more importantly, when they reduce their production costs, which is in this case the costs of the software development process. Finally, industry tends to see the contribution of multi-agent programming community as AI technology. The

main problems with such technologies are thought to be their theoretical purpose, scalability, and performance [45].

The general aim of this chapter is to stimulate the transfer of norm-controlled multi-agent programming technology to industry. We start with the following three observations. First, object-oriented programming languages and development frameworks have already found their way to industry. Second, it is a common practice in the object-oriented programming paradigm to use design patterns for often reoccurring problems. Third, multi-agent programming technology provides solutions to a variety of reoccurring problems in large-scale distributed applications with autonomous processes by means of dedicated programming constructs. Based on these observations and as argued in [142], we believe that multi-agent programming technology may find its way to industry by proposing an approach that guides the development of autonomous agents, norm enforcement and multi-agent systems in standard object-oriented programming technology.

The starting point of our approach is a selection of key high-level concepts and abstractions that are identified by the multi-agent programming community and for which language level constructs have been developed and operationalized in the existing well-established multi-agent programming languages. We then explain how the selected concepts and abstractions can be implemented in standard object-oriented technology to build autonomous agents and multi-agent systems. This is done by presenting an initial Java library of object-oriented design patterns for the identified concepts and abstractions. An assumption of our approach is that the selected concepts and abstractions are the best practices in the existing well-established multi-agent programming languages, which are in turn designed and developed to support applications that involve interacting autonomous systems (cf. [99]). It is thus important to emphasize that we do not claim that our proposed design patterns are directly based on the best industry practices, but indirectly via the existing well-established solutions proposed in the multi-agent programming community. We believe that proposing design patterns for the established and agreed constructs in multi-agent programming languages is a way to support and guide system developers to develop such applications in standard programming technology.

We consider multi-agent programming technology as a domain-independent and general purpose technology that aims to support the development of interacting autonomous applications. We are aware that the use of special purpose programming technologies is growing and constitutes an essential part of the programming practice in companies such as Google, Amazon and IBM. However, we see multi-agent programming technology as being concerned with specific data structures and processes that are designed to support the implementation of multi-agent system concepts such as knowledge, goals, plans, deliberation and decision making, norm, sanctions, monitoring and control. These data structures and processes can be introduced and supported by standard programming technologies such as Java, C++ and C# in order to build various applications that involve interacting autonomous systems. We would like to emphasize that there are already proposals to extend or build on existing standard programming technologies such as Java with agent related concepts in order to support the implementation of autonomous agents and multi-agent systems, e.g., Jade [23], Jack™ [35], and Jadex [109]. Our aim is not to extend or build on Java, but

rather to implement autonomous agents and multi-agents systems directly in Java.

The structure of this chapter is as follows. In Section 6.2 we provide a background in agent-oriented programming, the patterns that can be found in this field of research, and the general idea of mapping concepts and abstractions from agent programming to object-oriented programming. In Section 6.3 we discuss existing programming frameworks for norms, the background on aspect-oriented programming, and the general idea of mapping norm programming to aspect and object-oriented programming. We will then describe a library of Java classes and AspectJ aspects that directly implement these patterns in Section 6.4. Finally in Section 6.5, we discuss our approach and compare it with existing work.

6.2 Agents

In this thesis we have so far mainly considered norms. The concept of agents has not been discussed, other than that these are autonomous entities that may or may not be situated in an environment. We need to be specific of what we aim at in this chapter, because there are many different notions of agents throughout academia and everyday conversation. We are only interested in software agents. The work by Yoav Shoham is often considered to be the start of what is called the agent-oriented programming paradigm [119]. In this work he provides his own intended meaning of the word agent:

> **Quote 6.1** (Yoav Shoham [119])**:** *"An agent is an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments. These components are defined in a precise fashion, and stand in rough correspondence to their common sense counterparts. In this view, therefore, agenthood is in the mind of the programmer: What makes any hardware or software component an agent is precisely the fact that one has chosen to analyze and control it in these mental terms."*

In the spirit of this quote we may choose to analyze any program as an agent. However, Shoham argues immediately after this quote that we should not do this. By referring to John McCarthy's analysis of the ascription of mental qualities to programs in [93] he argues that agent software should have a precise underlying meaning for any of the mental components such as such as beliefs, capabilities, choices and commitments. This theory should also have a rough correspondence to common sense meanings of these words, otherwise we do not gain any insight by using these terms. By Shoham's own example, describing a lightswitch as an agent does not help us to gain insight, whereas a mental component description of a dog could provide insight in the dog [118].

Education in the agent programming paradigm has a big influence on how agent concepts are used in the research community, because new generations of researchers will gravitate towards the intuition behind concepts as they are described in the 'standard textbooks'. For agent-oriented systems two major books are "AI: A Modern Approach" [112] and "An Introduction to Multi-Agent Systems"[143]. In [112] a highly generic definition of agents is given:

> **Quote 6.2** (Stuart Russell, Peter Norvig [112]): *"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators."*

This is a more functional approach to agents with respect to Shoham's approach which was more geared towards classifying whether a program is an agent. In [112] an agent is a function from past input observations to a next output action. The kind of realization of this function is not required to match any mental components as Shoham's definition. In [143] an agent is also related to its environment, but it is emphasized that it is capable of autonomous action and is trying to achieve delegated objectives.

> **Quote 6.3** (Michael Wooldridge [143]): *"An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives."*

The two central concepts in Wooldridge's quote are autonomy and objectives. Agent autonomy is often described as that the agent need not be told explicitly what to do for each possible situation in which it might find itself. In that sense, autonomy can be achieved by generic planning procedures that given the observations of the current situation in which an agent finds itself, returns the next action or plan to perform. Autonomy is also often mentioned as the difference between agents and objects, by stating that agents may deny requests for functionality, whereas objects cannot [143]. In general, autonomy relates to the common sense notion of free will, which is also a concept that one may use to describe a system to provide insight to the system, much like the analysis of a program module by describing it as a mental component. For this it is not always necessary to provide a rigorous definition [95]. Objectives for an agent are often described as goals. Goal-oriented agents have themselves been the object of study in various works (cf. [70, 32, 43]).

Our approach towards agents builds mainly on the approach from Wooldridge, but is completely compatible with the other views as well. In our view, an agent should exhibit autonomous behavior in that no direct calls upon the agent can be made, but it is possible to trigger the agent into action. When an agent is triggered it will use a decision-making process to decide upon actions that it will execute. An agent might be triggered from external sources, e.g. events from the environment or the delegation of a goal from a user, or from internal sources, e.g. errors that occurred during execution or goals that are active in the agent. If an agent's behavior is driven by internal goals, then it will plan and execute action proactively in order to achieve its goals. Hence such an agent is often referred to as a proactive agent, whereas an agent without goals is often referred to as a reactive agent. Our own definition of an agent is as follows: *An agent is a potentially proactive decision-making process that can be triggered from either external or internal sources to execute actions.*

6.2.1 The Agent Programming Paradigm

The growing interest in autonomous and adaptive software systems and multi-agent systems requires a programming paradigm that supports their implementation. Ex-

amples of such software systems are algorithmic trading systems or software systems that are used to control autonomous vehicles, including unmanned vehicles and autonomous robots. Autonomous and adaptive software systems are assumed to be an effective solution in situations where software systems have to operate in dynamic environments and make optimal runtime decisions based on the actual situation. Such systems should be capable of generating plans to achieve their given objectives or to react to events, repair faulty plans, detect and recover from faults, and communicate with other (autonomous) systems. A popular model for autonomous agents is based on the belief-desire-intention (BDI) theory, which can be seen as a qualitative form of rational decision theory [47, 34]. Based on this theory, an autonomous software system, also called autonomous software agent, can be modeled as a cyclic process that continuously senses its environment, reasons about its state, and decides to act by selecting some plans to execute. Each cycle consists of sense, reason, and act operations. The agents can thus have autonomous behavior in the sense that they have ability to decide which actions or plans to select and execute at runtime to deal with the dynamics of their environment. An agent senses its environment by collecting new information, including received messages from other software agents, and updating its internal state accordingly. The ability to make decisions is often modeled by a set of decision rules that can be applied at runtime. A decision rule can be seen as a conditional plan/action that can be applied at runtime when its conditions are satisfied. The application of decision rules generate plans that are subsequently selected and executed.

Although the cyclic process of sense, reason and act seems simple, there are many subtle issues involved in modeling such a process. For example, updating the internal state in the sense operation can be as simple as changing some basic facts to a complex information revision process. Similarly, applying decision rules in the reasoning operation can be as simple as checking the conditions of the rule to a complex reasoning task involving reasoning about the agent's internal state, compatibility with already applied decision rules, and optimality of decisions. Finally, the execution of generated plans in the act operation can be as simple as parallel execution of generated plans to a more advanced synchronized and orchestrated execution of the generated plans. In addition to these issues, generated plans can consist of actions in the external environments such as access to a database or moving in a grid or physical world, communication actions such as send and reply actions, goal adoption actions for which the agent needs to select and execute new plans, or a combination of these actions by process composition operators such as sequencing, atomic plans, conditional choice and looping operators. The external actions in the environment can be either blocking or non-blocking. The execution of a blocking action that occurs in an agent's plan halts the further execution of the agent's plan until the action is performed in the environment, while the execution of a non-blocking action initiates a process in the environment and the execution of the agent's plan is continued. Since an agent may execute different plans simultaneously in an interleaving manner, an atomic plan can be used to ensure that the plan execution is not interleaved with any other plan execution. It is important to note that application of decision rules to generate plans is different from generating new plans from action descriptions. Reactive reasoning in the form of decision rules and its relation to classical planning is discussed in for

example [116, 64, 55].

Interaction is a central but challenging theme in the field of multi-agent systems. Autonomous agents may interact directly through messages or indirectly through the environment. Existing multi-agent programming languages provide communication actions to support message exchange. This includes send and receive actions, but also more advanced communication actions such as reply-to or forward of messages. The actual exchange of messages between software agents is often realized by means of a message transportation layer that is integrated in the platforms that are responsible for the execution of multi-agent programs. The interaction between autonomous software agents is a challenging issue as it is a source of undesirable emergent system-level properties. In order to ensure the overall desirable properties of interacting autonomous software systems, computational control and coordination mechanisms, either exogenous or endogenous to the agents, are indispensable. Multi-agent programming languages have focused on issues concerning control and coordination of interacting autonomous software systems, and have proposed various solutions for the development of control and coordination mechanisms. These proposals vary from synchronization techniques using tuple spaces to more advanced and socially inspired norm enforcement mechanisms that are the main focus of this thesis [103, 36, 46, 75, 73, 138, 28]. Synchronization through tuple spaces allow software agents to run in a coordinated manner, while the idea of norm enforcement is to ensure that the execution of software agents does not violate some desired behaviors specified as norms. It should be noted that existing object-oriented technologies such as Jini in Java can directly be used to support the implementation tuple spaces for coordination purposes.

The idea of norm enforcement is quite similar to the idea of runtime software verification where system executions are monitored at runtime to check whether system requirements (comparable to norms) are due to be violated (Chapters 5 and 3, [59, 115, 19]) and to execute appropriate measures when violations take place. In multi-agent programming languages, the idea of adopting a norm enforcement mechanism is to continuously monitor the agents' behavior and intervene by means of regimentation or sanctioning when norms are due to be violated [7, 8]. Norms can be state-based specifying that certain states are obliged or prohibited, action-based specifying that certain actions are obliged or prohibited, or they can be temporal specifying sequences of actions and states that are obliged or prohibited. Norms can also be conditional and come with deadlines in the sense that when the condition of the norm is satisfied, a concrete obligation or prohibition with a deadline is created. The created obligation or prohibition with a deadline is often called the detached norm, and the creation process itself is called norm detachment. In these proposals, norm monitoring boils down to checking the condition, content, or deadline of norms, which may be either some state formula, actions, or a temporal formula. We conceive these concepts and abstractions as the best practices suggested by the multi-agent programming and runtime verification communities to tackle coordination and regulation challenges in applications that involve interacting autonomous systems.

Existing programming languages for software agents and multi-agent systems provide a variety of constructs to support the implementation of a selection of these issues. Each proposal suggests a specific solution for the selected issues that are often

integrated in the operational semantics and the execution platforms of the proposed programming languages. Some of the programming languages such as 2APL, Jack™, Jason, GOAL and Jadex are specifically designed for programming autonomous agents that are based on rational decision theory, while others such as Jade, Brahms, Golog, Impact, CLAIM and METATEM are designed based on other paradigms such as behavior- or activity-based agents, planning, legacy code, mobility, and execution of logical specifications, respectively. A comprehensive discussion and analysis of these programming languages are out of the scope of this thesis, but can be found elsewhere, e.g., in [44].

The focus of this chapter is on autonomous and adaptive software agents based on BDI theory and control mechanisms based on norm enforcement. These are considered as software solutions for developing interacting autonomous systems that are designed to operate in dynamic environments and have to make runtime decisions for a variety of issues. We present object-oriented design patterns to support the implementation of autonomous and adaptive software agents and norm enforcement mechanisms to control and coordinate the behaviour of software agents. In order to guide system developers to exploit such solutions in standard programming technologies, we will present them as design patterns using the common format from Gamma et al. [61]. We are aware that design patterns should formalize the best industry practices. However, although we do not have access to industry practices regarding the development of interacting autonomous systems, we assume that the commonly agreed upon concepts and abstractions within the multi-agent programming community do reflect proper solutions to reoccurring problems in applications that involve interacting autonomous systems. These concepts and abstractions are established based on peer reviewed approaches and are believed to facilitate the development of multi-agent systems. In line with Gamma et al. [61], the examples throughout this chapter illustrate how an application of the patterns might look. They are not meant to compare the solutions resulting from the patterns with other possible solutions such as actor technology.

We shall draw our examples from a multi-agent system that we are developing for the Dutch railway organization (ProRail) in the context of simulations for smart infrastructures. The first step is to build a prototype simulation for a simple cargo transportation scenario where containers from the Rotterdam harbor should be transported by trains to different cities in Europe. The Dutch railway organization is interested in various ways to automate and optimize the transportation of containers by freight trains. We are developing a multi-agent simulation of a smart infrastructure where the transportation of cargo is managed by a set of auctions to allocate rail trajectories to trains and to allocate containers to trains. These auctions are set by different stakeholders such as the railway organization, the train companies, and the cargo companies. For this simulation scenario, we model the users as agents that buy and sell commodities such as slots on the railway (railway trajectory for a specific time), or a place on a specific train with an assigned railway slot. The railway agent organizes every day an auction to allocate railway slots to train agents. Subsequently, each train agent with a specific allocated railway slot organizes an auction for transporting containers. For this simulation, every agent has multiple interfaces to the simulation platform which expose functionalities such as organizing or bidding in an auction. Sometimes we want to restrict the use of these interfaces by imposing norms

on their usage. For instance an agent that organizes a Vickrey auction ought to notify participants whether their bid was rejected or accepted before the auction is closed. For our scenario we use Java to implement agents and AspectJ to implement norms. We show how the design patterns can be used to implement agents and norms.

6.2.2 Agent-Oriented Programming Languages

We first discuss some existing BDI-based agent programming languages that support the implementation of autonomous and adaptive software agents, in order to get a feel of the patterns that occur in these language designs. In particular, we focus on 2APL, GOAL and Jason as they provide dedicated programming constructs to directly implement decision rules, and moreover, because they come with operational semantics. The existence of operational semantics is crucial since they specify the exact operational meaning of their constructs and determine the behavior of corresponding programs. The operational semantics can guide us to propose design patterns for generating similar behavior. In these three programming languages a decision rule can be programmed by means of conditional plans. These are plans that are only applied if they are relevant and applicable to the agent's current situation. As an example, we shall show how in 2APL, GOAL and Jason an agent can be specified to react to the event that a new Vickrey auction is organized. In this example, such an agent has some information about its own budget that constitutes its beliefs. In both 2APL and GOAL the agent can also have an objective to own an item that constitutes its goals. However, as Jason does not provide an explicit construct for goals, the item to be owned by the agent needs to be specified as its belief. The agent has also plans to participate in auctions, which are often realized by a procedure to submit a bid. The plan that we show is the plan to decide on the bid that the agent wants to make. The condition of this plan is a check whether the agent actually wants the offered item, and whether its budget is above the minimum price. If both are true, then the agent will look at what it believes that the item is worth, and bid that amount, as this is the dominant strategy in a Vickrey auction.

For the 2APL example we have the following agent specification.

```
1  beliefs:
2    budget(slot4502,150).
3
4  goals:
5    own(slot4502).
6
7  message(Auctioneer, vickrey(Item, MinimumPrice)) <-
8    B( budget(Item, Value) and Value > MinimumPrice )
9    and G( own(Item) ) |
10   {
11     send(Auctioneer, bid(Item, Value));
12   }
```

In this 2APL program, lines 1-2 implement the agent's beliefs, i.e., the information the agent has about its budget for a railway slot, and lines 4-5 implement the agent's goal, i.e., to own a railway slot. Finally, lines 7-12 implement the agent's decision

rule, which prescribes to send a bid to the auctioneer for the item whenever the agent receives a message from an auctioneer informing it about a Vickrey auction with an item for a minimum price, and if the agent can afford the item and wants the item.

For the GOAL example we have the following agent program.

```
1  beliefs{
2    budget(slot4502,150).
3     }
4
5  goals{
6     own(slot4502).
7     }
8
9  if a-goal(own(Item)),
10    bel( percept(vickrey(Item, MinimumPrice, Auctioneer)),
11         value(Item, Value), Value >= MinimumPrice )
12 then do bid(Auctioneer, Item, Value).
```

In this GOAL program, lines 1-3 implement the information the agent has about its budget for a railway slot, while lines 5-7 implement the agent's goal, i.e., to own a specific railway slot. Finally, lines 9-12 implement the agent's decision rule that prescribes to bid for an item (in this case a railway slot) whenever the agent wants to own it, has perceived that the auction for that item is organized, and believes the item is affordable and worth at least the minimum price.

And for the Jason we have a highly similar agent program.

```
1 budget(slot4502,150).
2 want(slot4502).
3
4 +!vickrey(Item, MinimumPrice, Auctioneer) :
5   want(Item) & budget(Item, Value) & Value >= MinimumPrice <-
6     .send(Auctioneer, tell, bid(Item, Value)).
```

In this Jason program, line 1 implements the information that the agent has about its budget for a railway slot, and line 2 implements the agent's goal as belief, i.e., the agent believes it wants to own a specific railway slot. Lines 4-6 implement the agent's decision rule, which prescribes to send a bid to the auctioneer for the item whenever the agent receives an event from an auctioneer informing it about a Vickrey auction with an item for a minimum price, and if the agent believes it wants the item and can afford it.

All three languages follow a similar pattern of an agent specification consisting of information available to the agents (beliefs), the objectives (goals) or events (messages) that triggers agents to initiate actions, and a set of plans that prescribe which actions to perform to achieve its objectives or to respond to events. Each plan consists of a condition and a specification of the action/plan that has to be executed.

As noted, there are other BDI-based agent programming languages that are based on Java, e.g., Jadex [109] and Jack™ [35]. These programming languages do not come with operational semantics, but the general idea of their constructs are explained informally. Jadex, as proposed by Pokahr et al. [109], builds on Jade and extends it with programming constructs to implement BDI concepts such as beliefs, goals,

plans, and events. It uses XML notation to define and declare an agent's BDI ingredients and Java constructs to implement the agent's plans. Jack™, as presented by Winikoff [35], extends Java with programming constructs to implement BDI concepts. In both Jack™ and Jadex a number of syntactic constructs are added to Java to allow programmers to declare belief sets, to post events, and to select and execute plans. The execution of agent programs in both languages are motivated by the classical sense-reason-act cycle, i.e., processing events, selecting relevant and applicable plans, and execute applicable plans. Jadex and Jack™ come with integrated development environments.

6.2.3 Object-Oriented Agent Programming

We make a shift between concepts related to the specification of agents and concepts related to executing agent specifications. By agent specification concepts we refer to the elements of an agent that are described in for instance a 2APL, GOAL or Jason agent program file. By concepts relating to agent execution we refer to an agent's deliberation cycle and the platform on which an agent is situated.

In the object-oriented programming paradigm, an agent specification will be implemented by means of various classes. A belief base provides an agent with information that it may use during decision making. We capture an agent's belief by classes that are called *contexts*. Such contexts are not exactly the same as a belief base, but are intended to provide all necessary data retrieval for decision making and exposure of interfaces for performing actions. In addition, triggers implement events and goals which influence the selection and execution of plans. Triggers can be any object that implements the trigger interface. Goals however are a special type of trigger as these must implement a method to determine whether the goal is achieved. Finally, the so-called plan scheme classes implement procedures for both reactive and proactive behavior. The relevance and applicability checks of plan schemes are captured by a condition defined on triggers and the internal state of the agent including its context.

In object-oriented patterns such as the *strategy pattern* [61] a context class is utilized that can be used to provide information to strategies that implement some delegated functionality. We thus take this approach by describing a context class. A context serves as a belief base because it aggregates all relevant objects that carry information and are required for decision making. In many agent languages such as 2APL, GOAL and Jason the belief base is a logic-based reasoning engine such as Prolog. A context in the object-oriented representation of a belief base is not a logic engine, but a plain Java class. If the developer of an agent desires to perform reasoning with a Prolog engine, then the programmer can instantiate a Prolog engine and add it to the context as an attribute. For instance tuProlog and SWI-Prolog (with the JPL library) are easy to integrate in a Java program. During the execution of a plan the agent may retrieve the context with the Prolog engine and query it. It is important to note that because the context is an object, it may refer to, and hence use, other objects that are design-wise 'outside' of the agent. For instance, an agent in a market may directly call upon an object that is considered to be the environment in order to obtain its current credit status. We believe this direct access to environment data can in some scenarios be crucial for the multi-agent system

performance. For instance in a large scale multi-agent system it would be undesirable if an agent duplicates environment data which may be quite large.

The plan rules in an agent programming language specify when a plan is triggered, under which context condition it is applicable, and the plan itself that is created upon the triggering and application of the rule. Usually a plan is triggered by some event or goal. We capture a trigger as its own class which may be further extended to represent different kinds of triggers such as external events and messages. In object-oriented programming, a plan rule will be implemented by a class named plan scheme with a method to check for a given trigger and context whether that plan is applicable. If so, then the applicability check must return the plan that is to be executed. The returned plan is instantiated with the actual information from the triggering event and the context. Plans are implementations of a plan interface that requires at least one method to execute the plan. By default our library removes a trigger if no applicable plan scheme for the trigger is found, if that trigger represents an external/internal event or a message. Our library provides a special type of persistent trigger which we use to capture the concept of goals. A goal is considered as an event that continues to trigger plans until its associated condition is satisfied.

Hence the core classes that specify an agent are the context, plan schemes, plans and possible triggers. However, these classes alone are not enough to specify and execute an agent. In agent programming languages the execution of an agent program is controlled by its deliberation cycle, which in turn consists of various deliberation steps. We introduce various classes for the deliberation steps. In addition, a runtime configuration of the agent is required which maintains all references to objects that are required for executing the agent. These include the currently received but not processed triggers, the active goals, the context, the available plan schemes (put in a plan scheme base), the current plans, and the deliberation cycle (which is a list of deliberation steps). Finally a class named deliberation runnable is described which is a runnable to execute a single deliberation cycle. At the end of the cycle the runnable will check whether the cycle should be executed again if there are still active plans or goals, and otherwise will put the agent to sleep. If the agent receives external input whilst sleeping, then its deliberation runnable will be executed again.

In our developed library we have pre-programmed the execution components for agents. We implemented the deliberation steps that apply plan schemes on the different types of triggers that an agent might have. These steps are used for the deliberation steps that involve proactive behavior by applying plan schemes on goals, reactive behavior by applying plan schemes on received messages and external triggers, and finally self-healing behavior (i.e., the ability to cope with plan failures) by applying plan schemes on internal triggers. A plan scheme instantiates a plan if it is successfully applied. We also provide a deliberation step that executes the instantiated plans. In order to manage the multi-agent system we also preprogrammed a platform class that allows an administrator to instantiate new agents or kill existing ones. For this platform it is required that a developer implements an agent scheme. This scheme, in the spirit of the *factory pattern* [61], produces instantiations of classes that are required to create and execute an individual agent (which are mainly its context and plan schemes). The platform is at its core a standard Java executor service with some bookkeeping capabilities to maintain lists of existing agents. It is not required that

the platform is used to execute an agent, an agent might also be embedded in a Java project by for instance assigning it its own thread.

6.3 Norm-Oriented Programming

Throughout this thesis we discuss various norm-oriented frameworks for modeling and programming norms. In this section we will first show some examples of norm programming in order to highlight the pattern in these language designs. We then provide a quick overview of aspect-oriented programming as the reader needs to be familiar with its key concepts. We then discuss the translation of norms programming to aspect/object-oriented programming.

6.3.1 Norm Programming Languages

As an example norm we shall take that agents that organize auctions must notify the winner and losers of auctions before the auctions are closed. If the norm is violated then the organizer is blacklisted. The norm's implementation using the proposed patterns is given in section 6.4.2. In 2OPL [46] this norm could be implemented with the following norm/sanction constructs:

```
1 notify_participants(Organiser):
2     <started_auction(Organiser, Auction),
3      O(notifiedResultParticipants(Auction)),
4      closed(Auction)>
5
6 viol(notify_participants(Organiser)) => blacklisted(Organiser).
```

In this 2OPL program, line 1 is the norm label, lines 2-4 are the condition, deontic content (O stands for obligation) and deadline of the norm, respectively, line 6 implements the sanction rule that will be imposed when the norm is violated. In this case, the sanction is to blacklist the agent.

In NPL [73] we could program the norm as follows:

```
1 norm notify_participants:
2   started_auction(Organiser, Auction) ->
3     obligation(Organiser, notify_participants,
4         notifiedResultParticipants(Auction), 'now').
5 norm notify_participants_violated:
6   started(Auction, Organiser), closed(Auction,Organiser),
7   not notifiedResultParticipants(Auction) ->
8     obligation(controllAgent, notify_participants_violated,
9         blacklisted(Organiser), 'now').
```

In this NPL program, lines 1 and 5 are the norm labels, lines 2 and 6-7 specify the conditions of the norms, in this case the condition that the auction is started and the condition that the auction is closed without the participants being notified. Finally, lines 3-4 and 8-9 specify that the organizer is obliged to notify the participants, and that some controller agent is obliged to blacklist the organizer if it did not notify the auction participants.

The norm could be programmed as follows in the norm language that is proposed by Vázquez-Salceda et al. [138]:

```
1 Norm condition: OBLIGED(notifiedResultParticipants(Auction))
2                 IF started_auction(Auction, Organiser)
3 Violation condition: closed(Auction) AND NOT
4                      notifiedResultParticipants(Auction)
5 Detection mechanism: (omitted)
6 Sanction: (omitted)
7 Repairs: blacklisted(Organiser)
```

In this program, lines 1-2 specify the obligation to notify the participants of the auction results, lines 3-4 specify the violation condition, i.e., the auction is closed and the participants are not notified of the result, lines 5-6 are omitted fields for specific violation detection and sanction mechanism, and finally line 7 specifies how a norm violation can be repaired, in this case by blacklisting the organizer.

The main pattern that we discern from these examples is that a norm is programmed by specifying the condition under which the norm is detached, the condition that has to be fulfilled/avoided and the consequences of a norm violation. The conditions are based on the state of the environment or the actions that agents perform. Hence, the implementation of a norm can only take place if the environment is already designed/implemented. Also, if the environment changes, then the norm implementation may also change. However, the norm specification is not part of the environment's business logic.

6.3.2 A Primer on Aspect-Oriented Programming

Aspect-oriented Programming is a relatively new extension to the object-oriented programming paradigm. The most mature aspect programming framework is AspectJ, which is specified for Java and also our choice in this chapter to specify example code for norms. In this section we shall provide a quick overview of the most important terms and technicalities of aspect-oriented programming.

Consider the interface of an agent to an auction system. The functionality of this interface is to provide the agent the possibility to initiate auctions, provide bids, publish auction results, etc. However, often the business logic of an interface is accompanied by other functionalities such as code for debugging and, in our target domain, norms. Often these other functionalities form together a coherent *cross cutting concern* of an application. Typically a cross-cutting concern can only be implemented in an object-oriented application if through different classes the code for this concern is provided in the business logic of these classes. Aspect-oriented programming aims at concentrating the code for such concerns in what are called aspects. Aspects are programmed alongside the classes of a system. The code for a cross cutting concern in an aspect is added to the classes of the system during compilation time. The addition can be done either by adding lines of code in the source before actual compilation, or in the case of Java in the byte code. The process of adding the code is called weaving. For our purposes it is important to note that an aspect has to be specified before compile time. Hence, at runtime it is not possible to introduce new aspects unless the affected classes can be recompiled on the fly.

Every object-oriented program exposes many points of executions which are called *join points*. Typically these are the execution of a method or the update of a variable. In an aspect it is possible to specify a selection function on join points. For instance, a selection might occur on all methods that are public and have no arguments. The selection specification of join points is called a *pointcut*. During execution a pointcut might be activated. An *advice* specifies what happens when a pointcut is activated. The advice is usually a regular piece of code that, without aspects, would be included in the code where the pointcut of that advice is activated. An advice can be specified to execute code before the pointcut's flow of control is continued, afterwards, or around it.

In a conditionalized norm we have to specify a condition, obligation/prohibition and deadline. This specification often consists of either a query on the state of the environment or an action that conforms to some arguments. Actions in an object-oriented MAS are similar to method calls in an environment interface, and hence can be captured quite intuitively by specifying pointcuts for those join points that are execution calls of environment interface methods. However, we note that also for state-based norms there exists possibilities to implement them with aspects. If a variable changes, then this is also a join point that can be selected by a pointcut. For instance, assume we have some interface that contains a credit counter and a limit on how negative an agent might be. If we want to check whether the limit is always lower than the credit, then we could define a pointcut either by a) selecting all join points of method calls that may alter either variable, or b) selecting all join points where either the credit or the limit is altered. Using these pointcuts we can check the state of the interface only when it is relevant.

6.3.3 From Norms to Aspects, Objects and Classes

Normative constructs are prevalent among organizational coordination frameworks for multi-agent systems. Norm-based regulation mechanisms can be introduced using existing technologies such as aspect-oriented programming. Aspects allow crosscutting concerns to be programmed separately from the system's core business logic. Though unusual, norm-based regulation mechanisms can be presented as design patterns based on aspects. The key correspondence is to use pointcuts from aspect-oriented programming to specify where a norm applies (norm condition), and pointcut advices to check if a norm is violated and how to react to this violation (deadlines and sanctions). The use of aspects may raise concerns about the open nature of multi-agent systems because the source code of the target processes is required in order to program aspects and weave them in the processes at compile time. However, various works promote the use of organizational interfaces for the interaction between agents and the environment, or between agents themselves (cf. controllers in [96] and OrgBoxes in [75]). In such methods it is feasible that the norms are separately developed and maintained from the business logic of the interfaces to the environment and the agents that use those interfaces.

Organizations and institutions are often specified in terms of concepts such as roles, social empowerment and norms. We focus on norms and ignore other concepts such as roles and social empowerment. We believe that norms are the basic concept

underlying higher-level concepts such as roles or social empowerment. For example, a role can be specified in terms of a set of norms that should be respected by the agent playing the role and social empowerment can be implemented by means of the roles that an agent can play. In particular, we focus on exogenous norms, which are explicitly programmed norms that have to be monitored and enforced on a multi-agent system. An exogenous norm specifies desirable behavior for agents which is not incorporated inside of the agent specification. This type of norm is particularly useful for controlling multi-agent systems where the source code of the software agents might not be available. Also, we discuss conditional obligations and prohibitions, as these are common types of norms [46, 75, 51]. Note that permission is specified in deontic logic as the negation of an obligation or prohibition. However, since we would like to operationalize norm enforcement for coordination purposes we ignore permissions and focus on atomic obligations and prohibitions.

We assume that the minimal point of control that a multi-agent system developer has, is the environment interfaces from agents. In particular, we assume that a multi-agent developer can exercise his/her control on software agents by authorizing and realizing the effect of their actions in the environment. Our approach to enforcing norms is thus to implement the norms in the environment interfaces. Thus, whether the environment source code is given or not, one has always to implement the interface to the environment that agents use. Note that this is in line with many norm programming frameworks where an institution or organization mechanism observes the agents' actions and/or environment states to detect norm violations, and intervenes by authorizing and effectuating the agents' actions in the environment [46, 96]. The core business logic of such interfaces is the requests from agents for performing actions and relay them to the environment with specific effect. Hence a desideratum is that norms are developed as a separate concern. The environment interface should not change if the norms change. To realize a separation of concerns we use aspect-oriented programming. At compile time the norms that are specified in aspects will be woven into the environment interfaces. An aspect that implements a norm specifies pointcuts for the condition of a norm, the obligation or prohibition, and the deadline. The advices for these pointcuts are related to detaching the norm (creating an object that represents an active obligation/prohibition) and removing a norm detachment[1], possibly followed by a sanction. A norm detachment, which represents an active obligation/prohibition, has its own class, and instances thereof indicate that the norm's condition was met sometime in the past. We also include the publication of detachments in case agents might be norm-aware in the sense that the agents can process the published detached norms and incorporate them in their decision-making deliberation [6].

In general, norms can be enforced through regimentation and sanctioning [48, 7]. For regimentation it is necessary that a norm's violation will not occur. A violation of an obligation can only occur when the deadline is met. Hence, we should realize a forward looking mechanism that checks whether the deadline is about to be reached, and if so, also checks whether the obligation is fulfilled. In the case that the obligation is not fulfilled, either the deadline should be prevented or, if this is not

[1] In the case of Java 'removing a detachment' means that no references are made anymore to the object that represents an active obligation/prohibition.

possible, the system execution should be halted. A violation of a prohibition can only occur when the prohibited action/state condition is about to be met. Hence, if the prohibited condition is about to be met then the action/state change that causes this has to be nullified. This is a difficult issue to realize in general for object-oriented programming since it is not possible to for instance remove a method call from the execution stack as if it never occurred. As we will demonstrate in Section 6.4.2, we propose that a regimented obligation's deadline or regimented prohibition condition is only specified on methods that do not have a return value. This way we can utilize the around advice of aspect-oriented programming to circumvent the execution of a norm-violating method call. For sanctioning it is required that upon the violation of a norm the sanction is executed. We model the sanctioning mechanism through advices from aspect-oriented programming and implement sanctions as method calls. It is up to the developer whether the sanction has immediate effect (as for instance in [46]) or whether the sanction is to send an obligation to another agent to handle the actual sanctioning (as for instance in $\mathcal{S} - \mathcal{M}$OISE [75]).

6.4 Design Patterns for Autonomous Agents and Norms

For the description of the patterns we use the widely accepted approach from Gamma et al. [61]. Their methodology to describing a pattern is to document the following items. The *name and classification* of a pattern describe in a few words what the pattern is about. The *intent* behind the pattern describes what the pattern tries to achieve and why it should be used. The *motivation* of a pattern consists of a natural problem example for which the pattern is suitable. The *applicability* of the pattern describes the typical problem symptoms for which the pattern is intended. The *structure* is a graphical display of the top-level concepts or classes. The *participants* description is a textual description of the most important concepts. The *collaboration* description describes how participants interact. The *consequences* describe how the pattern achieves its intent. *Implementation* and *sample code* describes how the pattern should be applied. *Known uses* are existing solutions which use the pattern. *Related patterns* are those that are strongly related to the documented pattern.

6.4.1 Object-Oriented Agent Pattern

In the rest of this section we will document the *object-oriented agent* pattern that can be used to program agents.

Name and Classification

The *object-oriented agent* pattern is a concurrency pattern. This design pattern has no other known names.

Intent

The intent of the *object-oriented agent* pattern is to separate the business logic of decision making and processing of triggers from the rest of an application. The

described pattern promotes modular development of autonomous behavior. It also provides a clear separation between the specification of autonomous behavior and the execution of this specification. Finally the pattern supports concurrency in terms of concurrent decision making.

Motivation

Consider a simulation of a real world process that involves many decision-making entities such as humans and software systems. During the development of such a simulation new sensors and actuators may become available to these entities and existing ones may change. If the decision making business logic is scattered across the project then it is hard to maintain the code if the available sensors and actuators change. Additionally, it often becomes hard re-use decision making functionalities if there is no generic methodology used.

Modeling these entities as agents using the *object-oriented agent* pattern allows developers to concentrate all the decision making in a few classes that expose exactly what can be observed, what actions can be made, and how the connection between observations and action is realized. These classes can be stored and extended from a library so that the decision making can be reused for another project.

Applicability

The *object-oriented agent* pattern is best used when:

- There are multiple triggers that have to be processed by the same entity. The pattern supports the implementation of the decision making logic by specifying what plans needs to be executed for which triggers.
- The business logic for decision making can change over time. For instance new information and/or action possibilities can emerge while the project develops.
- The autonomous entities are fairly heavyweight. For lightweight concurrent processes one should consider for instance actor-based programming languages (cf. [67]).
- The application's high level concepts match with the agent paradigm. The pattern is inspired by the BDI paradigm.
- Proactiveness is required. The pattern supports goal-based execution.

Structure

The structure of the *object-oriented agent* pattern is shown in Figure 6.1.

Participants

The participants are as follows:

- **Trigger.** An object that represents an event, message or goal to be processed.

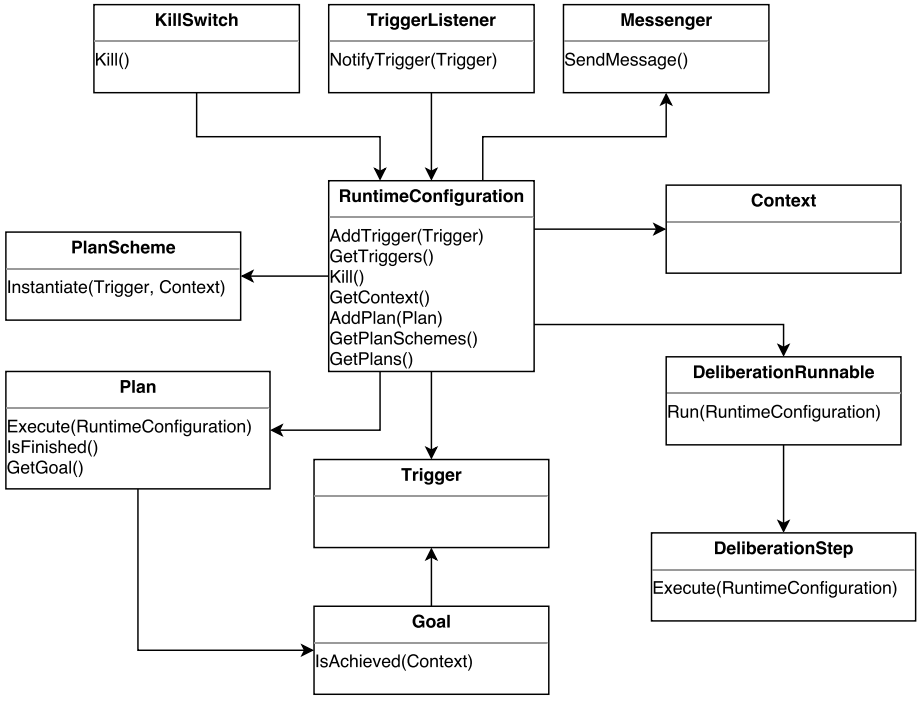

Figure 6.1: *Structure of the elements that form an agent.*

- **Goal.** Persistent trigger that represents a goal of the agent.
- **Plan.** Specifies the business logic for processing a trigger.
- **PlanScheme.** Specifies when a plan is applicable.
- **Context.** Exposes all required information for determining whether a goal is achieved and the execution of plans.
- **Messenger.** Allows the agent to send messages to other agents.
- **DeliberationStep.** Selects and/or executes relevant plans.
- **DeliberationRunnable.** Contains a run method that executes a sequence of deliberation steps (called deliberation cycle).
- **RuntimeConfiguration.** This class is the main data container for a single agent.
- **TriggerListener** Exposes the functionality to add to the agent triggers (messages, events, percepts, etc.).
- **KillSwitch.** Contains a method to cause the agent to stop executing.

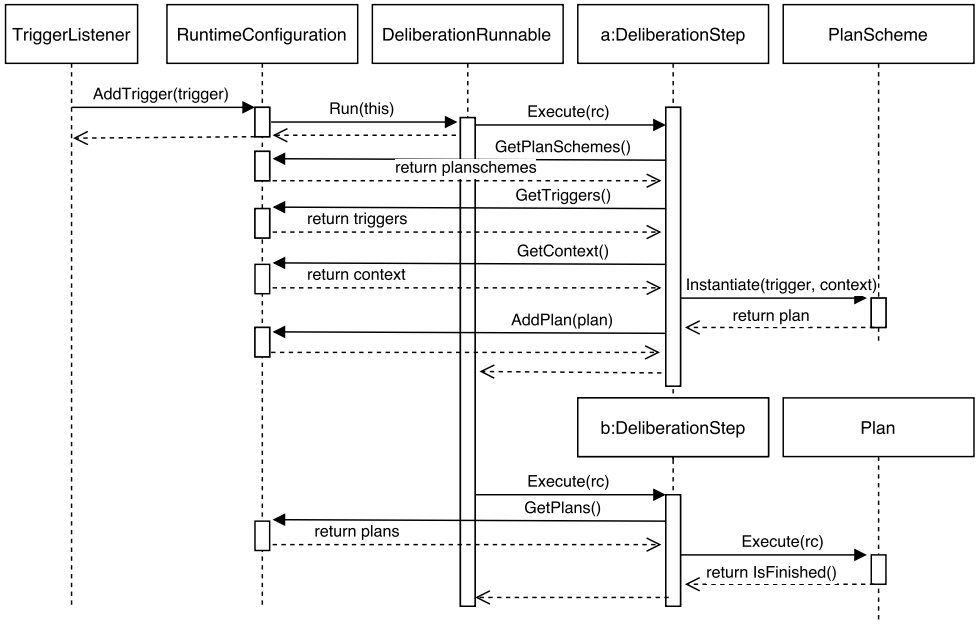

Figure 6.2: *The core interactions of the agent components.*

Collaborations

The core component of an agent is its runtime configuration. It mainly collaborates with other components by storing and returning objects that are required during the execution of the agent. Most collaborations occur when an agent receives a trigger. Those collaborations are depicted in Figure 6.2. Trigger listeners can add triggers to the runtime configuration. This will cause the agent to call upon its deliberation runnable to execute a deliberation cycle. A deliberation cycle is a sequence of deliberation steps. Each deliberation step contains an execute method that given a runtime configuration progresses the agent. In our library we instantiated two types of deliberation steps that we consider to be basic progressions of an agent. The first is depicted in Figure 6.2 as step 'a', and progresses the runtime configuration by obtaining the plan schemes, current triggers and context to instantiate plans schemes, and storing the potentially returned instantiated plans in the runtime configuration. A second step 'b' obtains the instantiated plans from the runtime configuration and executes them with the configuration's context and messenger (for communicating with other agents). In our library we chose to put the agent to sleep if it contains no more active plans or triggers, otherwise it will execute the cycle again. A kill switch can halt an agent. The execute method of a plan may retrieve the context and messenger of the agent and adopt new triggers, including goals. A plan maintains a Boolean that indicates whether the plan should be executed again in the next deliberation cycle. The deliberation cycle cannot continue if the method of a plan does not return. Hence, if a plan is to be applied repeatedly then it is best to do so by setting it to not being finished until the repetitions are done instead of using a '`while(true)`' loop.

We note that the depicted classes in Figure 6.1 serve to specify the agent and need to collaborate with a surrounding system in order to gain execution time. In our library we opted to enrich a standard executor service that maintains a thread pool for executing agents, the kills switches, a messenger infrastructure and agent initialization factories.

Consequences

The pattern supports its modularity objectives by focusing decision making business logic in the plan classes and the observation capabilities in the context. It is possible to define suitable plan schemes, plans and contexts for an application and re-use them in another application. For instance, a set of plan schemes, plans and a context can be designed for auctions where, if imported from a library, the agent programmer only has to implement a valuation method for something that is traded.

The objective of achieving pro-active behavior is supported by the usage of goals. Goals persist in the agent's runtime configuration for as long as they are not considered as achieved given the context of the agent. This will cause the agent to initiate new deliberation cycles for as long as the goals are not achieved. It should be noted that liveness properties can be proved for the BDI programming languages that have formally specified operational semantics.

The trade-off is that a certain overhead is introduced. Especially for small agents it might be more beneficial to resort to for instance actor-based programming.

Implementation

The library of classes that we provide as part of this research contains an implementation of this pattern. We have made the choice to divide triggers in four categories: a) triggers that come from external sources to the agent system, such as events from the environment, b) messages of the agent system, c) goals of the agent and d) triggers that occur internally in the agent. We also divided the plan schemes in these categories and maintain them in a plan scheme base (rather than the runtime configuration containing one list of plan schemes). Accordingly, we have implemented a separate deliberation step for the processing of each of these trigger types. We opted for these choices to increase the modularity and reusability of agent source code. For the same reason we use a context container with multiple contexts rather than a single context. The context container of an agent is implemented by the *typesafe heterogeneous container* pattern [24]. As a consequence, the key of a context in the container is its class. We use internal triggers to create a self-healing mechanism. If a plan fails for any reason, then the user can specify an exception to be thrown which is adopted as an internal trigger. Accordingly, plan schemes for handling such exceptions can be programmed. For good coding practices we adopt many interfaces between the components of an agent to ensure correct exposure of the runtime configuration to various other components.

There are many reoccurring challenges in the design of agent systems that the proposed pattern does not cover. For instance if an agent shares a thread with other agents, then its deliberation cycle should not block the execution of other agents due to for example waiting on the return value of an environment call. Also concurrency

issues regarding the objects that agents call upon have to be addressed. There are various patterns and technique to help this, such as the use of futures and explicit environment design frameworks such as Cartago [111].

Sample code

We shall exemplify a goal, plan scheme, plan and context, as these are the core components that an agent programmer has to program. The current example is based on the aforementioned example of an agent that participates in auctions. We assume that all auctions are Vickrey auctions. As an example context we take a context named TraderContext. Its functionalities include checking whether an item is owned, obtaining a relevant auction for an item and determining the personal value of an item.

```
1 public class TraderContext implements Context {
2   ...
3   public boolean isInInventory(Item item){ ... }
4   public double value(Item item){ ... }
5   public Auction getAuctionFor(Item item){ ... }
6   ...
7 }
```

The method in line 3 returns true if a given item is in the inventory of the agent. The method in line 4 returns the value that a given item holds for the agent. The method in line 5 returns an available auction for a given item, or null if no such auction exists.

As an example goal we will take a class named GoalItem, of which instances represent the goal to own a specific item. This goal is achieved when the item is in the inventory of the agent. For as long as this is not the case it should be tried to instantiate plans for this goal.

```
1  public class GoalItem implements Goal {
2    private Item item;
3    public GoalItem(Item item){ this.item = item; }
4
5    public boolean isAchieved(ContextContainer contextContainer){
6      TraderContext context =
7          contextContainer.getContext(TraderContext.class);
8      return context.isInInventory(item);
9    }
10 }
```

Line 2 contains the item that the agent wants. Line 3 is the constructor of the goal. Lines 5-9 are called when it is needed to check whether the goal has been achieved. Lines 6-7 show how the context is obtained in our library. In line 8 it can be seen that the goal is achieved if the desired item is in the inventory of the agent.

As an example plan scheme and accompanying plan we take the plan scheme that specifies a bidding strategy in a Vickrey auction. This plan scheme can be instantiated, i.e. create a plan, if there is a trigger that is a GoalItem (recall that Goals are a type of Trigger). For a specific trigger, checking whether that trigger

is of a certain type is similar to checking the head of a plan rule in for instance 2APL, Jason and GOAL. Also the TraderContext is used to check whether there is an auction available. This can be seen as a belief query, or guard check of a plan rule. In a Vickrey auction the best strategy is to bid on an item the true value. So the actual plan is to determine the goal item's value, and bid that amount in the auction. Note that if other auction types were available that then for instance for each auction type a different plan could have been specified. In that case the plan scheme could instantiate a plan depending on the type of the auction.

```
1  public class BidScheme extends PlanScheme {
2    public Plan instantiate(Trigger trigger,
3        ContextContainer contextContainer){
4      if(trigger instanceof GoalItem){
5        GoalItem goalItem = (GoalItem)trigger;
6        TraderContext context =
7          contextContainer.getContext(TraderContext.class);
8        Auction auction = context.getAuctionFor(goalItem);
9        if(auction != null){
10         return new VickreyBidPlan(auction, goalItem.getItem());
11     }
12     return Plan.UNINSTANTIATED;
13   }
14   public final class VickreyBidPlan extends Plan {
15     private Auction auction;
16     private Item item;
17
18     public VickreyBidPlan(...){ (constructor) }
19
20     public void execute(PlanToAgentInterface planInterface)
21         throws PlanExecutionError {
22       TraderContext context =
23         planInterface.getContext(TraderContext.class);
24       auction.bid(context.value(item));
25       setFinished(true);
26     }
27   }
28 }
```

Lines 2-13 describe when the plan should be instantiated and how it is instantiated. Line 4 conditionalizes the instantiation on the type of the trigger that is being processed. Lines 6-9 let the agent check whether there is an auction available. If so, then in line 10 a new bidding plan will be created. Lines 14-27 specify the plan. The agent will bid the value that the agent thinks is the item's true value (line 24). The plan is set to finish at line 25. This will remove the plan from the runtime configuration.

Known uses

The pattern is visible in numerous agent-oriented programming frameworks/languages such as Jade, 2APL, GOAL and Jason. In these languages the business logic for

making decisions and executing actions is concentrated in behavior for Jade or plan rules in 2APL, GOAL and Jason. There a plan rule is composed of a head, guard an plan itself which can be triggered by an event, message or goal. In this pattern a plan rule is the same as a plan scheme and its corresponding plan, and the match between event/message/goal and the head of a rule is the same as the applicability check for a plan scheme. Instantiating and storing a plan for later execution if the scheme is applicable is similar to 2APL and Jason. In GOAL the plan is executed immediately, which is also supported by the pattern. The notion of a belief (and knowledge) base is integral to the latter languages, and is here represented as a context.

Related patterns

The most related is the *active object* pattern [61]. It too has this structure where calls are made through a proxy and are processed independently, much like our setup where triggers are send to the agent to be processed independently. However, it does not contain the notion of plan schemes, nor is proactivity possible through something like goals.

Another important related pattern is the *strategy* pattern. It contains the solution to problems where a different execution strategy is needed under different circumstances. This relates much to our plan schemes. In the *reactor* pattern and its asynchronous variant the *proactor* pattern [114] applications can register event handlers in an initiation dispatcher. Clients can then send events to the initiation dispatcher which notifies the correct handlers when they can process the events without a block. This is related to our pattern, because it resembles the structure of sending events (in our case triggers) to an entity that decides when and how these events are processed. However, the selected plan schemes in our pattern do not solely depend on the type of trigger that is received. Also note that the *reactor* and *proactor* patterns are of a reactive nature. In contrast, our pattern introduces proactiveness by pursuing goals until their achievement.

Notes

In the example code the chosen plan can be instantiated and stored for later execution as in 2APL and Jason, or can be executed immediately such as in GOAL. Note that in some cases it is desirable that one plan or a part of the plan is executed at once without interleaving its execution with the execution of other plans. This is for example done in 2APL by introducing atomic plans. In general, we believe that these issues should be decided by the programmer and may not be generic enough to introduce as part of the design pattern. The same holds for a specification of how beliefs and goals can be managed. An integral part of agent programming languages is a specification of how an agent deals with its beliefs and goals. For example, how they may change over time. A management system for goals and beliefs would be part of the context of the object-oriented agent. If desired one can create a context with a pointer to a Prolog engine or object-oriented database to represent beliefs. This way a programmer can still use a Prolog engine to for instance manage beliefs, without forcing all programmers to use Prolog.

6.4.2 Object-Oriented Norm Pattern

Norms are related to constraints. But the term constraint already has a defined meaning in object-oriented programming (from the *constraint* pattern). Hence we refer to the counterpart of norms as normative constraints; constraints that can be violated albeit with consequences. In this section we outline how norms can be captured using object-oriented technology.

Name and classification

The *normative constraint* pattern is a behavior pattern.

Intent

The intent of the *normative constraint* pattern is to realize norms for open multi-agent systems separately from the environment and agents. The described pattern promotes a modular development approach to developing the capabilities of autonomous agents (the environment interfaces) and the restrictions on behavior (the norms).

The pattern that we describe captures many possible forms of normative behavior. We observe in the scientific community a preference for conditional obligations and prohibitions with deadlines. Such norms require an agent to achieve/prevent some state or (not) execute some action after a specified condition is met and before a specified deadline occurs. The enforcement of these norms is realized through either regimentation, that nullifies an action or state change that causes a norm violation, or sanctioning, which applies a compensating procedure after a norm is violated. The described pattern in this section captures both.

We promote a design approach where norms are specified in their own aspects. These aspects are interwoven in the environment interfaces at compile time. Hence, at runtime it is not possible to retract a norm. However, we do note that it is quite straightforward to specify conditions under which the norm does (not) apply. For a sanctioning norm for instance it is possible to check an applicability condition before a sanction is applied. This way a norm can be 'switched off' at runtime. The norm-oriented programming languages on which this pattern is based take an approach where norms are specified together in an institutional or organizational entity. The conditions under which a norm applies are often related to for instance a role that an agent has within the multi-agent system. We do not capture organizational concerns other than norms in this chapter. However, we do note that multiple norms can for instance synchronize on an organizational context to capture these dynamics.

Motivation

The natural scenarios for the *normative constraint* pattern include resources and services that autonomous processes such as agents may use but which are not under full control of the system designer. Yet the usage of these resources and services should be limited to desirable behavior. The system designer can in such scenarios implement the constraints on behavior in the interfaces between the autonomous processes and the resources and services. However, if the specification of correct and

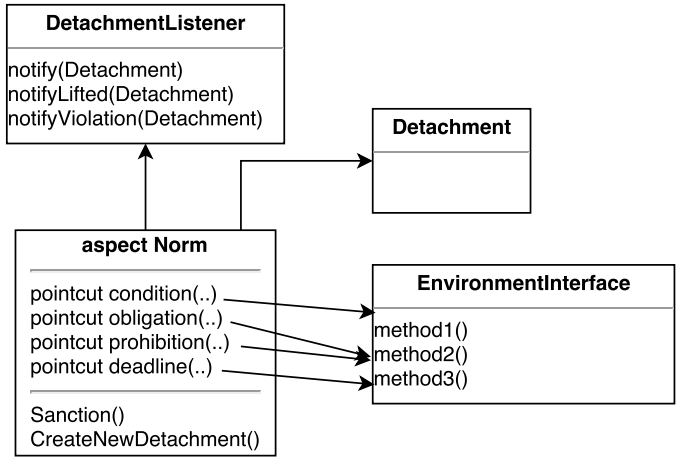

Figure 6.3: *Global structure of the* normative constraint *pattern.*

incorrect behavior depends on the usage of multiple interfaces, then soon the code becomes hard to maintain because the business logic of a constraint is spread over multiple classes.

Applicability

The *normative constraint* pattern is best used when:

- The system designer has no access to the source of the processes that it tries to control, but does have control over the interfaces that these processes use.
- Desirable behavior can be specified in terms of state changes and method calls in interfaces between autonomous process and the services and resources that they use.
- The desirable behavior specification does not change at runtime.

Structure

The structure of the *normative constraint* pattern is shown in Figure 6.3.

Participants

The participants are as follows:

- **EnvironmentInterface.** Exposes the relevant functionalities of a resource or service to another process.
- **Detachment.** A detachment of the norm, i.e., the generation of an obligation or prohibition with deadline. It contains relevant data from when the detachment occurred, which can be used to check whether the constraint is violated or not.

- **DetachmentListener.** Allows the norm to publish the detachment, i.e., an obligation with deadline. The listener relays the detachment to relevant processes.
- **Norm.** Specifies the condition, obligation or prohibition and deadline of the norm. Also maintains the current detachments and specifies how violations of the norms should be sanctioned, if sanctioning is applied.

Collaborations

The pointcuts in the norm are specified on execution points in the environment interface. The condition poincut's advice is an after-advice (it is executed after the condition pointcut is passed). When the condition pointcut of the norm is reached, then the condition advice will check whether the norm should be instantiated and detached. If so, then a detached norm (implemented as a Detachment; see Figure 6.3) is created and any detachment listener is notified. The obligation advice is also an after-advice. If the obligation pointcut of the norm is reached, then the obligation advice will check whether for any detached norm whether it can be removed, i.e. whether the obligation was met. If this is the case, then the detachment listeners will be notified that the obligation is lifted. The advice of the prohibition is an around-advice. This advice checks whether a detachment exists for which now the prohibition is violated. The advice will not proceed the pointcut's method call if the prohibition is violated and the norm is regimented. If the norm is sanctioned, then the sanction will be executed upon violation. The detachment listeners will be notified of the violation. The advice of the deadline pointcut is also an around advice and will not proceed the call if the norm is an obligation and is regimented, and otherwise will execute the sanction if the norm is not regimented. The detachment listeners are notified in case the obligation is violated. If the norm is a prohibition, then the deadline will simply remove detachments to which the deadline applies and the detachment listeners are notified that the prohibition is lifted.

Consequences

The main objective is to separate the norm from the environment interfaces. This is inherently the case because of the usage of an aspect. The separation between condition, obligation and deadline provides a clear specification of the temporal aspect of a detachable norm. With this pattern a system designer has the possibility to independently design complex rule structures for different use cases, where the rule can be violated. The trade off is that the flow of control is harder to grasp because of the use of aspects.

Implementation

Care has to be taken that the norm is not detached extremely often, because each detachment requires memory. If the detachments can somehow be ordered, then a heap or other sorted data structure is preferable to an iterable due to run time complexities. Memory issues can occur easily if the deadlines and obligations/prohibitions are met in a slower pace than that the norm is detached.

On a higher conceptual level it might be beneficial to approach the design and development of norms with the use of the multi-agent organizational paradigm. Norms are a key concept in this paradigm which are usually tightly related to other notions such as roles that agents may have and organizational empowerment. If a norm should only be detached under specific organizational circumstances (such as the roles that a particular agent fulfills), then these checks can for instance be performed in the advice of the condition pointcut of the norm aspect.

Example code

The arguments of the pointcuts and methods in a norm heavily depend on the application for which the norm is implemented. In this section we give an example of an abstract sanctioning norm aspect where the arguments of the condition, obligation/prohibition and deadline pointcuts only consist of the environment interface to which the norm applies. We also demonstrate a norm that instantiates the abstract norm. We expect that in many applications different arguments might be needed for the aspect components, such as the arguments that were used in the call of an environment method.

The following norm specification requires the programmer to provide the type that is used for norm detachments (D) and the environment interface to which it applies (I). It also requires the programmer to specify the pointcuts for the condition, obligation/prohibition and deadline. The arguments for these pointcuts are forced to be the environment interface instance to which the norm applies. The norm aspect takes care of the operational semantics of the norm; when it is detached, when detachments are removed, and when sanctions should be applied. We also allow the norm to be regimented instead of being enforced through sanctioning. The effect of regimentation is that the call is not executed if the norm will be violated by this call. Therefore we use an around advice which proceeds only if there is not a detachment that now constitutes a violation. Note that this advice also does not remove norm detachments. Finally we assume in the following code that the deadline pointcut was specified on some method call that has no return value. We opted for a form of obligation and prohibition where after violating the norm the prohibition and obligation are also removed. This can be easily altered to accommodate for other operational semantics. Also note that we make heavy use of lambda expressions and the streams library that were introduced in Java 8.

```
1  public abstract aspect Norm<D, I> {
2    private Collection<D> detachments;
3    private Collection<DetachmentListener<D, I>> listeners;
4    private boolean isObligation, isSanctioning;
5
6    public Norm(boolean isObligation, boolean isSanctioning){
7      this.isObligation = isObligation;
8      this.isSanctioning = isSanctioning;
9      // (omitted initialization code)
10   }
11
12   protected abstract pointcut condition(I i);
```

```
13    protected pointcut obligation(I i);
14    protected pointcut prohibition(I i);
15    protected abstract pointcut deadline(I i);
16    protected boolean obligationApplies(D d, I i){ return false; }
17    protected boolean prohibitionApplies(D d, I i){ return false; }
18    protected abstract boolean deadlineApplies(D d, I i);
19    protected void sanction(D d, I i){};
20    protected abstract D createNewDetachment(I i);
21
22    after(I i) : condition(i){
23      D d = createNewDetachment(i);
24      detachments.add(d);
25      for(DetachmentListener listener : listeners)
26        listener.notifyNewDetachment(d, i);
27    }
28
29    after(I i) : obligation(i){
30      detachments = detachments.stream()
31        .filter((D d) -> {
32              if(obligationApplies(d, i)){
33                for(DetachmentListener listener : listeners)
34                  listener.notifyLifted(d, i);
35                return false;
36              } else return true;
37          }).collect(Collectors.toList());
38    }
39
40    void around(I i) : prohibition(i){
41      AtomicBoolean block = new AtomicBoolean(false);
42      detachments = detachments.stream()
43        .filter((D d) -> {
44              if(prohibitionApplies(d, i)){
45                if(this.isSanctioning)
46                  sanction(d, i);
47                else block.set(true);
48                for(DetachmentListener listener : listeners)
49                  listener.notifyViolation(d, i);
50                return false;
51              } else return true;
52          }).collect(Collectors.toList());
53      if(!block.get()) proceed(i);
54    }
55
56    void around(I i) : deadline(i){
57      AtomicBoolean block = new AtomicBoolean(false);
58      detachments = detachments.stream()
59        .filter((Detachment d) -> {
60              if(deadlineApplies(d, i)){
61                if(this.isObligation){
62                  if(this.isSanctioning)
```

```
63                    sanction(d, i);
64                  else block.set(true);
65                  for(DetachmentListener listener : listeners)
66                    listener.notifyViolation(d, i);
67                } else {
68                  for(DetachmentListener listener : listeners)
69                    listener.notifyLifted(d, i);
70                }
71                return false;
72              } else return true;
73          })
74        .collect(Collectors.toList());
75      if(!block.get()) proceed(i);
76    }
77  }
```

Lines 2-10 initialize the norm and specify whether it is a conditional obligation or prohibition and whether enforcement occurs through regimentation or sanctioning. Lines 12-20 specify what a norm developer has to implement when extending the norm aspect. The obligation and prohibition are optional, but it is intended that exactly one of them is overridden. The same holds for the obligationApplies and prohibitionApplies methods. Lines 22-27 check whether the norm has to be detached. Lines 29-38 specify that detachments for which the obligation is met will be lifted. Lines 40-54 specify that detachments for which the prohibition holds are violated. The sanction is executed (line 46) or the call is not proceeded (line 53), depending on the mode of enforcement. Lines 56-76 specify the deadline advice. If the norm is an obligation then detachments are checked for violations (lines 60-67). Else if the norm implements a prohibition then relevant detachments are lifted. Finally if the norm is an obligation and there was a violation, then the call is not proceeded (line 75).

The following code is the example where an organizer of an auction ought to inform the participants of the auction of the auction results, before the auction is closed. Violations are sanctioned by blacklisting the organizer. We assume the following classes in this scenario (aside from the participants in this pattern):

- Auction: The environment interface that agents use to set up an auction. This class has methods to start the auction, publish the result, get the organizer, and close the auction.
- AuctionDetachment: The class for detachments of the auction norm. Contains a field to store the relevant auction interface for which the norm was instantiated.
- BlackList: A blacklist on which auction organizers can be added.

```
1  public aspect AuctionNorm extends
2        Norm<AuctionDetachment, Auction> {
3    private BlackList blacklist;
4
5    public AuctionNorm(){
6      super(true, true);
```

```
7       // (omitted initialization code)
8     }
9
10    protected pointcut condition(Auction auction) :
11      call(public * Auction.startAuction(..)) && target(auction);
12
13    protected pointcut obligation(Auction auction) :
14      call(public * Auction.publishResult(..)) &&
15      target(auction);
16
17    protected pointcut deadline(Auction auction) :
18      call(public * Auction.removeAuction(..)) &&
19      target(auction);
20
21    protected AuctionDetachment createNewDetachment(Auction auction){
22      return new AuctionDetachment(auction);
23    }
24
25    protected boolean obligationApplies(AuctionDetachment d,
26                                        Auction auction){
27      return d.getAuction().equals(auction);
28    }
29
30    protected boolean deadlineApplies(AuctionDetachment d,
31                                      Auction auction){
32      return d.getAuction().equals(auction);
33    }
34
35    protected void sanction(AuctionDetachment detachment,
36                            Auction auction){
37      blacklist.add(auction.getOrganiser());
38    }
39  }
```

The pointcuts of the condition, obligation and deadline are the method calls for starting the auction, publishing the results and removing the auction, respectively (lines 10-19). The creation of an auction is specified in line 22. A detachment obligation and deadline applies if its auction matches with the auction for which the current auction that triggered the aspect (lines 27 and 32). Finally, the sanction is to blacklist the violating organizer (line 37).

Known uses

There is quite a lot of work on norms with a condition, obligation/prohibition and deadline, e.g. 2OPL and NPL. In object-oriented programming this is typically indicated by some boolean flag in code that signals whether some condition was met before and that is being used to steer execution at a later point.

Related patterns

Patterns with contracts among objects are also used to ensure behavior over time. A related work is for instance Contract4J [139]. In design by contract for programs, contracts consist of preconditions, postconditions and invariants. A client must fulfill the precondition so that a server can perform an operation which fulfills the postcondition. Invariant constraints must hold at all times. If a contract is violated, then the program halts (in contrast to normative constraints).

Another related concept, though no pattern, is the Object Constraint Language (OCL), which is a part of UML. OCL is a design tool that allows a designer to specify very specific constraints such as the range of an integer attribute of an object. However, these constraints are also meant as constraints that cannot be violated.

Notes

The presented pattern captures conditional norms in a very generic and basic form. We did not go into details on various topics in normative system research such as different types of norms, the dynamics of norms, norm conflicts and norm awareness. Future efforts will be focused to these topics. We also did not address many synchronization and related issues for interacting autonomous systems. Depending on how the normative pattern is used, it can be combined with patterns for decentralized computation. We note though that aspects inherently allow us to use some form of decentralization. The pointcuts that are defined can be triggered by different processes. The Java virtual machine ensures partly that the aspect will be correctly applied. If norm aspects require use of each other's monitoring/control capabilities, then this can be realized through standard Java means. This is why we do not provide special constructs for decentralized runtime norm enforcement. We would like to emphasize that our approach may suggest that the source code of various interacting components must be available, which may not always be the case, e.g., in open systems. We do assume that the components interact through interfaces that are under the control of the system designer. The system designer can therefore introduce the pointcuts in the interface source code and thereby control the execution of the components.

6.5 Related Work

The idea of agent-based design patterns has grabbed the attention of many researchers in the field. There have been several proposals focusing on various categories of design patterns (for an overview see [77]). Some of the earliest agent-oriented design patterns are proposed by Aridor and Lange [12]. They proposed agent design patterns for mobile agent applications and classified them into travelling patterns, task patterns and interaction patterns. The mobility patterns can be used to enforce encapsulation of mobility management. An example of traveling patterns is the *itinerary* pattern that defines routing schemes for multiple destinations and handles special cases such as non existent destination. The task patterns are concerned with decomposing tasks and their delegation. An example is the *master-slave* pattern that allows task delegation from master to slave. Finally, the interaction patterns are concerned with agents'

communication and cooperation. For example, the *meeting* pattern allows agents to dispatch themselves to a specific destination (a meeting place) and engage in local interaction. Our proposed design patterns are complementary as we are not concerned with mobile agent applications, but with the internal design of autonomous agents and how such agents can be controlled and coordinated by means of norms.

Sauvage presents different classes of patterns such as Meta patterns, Metaphoric patterns and Architectural patterns [113]. Examples of meta patterns are *organisation* and *protocols* which are defined in terms of roles, their relations, and messages. An example of metaphoric patterns is the *marks* pattern, which describes an indirect communication model via environment. Examples of architectural patterns are *BDI architecture* consisting of knowledge bases and *horizontal architecture* consisting of parallel modules (e.g., deliberation and act modules). Our proposed design patterns for autonomous behavior and norm-based coordination are related to the *BDI architecture* pattern and *organisation* pattern. Although Sauvage provides only a two lines description of *BDI architecture* pattern, we provide an extensive description and possible refinements of it. Moreover, Sauvage conceives an *organisation* pattern as being defined in terms roles and their interactions while our organization is defined in terms of norms being monitored and norm violations being sanctioned.

Separating coordination among processes as a concern has been argued by Gelernter et al. [63], where the case is made for special coordination frameworks such as Linda. A coordination framework manages coordination separately from the business logic, which is programmed in a different language. With the use of aspects we can make reusable coordination oriented norms, while staying very close to the computational language of the business logic. Though the use of a separate coordination framework will often remain the preferred choice for interacting autonomous systems where processes are made with different implementation languages.

In order to organize interacting intentional software entities in multi-agent systems, social patterns are introduced by Do et al. [54]. Two specific categories of patterns introduced here are pair patterns and mediation patterns. The pair patterns describe direct interaction between intentional agents while mediation patterns describe intermediate agents that aim at reaching agreement between other agents. An example of pair patterns is the *booking* pattern for booking resources from a service provider, and an example of mediation patterns is the *monitor* pattern that allows receiving notification of changes of state. Our proposed design patterns for autonomous behaviour are complementary to the patterns proposed by Do et al. in [54] and describe the internal design of individual agents. Moreover, our norm-based design patterns differ from patterns proposed by Do et al. in [54] as ours are not concerned with explicit interaction between agents.

In Weyns el al. [141] a pattern language is presented to capture various patterns in the design of multi-agent systems. The language consists of five interrelated patterns that together capture the different aspects of agent systems. The *virtual environment* pattern captures the design of an environment in which agents are situated. Those agents are captured with the *situated agent* pattern. It is very common that agents have a limited view on the system, which is documented as the *selective perception* pattern. For the coordination of agents the language contains two patterns: *protocol-based communication* and *roles & situated commitments*. The patterns are

described in a architectural design language whereas we focused on object-oriented programming. That is less general, but easier to adopt.

Probably the closest agent-oriented design patterns to ours are those proposed by Morreale et al. in [98], which aim at supporting the development of BDI agent-based systems. They use the PRACTIONIST framework, which allows the development of goal oriented agents based on BDI models, to introduce various BDI agent patterns. In particular, they propose four agent design patterns called *dynamic strategy selection* pattern, *intention decomposition* pattern, *mutually exclusive intentions* pattern, and *necessary intention* pattern. For example, the *dynamic strategy selection* pattern describes how an agent's intention can be achieved by the best strategy from a set of strategies at run time. Our design patterns for autonomous behavior are similar to *dynamic strategy selection* pattern. But, in contrast to this pattern, we distinguish two different refinements for both reactive and proactive behavior. In our view this distinction is crucial as they generate two important types of behavior, i.e., reactive behavior triggers only one single plan as a response to an event while proactive behavior keeps triggering new plans until the goal is achieved. Moreover, our norm-based design pattern are complementary to the patterns introduced by Morreale et al. in [98].

Similar agent design patterns are introduced by Kendall et al. [80]. In this article, reactive and deliberative agent patterns are presented as instances of sensory, beliefs, reasoning patterns. These patterns are described briefly and informally in terms of the problem, forces, solution and known uses of the patterns. The problem of a pattern is merely an informal description of the agent type. For example, the problem of a deliberative agent is described as how an agent can select a capability to proactively achieve a goal. The forces represent some requirements and properties of the pattern. For example, the forces of a reactive agents consists of the requirement that an agent needs to be able to respond to a stimulus or a request. The solution explains how the problem should be solved. For example, for a reactive agent patterns it is indicated that the agent acts using a stimulus/response type of behavior. Finally, the known uses refer to other work that use similar agent types. For example, the authors refer to the work of Cohen and Levesque [40] as a use of deliberative agents.

Finally, Oluyomi et al. [102] presents a two dimensional classification in order to analyze, classify, and describe some existing agent-oriented patterns. The vertical dimension is based on the stages of agent-oriented software engineering and distinguishes seven stages from requirement analysis to implementation and testing phases. The horizontal dimension is based on tasks and activities that are relevant at each stage of software development. For example, at the multi-agent system architectural level, the tasks are to design the system, the involved agents, and their interaction. The vertical and horizontal dimensions identify categories of agent-oriented design patterns. For example, the category defined by the multi-agent system architectural level (vertical dimension) and system design activity (horizontal dimension) is identified as a structural patterns which describe the structure of agent organizations in terms of architectural components including knowledge component and environment. An example of an agent-oriented pattern that belongs to this category is the *embassy* pattern. This pattern introduces an agent responsible for the interaction between a multi-agent system and other heterogeneous domains. Our design patterns

for autonomous behavior can be seen as a member of the category Agent Internal Architecture - Interaction patterns and our design pattern for norms as a member of the category agent-oriented Analysis - Organizational patterns.

6.6 Conclusion

The adoption of multi-agent programming tools and technologies by industry is a major challenge that still needs to be met by the multi-agent programming community. One possible way to meet this challenge is by transferring multi-agent programming technologies to the standard software technologies. An idea is to start with the high-level concepts and abstractions for which the multi-agent programming research field has provided computational models and programming constructs, and propose either corresponding language level supports in the standard programming languages (e.g., C++ or Java), or alternatively propose corresponding design patterns, i.e., general reusable solutions to problems such as proactivity, reactivity, adaptivity, monitoring and control. The language level support can either be realized by standard programming approaches such as meta-programming or aspect-oriented programming, where concepts such as deliberation and control can be considered as different concerns that can be programmed either by meta-programs or aspects. Although these suggestions are not mature and need to be worked out both in details and in practice, attempts along these lines can bring the multi-agent community closer to industry.

The design patterns in this chapter are only scratching the surface of all the contributions of the multi-agent programming community. The goal of this chapter is to take part in the discussion of how we can engineer multi-agent systems with object-oriented technology, so that we can promote the agent paradigm to a wider audience. We will further develop design patterns to deal with for instance concurrency issues, repair strategies and interaction among behavior and agents. With the input from the multi-agent community we shall reach out to other platforms where object-oriented technology is discussed. We shall also release open-source example code to illustrate what projects look like when they are implemented according to the proposed design patterns.

7 Smart Infrastructure Simulation

In this chapter we describe an application for decentralized norm-based control, which also touches upon the notion of norm-awareness. We continue upon the theme of smart infrastructures, which is a topic from which we draw many examples throughout this thesis. Autonomous vehicles are commonly predicted to be a major component of future traffic. The advent of autonomous vehicles allows us to explore innovative ideas for traffic control such as the application of norm-based control. Norm-based traffic controllers monitor traffic and realize sanctions in case vehicles violate norms. The law provides the rules of traffic, and hence norms represent traffic regulations. We approach the experimentation with norms by traffic simulations, for which we use the traffic simulator SUMO [84]. We present an extension of SUMO that enables the user to apply norm-based traffic controllers to traffic simulations. In our extension, named TrafficMAS, vehicles are capable of making an autonomous decision on whether to comply with norms. We provide a description of the extension, a summary on its implementation and demonstrative experiments.

7.1 Introduction

The automotive industry steer towards a future where autonomous vehicles become a part of everyday traffic (cf. [2],[92],[1]). Such vehicles offer the possibility for the infrastructure and other vehicles to communicate with them. We refer to infrastructures that are enriched with controllers that communicate with vehicles as smart roads or smart infrastructures. Clearly human drivers are different from software programs that operate vehicles. For instance, the response of human drivers to receiving information is considerably slower and less accurate in comparison with a computer program. These differences pose new challenges and opportunities [18]. For example, delegating cruise control to the vehicles' on board computers allows vehicles to coordinate and form platoons, which improves the traffic flow [79].

In this chapter we are particularly interested in future challenges and opportunities for traffic control. Traffic controllers can exert some level of influence on vehicles in order to improve traffic flow and safety. Traffic is currently controlled by means of traffic laws and signs which require the education of general traffic regulations and the interpretation of signs. The government as a regulator creates incentive to follow

the regulations by imposing sanctions on anybody who is caught violating them. This control mechanism is tailored for humans, as they are currently the road's only occupants. For example speed limits are given in easy round numbers, as we expect humans to only approximate their limits within an error margin. An autonomously controlled vehicle has a more precise control over its velocity and hence its error margins are different. This allows us to give an autonomous vehicle more precise directives. We shall address the question of how we can design, analyze and test new traffic controllers where (a portion of) the vehicles is driven by autonomous software systems.

We envision a future where the traffic infrastructure consists of smart roads, enriched with software systems that control traffic. We will call such a software system a traffic controller. The function of a traffic controller is to observe and evaluate traffic, and communicate personalized directives to vehicles. Such a traffic controller is required to respect the autonomy of the vehicles as autonomous vehicles are assumed to be self-interested with possibly personal incentives to violate traffic regulations [14]. Traffic controllers will be allowed to impose sanctions on autonomous vehicles as a way to promote desired behavior. Thus, in our vision autonomous vehicles, which are aware of traffic regulations and their corresponding sanctions, may still violate traffic regulations and accept the imposed sanctions whenever their personal objectives are worth the incurred sanctions. Traffic controllers should also be easily maintainable in terms of the traffic regulations. In our approach, we focus on drivers, the state of traffic, the regulations and the traffic controllers. We shall argue that a suitable paradigm for these kind of control systems is that of norm-based control for multi-agent systems. We already note that the enforcement of the traffic regulations by sanctioning is quite similar to norm enforcement by sanctioning. As discussed in other chapters, aside from sanctioning we may consider regimentation. In this chapter we focus only on sanctioning. The reason is that current regulations fit sanctioning whereas regimentation requires invasive control capabilities on the enforcer's side. In the case of a speed limit this idea would not fit, as a traffic controller cannot physically control a vehicle. We do note that digital actions of an autonomous vehicle regarding the infrastructure, such as requesting permissions of any kind (e.g., the priority lane usage in Chapter 1), could be regimented. However, we do not discuss that type of actions in this chapter.

Aside from design we also want to demonstrate norm-based traffic control. Such demonstrations usually rely on traffic simulations as real world experiments with autonomous vehicles and smart roads tend to be an expensive affair. We also use traffic simulation for this same reason. We choose SUMO [84] as a simulation platform because it is open-source, has a track record of research behind it, and performs well. We did observe however that SUMO does not provide a straightforward platform to implement concepts from the paradigm of norm-based control for multi-agent systems. In this chapter we discuss an extension to SUMO, named TrafficMAS, that allows the user to specify and execute norm-based traffic controllers for traffic.[1] The demonstrations with SUMO serve as a proof-of-concept for norm-based traffic control for the future.

The running example in this chapter is a ramp merging scenario, where two traffic

[1]This software is is open-source and can be found at https://github.com/baumfalk/TrafficMAS.

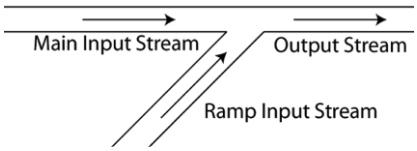

Figure 7.1: *Example scenario where two traffic streams must merge.*

streams have to merge together. For a schematic overview, see Figure 7.1. There is one main input stream and one ramp input stream, resulting into a single output stream. The goal is to make optimal use of the output capacity of the network whilst not causing unnecessary traffic jams for the ramp input stream or compromising safety. An analysis of this scenario can be found in [18]. Our approach is to use a traffic controller that assigns individual directives to vehicles. In particular, vehicles are obligated to move or stay on a lane and/or adopt a certain target speed until they are released of this directive. Furthermore, the vehicles receive a notification of this directive. This will allow them to decide on whether they want to violate the directive or obey it.

This chapter is structured as follows. We discuss in Section 7.2 norm-based control systems as traffic controllers. We will then show how the norm-based control system for our SUMO application is designed, and discuss its application in our example scenario in Section 7.3. Following that, we describe how vehicles can reason about the norms that are directed to them by the traffic controller in Section 7.4. This leads to our driver model (i.e. the model of the agents that control vehicles) that is described in Section 7.5. In Section 7.6 we demo norm-based traffic control through a series of experiments that highlight different aspects of our contribution. Finally, we look at related work and compare it with our approach in Section 7.7.

7.2 Norm-Based Traffic Controllers

Norm-based control systems are a popular technology for coordination in the Multi-Agent Systems community (see Chapter 2 for more background information on norm-based control). A multi-agent system consists of a set of agents that interact within a shared environment. The agents may communicate with each other or perform actions in their shared environment. Autonomous agents are assumed to have their own objectives (goals) for which they proactively initiate actions in order to achieve them, in addition to reactively responding to environmental changes. Aside from its objectives, the knowledge/belief of an agent may influence the actions that it decides upon to perform. A simple model of an agent's internal process is the sense-reason-act cycle. In this cycle the agent first senses what the state of the environment is, then reasons about its goals, preferences, etc, to determine what action it wants to perform, and then executes the action. Although the agents' behavior is not always predictable or controllable, multi-agents systems are often required to satisfy some

global properties. For our purposes we consider smart roads as a multi-agent system where the state of traffic as well as the infrastructure is seen as the environment, and the drivers of vehicles, whether human or not, are seen as the agents in a traffic MAS. The throughput and safety of smart roads are considered as the global properties that are required to be satisfied.

We will use the same vocabulary regarding norms as in the other chapters. With the concept norm we refer to the specification of the circumstances after which a specific agent is obliged to achieve a system state, and a sanction is a compensating action that will be imposed should this obligation not be met before a certain deadline (cf. [133]). With the concept norm detachment we refer to a specific directive that is in effect for a specific agent. In general, norms can take the form of an obligation, prohibition or permission, but the scope of this chapter concerns only obligation norms. The enforcement of norms requires continuous monitoring of the behavior of individual agents, evaluation of their behavior with respect to the specified norms, and assurance that norm violating agents are sanctioned. This approach maintains the agents' autonomy and can still promote desirable behavior. We observe that traffic regulations can be formulated following the normative approach. For instance one can straightforwardly formulate a speed limit measure in terms of a condition (entering the road), obligation (maintaining maximum velocity), deadline (passing a camera), and sanction (a fine).

A norm-based traffic controller monitors vehicle behavior and reacts to it. A collection of sensors enables the monitoring task by sensing the state of the environment, in this case the state of traffic and the infrastructure. Examples of these sensors are inductive-loop detectors or cameras that can perform image processing aside from speed measurements. The monitoring functionality of traffic controller serves two purposes. The first purpose is to detach norms and detect violations of norms, such as speeding, tailgating or driving on a priority lane without a permit. If such a violation is detected, then a sanction coupled with this violation will be issued towards the violating vehicle. The second purpose of monitoring is to issue new norms. If the traffic controller continues to observe situations where either the throughput or safety of vehicles declines, then a norm might be issued to improve the situation. This norm might be global, or tailored to a specific vehicle. This allows the traffic controller to be very adaptive to traffic dynamics. Furthermore, the severity of a sanction might be increased if the coupled violation either occurs an excessive number of times, or seems to be the cause of problematic situations. In our framework for norm-based traffic control, both autonomous vehicles and the traffic controller show adaptive behavior towards the ever changing traffic flow and enable the system to cope with difficult and dynamic traffic situations. We announce norm detachments to the autonomous vehicles, just like drivers are assumed to be aware of traffic regulations. Norm-awareness allows an autonomous vehicle to reason and decide whether or not to comply with the norms (cf.[6],[94]). Norm-awareness is crucial for future traffic on smart roads as autonomous vehicles need to know the consequences of norm violations before deciding whether or not to comply with the norms.

Some applications are inherently distributed and require distributed control systems. There are various benefits for distributed control such as increased robustness, parallel processing of data, less communication of data and modular maintenance

(cf. [129], Chapter 3). We believe this is also strongly the case for future traffic in smart roads where sensors and traffic controllers are geographically distributed, different sections of road networks are governed by different sets of norms and regulations, and the amount of generated data to be processed is large. Therefore, we aim at decentralized traffic control consisting of decentralized monitoring of sensor data as well as distributed enforcement of norms. The implementation of our TrafficMAS platform allows for a straightforward implementation of individual traffic controllers and their communication. A norm-based traffic controller may have subscribers and subscriptions. In a decentralized setting, traffic controllers can subscribe to each other in order to receive sensor data which they cannot obtain locally. Thus we obtain a form of decentralized norm-based traffic control.

7.3 Design of the Norm-Based Traffic Controller

In our use-case scenario we illustrate a fairly simple norm-based traffic controller, which at first is not decentralized. At the core of a controller lies a cyclic process that is similar to how agents operate. The process is executed after every tick of SUMO and repeats the cycle of sensing the environment (sense phase), evaluating and creating norm detachments (reason phase), and imposing sanctions when norms are violated (act phase). In the sense phase, traffic controllers have two ways of observing the environment: i) obtain data from the sensors that are placed in the environment, and ii) obtain data from other traffic controllers (by means of communication) to which the norm-based controller is subscribed. We use SUMO's lane detectors that can sense the vehicles that drive along the area that they cover. Specifically, each sensor can detect the identity[2], velocity and position of each vehicle on the sensor's area. Further parameters such as the maximum velocity, acceleration and deceleration capacities can be assumed within reasonable margins. Immediately after the data is received from the sensors, information is communicated between the traffic controllers. In the reason phase the traffic controllers apply their information to their norms in order to create detachments, if possible. If the traffic controller detaches a norm, then the vehicle for which the norm is detached is notified of the directive that it has to fulfill and the associated sanction that will be imposed if the directive is not followed. This gives drivers a chance to adapt their behavior. Furthermore, the traffic controllers act by imposing sanctions on the drivers when they fail to comply to existing obligations when a deadline has been reached.

A schematic representation of the aforementioned ramp merging scenario is given in Figure 7.2. Triangles are vehicles that travel in the direction towards they point. White vehicles are the ones that have not yet received their directive from the traffic controller, while the black vehicles have passed a sensor and have thus received a personalized norm detachment. The vehicles without a norm detachment in Figure 7.2 are vehicles A, B, C and D. Lane sensors (s_1 to s_5) are placed on the roads. These sensors can detect the status of vehicles that are driving over them. For the scenario to work correctly, it is necessary that either the sensors are sufficiently long, or the vehicles sufficiently slow, so that no vehicle can pass the sensors undetected. The

[2]In practice we may use existing license plate detection.

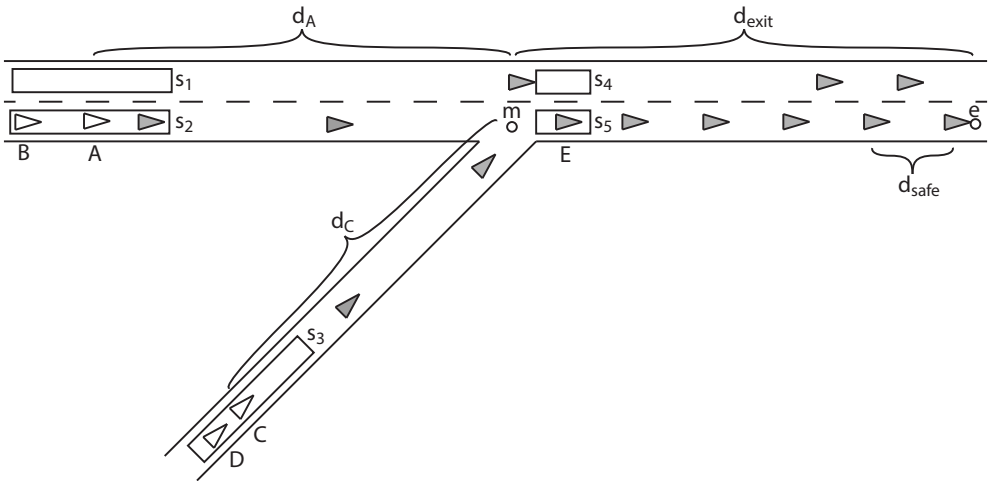

Figure 7.2: *Scenario with two lanes.*

sensors should also be placed at a distance far enough from the merge point m such that vehicles have enough time to comply with the directive before the deadline. There are two important points on the road, point m where the two roads merge, and point e where the vehicles exit the scenario. Distances d_A and d_C are agent's A and C's distances to m, and d_{exit} is the distance from m to e, and d_{safe} is the distance between vehicles that is deemed safe (i.e., the minimal gap between cars). Ideally A and C traverse d_A and d_C such that they arrive at m with a distance d_{safe} and can accelerate to their maximum speed within the distance d_{exit}.

The standard traffic rule in the ramp merging scenario is that the stream that originates on the main road has priority over the ramp road's stream. However, if the main road is busy, this may lead to large traffic jams on the ramp road. For our scenario, the traffic controller creates vehicle specific norm detachments in such a way that vehicles cannot arrive on the merge point at the same time if they comply with their directives. This will cause vehicles to slow down considerably on the main road. If they have to slow down too much then they will be obliged to move to the left lane which is assumed to be more free flowing. Therefore, the traffic controller uses the traffic data from sensors 1 to 3 and determines the optimal velocities of the vehicles in such a way that the traffic streams merge smoothly together at the merge point. More specifically, the traffic controller notifies vehicles passing sensor 1 to stay on the left lane. Sensor 2 and 3 are used by the traffic controller to coordinate the scheduling of vehicles on the merge point. If a vehicle passing sensor 2 has to slow down too much, i.e. to a velocity less than a given threshold, then it receives the directive to move to the left lane.

In order to not overcomplicate the scenario we decided to simplify some aspects of the traffic controller. The sanction that a vehicle can receive for violating a norm is modeled by either a low or a high fine. A directive that a vehicle can receive is an obligation to be on the left or right lane of the main road at a certain target velocity. For instance $(right, 10)$ is read as the obligation to be on the right lane

at 10 m/s. Given that autonomous vehicles are accurate, we take real numbers for the specification of target velocity rather than multiples of 10km/h as is common in current traffic controllers. A norm detachment consists of a directive that is paired with a sanction. The sets of possible sanctions, directives and norm detachments for this scenario are global throughout this chapter, and given by:

- $S = \{low, high\}$ are the possible sanctions.
- $O = \{left, right\} \times \mathbb{R}$ are the possible directives.
- $N = O \times S$ are the possible norm detachments[3].

We will regulate the scenario with two separate norms; one for regulating smooth merging, and one for keeping left-lane vehicles on their lane. The pseudo code of the traffic merging norm is given in Algorithm 1. Note that the code bears strong resemblance to the norm design pattern in Chapter 6. For this pseudo code we use Δ as a store for the norm detachments per vehicle for the traffic controller. We also assume that there exists a function *sanction* that given a vehicle and fine issues the fine for that vehicle, and a function *read* that returns the vehicles on a sensor's area that have not been seen before by that sensor.

We begin with the detachment of the norm (lines 1-7). Initially we read sensors 2 and 3 and merge the readings using the algorithm of Wang et al. [140] (line 1). The result is an ordered list of agents, which, if they continue as they are, will arrive at the merge point in the same order. With $optimalVelocity$ we calculate the optimal speed for an agent such that it will arrive on the merge point at the time that the previous vehicle will arrive plus some safe margin, or later if the agent cannot make it in time physically (line 3). If the agent is at the right lane of the main road and the optimal velocity is below a predefined threshold, then it is obliged to move to the left lane (line 5), otherwise it is obliged to adapt its velocity to the optimal velocity and pass the merge point on the right lane (line 7). Next we check at sensor 4 whether the agents have complied with any possible directive that told them to go to the left lane or stay on the right (lines 8-12). Directives regarding agents that complied with this norm are retracted (line 10) and agents that were supposed to stay on the right lane are sanctioned (lines 11-12). Finally sensor 5 is checked in order to determine whether any agents have stayed on the right lane against their directive (lines 15-17) or whether any agents did not comply with their velocity directive (lines 19-21).

We also have the other norm that obliges vehicles that enter on the left lane of the main road to stay on their lane at a preset maximum velocity v_{MAX}. This scheme's pseudo code is given in Algorithm 2. We begin with creating new norm detachments (lines 1-3). In this case sensor 1 is read. In Figure 7.2 it can be seen that this sensor detects all vehicles that enter the scenario on the left lane of the road. Hence, we give each vehicle on the left lane the directive to stay on the left lane, and also obtain a preset maximum velocity to ensure that the flow stays high (line 3). We then continue by checking sensor 4 (lines 4-8). Vehicles with detachments of this norm that pass sensor 4 fulfilled their directive to stay on the left lane. However, if their velocity is

[3]Unlike other chapters we include the sanction in the norm detachment so that we only have to communicate this construct to agents.

Algorithm 1 Pseudo code for the merge norm scheme

```
1:  L ← merge(read(s_2), read(s_3))
2:  for all agent ∈ L do
3:      s ← optimalVelocity(agent)
4:      if agent.lane = right & s < v_threshold then
5:          Δ ← Δ ∪ {(agent, ((left, v_MAX), high))}
6:      else
7:          Δ ← Δ ∪ {(agent, ((right, s), high))}
8:  L ← read(s_4)
9:  for all agent ∈ L do
10:     Δ ← Δ\{(agent, ((left, v_MAX), high))}
11:     if (agent, (right, s), high)) ∈ Δ then
12:         sanction(agent, high)
13: L ← read(s_5)
14: for all agent ∈ L do
15:     if (agent, (left, v_MAX, high)) ∈ Δ then
16:         sanction(agent, high)
17:         Δ ← Δ\{(agent, ((left, v_MAX), high))}
18:     else if (agent, (right, s), high)) ∈ Δ then
19:         if s ≠ agent.v then
20:             sanction(agent, high)
21:         Δ ← Δ\{(agent, ((right, s), high))}
```

not the obliged velocity, they receive a low fine (lines 6-7). After passing this sensor the vehicles are relieved of their directives (line 8). The same holds for sensor 5 (line 9). However, if a vehicle passes sensor 5 then it means that it switched to the right lane. Hence, each vehicle that has received a directive to stay left but passes sensor 5 is fined (line 12).

7.4 Norm-Aware Vehicles

We have discussed norm-based traffic controllers, but not yet the vehicles/agents that are subjected to the norms. We assume that individual vehicles have their own objectives (e.g. destination, arrival time, travel cost, etc.) and are able to deliberate and decide on actions that achieve their objectives. This implies that a vehicle can choose to obey/violate a norm, when this contributes to the achievement of its objectives. In this section, we will discuss norm-awareness and how we adopt it in our agent system.

We have explained norm-based controllers in a multi-agent system as being external entities to the agents, which may oblige/forbid certain behavior and impose sanctions. This abstraction comes from the human way of organizing. It is beneficial to make agents in a multi-agent system norm-aware, especially in simulations where we want the agents to change their behavior due to the norms as humans do. Norm-

Algorithm 2 Pseudo code for the stay-on-lane norm scheme

1: $L \leftarrow read(s_1)$
2: **for all** $agent \in L$ **do**
3: $\quad \Delta \leftarrow \Delta \cup \{(agent, ((left, v_{MAX}), low))\}$
4: $L \leftarrow read(s_4)$
5: **for all** $agent \in L$ **do**
6: $\quad$ **if** $(agent, ((left, v_{MAX}), low)) \in \Delta \& agent.v < v_{MAX}$ **then**
7: $\quad\quad sanction(agent, low)$
8: $\quad \Delta \leftarrow \Delta \backslash \{(agent, ((left, v_{MAX}), low))\}$
9: $L \leftarrow read(s_5)$
10: **for all** $agent \in L$ **do**
11: $\quad$ **if** $(agent, ((left, v_{MAX}), low)) \in \Delta$ **then**
12: $\quad\quad sanction(agent, low)$
13: $\quad \Delta \leftarrow \Delta \backslash \{(agent, ((left, v_{MAX}), low))\}$

awareness falls under the umbrella of organizational awareness [135]. Organizational awareness is about allowing agents to reason about their role within an organization. However, for traffic simulation purposes all agents have the same role (i.e. being a driver), hence we will focus on norm-awareness only. Being norm-aware means that an agent can reason about the norms that it receives. The directives that an agent receives are unlikely to match its goals and desires. Otherwise, there would be no need for sanctions. Reasoning about norms hence entails weighing for a course of actions the benefits (reaching goals, fulfilling desires, etc) and the penalties (expected sanctions from the norm-based controller). In general, the more complex the planning mechanism of the agent is, the harder it is to incorporate reasoning about norms. For an example language for programming norm-aware agents that can deliberate about norms with respect to their own goals and plans we refer the reader to N-2APL [6].

Agents in traffic simulations are implemented as driver models. In order to model norm-awareness we deviate from the standard SUMO driver models [85]. There are several reasons for this. First, vehicles in SUMO are goal-directed only in a limited way. For example, the goal of SUMO drivers is to follow a certain route, as opposed to the goal of having a specific location as the destination. Second, SUMO agents are preprogrammed to follow a specific route. They only respond reactively to their environment, instead of deliberating on what action would best suit them. Third, vehicles in SUMO are inherently incapable of deciding to break the rules. For example, they always stop for a red light and they obey to the right of way. This is because vehicles in SUMO move according to specific car-following and lane-changing rules. The car-following model is not created with norm-aware vehicles in mind. The most commonly used car-following model made by Stefan Krauss is designed purely to create realistic traffic flows in general since in most traffic simulations individual movement on the microscopic level is not interesting [85]. In contrast, we aim at designing futuristic autonomous cars with a more fine grained sense of control and most importantly, the ability to violate norms. In our work we model drivers with different possible actions, a belief state, and personal preferences. The available

actions of a vehicle depend on the scenario. The belief state of a vehicle consists of:

- A description of its runtime variables which contain its velocity, position (given by a lane and distance from that lane's start) and current norm detachments.
- An expected arrival time (at the goal location) function that reflects for instance the GPS planning tools that vehicles have available. This function is equal for all vehicles, but can be parameterized in order to make more optimistic/pessimistic drivers.
- A local effect function that returns the next expected runtime variable configuration, given the runtime variables of a vehicle and an action. For instance, if the current velocity is 20 m/s and a vehicle accelerates by 5 as an action, then it expects for the next simulation tick to be at 25 m/s if this is possible within its acceleration capabilities. This function is also equal for all vehicles.
- A directive distance function that returns a positive expectancy of whether the vehicle can fulfill the directive in time before it is sanctioned, given runtime variables and a directive. The precise definition of this function depends on the possible directives and driver specifics in an application. However, a high distance should mean that it is likely that the sanction will be incurred in the future whereas a distance of zero should indicate that the current state fulfills the directive. This function is also equal for all vehicles.

The personal preferences of a vehicle are given by its personal profile. This profile consists of:

- A maximum desirable velocity of the vehicle.
- A sanction grading function that returns how bad a sanction is to the vehicle. The higher the number, the worse the sanction. This can be used to model how affluent or greedy a vehicle is.
- An arrival time grading function that returns how good or bad an arrival time is. This encodes the desired arrival time of the vehicle. Anything before the desired time should be evaluated to zero or less, and everything after it should monotonically increase in the evaluation.

Our vehicles use a sense-reason-act cycle. In our extension this cycle is performed at each simulation tick of SUMO. In practice it is not required that the vehicles run synchronously with the traffic controllers and/or other vehicles. In practice a vehicle perceives its road environment by its on-board sensors. In our extension this information is obtained from SUMO, which gives a vehicle its local information. A vehicle also receives newly instantiated or retracted directives that apply to itself. This information updates the vehicle's belief state (its runtime variables). Using its knowledge of the road network, the vehicle estimates the utility of each of its actions by balancing between the value of achieving its objectives and possible sanctions that will occur if it performs the action. The arrival time and expected sanctions are used to calculate the utility for each action.

The action with the highest utility is chosen by the agent's action selection function, using a priority based tie-break mechanism for ties. The tie-break mechanism is used when two or more actions have the highest utility. If such a situation occurs, then the action with the highest priority is chosen. In our implementation, the less an action changes the agent state, the higher its priority is. For example, in a tie-break situation, doing nothing is preferred to increasing velocity to a small amount, which in turn is preferred to increasing velocity to a larger amount, which in turn is preferred to changing lane. We chose for this 'least impactful action' tie-break ordering since we believe this to be in line with human behavior. However, we stress that this ordering is not essential to our framework and can be replaced by arbitrary tie-break orderings that correspond more closely to human behavior. Finally the simulator executes this action.

Recall that S is the set of possible sanctions, O the set of possible directives and N the set of possible norm detachments. The relevant components of an agent are modeled as follows. An instantiation of these components and example is given in the next section.

- A is the set of actions to choose from.
- $\langle v, l, d, \Delta \rangle$ is a specification of the runtime variables of a vehicle, where v is the current velocity, l is the current lane, and d is the distance from the starting point of that lane and $\Delta \subseteq N$ are norm detachments.
- B is the set of all possible runtime variables configurations.
- $f : B \mapsto \mathbb{N}$ is the expected arrival time function.
- $e : B \times A \mapsto B$ is the local action effect function.
- $\delta : B \times O \mapsto \mathbb{R}$ is the directive distance function.
- $\langle v_{max}, g_s, g_t \rangle$ is a specification of a vehicle's personal profile, where v_{max} is the maximum velocity, $g_s : S \mapsto \mathbb{R}$ is the sanction grading function and $g_t : \mathbb{N} \mapsto \mathbb{R}$ is the arrival time grading function.
- P is the set of all possible personal profiles.
- $u : B \times P \times A \mapsto \mathbb{R}$ is the utility function, given by:

$$u(b, p, a) = g_t(f(b')) + \sum_{(o,s) \in n} (\delta(b', o) \cdot g_s(s)),$$

  where $b = \langle v, l, d, \Delta \rangle$, $p = \langle v_{max}, g_s, g_t \rangle$ and $b' = e(b, a)$.

- $\alpha : B \times P \to A$ is the action selection function given by:

$$\alpha(b, p) = \max_{a \in A} u(b, p, a)$$

7.5 Application of Norm-Aware Driver Models

In this section we shall go into detail how the norm-aware driver models are specified for our scenario. We give a specification of the scenario's specific components and discuss utility calculations.

7.5.1 Vehicle Driver Specification

A vehicle reasons about all its actions when it deliberates about a next action. Among all actions in our scenario are de-/accelerating actions. We have simplified this by discretizing the possible de-/acceleration values. The other possible actions available to a vehicle are switching a lane to the left or right. More specifically, the set of actions A that a vehicle can decide upon to perform are:

$$A = \{a_x \mid x \in \{0, 0.1, -0.1, 1, -1, 5, -5, 10, -10, 20, -20, 50, -50\}\} \cup \{l_{left}, l_{right}\},$$

where a_x is read as adding x to the current velocity (i.e. de/accelerating) and l_{left}/l_{right} is read as moving a lane to the left or to the right.

We also specify how the directive distance measure is calculated given the possible directives in our scenario. In our scenario the factors that a vehicle considers are the time it takes to fulfill the directive, the current time and the expected time that the sanction will be issued if the directive is not followed. We have implemented vehicles in such a way that they expect that the traffic controller will check whether a directive is followed somewhere between the current time t and the current expected exit time t_{exit} for the vehicle. The minimal amount of time needed to adhere to the norm is denoted δ_t. For instance, if a vehicle at time step t is being instructed to drive 25 m/s, and can accelerate to this speed in minimally three time units, then $\delta_t = 3$. If we need zero steps to adhere to the norm, the distance is zero. Otherwise the distance proportionally moves to 1 given the current time. If the traffic controller will check directive fulfillment before the driver can achieve compliance (i.e.$t_{exit} - t < \delta_t$), then δ should be 1. Hence the directive distance measure is given by:

$$\delta(b, (l_o, v_o)) = \frac{\min(\delta_t, t_{exit} - t)}{(t_{exit} - t)},$$

where t is the current time, t_{exit} is the expected exit time and δ_t is the minimal number of steps needed for compliance of (l_o, v_o) given the current runtime variables b.

Note that this means that the drivers expect the traffic controller to issue a sanction if a directive is not followed between now and $t_{exit} - t$ time units later. This distance measure can be modified easily in our framework.

7.5.2 Utility Calculations: Example

To illustrate this notion suppose we have two drivers, a poor one and an affluent one in an identical situation. They currently drive 20 m/s, their maximum speed is 30

m/s, they can accelerate or decelerate with 10 m/s and their travel distance is 1080 meters. Both are in a hurry, so their g_t is defined as $g_t(time) = \frac{bestTime}{time}$.

Here, $bestTime$ is defined by the minimal travel time, i.e. the time it would take the drivers to travel the distance if they could go their maximum speed all the time. In this case, $bestTime = 1080/30 = 36$. However, the road the drivers travel on has a speed norm, with the maximum speed being 10 m/s. Not complying with this norm gives a high fine. The poor agent cannot afford this fine, so it has $g_s(high) = -20$. The affluent driver can easily afford this fine, so it has $g_s(high) = -0.2$. Suppose for this example, that drivers can only take the actions a_0, a_{10} and a_{-10}.

In Table 7.1, we see the utilities for each of these actions for both drivers. Here we see that the highest rewarded action for the poor driver is the one that obeys the norm since it cannot afford the fine, while the affluent driver is in a position to violate the speed norm since it can afford the fine. In fact, the affluent driver will increase its speed since it then maximizes its time grade and thereby its utility.

Table 7.1: *Example of the deliberation of a poor and affluent agent.*

	a_0	a_{10}	a_{-10}
Speed after action	20m/s	30m/s	10m/s
Norm speed	10m/s	10m/s	10m/s
Travel time remaining	54s	36s	108s
g_t	0.67	1.00	0.33
Steps needed to oblige the norm	1	2	0
δ	0.02	0.06	0.00
Utility poor agent	0.30	-0.11	0.33
Utility rich agent	0.66	0.99	0.33

7.6 Demonstrations

We demonstrate the norm-based traffic control system with four experiments. In these experiments variations of the ramp-merging scenario, as explained in Section 7.3, are used. The first experiment considers a ramp-merging scenario where the main road consists of a single lane, while in the second, third and fourth experiment the main road has two lanes. In the second experiment the second lane is accessible for all drivers, but in the third experiment the second lane is marked as an "emergency only" lane. Finally, in the fourth experiment, the ramp-merging scenario is used twice in succession in order to demonstrate the use of communication and coordination between decentralized traffic controllers. We stress that these experiments are intended as proof-of-concepts for norm-based traffic control.

The experiments were set up as follows. Each experiment is run for a length of one simulated hour (3600 ticks). The spawn rate shown in the tables of the experiments is defined as the chance of a vehicle spawning every tick. If there is not enough room to spawn a vehicle at a certain time, then SUMO puts the vehicle on hold and spawns it at the earliest possible time when space is available. The maximum speed at the merge point, v_{max}, was set to 80 km/h. Furthermore, the four experiments consists

of comparing two scenarios, both ran for one hundred times. The values displayed in the tables are the averages over the hundred runs. The throughput is defined as the number of vehicles leaving the simulation every simulated minute. Average speed is the average speed over all runs in m/s, and finally the average gap is the average distance between two vehicles in meters. Also we define for each experiment the maximum (expected) throughput. This is the expected throughput if each vehicle could keep driving its maximum speed throughout the scenario and can be calculated by the following formula: $throughput_{max} = 60p$, where p is the probability of a car entering the simulation on that tick.

7.6.1 Experiment 1: SUMO and TrafficMAS

The first experiment illustrates the behavior difference between the default SUMO vehicles and the norm-aware driver models implemented in the TrafficMAS extension using the merge norm from Section 7.3. In this scenario a classic ramp-merging situation is implemented, where both the main road and ramp consist of a single lane. A single traffic controller observes the vehicles in the simulation and communicates personal norm detachments to each vehicle. In this experiment a norm detachment is simplified to just a target velocity since there is no choice of lanes on the main road. The expected result is that the enforcement of the norm results in a higher average velocity and a better throughput of vehicles since traffic jams will be prevented. The spawn rate of the vehicle input stream will be slightly higher on the main road to resemble a realistic traffic situation. In the TrafficMAS scenario three sensors are placed on the road, one on the main road, one on the ramp and a control sensor on the output road. The traffic controller creates norm detachments, removes norms or applies sanctions when the vehicles are detected by the sensors. In the SUMO scenario the main road has priority over the ramp road, comparable to real life merging situations.

As is clear from the results in Table 7.2, there is an increase in both throughput, average speed and the average amount of space between the vehicles. This is the case since in the SUMO scenario a traffic jam instantly forms on the ramp, because of the relatively high density of cars on the main road (Figure 7.3). These results confirm our expectation of coordination by a norm-based traffic controller improving on classic ramp-merging scenarios. Note that the throughput percentage value exceeds a hundred percent, this is possible because the spawn rate is probability based and thus can exceed the maximum expected throughput.

7.6.2 Experiment 2: Simple Norms and Advanced Norms

The goal of the second experiment is to compare traffic controllers using simple and advanced norms. We will now simulate two lanes on the main road. The traffic controller in the simple scenario observes and controls the same norm as in Experiment 1. In the advanced scenario the traffic controller can also issue directives for the vehicles to change lanes in order to relieve the rightmost lane traffic and prevent congestion. The lane change directive will be given to a vehicle when its calculated velocity on the merge point is below a certain threshold. For this experiment the

Table 7.2: *The results for the first experiment.*

	SUMO agents	Norm-aware agents
Main road Spawn rate	20%	20%
Ramp Spawn rate	15%	15%
Throughput	16.16	21.01
Max throughput	21	21
Throughput %	76.95%	100.05%
Average Speed	3	20.97
Average Gap	13.81	101.82

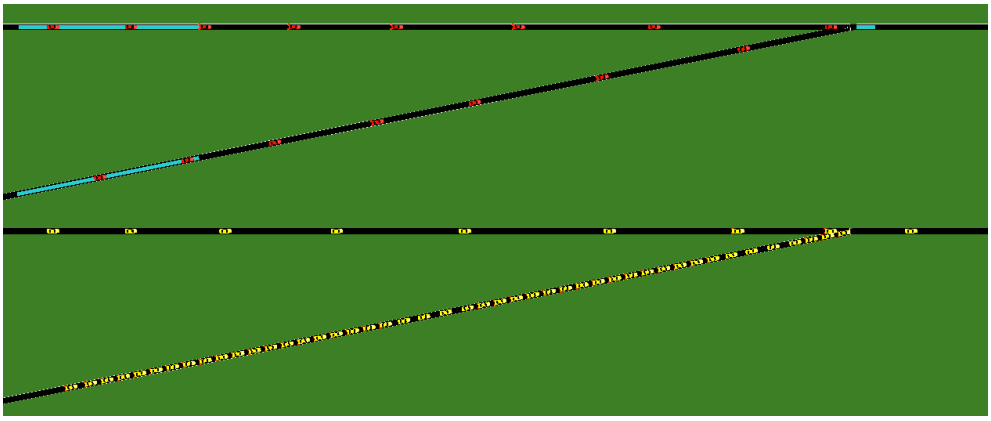

Figure 7.3: *Screenshot depicting the difference in performance in Experiment 1. The top scenario uses our framework and merge norm. The bottom scenario uses the default SUMO driver models.*

Table 7.3: *The results for the second experiment.*

	Simple	Advanced
Main road Spawn rate	30%	30%
Ramp Spawn rate	20%	20%
Throughput	20.38	29.91
Max throughput	30	30
Max throughput %	67.93%	99.70%
Average Speed	3.31	14.91
Average Gap	14.2	61.49

threshold was set to $0.5v_{max}$. The setup for the simple scenario is a copy of the TrafficMAS scenario in experiment 1, except that in this case the main road has two lanes instead of one (both spawn vehicles), and moreover, the input stream of vehicles of both roads are increased. The advanced scenario implements extra sensors on the second lane, but is exactly the same in every other aspect. Our expectation is that in this multi-lane scenario, the traffic controller with the advanced norm can successfully cope with a higher input stream of vehicles.

As can be observed from the results in Table 7.3, the simple norm cannot cope properly with the increased spawn rate of vehicles in this scenario. The average speed has diminished severely, as well as the average gap between vehicles. This means congestion is abundant in the simple scenario. However, the advanced scenario seems to cope very well with the increased input stream of vehicles. In this scenario the throughput approximates the maximum expected throughput by a factor 0.3%, which indicates that the vehicles move throughout the simulation without much congestion.

7.6.3 Experiment 3: Sanction Severity

The third experiment illustrates norm-awareness and its impact. Experiment 1 has shown that drivers are norm-aware. However, TrafficMAS agents also have the capabilities to violate norms if these violations do not have significant impact to them. In this experiment the leftmost lane is an emergency lane, reserved for certain traffic in order to help with accidents and other emergencies (e.g., as the permit lane in the examples from Chapter 2). Therefore regular drivers will get sanctioned if caught driving on this lane. Since this lane remains mostly empty, this is a viable option for drivers who greatly value a faster arrival time and are in a financial position which makes them willing to accept a fine. This experiment is set up in the same way as Experiment 2, except that the leftmost lane is reserved for emergencies and the spawn rates are lowered. In the Poor Drivers scenario, the input stream consists of drivers who are impatient, but in a substandard financial position. The Affluent Drivers scenario spawns drivers who care about sanctions, but are willing to accept fines if by doing so they can arrive earlier to their destinations. We expect that the more affluent drivers will choose to accept sanctions in order to improve their arrival time, resulting in distinct behavior between the two groups of drivers.

Table 7.4: *The results for the third experiment.*

	Poor drivers	Affluent drivers
Main road Spawn rate	20%	20%
Ramp Spawn rate	15%	15%
Throughput	20.48	20.88
Max Throughput	21	21
Max Throughput %	97.52%	99.43%
Average Speed	12.95	14.42
Average Gap	48.37	69.29
Sanctions	0	133.12

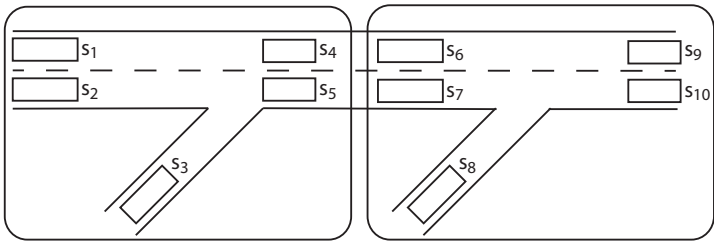

Figure 7.4: *Distributed traffic control setting. Rounded boxes indicate local traffic controllers. The left controller is connected to sensors 1 to 5 and the right controller to sensors 6 to 10*

The results of this experiment are listed in Table 7.4. With this experiment, the differences in throughput, average speed, and average gap are much smaller, and not significant enough to lead to any conclusions about improvement. However, a significant distinction in the number of sanctions can be seen. This indicates a difference in behavior between the groups of drivers. On average about 133 affluent drivers decide to drive on the emergency lane in an hour of simulation. This shows a clear difference in behavior from the poor drivers, who never decide to change lanes.

7.6.4 Experiment 4: Distributed Traffic Control Systems

The final variation of the merging scenario that we consider demonstrates the decentralized version of our framework. Specifically we demonstrate the ability of one traffic controller to share data about traffic with another traffic controller so that the receiver can adjust its norms. In this variation there are two merge points in sequence (Figure 7.4). Each of the merge points is controlled by a local traffic controller as in the previous scenarios. For this they have their own local sensors.

When running this scenario without communicating traffic controllers, we observed that the traffic streams tend to flow like the top situation in Figure 7.5. Traffic that arrives on the left lane of the main road keeps that lane, as it is faster than switching

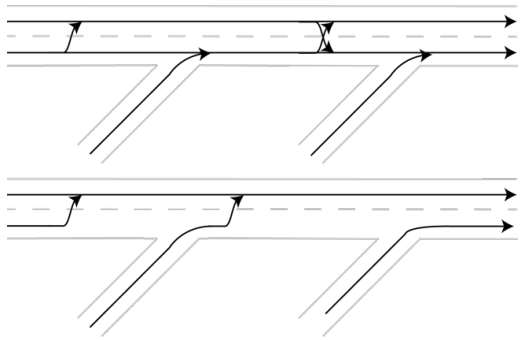

Figure 7.5: *Top: Traffic streams (arrows) without coordination. Bottom: traffic streams with coordination.*

to the right lane. The ramp traffic streams merge in on the right lane of the main road. After the merge scenario the vehicles can freely move from left to right and back. However, if the stream of vehicles in the second ramp is too dense, then congestion occurs at the second merge point. The problem is that the second merge point has to process too many vehicles.

A solution might be to redirect all traffic observed by the left traffic controller to the left lane of the main road when high density streams are observed at the ramp road of the right traffic controller (the second ramp road). This way all traffic on the second ramp road can continue through on the right lane of the main road without being obstructed by oncoming traffic. However, the left traffic controller can only sense the traffic situation using its local sensors. Therefore the right traffic controller needs to inform the left traffic controller about the traffic density on the second ramp.

This coordination is realized as follows. The left traffic controller subscribes to traffic density observed by sensor s_8 of the right traffic controller. If the left traffic controller detects a high traffic density on the second ramp, it will issue new norm detachments that oblige vehicles to move to the left lane. As a result the input traffic streams of the right traffic controller should be easily manageable as the vehicles on the main road are obliged to stay left such that the vehicles on the second ramp roads can move on the right lane of the main road (see the stay-on-lane norm of Section 7.3). The resulting traffic streams should resemble the streams in the bottom depiction of Figure 7.5. We expect that the coordinating traffic controller perform better in terms of throughput, average speed, and average gap, since less congestion should occur at the second merge point.

The results of experiment four are listed in Table 7.5. A small increase in the throughput and a larger increase the average speed and gap in the coordinated traffic controllers scenario compared with uncoordinated traffic controllers scenario can be observed. Thus, giving vehicles on the main road the obligation to change to the left lane quickly after the first merging point appears to prevent the delays as observed in the original scenario. These preliminary results support our hypothesis that observation sharing and communication between traffic control systems can be effective for

Table 7.5: *The results for the fourth experiment.*

	No Coordination	Coordination
Main road Spawn rate	25%	25%
Ramp #1 Spawn rate	15%	15%
Ramp #2 Spawn rate	35%	35%
Throughput	43.81	44.74
Max throughput	45	45
Max throughput %	97.36%	99.36%
Average Speed	11.32	14.60
Average Gap	62.47	66.03

traffic regulation.

7.7 Contributions and Related Work

The results of our experiments were positive in the sense that the advanced version of our framework performed better than the SUMO baseline/simpler versions of our framework. However, as noted previously, some scenarios were not completely realistic. On an actual highway there is an acceleration lane which allows vehicles to adjust to the main road's average speed. The benefit of the acceleration lane is that vehicles have a longer window of opportunity to switch lanes onto the main road. This allows them to merge more efficiently since they may adjust their speed to fit into a gap on the main road. Our merge scenario did not have an acceleration lane aligned with the main road. Moreover, in the final experiment the traffic density of the second merge lane was higher than one would expect in the real life. However, the aim of our experiments was not to show simulate calibrated realistic scenarios. Our goal was rather to show that our extension is an enabling technology for the specification and testing of norm-based traffic control, which can be extended into a more complex and feature rich framework.

The contributions of this chapter are as follows. First of all, we have created a lightweight framework for autonomous norm-aware vehicles and norm-based traffic controller on top of SUMO. This framework is easy to extend with different types of driver profiles. It also allows for the easy usage of a different simulation package. Secondly, we have created the possibility to conduct traffic experiments and measuring the impact of a norm-based traffic controllers.

Our approach has some similarity with Baines et al. [15] since they employ autonomous agents and use governing institutions to influence agents to have desirable behavior. However, Baines et al. concentrate on the agents' internal architecture, situational awareness, and the communication between agents. The project is set up with realistic maps imported from the Open Street Map foundation and used real-world data from a highway in the UK, the M25. While our framework is related to the work done by Baines et al., the aim of our research is different. Our driver

model is deliberately kept simple in order to focus on the interaction between traffic controllers on the one hand, and between agents and traffic controller on the other hand. Finally, our framework supports decentralized traffic controllers while Baines et al. focus on a single, all knowing, institution.

Another comparable line of research has been done by Balke et al. [16]. In this extended abstract, Balke et al. discuss the difference between off-line and on-line reasoning of institutions (similar to norm-based traffic controllers) governing open multi-agent systems. They state that most research up to that point had been focused on the off-line reasoning of institutions, which can be used to research the static properties of institutions. The on-line reasoning of institutions concerns the monitoring and controlling of agents, observing if norms are being violated and informing agents if this is the case. In this implementation, there is a single institution with the title "The Governor" with which agents can communicate and receive information regarding possible consequences of their actions. Our approach is most related to the on-line reasoning as described by Balke et al. However, communication between the agents and the institution is handled in a different way. With our framework, the information provided by agents to the traffic controller is acquired via sensors. This is a more realistic representation of traffic situations, since it is often beneficial for the agent to not disclose any information about itself. Furthermore, in TrafficMAS, multiple traffic controllers are present, creating a more robust and better controlled system through communication and coordination between these institutions.

7.8 Conclusion

Our goal was to create a traffic simulation multi-agent system that demonstrates (decentralized) norm-based traffic control, and where vehicles should generally follow traffic regulations, yet are able to ignore these regulations in certain circumstances without implementing hard constraints on the agents themselves. We used norm-based traffic controllers since they can properly deal with these kinds of situations. Our work is an extension to SUMO. It features a norm-based traffic controller which monitors and possibly sanctions the vehicles. We assume deliberative proactive drivers that make autonomous decisions according to their goal and received sanctions. The extension features i) driver profiles which model different types of behavior, ii) traffic controllers and norms to control vehicles, and iii) an easy way to add new driver profiles, traffic controllers and norms. This plug and play extension to SUMO can serve as a testing suite for experiments concerning norm-based traffic control.

We showed in our demonstrations that the performance of our example normative system is better than the default behavior in a ramp-merge scenario. Furthermore, we presented that complex norms allow for a more fine grained steering of behavior in complicated scenarios. Moreover, we illustrated the autonomy of drivers, by demonstrating a different in behavior between poor and affluent drivers. Finally, we demonstrated the ability of traffic controllers to coordinate their activities, yielding better results in certain scenarios.

8 Conclusion

In this chapter we reflect on the thesis as a whole. We also provide topics of interest which we did not discuss in this thesis but are relevant and interesting for future research.

8.1 Answering the Questions

In this section we revisit and answer the questions from the introduction.

8.1.1 Question 1

The first question, which we mainly addressed in Chapter 2, is:

Research Question 1: *What are the core concepts for modeling decentralized runtime norm enforcement mechanisms?*

The short answer to this question is that norms, decentralized runtime monitoring and control, and the assignment of norms to institutions, are the core concepts for decentralized runtime norm enforcement. In a decentralized setting we have to deal with the locality of observation and control capabilities, which individually may be insufficient for norm enforcement. Norms tend to be specified as global rules of behavior which have to be enforced by local institutions. For an institution to enforce a norm it is necessary that it has access to the appropriate observation and control capabilities. In a decentralized setting such capabilities may only be accessible through collaboration with other institutions. Furthermore, the task of checking whether a norm violation has occurred (monitoring) is different from the task of executing a sanction or halting execution altogether (control), and the task of determining what sanction to apply given a norm violation. All these three tasks, that are required for each norm, might be distributed across multiple institutions, leading to different types of enforcement (centralized, decoupled and decentralized enforcement). We proposed a model for decentralized institutions in Chapter 2 that was used to highlight these concepts and aspects of decentralized runtime norm enforcement mechanisms.

8.1.2 Question 2

The second question, which we addressed in Chapters 2, 3, 4 and 5, is:

Research Question 2: *How to computationally model runtime monitoring and control mechanisms for decentralized runtime norm enforcement?*

To answer this question we first require a model of norms. As an initial model we used in Chapter 2 counts-as rules for this purpose. Norms are a means to specify desirable behavior, and behavior can be seen as patterns in sequences of states or actions that agents (or other software) bring about. Linear temporal logic (LTL) is a suitable logic for reasoning about sequences of states or actions, and this is why we turned to LTL to model norms in Chapters 3, 4 and 5. This logic proved to be particularly useful regarding our runtime setting as a run of a system is a linear sequence of states or actions that is revealed incrementally.

In Chapters 3 and 4 we discussed progression based monitoring from [19] and proposed two new monitoring frameworks; called delay and aggregation based monitoring. Each of the frameworks has its own advantages and disadvantages (Section 4.2.3) that make them applicable in different situations. These models differ from centralized monitoring because we have to consider what information is communicated between monitors and under what conditions. In Chapter 2 we mentioned that norm assignments are an important concept for decentralized runtime norm enforcement mechanisms. With respect to monitoring we showed for this concept that progression based monitoring is suitable when multiple monitors have to collaboratively monitor the violations of a single norm. Aggregation based monitoring is suitable when for a collection of monitors some monitors have a norm for which they much check whether norm violations occur. Finally, delay monitors are suitable when a monitor can have a set of norms for it much check whether violations occurred.

Literature on the control of discrete event systems contains various models of automaton-based controllers that 'rewrite' the sequence of actions that a target system produces. We adopted the notion of edit automata [90] to runtime norm enforcement by establishing the connection between norm enforcement and such control theories. We proved that regimentation and sanctioning based enforcement of norms is related to the notions of precise and effective enforcement from discrete event control systems. We discussed the use of individual controllers and showed how they can be combined into collaborative controllers in order to obtain decentralized runtime control for norm enforcement. The decentralized setting forces us to address the issue of making the controllers compatible with each other. We formally defined decentralized regimentation, sanctioning and the combination thereof. For the combination of regimentation and sanctioning we showed that we must decompose the challenge to first designing a collaborative controller for the norms that have to be regimented, and then apply consecutively to the result a collaborative controller that enforces the remaining norms through sanctioning.

8.1.3 Question 3

The third question, which we addressed in Chapters 2, 3, 4 and 5, is:

Research Question 3: *What are the desirable properties of decentralized runtime norm enforcement mechanisms?*

An initial question that we must ask ourselves is in which situations a runtime norm enforcement mechanism is actually useful, since usefulness is a desirable property of any mechanism. To this end we discussed the monitorability and enforceability of norms that are expressed in LTL in Chapters 3 and 5. We specified monitorability with respect to a target system and the currently revealed behavior of that system, which was modeled as a word of states or actions. This way we can specify when a norm becomes non-monitorable at runtime and consequently when we can shutoff a norm enforcement mechanism.

If a norm enforcement mechanism is useful, then we also want it to correctly identify norm violations and being able to discover all violations, and that it correctly enforces the norm through regimentation or sanctioning. We proved for our decentralized runtime monitor mechanism for norm enforcement in Chapter 3 that it correctly monitors LTL formulas and hence can correctly identify all norm violations of a norm that is specified as an LTL property. In Chapter 5 we proved that for a set of norms that is to be regimented, or a set that is to be sanctioned, that there exists a collaborative controller such that none of the norms are violated in case of regimentation, or that each of the violations is properly sanctioned, in case of sanctioning.

In Chapter 3 we discussed that in a decentralized setting we might be specifically interested in limiting the number of messages that are sent among monitors, and the delay with which a norm violation is detected. We showed that all the monitoring frameworks have an upper bound (which is the number of local monitors) on the number of messages that have to be communicated between monitors at each tick of the target system. In a decentralized setting it is possible that the detection of a norm violation is delayed due to the monitor's communication model. For the delay monitor framework we showed that the maximum number of ticks after which a norm violation is detected equals the number of local monitors (this also holds for progression based monitoring [19]). In case of aggregation based monitoring we did not model delays of communication, hence that model does not include a delay in which norm violations are detected.

We also looked at the security and robustness aspects of decentralized runtime monitoring. Wireless sensor networks (WSNs) are an example of practical decentralized monitoring systems. Literature from the WSN community contains many proposals for solutions to robustness and security concerns. Some of these solutions can be transferred to decentralized monitoring to check for norm violations. For instance, one common aspect of WSNs is data aggregation. Aggregation is used to combine the input of child sensors and combine it with local observations, which in turn is shared with parent sensors. This way the parent does not know exactly what data brought about the input of its children. If an attacker pretends to be a parent, then the revealed information is also limited. In Chapter 4 we transferred this idea to a decentralized LTL monitoring mechanism which is the aggregation based framework. We also proposed metrics for formal models of such monitors so that we can identify which monitors require extra attention when it comes to security and robustness.

8.1.4 Question 4

The fourth question, which we addressed throughout all the chapters, is:

Research Question 4: *How to develop a decentralized runtime norm enforcement mechanism?*

For our models of decentralized runtime montoring mechanisms we described by examples how they should execute at runtime in Chapters 3 and 4. For the model of decentralized runtime control we gave operational semantics in Chapter 5.

On the engineering side we drew inspiration from the field of design patterns for object-oriented programming. Design patterns are used in various programming fields in order to make solutions to often reoccurring problems reusable. For engineering normative multi-agent system we also have often reoccurring problems such as the specification of agents and norms. We provided for agents and norms design patterns, where for norms we used aspect-oriented programming. We argued that aspect-oriented programming is a suitable implementation technique for norms since norms are a separate concern from the business logic of the system that is being controlled.

We made a proof-of-concept application where decentralized norm enforcement is applied to highway traffic. We used an existing traffic simulator (SUMO [84]) as a basis and expanded it for our scenarios. The application shows that if we can give very precise directives to (autonomous) traffic, that this can then lead to better throughput and safety. It also shows that we can model the willingness of agents to adhere to the norms and see what happens when they are more, or less, inclined to follow the norms. A smart roads application that applies decentralized norm enforcement thus is a decentralized information gathering system which evaluates sensor data, sees which norms apply to what agents, notifies of the norms that apply to them, and sanctions agents that violate the norms.

8.1.5 Main Question

Our main question is:

Main Research Question: *How can a decentralized runtime norm enforcement mechanism be modeled, analyzed and developed?*

As a summary, we can model, analyze and develop a decentralized runtime norm enforcement mechanism by:

- Identifying the observation and control capabilities of the norm enforcement mechanism and grouping them together in meaningful institutions.
- Modeling the norms as LTL properties.
- Analyzing whether the norms can be meaningfully enforced; are they monitorable and enforceable?
- Determining whether the monitoring part of the enforcement mechanism monitors a single norm or a set of norms, and which monitor can and should monitor what norm(s).

- Choosing an approach towards observation sharing and using that to design a decentralized runtime monitor for detecting norm violations.
- Investing extra attention in those components of the monitor that are critical to robustness and safety. For the analysis of security and robustness we can use the metrics that are provided in Chapter 4.
- Designing the individual controllers that may regiment or sanction norms. The controllers can be applied concurrently using a collaborative controller model, which requires us to design a selection function over action revisions for when controllers propose different revisions at the same time.
- Developing the decentralized runtime norm enforcement mechanism by implementing the intended execution and the operational semantics of the monitoring and control mechanisms. For conditional obligations and prohibitions we may use aspect-oriented programming.
- Transforming the often reoccurring solutions in the development process into design patterns.

8.2 Main Contributions

We strongly hope that the contributions of this thesis will stimulate the transfer of theory from the field of normative multi-agent systems to practice, which hopefully will attract more attention for the field as a whole. In summary, our contributions in this thesis are:

- A conceptual model for decentralized institutions (Chapter 2).
- Two models for decentralized monitoring with the purpose of detecting norm violations (Chapter 3 and Chapter 4). We also compared these models with a related model from literature that is based on formula progression.
- A discussion and analysis of robustness and security aspects with respect to decentralized monitoring (Chapter 4).
- A theoretical bridge between runtime property enforcement from the field of discrete event control systems and norm enforcement (Chapter 5).
- A model for combining individual controllers into collaborative controllers for norm enforcement (Chapter 5).
- A specification of design patterns to support the development of normative multi-agent systems with the object-oriented programming paradigm, and an accompanying library of Java classes and aspects (Chapter 6).
- A proof-of-concept application that illustrates the use of norm enforcement for the regulation of future traffic with autonomous vehicles on smart roads (Chapter 7).

8.3 Future Work

Throughout the different chapters we pointed out some interesting future research directions for decentralized runtime norm enforcement. One future direction includes a more fundamental investigation of decentralized runtime norm enforcement through formal modeling. For instance, we have separately discussed monitoring and control, but these could be combined in a single framework that integrates the decentralized monitoring frameworks with collaborative controllers. Also, we may investigate the relation between theoretical models and efficient implementations. E.g. are some classes of norms more efficiently implementable than others? For the transfer of theory to practice it is also interesting to have in the future algorithms that provide, given a decentralized enforcement mechanism and set of norms, the most efficient distribution of those norms over the mechanism.

Future work also includes institution dynamics. A given infrastructure does not have to be permanent. We can imagine some decentralized institution to be expandable with new institutions and communication relations among them. For instance a highway network can be expanded with new highway institutions, or become connected to other institutions. It would be interesting to investigate enforcement under the possibility of a dynamic decentralized institution. It would be ideal to have a running network of institutions that in case of failing institutions or dynamic circumstances can restore itself and keep enforcing the norms.

On the engineering side we can still further expand upon the design patterns for normative multi-agent systems. For the development of agents in normative systems we still need patterns for dealing with norm-awareness. We also aim at further developing the accompanying library of classes and aspects from this thesis. Among other topics, we want to include support for debugging normative multi-agent systems, which is still a notoriously hard problem.

Lastly, we have paid little attention to agents. We followed lines of earlier research where institutions can observe and act on their own as entities, rather than being implemented endogenously inside agents. This does leave questions open such as whether norm-awareness in decentralized institutions is different from centralized institutions. Or if an institution depends on an agent to control an environment state, what happens if that agent leaves?

In conclusion, we have provided several theoretical and practical contributions for decentralized norm enforcement, but those are definitely not the end of the research in this topic. Many exciting research efforts still lie ahead.

Bibliography

[1] G. Abdelkader. Requirements for achieving software agents autonomy and defining their responsibility. In *Proceedings of the Autonomy Workshop at AAMAS 2003*, volume 236, 2003.

[2] E. Ackerman. Tesla working towards 90 percent autonomous car within three years. `http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/tesla-working-towards-90-autonomous-car-within-three-years`. Accessed: 2015-06-29.

[3] T. Ågotnes, W. Van Der Hoek, J. Rodriguez-Aguilar, C. Sierra, and M. Wooldridge. On the logic of normative systems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1181–1186. AAAI Press, 2007.

[4] T. Ågotnes, W. van der Hoek, and M. Wooldridge. Normative system games. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2007)*, pages 1–8, New York, NY, USA, 2007. ACM.

[5] N. Alechina, N. Bulling, M. Dastani, and B. Logan. Practical run-time norm enforcement with bounded lookahead. In *Proceedings of the 14th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2015)*, pages 443–451, 2015.

[6] N. Alechina, M. Dastani, and B. Logan. Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2012)*, volume 2, pages 1057–1064. International Foundation for Autonomous Agents and Multi-Agent Systems, 2012.

[7] N. Alechina, M. Dastani, and B. Logan. Reasoning about normative update. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, pages 20–26. AAAI Press, 2013.

[8] N. Alechina, M. Dastani, and B. Logan. Norm approximation for imperfect monitors. In *Proceedings of the 13th International conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2014)*, pages 117–124, 2014.

[9] N. Alechina, J. Y. Halpern, I. A. Kash, and B. Logan. Incentivising monitoring in open normative systems. In *AAAI*, pages 305–311, 2017.

[10] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[11] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[12] Y. Aridor and D. B. Lange. Agent design patterns: Elements of agent application design. In *Proceedings of the 2nd International Converence on Autonomous Agents (AGENTS 1998)*, pages 108–115. ACM Press, 1998.

[13] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence*, 22(1-2):5–27, 1998.

[14] V. Baines and J. Padget. On the benefit of collective norms for autonomous vehicles. In *Proceedings of 8th International Workshop on Agents in Traffic and Transportation*, 2014.

[15] V. Baines and J. Padget. A situational awareness approach to intelligent vehicle agents. In *Modeling Mobility with Open Data*, pages 77–103. Springer, 2015.

[16] T. Balke, M. De Vos, J. Padget, and D. Traskas. On-line reasoning for institutionally-situated BDI agents. In *The 10th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2011)*, volume 3, pages 1109–1110. International Foundation for Autonomous Agents and Multi-Agent Systems, 2011.

[17] T. Balke, M. De Vos, and J. A. Padget. Evaluating the cost of enforcement by agent-based simulation: A wireless mobile grid example. In *Proceedings of the 16th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2013)*, pages 21–36. Springer, 2013.

[18] L. D. Baskar, B. De Schutter, J. Hellendoorn, and Z. Papp. Traffic control and intelligent vehicle highway systems: a survey. *IET Intelligent Transport Systems*, 5(1):38–52, 2011.

[19] A. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[20] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[21] J. Baumfalk, B. Poot, B. Testerink, and M. Dastani. A SUMO extension for norm based traffic control systems. In *Proceedings of the SUMO 2015 User Conference Intermodal Simulation for Intermodal Transport*, volume 28, pages 63–82, 2015.

[22] D. Beauquier, J. Cohen, and R. Lanotte. Security policies enforcement using finite edit automata. *Electronic Notes in Theoretical Computer Science*, 229(3):19–35, 2009.

[23] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with Jade*. Wiley, 2007.

[24] J. Bloch. *Effective Java (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[25] G. Boella and L. van der Torre. Norm governed multiagent systems: The delegation of control to autonomous agents. In *Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology (IAT 2003)*, pages 329–335. IEEE, 2003.

[26] G. Boella and L. van der Torre. Regulative and constitutive norms in normative multi-agent systems. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004)*, pages 255–266. AAAI Press, 2004.

[27] G. Boella, L. Van Der Torre, and H. Verhagen. Introduction to normative multiagent systems. *Computational & Mathematical Organization Theory*, 12(2-3):71–79, 2006.

[28] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, 2013.

[29] R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multi-Agent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[30] R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Tools and Applications*. Springer, 2009.

[31] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[32] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In *Proceedings of the Workshop on Programming Multi-Agent Systems (ProMAS) at AAMAS 2004*, volume 3346, pages 44–65. Springer, 2004.

[33] J. Broersen, F. Dignum, V. Dignum, and J.-J. C. Meyer. Designing a deontic logic of deadlines. In *Deontic Logic in Computer Science*, pages 43–56. Springer, 2004.

[34] N. Bulling. A survey of multi-agent decision making. *KI*, 28(3):147–158, 2014.

[35] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK intelligent agents - components for intelligent agents in Java. Technical report, Agent Oriented Software Pty. Ltd, 1999.

[36] M. Cabano, E. Denti, A. Ricci, and M. Viroli. Designing a BPEL orchestration engine based on respect tuple centres. *Electronic Notes Theoretical Computer Science*, 154(1):139–158, 2006.

[37] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[38] E. Chang, Z. Manna, and A. Pnueli. The safety-progress classification. In F. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94, pages 143–202. Springer, 1993.

[39] O. Cliffe, M. D. Vos, and J. Padget. Specifying and reasoning about multiple institutions. In *Proceedings of the Workshop on Coordination, Organization, Institutions and Norms in agent systems at AAMAS 2006*, volume 4386, pages 67–86. Springer, 2006.

[40] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[41] R. Conte and C. Castelfranchi. *Cognitive and Social Action*. UCL Press, 1995.

[42] S. Cranefield. A rule language for modelling and monitoring social expectations in multi-agent systems. In O. Boissier et al., editor, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, volume 3913 of *LNCS*, pages 246–258. Springer Berlin Heidelberg, 2006.

[43] M. Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008.

[44] M. Dastani. Programming multi-agent systems. *Knowledge Engineering Review*, 30(4):394–418, 2015.

[45] M. Dastani and J. Gomez-Sanz. Programming multi-agent systems. *The Knowledge Engineering Review*, 20(2):151–164, 2006.

[46] M. Dastani, D. Grossi, J.-J. C. Meyer, and N. Tinnemeier. Normative multi-agent programs and their logics. In J. Meyer and J. Broersen, editors, *Knowledge Representation for Agents and Multi-Agent Systems*, volume 5605 of *Lecture Notes in Computer Science*, pages 16–31. Spring, 2008.

[47] M. Dastani, J. Hulstijn, and L. W. N. van der Torre. How to decide what to do? *European Journal of Operational Research*, 160(3):762–784, 2005.

[48] M. Dastani, J. C. Meyer, and D. Grossi. A logic for normative multi-agent programs. *Journal of Logic and Computation*, 23(2):335–354, 2013.

[49] M. Dastani and B. Testerink. From multi-agent programming to object-oriented design patterns. In *International Workshop on Engineering Multi-Agent Systems at AAMAS 2014*, pages 204–226. Springer, 2014.

[50] M. Dastani, N. A. Tinnemeier, and J.-J. C. Meyer. A programming language for normative multi-agent systems. *Multi-Agent Systems: Semantics and Dynamics of Organizational Models*, pages 397–417, 2009.

[51] M. Dastani, L. W. N. van der Torre, and N. Yorke-Smith. A programming approach to monitoring communication in an organisational environment. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2012)*, volume 2, pages 1373–1374, 2012.

[52] F. Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, 1999.

[53] V. Dignum. *A model for organizational interaction: based on agents, founded in logic*. PhD thesis, Universiteit Utrecht, 2004.

[54] T. T. Do, M. Kolp, S. Faulkner, and A. Pirotte. Agent-oriented design patterns: the SKwyRL perspective. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ECEIS 2004)*, pages 48–53, 2004.

[55] M. Drummond. Situated control rules. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–113, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[56] C. Ellis. Team automata for groupware systems. In *Proceedings of the international ACM SIGGROUP conference on supporting group work: the integration challenge*, pages 415–424. ACM, 1997.

[57] M. Esteva. ISLANDER: an electronic institutions editor. In *Proceedings of the 1st International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 1045–1052. ACM Press, 2002.

[58] M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. L. Arcos. Ameli: An agent-based middleware for electronic institutions. In *Proceedings of the 3d International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004)*, volume 1, pages 236–243. IEEE Computer Society, 2004.

[59] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[60] D. Gaertner, A. García-camino, P. Noriega, and W. Vasconcelos. Distributed norm management in regulated multi-agent systems. In *Proceedings of the 6th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2007)*, pages 624–631, 2007.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[62] R. Gay, H. Mantel, and B. Sprick. Service automata. In *Formal Aspects of Security and Trust*, pages 148–163. Springer, 2012.

[63] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[64] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the 6th National Conference on Artificial Intelligence - Volume 2*, AAAI'87, pages 677–682. AAAI Press, 1987.

[65] G. Governatori, J. Hulstijn, R. Riveret, and A. Rotolo. Characterising deadlines in temporal modal defeasible logic. In *Proceedings of Advances in Artificial Intelligence (AI 2007)*, pages 486–496. Springer, 2007.

[66] A. Grizard, L. Vercouter, T. Stratulat, and G. Muller. A peer-to-peer normative system to achieve social order. In *Coordination, organizations, institutions, and norms in agent systems II*, pages 274–289. Springer, 2007.

[67] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[68] M. Hannoun, O. Boissier, J. S. Sichman, and C. Sayettat. Moise: An organizational model for multi-agent systems. In *Advances in Artificial Intelligence*, pages 156–165. Springer, 2000.

[69] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 135–143. IEEE, 2001.

[70] K. V. Hindriks, F. S. d. Boer, W. v. d. Hoek, and J.-J. C. Meyer. Agent programming with declarative goals. In C. C. and L. Y., editors, *Intelligent Agents VII Agent Theories Architectures and Languages*, volume 1986 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2001.

[71] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[72] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Computing*, 5(4):38–46, Oct. 2006.

[73] J. Hübner, O. Boissier, and R. Bordini. A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence*, 62:27–53, 2011.

[74] J. Hübner, J. Sichman, and O. Boissier. Moise+: towards a structural, functional, and deontic model for MAS organization. In *Proceedings of the 1st International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002)*, pages 501–502, New York, NY, USA, 2002.

[75] J. Hübner, J. Sichman, and O. Boissier. $\mathcal{S} - \mathcal{M}oise^{+}$: A middleware for developing organised multi-agent systems. In O. Boissier et al, editor, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems*, volume 3913 of *Lecture Notes in Computer Science*, pages 64–77. Springer Berlin / Heidelberg, 2006.

[76] J. F. Hübner, J. S. Sichman, and O. Boissier. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In B. G. and R. G.L., editors, *Advances in Artificial Intelligence*, volume 2507 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2002.

[77] J. Juziuk, D. Weyns, and T. Holvoet. Design patterns for multi-agent systems: A systematic literature review. In O. Shehory and A. Sturm, editors, *Agent-Oriented Software Engineering*, pages 79–99. Springer Berlin Heidelberg, 2014.

[78] C. Karlof and D. Wagner. Secure Routing in Wireless Sensor Networks: Attacks and Countermeasures. *Elsevier's AdHoc Networks Journal, Special Issue on Sensor Network Applications and Protocols*, 1(2–3):293–315, September 2003.

[79] P. Kavathekar and Y. Chen. Vehicle platooning: A brief survey and categorization. In *Proceedings of the International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (ASME 2011)*, volume 3, pages 829–845. American Society of Mechanical Engineers, 2011.

[80] E. Kendall, P. Krishna, C. Pathak, and C. Suresh. Patterns of intelligent and mobile agents. In *Proceedings of the 2nd international conference on Autonomous Agents*, pages 92–99. ACM Press New York, NY, USA, 1998.

[81] S. C. Kleene. *Introduction to metamathematics*. Ishi Press, 1952.

[82] M. Knobbout. Modelling and verifying normative multi-agent systems. *PhD thesis, SIKS Dissertation Series*, 2016.

[83] R. Koymans, J. Vytopil, and W. P. de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the 2nd annual ACM symposium on Principles of distributed computing*, pages 187–197. ACM, 1983.

[84] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker. Recent development and applications of SUMO–simulation of urban mobility. *International Journal On Advances in Systems and Measurements*, 5(3&4), 2012.

[85] S. Krauss, P. Wagner, and C. Gawron. Metastable states in a microscopic model of traffic flow. *Physical Review E*, 55(5):5597, 1997.

[86] B. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 575–578. IEEE, 2002.

[87] L. Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, 2:125–143, 1977.

[88] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 383–392. IEEE, 2002.

[89] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Berlin Heidelberg, 1985.

[90] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[91] F. Lin and W. Wonham. Decentralized supervisory control of discrete-event systems. *Information Sciences*, 44(3):199 – 224, 1988.

[92] J. Markoff. Google cars drive themselves, in traffic. `http://www.nytimes.com/2010/10/10/science/10google.html`. Accessed: 2015-06-29.

[93] J. McCarthy. Ascribing mental qualities to machines. *Technical report, Stanford University AI Lab, CA 94305*, 1979.

[94] F. Meneguzzi, W. Vasconcelos, N. Oren, and M. Luck. Nu-BDI: Norm-aware BDI agents. In *Proceedings of the 10th European Workshop on Multi-Agent Systems, Dublin, Ireland*, 2012.

[95] M. Minsky. *The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind*. Simon and Schuster, 2007.

[96] N. Minsky and V. Ungureanu. Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9:273–305, 2000.

[97] S. Modgil, N. Faci, F. Meneguzzi, N. Oren, S. Miles, and M. Luck. A framework for monitoring agent-based normative systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2009)*, volume 1, pages 153–160. International Foundation for Autonomous Agents and Multi-Agent Systems, 2009.

[98] V. Morreale, G. Francaviglia, F. Centineo, M. Puccio, and M. C. Bc. Goal-oriented agent patterns with the PRACTIONIST framework. In *Proceedings of the 4th Workshop on Multi-Agent Systems (EUMAS 2006)*, 2006.

[99] H. S. Nwana. Software agents: An overview. *The knowledge engineering review*, 11(3):205–244, 1996.

[100] F. Okuyama, R. Bordini, and A. da Rocha Costa. A distributed normative infrastructure for situated multi-agent organisations. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems (AAMAS 2008)*, volume 3, pages 1501–1504, Richland, SC, 2008.

[101] F. Y. Okuyama, R. H. Bordini, and A. C. Rocha Costa. ELMS: An environment description language for multi-agent simulation. In D. Weyns, H. Dyke Parunak, and F. Michel, editors, *Proceedings of the 1st International Workshop on Environments for Multi-Agent Systems (E4MAS 2004)*, pages 91–108. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[102] A. Oluyomi, S. Karunasekera, and L. Sterling. A comprehensive view of agent-oriented patterns. *Autonomous Agents and Multi-Agent Systems*, 15(3):337–377, 2007.

[103] A. Omicini and E. Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, Nov. 2001.

[104] A. Pathan, H.-W. Lee, and C. S. Hong. Security in wireless sensor networks: issues and challenges. In *Proceedings of the 8th International Conference on Advanced Communication Technology (ICACT 2006)*, volume 2, pages 1043–1045, Feb 2006.

[105] A. Perrig, J. Stankovic, and D. Wagner. Security in wireless sensor networks. *Communications of the ACM*, 47(6):53–57, 2004.

[106] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, Sept. 2002.

[107] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[108] A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification Via Testers. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer Berlin Heidelberg, 2006.

[109] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multi-Agent Systems, Artificial Societies, and Simulated Organizations*, pages 149–174. Springer, 2005.

[110] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989.

[111] A. Ricci, M. Viroli, and A. Omicini. CArtAgO: A framework for prototyping artifact-based environments in multi-agent systems. In *Environments for Multi-Agent Systems III*, pages 67–86. Springer, 2006.

[112] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[113] S. Sauvage. Design patterns for multi-agent systems design. In *Proceedings of the 3rd Mexican International Conference on Artificial Intelligence (MICAI 2004)*, volume 2972 of *Lecture Notes in Computer Science*, pages 352–361, 2004.

[114] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[115] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[116] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI 1987)*, volume 2, pages 1039–1046, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[117] J. Searle. *Construction of social reality*. Free Press, 1995.

[118] Y. Shoham. Time for action. In *Proceedings of The 11th International Joint Conference on Artificial Intelligence (IJCAI 1989)*, volume 2, pages 954–959. Morgan Kaufmann Publishers Inc, 1989.

[119] Y. Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[120] Y. Shoham and M. Tennenholtz. On social laws for artificial agent societies: off-line design. *Artificial Intelligence*, 73(1-2):231–252, 1995.

[121] M. H. Ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in team automata for groupware systems. *Computer Supported Cooperative Work (CSCW)*, 12(1):21–69, 2003.

[122] M. H. ter Beek, G. Lenzini, and M. Petrocchi. Team automata for security: A survey. *Electronic Notes in Theoretical Computer Science*, 128(5):105–119, 2005.

[123] B. Testerink. Norms for distributed organizations: Syntax, semantics and interpreter. Master's thesis, Utrecht University, the Netherlands, 2012.

[124] B. Testerink, N. Bulling, and M. Dastani. A model for collaborative runtime verification. In *Proceedings of the 14th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2015)*, pages 1781–1782, 2015.

[125] B. Testerink, N. Bulling, and M. Dastani. Security and robustness issues in collaborative runtime verification. In V. Dignum, P. Noriega, M. Sensoy, and J. Sichman, editors, *Coordination, Organizations, Institutions, and Norms in Agent Systems XI*, volume 9628 of *Lecture Notes in Computer Science*, pages 376–395. Springer, 2016.

[126] B. Testerink and M. Dastani. A norm language for distributed organizations. In *Proceedings of the 24th Belgium-Netherlands Artificial Intelligence Conference (BNAIC 2012)*, pages 234–241, 2012.

[127] B. Testerink, M. Dastani, and N. Bulling. Distributed controllers for norm enforcement. In *Proceedings of the 22nd European Conference on A.I. (ECAI 2016)*, volume 285, pages 751–759. IOS Press, 2016.

[128] B. Testerink, M. Dastani, and J.-J. Meyer. Norm monitoring through observation sharing. In *Proceedings of the European Conference on Social Intelligence*, pages 291–304, 2014.

[129] B. Testerink, M. Dastani, and J.-J. Meyer. Norms in distributed organizations. In *Coordination, Organizations, Institutions, and Norms in Agent Systems IX*, pages 120–135. Springer, 2014.

[130] N. Tinnemeier. Organizing agent organizations: syntax and operational semantics of an organization-oriented programming language. *SIKS Dissertation Series*, 2011(02), 2011.

[131] N. Tinnemeier, M. Dastani, and J.-J. Meyer. Roles and norms for programming agent organizations. In *Proceedings of The 8th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2009)*, volume 1, pages 121–128. International Foundation for Autonomous Agents and Multi-Agent Systems, 2009.

[132] N. Tinnemeier, M. Dastani, and J.-J. Meyer. Programming norm change. In *Proceedings of the 9th International Conference on Autonomous Agents and Multi-Agent Systems: volume 1-Volume 1*, pages 957–964. International Foundation for Autonomous Agents and Multi-Agent Systems, 2010.

[133] N. A. Tinnemeier, M. Dastani, J.-J. Meyer, and L. Torre. Programming normative artifacts with declarative obligations and prohibitions. In *Proceedings of The IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2009)*, volume 2, pages 145–152. IET, 2009.

[134] N. A. Tinnemeier, M. Dastani, and J.-J. C. Meyer. Orwell's nightmare for agents? programming multi-agent organisations. In *International Workshop on Programming Multi-Agent Systems (PROMAS 2008)*, pages 56–71. Springer, 2008.

[135] M. B. van Riemsdijk, K. Hindriks, and C. Jonker. Programming organization-aware agents. In *Engineering Societies in the Agents World X*, pages 98–112. Springer, 2009.

[136] P. Varaiya. Smart cars on smart roads: problems of control. *IEEE Transactions on automatic control*, 38(2):195–207, 1993.

[137] W. W. Vasconcelos, A. García-Camino, D. Gaertner, J. A. Rodríguez-Aguilar, and P. Noriega. Distributed norm management for multi-agent systems. *Expert Systems with Applications*, 39(5):5990 – 5999, 2012.

[138] J. Vázquez-Salceda, H. Aldewereld, and F. Dignum. Implementing norms in multi-agent systems. In G. Lindemann, J. Denzinger, I. Timm, and R. Unland, editors, *Multi-Agent System Technologies*, volume 3187 of *LNCS*, pages 313–327. Springer Berlin Heidelberg, 2004.

[139] D. Wampler. Contract4J for design by contract in Java: Design pattern like protocols and aspect interfaces. In *Proceedings of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, 2006.

[140] Z. Wang, L. Kulik, and K. Ramamohanarao. Proactive traffic merging strategies for sensor-enabled cars. In *Proceedings of the 4th ACM international workshop on Vehicular ad hoc networks*, pages 39–48. ACM, 2007.

[141] D. Weyns. A pattern language for multi-agent systems. In *Proceedings of the 3rd European Conference on Software Architecture (ECSA 2009)*, pages 191–200, Sept 2009.

[142] D. Weyns, A. Helleboogh, and T. Holvoet. How to get multi-agent systems accepted in industry? *International Journal of Agent-Oriented Software Engineering*, 3(4):383–390, May 2009.

[143] M. Wooldridge. *An introduction to multiagent systems.* John Wiley & Sons, 2009.

[144] Z. Yang, A. Hanna, and M. Debbabi. Team edit automata for testing security property. In *Proceedings of the 3rd International Symposium on Information Assurance and Security (IAS 2007)*, pages 235–240. IEEE, 2007.

Summary

The introduction of autonomous vehicles on highways offers new opportunities for traffic control. One of the opportunities is that we may enrich the infrastructure such that it can communicate with individual vehicles. Such an infrastructure is also called a smart road. A smart road aims to maximize the throughput and safety of traffic. To help with this, it may send personalized instructions to vehicles. As an example, the instructions might be target velocities such that two traffic streams can merge perfectly. Chapter 7 describes a simulation of this example. If a vehicle is not complying with its instructions, then an appropriate reaction such as a fine is required. This gives us two basic tasks for a smart road. On the one hand does it have to monitor the traffic in order to determine the instructions for vehicles and whether they comply with their instructions, on the other hand does it have to process the violations of instructions.

The term agent is often used to describe software which determines its own reaction when it receives input from its environment (such as speed limits) and tries to achieve personal goals (such as arriving at a target location). Chapter 6 discusses agents in more detail. In this thesis we view the tasks of a smart road as an example of the more generic task of making sure that agents behave according to preset guidelines. A specification of how agents ought to behave and the measures when they fail to comply, is what we call a norm. In our traffic example we may consider a norm to be comparable with a traffic rule. The task of a smart road can be reformulated as 'the enforcement of one or more norms'. The traffic situation on one road may affect the situation on another. Therefore, it is often appropriate to consider norms not in the context of a single smart road but in a network of them. Smart roads have to collaborate in such cases in order to enforce the norms. We call this decentralized enforcement since there is no central entity that enforces all the norms. We are particularly interested in the limitations and possibilities of decentralized norm enforcement when we want to apply it in an application such as a smart road network. To this end, we focus specifically on decentralized runtime norm enforcement[1]. We

[1]Runtime enforcement means that the enforcement takes place while the system in which we are enforcing norms is running. Enforcement that is not runtime can for instance take the form of system design changes such that the system cannot violate any norms. Note that for traffic this is not feasible since the government cannot exert full control over autonomous vehicles.

formulate formal models in Chapters 2-5 for decentralized norm enforcement in order to analyze the concept.

Our main conclusions and contributions are as follows. Decentralized runtime norm enforcement is composed of monitoring and control (Chapter 2). Monitoring for decentralized runtime norm enforcement can be equated to verifying whether a system satisfies a linear temporal logic (LTL) formula (Chapter 3). Several proposals exist for runtime LTL verification, to which we have added two complementary proposals (Chapters 3 and 4). We also discussed security and robustness for a decentralized monitor (Chapter 4). The control task of decentralized runtime norm enforcement is comparable to the control of discrete event systems (Chapter 5). We took a runtime controller model from the literature of discrete event control systems and have shown how this model can be reapplied for norm enforcement. A typical property of decentralized enforcement is that multiple controllers can operate in a concurrent manner, such as multiple smart roads that are coordinating traffic concurrently. We describe in Chapter 5 how the model for controllers can be expanded to the concurrent application of controllers, which results in a collaborative controller. In Chapter 6 we move from the theory of decentralized runtime norm enforcement to its implementation. It is important to consider conventional programming paradigms in order to promote the development of agent systems with norm enforcement. We proposed design patterns for object-oriented programming which capture often reoccurring solutions in the agent programming literature. We also showed how an object-oriented implementation can be expanded with norm enforcement using aspect-oriented programming. Finally, we made an example simulation of a smart roads scenario (Chapter 7). This simulation illustrates how we may view the task of a smart road as the decentralized runtime enforcement of norms.

Samenvatting

De opkomst van autonome voertuigen biedt nieuwe mogelijkheden voor het reguleren van verkeer. Een van de mogelijkheden is dat we de infrastructuur uitbreiden zodat deze op een individuele basis kan communiceren met voertuigen. Een dergelijke infrastructuur wordt ook wel een smart road genoemd. Een smart road houdt het verkeer in de gaten en helpt om de doorstroom en veiligheid te maximaliseren. Een smart road kan voertuigen gepersonaliseerde instructies geven. Denk hierbij aan bijvoorbeeld de optie om voertuigen doelsnelheden op te leggen zodat ze perfect ritsen. Hoofdstuk 7 beschrijft hier een simulatie van. Als een voertuig zich niet houdt aan de ontvangen instructies, dan moet er een gepaste reactie komen zoals een boete. Dit levert twee basistaken op voor de smart road. Enerzijds moet het verkeer in de gaten gehouden worden om te bepalen wat de voertuigen zouden moeten doen en of ze dat doen, en anderzijds moeten overtredingen afgehandeld worden.

De term agent wordt gebruikt voor software dat zelf de reactie bepaalt op input van de omgeving (zoals snelheidsborden) en persoonlijke doelen najaagt (zoals het bereiken van een doellocatie). Hoofdstuk 6 gaat meer in op de achtergrond van agenten. In deze thesis beschouwen we de taken van een smart road als een voorbeeld van de meer algemene taak om agenten in het gareel te houden. Een specificatie van hoe agenten zich zouden moeten gedragen, in combinatie met de maatregelen wanneer ze dit niet doen, heet een norm. In ons verkeersvoorbeeld is een norm vergelijkbaar met een verkeersregel. De taak van een smart road kan geherformuleerd worden als 'het handhaven van één of meer normen'. De verkeerssituatie op de ene weg kan gevolgen hebben voor situatie op een andere weg. Daarom passen normen vaak niet in het kader van een enkele smart road, maar is het meer gepast om ze te plaatsen in het kader van een wegennet. In zulke gevallen moeten de wegen samenwerken om de normen te handhaven. Dit heet gedecentraliseerde handhaving omdat niet een centrale entiteit alle normen handhaaft. In het bijzonder zijn we geïnteresseerd in de beperkingen en mogelijkheden van gedecentraliseerde normhandhaving wanneer we dit toepassen in een applicatie zoals een smart road. Daarom richten wij ons op runtime handhaving [2]. Hiervoor maken we in de hoofdstukken 2-5 formele runtime

[2]Runtime handhaving houdt in dat de handhaving plaatsvindt terwijl het systeem dat we controleren draait. Handhaving die niet runtime is kan bijvoorbeeld de vorm hebben van ontwerpaanpassingen aan het systeem zodat het systeem niet anders kan dan aan de normen voldoen. Merk

modellen van handhavende entiteiten (zoals smart roads).

Onze bevindingen en contributies zijn als volgt. Gedecentraliseerde normhandhaving is een samenspel tussen 'monitoring' en 'control' (Hoofdstuk 2). Monitoring voor gedecentraliseerde runtime normhandhaving kan gelijkgesteld worden met het controleren of een systeem een zogeheten 'linear temporal logic' (LTL) formule vervult (Hoofdstuk 3). Voor dit soort controles zijn in de literatuur enkele voorstellen te vinden van mechanismen, waaraan we twee nieuwe voorstellen hebben toegevoegd (Hoofdstukken 3 en 4). Ook hebben we de veiligheid en robuustheid van een gedecentraliseerde monitor beschouwt (Hoofdstuk 4). Control voor gedecentraliseerde normhandhaving is overeenkomstig met control voor zogeheten discrete event systems (Hoofdstuk 5). Uit de discrete event systems literatuur hebben we een controller model genomen en getoond hoe deze kan worden toegepast voor het handhaven van normen. Een typische eigenschap van gedecentraliseerde handhaving is dat twee of meerdere controllers parallel aan elkaar kunnen opereren, net als twee smart roads die tegelijk het verkeer in de gaten houden. We hebben in Hoofdstuk 5 omschreven hoe de parallelle toepassing van controllers kan worden gemodelleerd als een collaboratieve controller voor normhandhaving. In Hoofdstuk 6 stappen we over van de modellering van normhandhaving naar de implementatie daarvan. Het is belangrijk om aan te sluiten bij gangbare programmeerparadigma's als we de ontwikkeling van agentsystemen met normhandhaving willen promoten. Daarom hebben we veel voorkomende oplossingen in de in de literatuur rondom agentprogrammeren gevangen in een design pattern voor objectgeoriënteerde programmeertalen. Daarnaast hebben we aangegeven hoe normen in een objectgeoriënteerde applicatie kunnen worden toegevoegd door middel van aspectgeoriënteerde methoden. Als laatste hebben we een simulatie gemaakt van een smart roads scenario (Hoofdstuk 7). Deze simulatie illustreert hoe we de smart roads taken kunnen zien als gedecentraliseerde runtime handhaving van normen.

op dat voor verkeer dit niet mogelijk is omdat de overheid niet volledige controle heeft over hoe de autonome voertuigen werken.

Dankwoord

Als klein kind wist ik zeker dat ik op mijn huidige leeftijd een archeoloog zou zijn. Nu schrijf ik het dankwoord voor mijn thesis met een onderwerp uit de artificiële intelligentie. Mijns inziens is dit een goede ontwikkeling geweest. Achteraf gezien zit ik liever comfortabel achter een computer dan dat ik dinosaurusbotjes moet opgraven in een stoffige woestijn. Bij deze wil ik graag iedereen bedanken die heeft bijgedragen aan het mogelijk maken van deze thesis.

Allereerst wil ik mijn dagelijkse begeleider Mehdi Dastani bedanken voor zijn vertrouwen in mij en zijn begeleiding. Ik heb ontzettend veel geleerd van onze samenwerking. Ook mijn promotor John-Jules Meyer wil ik bedanken voor onze samenwerking. Daarnaast wil ik Next Generation Infrastructures bedanken voor de financiering van mijn onderzoek.

Ik wil Barend Poot en Jetze Baumfalk bedanken voor onze samenwerking bij het simuleren van verkeer. Ook Alexander Verbraeck wil ik bedanken voor zijn feedback op eerdere versies van OO2APL en zijn tips om mijn programmeerniveau omhoog te tillen. Ik wil mijn voormalige kantoorgenoten Max Knobbout en Sjoerd Timmer bedanken voor hun steun en adviezen tijdens mijn PhD traject.

Next I want to thank Nils Bulling for our collaboration on runtime monitoring with aggregation. I want to thank my reading committee members Alexander Verbraeck, Amal El Fallah Seghrouchni, Brian Logan, Carles Sierra and Jürgen Dix for evaluating this thesis and providing feedback. In particular I'd like to thank Brian Logan for his detailed comments and our collaboration at the start of my PhD. Also I want to thank all my current and past colleagues at Utrecht University. I very much enjoyed our lunch discussions, board game sessions and activities.

I'm very grateful towards Loïs Vanhée, with whom I've developed a close friendship that I value a lot. The support, inside jokes and fun times are much appreciated. Also my thanks towards Urlagh for regularly being the victim of those fun times.

Uiteraard wil ik ook mijn vriendin, Lotte Slenders, en mijn familie bedanken. Jullie hebben altijd een onuitputtelijk vertrouwen in mijn getoond. Ik heb het goed getroffen met zulke fantastische mensen in mijn leven!

Driebergen-Rijsenburg,
07/10/2017

Bas Testerink

Curriculum Vitae

Work Experience

Mar 2016-*Current*	Post-doc on conversational interfaces for the Dutch National Police at Utrecht University, Utrecht I work on the application of argumentation techniques to argue for required follow-up questions during automated intake processes.
Sep 2012-*Nov 2015*	PhD on decentralized monitoring and control of multi-agent systems at Utrecht University, Utrecht Promotor: Prof. dr. J-.J.Ch. Meyer, Co-Promotor: Dr. M.M. Dastani
Apr 2011-Aug 2012	Student Assistant at Utrecht University, Utrecht Teaching assistance and educative software development.
Dec 2006-Jan 2011	Technical Support Employee at Solcon, Dronten Technical support for DSL and fiber internet connections, digital TV, Voice over IP, and web hosting.

Education

Jul 2012	Msc. Degree in Computer Science at Utrecht University Programme: Technical Artificial Intelligence. Thesis: "Norms for Distributed Organizations" — Advisor: Dr. M.M. Dastani GPA: 4.0/4.0, Judicium: Cum Laude
Jul 2010	Bsc. Degree in Cognitive A.I. at Utrecht University Specialization: The focus area was informatics. Thesis: "Betrouwbaarheid op basis van overtuigingsdialogen" ("*Trust based on persuasion dialogues*") — Advisor: Prof. Dr. H. Prakken GPA: 4.0/4.0, Judicium: Cum Laude
Jul 2007	First year Informatics certificate at Windesheim Zwolle

SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
09 Tim de Jong (OU), Contextualised Mobile Media for Learning
10 Bart Bogaert (UvT), Cloud Content Contention
11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets
15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior
25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems
06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes
37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

---

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing

03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval
21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

---

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support
12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare

16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.
45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning

22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs
33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

---

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It
31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design

39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees
13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR

40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering