Saarland University
Reactive Systems Group
Department of Computer Science

Bachelor Thesis

From Knowledge- to Belief-Based Programming for Mobile Ad-Hoc Networks

Towards a Belief-Oriented MANET Middleware

Maximilian A. Köhl

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Advisor
Felix Klein, M.Sc.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr.-Ing. Holger Hermanns

November 2, 2017

Contents

Chapter 1

Introduction

The advent of wireless networks paved the way for untethered, everywhere, and anytime use of the internet and other communication services. Cellular networks are the backbone of modern society, and free WiFi is everywhere. Without us noticing, an enormous network machinery ensures behind the scenes that we can exchange instant messages with our friends, browse the internet while lining up at the supermarket checkout, make telephone calls with our loved ones, and find our way through unknown cities.

Most wireless networks we rely on every day are organized hierarchically and require central infrastructure, like cellular base stations and WiFi access points. However, what happens if there is a major catastrophe which destroys the needed infrastructure, like a natural disaster or a terror attack? Despite the lack of central infrastructure in such cases we would like to organize efficient and effective disaster recovery measures which undeniably require communication. Furthermore, there are also other scenarios where central infrastructure is not available, difficult to deploy, or economically inefficient, but nevertheless, communication is required. Those scenarios include for example communication between soldiers in combat zones or reconstruction workers in areas destroyed by civil war.

In all those scenarios where central infrastructure is destroyed or not feasible mobile ad-hoc networks (MANETs) pledge flexible deployments of failure-resistant communication mechanisms. In contrast to infrastructure-based networks, MANETs do not require any infrastructure, as they are constructed by directly connecting end devices like smartphones and laptops. The advantage to do without central infrastructure, however, comes at the expense of an increased complexity for applications and network protocols due to the decentralized and highly dynamic nature of mobile ad-hoc networks. It is thus hardly a surprise that in order to simplify MANET application development a multitude of network protocols and middleware solutions have been proposed [27] — most of which make use of traditional programming paradigms for distributed systems [27] like remote procedure calls, the event-based publish/subscribe paradigm, or tuple spaces.

Knowledge-based programming [9, p. 253 ff.] is the idea to use knowledge an agent, e.g., a node in a network, possess to program its behavior. Knowledge-based programs are based on epistemic logic [9, p. 15 ff.] which further allows the specification and verification of epistemic properties of multi-agent systems [18]. When it comes to highly dynamic systems like mobile ad-hoc networks, however, knowledge-based approaches for verification and programming fail, as, in general, very little about the system is knowable by agents. For instance, in mobile ad-hoc networks due to frequent and unpredictable network topology changes, nodes cannot know which neighbors they have. Nonetheless, to enable efficient communication, network protocols have to take into account the topology. Therefore, MANET network protocols usually rely on some informal notion of beliefs, e.g., they use beliefs about the topology to route packets. In contrast to knowledge, beliefs may be incorrect and thus allow agents to act upon falsehoods.

The goal of this thesis is to capture this informal notion of beliefs within a formal framework and thereby enable us to reason about beliefs, specify correctness criteria for them, and use them as a basis for a belief-based generalization of knowledge-based programming. Belief-based programming for mobile ad-hoc networks and distributed systems, in general, is inspired by the belief-based human decision-making process.

Every day beliefs and desires guide actions we perform and decisions we make. A rational being — a rather naive characterization would say — does what it believes fulfills its desires most [31, cf.]. For instance, if you have the all-things-considered desire to eat a cookie and you believe that there are cookies in your desk drawer, then it would be rational for you to open the drawer and eat one of them if there actually are some. If on the other hand, you open the drawer and it turns out, that there are no cookies in it, then it would be rational for you to revise your beliefs accordingly to include the new evidence. Without going into further detail regarding the philosophy behind rational decision-making and belief-revision, this makes it plausible that the human decision-making process is based on beliefs, desires, and the options for action available.

The larger goal we have in mind, but is out of the scope of this thesis, is to develop a general belief-centric middleware for MANETs and distributed systems in general which provides an interface inspired by the human decision-making process and uses the formalism of belief-based programming we introduce as a theoretical foundation.

The Digital Message Board Example

Besides the already introduced cookie example, we further use the example of a digital message board used to organize a huge public event where no central infrastructure exists, e.g., a music festival at a bunch of meadows between villages. To efficiently prepare the event several teams of workers with fixed responsibilities exist. For instance, one team is responsible for building the stage and another for preparing the catering. To work together, the workers post status updates on a digital message board. However, not everyone is interested in all messages. Rather there are team specific, i.e., intra-team, messages and team unspecific, i.e., inter-team, messages. For this purpose, the message board provides topics workers can subscribe to. Everyone then gets only those messages corresponding to subscribed topics of interest. To bring new workers up to speed the message board needs to ensure that workers who join will not only get all messages belonging to their topics of interest posted after their arrival but rather also all previous ones.

1.1 Structure

Mobile ad-hoc networks provide the general frame and the example case of this thesis, consequently, we begin this thesis with an introduction into the general subject matter of MANETs. We introduce necessary terminology and discuss particular challenges in the design of distributed MANET applications. We later come back to those challenges and see how a belief-based MANET middleware would address them.

We then introduce a formalism for open multi-agent systems, like MANETs. In contrast to multi-agent systems, open multi-agent systems allow agents to join and leave at runtime [2]. The formalism serves as the general basis of the remainder.

As a first step towards belief-based programming for mobile ad-hoc networks, we subsequently adapt knowledge-based programs as introduced in [9] to open multi-agent systems using the presented formalism. We further study the relation between knowledge-based programs for multi-agent systems and our adaption to open multi-agent systems.

Knowledge-based programs as introduced in [9] and our adaption to open multi-agent systems have the disadvantage that they are not all implementable [30] making them unsuitable as a basis for a general MANET middleware. The behavior of a knowledge-based program depends on the knowledge the program posses which again depends on the behavior of the program leading to a possibly vicious infinite regress. We circumvent this problem by introducing a notion of knowledge called *nomological knowledge* using the concept of *nomological systems*. Nomological systems explicitly encode the options for action the agents have by means of *nomological nondeterminism*. A *nomological knowledge program* chooses which of the available options to take based on nomological knowledge, i.e., knowledge about the nomological system, which eliminates the possibly vicious infinite regress. We use this *nomological framework* both, to specify correctness criteria for beliefs and as the formal basis for a general belief-oriented MANET middleware.

Afterwards, we introduce belief-based programming as a belief-based generalization of knowledge-based programming. To this end, we introduce a formal framework to model beliefs and study various adequacy criteria for beliefs and their revision in response to observations similar to rationality requirements for humans. We further demonstrate the usefulness of our approach by an example protocol defined using belief-based programs and informally prove certain correctness properties thereof. Finally, we sketch a general belief-centric MANET middleware which also gives an outlook on future work.

The main contributions of this thesis are, a notion of knowledge-based programs for open multi-agent systems which use agent-type relative knowledge, a formal framework for reasoning about nomological systems, and the concept of belief-based programming suitable to be used as a basis for a belief-centric MANET middleware.

Chapter 2

Mobile Ad-Hoc Networks

In this chapter, we have a detailed look at wireless networks especially mobile ad-hoc networks, which provide the general frame for this thesis. This chapter aims to serve as a starting point for our journey towards a general belief-oriented MANET middleware by providing essential preliminaries, necessary terminology, and background information. We begin with an introduction of basic terminology used throughout the thesis and subsequently, discuss particular challenges for the design of distributed MANET applications. Thereinafter, we motivate and introduce the concept of distributed systems middleware for the simplification of application development for mobile ad-hoc networks. Finally, we introduce a model of MANETs which we refine throughout the thesis.

So let's get started by importing the necessary terminology from [16]. A network comprises multiple *nodes* which are connected to each other via *links*. We refer to the graph constituted by the nodes and links of a network as the *network topology*. Wireless networks are networks comprising two sorts of nodes, *wireless hosts* and *base stations*. Wireless hosts run the end user applications and may or may not be mobile, i.e., they may or may not move around more or less freely in their environment. Base stations relay traffic between the wireless hosts and optionally other possibly wired networks. We distinguish two converse architectures for wireless networks, *infrastructure-based* and *infrastructure-less* networks. Infrastructure-based networks contain network infrastructure such as base stations whereas infrastructure-less networks do not contain any infrastructure, i.e., they solely comprise wireless hosts. A *wireless link* connects a wireless host to a base station or another wireless host within its *range*. See Figure 2.1 for an example of each architecture.

Wireless links allow the direct transmission of *packets*, i.e., packages of information, usually a byte string, between the two connected nodes. Depending on the structure of the network it might be necessary for a packet to travel from its *source* through multiple nodes to

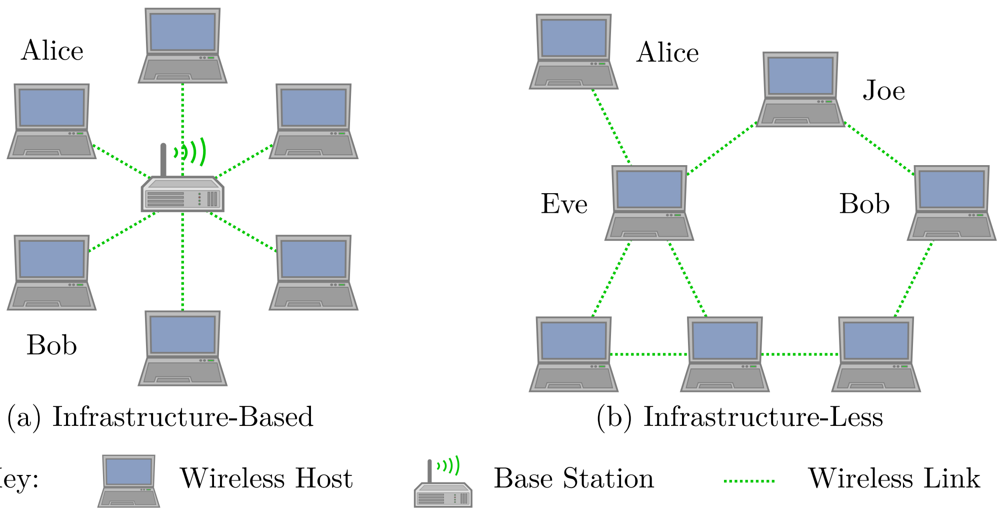

Figure 2.1: Wireless Network Architectures

reach its *destination*. We refer to this process as *routing* and to the number of nodes a packet needs to pass until it reaches its destination as *hops*. For instance consider the network shown in Figure 2.1b. If Alice wants to send a message to Bob, she can route it through Eve and Joe, which constitutes two hops. In addition to the classification into infrastructure-based and infrastructure-less networks we further distinguish between *single-hop* and *multi-hop* networks. In a single-hop network, each node can reach every other node within one hop, while in multi-hop networks it might be necessary to route a message through multiple hops until it reaches its destination. Figure 2.1a shows a single-hop and 2.1b a multi-hop network. A *mobile ad-hoc network* (MANET) is an infrastructure-less usually multi-hop network between multiple mobile wireless hosts [16, p. 518]. In this thesis, we will except for comparison reasons only consider MANETs.

To deliver a packet from one node to another or to enable other kinds of communication between nodes, e.g., exchange of instant messages, network protocols are required. A *network protocol* defines rules and formats for packet exchanges between nodes to achieve a particular communication task [16, cf. p. 9]. For instance, routing requires particular routing protocols on each node. In our example above, a routing protocol would tell Eve and Joe to relay the packet from Alice such that it eventually reaches Bob.

In case of the digital message board example, the mobile ad-hoc network is created by directly connecting the end devices, e.g., smartphones, of the workers. The task of an appropriate network protocol is to make sure, that each worker gets the messages of all topics of interest. For instance, the workers responsible for building the stage need to get all messages which belong to the stage-team topic and in addition all messages belonging to the inter-team topic. Since workers may arrive later, the protocol also needs to make sure, that everyone receives all messages posted prior to their arrival. Designing appropriate network protocols for specific communication tasks, e.g., the digital message board example, is, in general, a great challenge and even more so for MANETs — but, what makes building applications for MANETs a particular challenge?

2.1 Challenges for MANET Applications

Due to their highly decentralized and dynamic nature, MANETs exhibit several unique characteristics. So, let's see, what those unique characteristics are and which specific challenges they raise for the design of MANET applications.

In general, distributed applications can be implemented in a centralized and decentralized manner [29, p. 36 ff.]. Centralized applications are composed of two distinct parts, namely a server and a client part, usually running on specific server and client nodes. In contrast to that, fully decentralized applications do not draw this distinction and rather treat all nodes more or less equally. In case of our example a centralized application could look like this: A central server node stores all the information about which node is interested in which topics. The client runs on the devices of the workers and sends the messages to the server node which subsequently distributes them among all other interested client nodes. This approach has the advantage that it is straightforward compared to a decentralized solution. However, this simplicity comes at the expense of decreased reliable and performance. Wireless hosts in MANETs are usually not considered very reliable, and even if the server node is more reliable than other nodes in the network, we introduce a single-point of failure. Performance-wise all the traffic in the network is concentrated around the server node, as everything, e.g., subscribe requests and messages, has to travel to that node and in some cases back again, e.g., messages to all interested nodes. A decentralized application design on the other hand, if done right, distributes, for example, the information about interests evenly and replicates it among multiple nodes. This not only increases reliability, because, depending on the level of replication, a very high number of nodes has to fail until the application goes down, but also spreads the traffic more evenly among all nodes avoiding traffic concentration and congestion. Traffic congestion and reliability are issues when it comes to MANETs. Hence, a decentralized application design, although making the application more complicated, is preferable if not mandatory.

Another characteristic of MANETs is that nodes can move around more or less freely. Thus, the topology of MANETs is subject to frequent unpredictable changes [27, p. 4]. This makes routing in MANETs an exceptional challenge. For instance, routes might frequently break and require repairing. Furthermore, route discovery may consume a good deal of network resources. In contrast routing overhead in infrastructure-based networks is usually much less, as the topology is much simpler (especially in single-hop networks) and does not change frequently. Besides, wireless hosts moving from one base station to another can be handed over gracefully [16, p. 541 f.]. Also, infrastructure is often much more reliable than the wireless hosts which comprise a MANET. Hence, if infrastructure is available or easy to deploy and routing is required, then infrastructure-based networks should be used. MANET applications and network protocols need to deal with frequent topology changes as well as node failures efficiently and effectively [27, p. 4]. Done right, however, MANETs may be more reliable than infrastructure-based networks, since they do not introduce a single-point of failure — in an infrastructure-based network, every node is cut off immediately if the base station is destroyed — but rather tolerate a higher number of node failures. While in traditional networks usually the application does not need to be aware of the topology, it has been argued, that in MANETs due to their dynamic nature some sort of cross-layer architecture which gives an application access to topology and other information about the network itself is more efficient [6, e.g.].

Furthermore, MANETs often lack a central authority. This opens the doors for adversarial nodes which may tamper with the network, applications, and protocols. Unsurprisingly, making MANETs tolerant towards such nodes is again a difficult challenge. Since infrastructure is usually provided by a trustworthy authority, this problem does not or at least not to that extent arise for infrastructure-based networks and centralized applications where server nodes are provided by a trustworthy authority as well.

While much more could be said about challenges in the design of distributed MANET applications, the three major points mentioned, i.e., the need for fully decentralized applications, the need to deal with frequent topology changes, and the lack of trustworthy authorities, show that the advantages MANETs offer, e.g., the flexible and easy deployment of communication mechanisms and increased reliability, come at the expense of an increased complexity for both, applications and underlying network protocols. Hence, it is hardly a surprise that several middleware solutions and libraries emerged aiming to simplify the development of distributed and decentralized applications for MANETs [27].

2.2 MANET Middleware

A *middleware* is a piece of software between the operating system and the application (see Figure 2.2) [3] — or, to be more precise, in our case a piece of software between

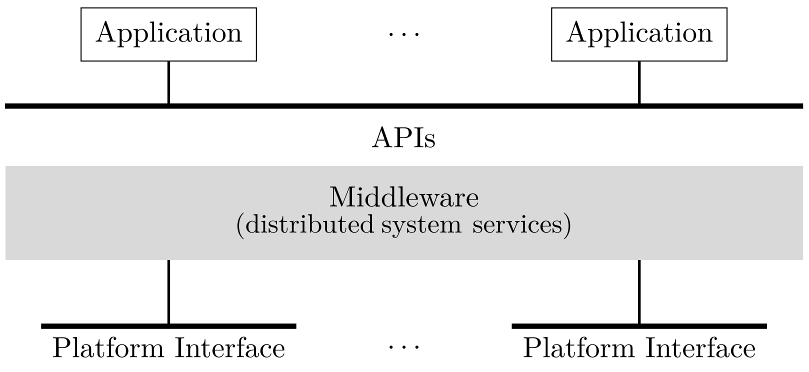

Figure 2.2: Distributed System Middleware

the application and the network primitives (the platform interface) provided by the operating system. The middleware provides a high-level *application programming interface* (API) to the application and handles all the low-level communication details [3, cf.]. The fundamental idea of distributed systems middleware is to abstract away much of the underlying complexity and provide an easy to use and reason about high-level programming interface with well-defined semantics. The application programming interface is supposed to be generic such that it can be used for a variety of different applications.

2.2.1 Application Programming Interface

Depending on the type of middleware its API provides different high-level abstractions. To get a rough idea of existing solutions in the field of MANET middleware, we broadly give an overview over two existing MANET middleware approaches in this subsection, namely remote procedure calls and publish/subscribe systems. We also evaluate whether we could use them for the example of the digital message board.

Remote Procedure Calls

A classical paradigm for the development of distributed applications, which is very popular for traditional networks as well, are remote procedure calls (RPCs) [22]. The key idea of remote procedure calls is to provide an API based on a set of procedures which when called transparently execute on a remote machine. From the application's point of view, a remote procedure call just looks like calling a local procedure. Under the hood the middleware marshalls the provided arguments into a form suitable for transmission, then sends them over to the remote machine, where they are unmarshalled again and passed to the corresponding local procedure. When the corresponding local procedure returns from execution the result gets marshaled by the middleware and is then transmitted back to the calling machine, where it is unmarshalled and returned to the caller.

Under the assumption that neither the network nor the remote machine will fail, an RPC middleware can provide nearly complete transparency between local and remote calls. Just the latency is usually much higher because the function call is done over the network. In practice, however, the amount of transparency is limited, especially when it comes to largely dynamic networks like MANETs. While a local call might succeed or throw an error if something with the actual task or the callee, e.g., invalid supplied arguments, went wrong, remote procedure calls may fail in two additional ways — there could be a network failure or the node which should run the procedure could have failed. However, often these kinds of failures are indistinguishable by the caller.

Remote procedure calls provide a synchronous one-to-one interface, i.e., one caller calls one callee and blocks until the call either failed or succeeded. Thus, it merely allows two nodes to communicate with each other. In our example case of the digital message board, a one-to-one interface is unsuitable. Instead, what is required, is a one-to-many interface, e.g., transmit a new message to many nodes, namely those that are interested. In general, remote procedure calls are not well suited for mobile environments, because they neither support group communication nor asynchronous communication and present a limit scalability [27, p. 3]. Asynchronous communication abstractions, unlike the synchronous RCP communication abstractions, do not block the application.

Remote procedure call middleware, in general, requires a unicast routing protocol between the caller node and the callee node. A unicast routing protocol routes messages between two nodes. Thus, ordinary remote procedure call middleware can in principle be used on top of any routing protocol providing unicast routing. There, however, also exist MANET specific RPC-like middleware, e.g., implementing MANET suited many-to-many semantics [14] which requires more sophisticated routing than unicast to be efficient.

Although we could implement our digital message board on-top of an RPC middleware the provided interface does not fit our needs well, as we need a one-to-many interface, but an RPC middleware provides us with a one-to-one interface. In the following, we, therefore, have a look at a presumably more suitable middleware approach.

Publish/Subscribe Middleware

Another famous middleware paradigm is the publish/subscribe pattern which is asynchronous and provides a one-to-many interface. This makes it an interesting candidate for the digital message board. The key idea of the publish/subscribe paradigm is that nodes subscribe to certain events and get notified when these events occur. In the example of the digital message board, we could have a message-posted event for each topic which is emitted whenever there is a new message regarding a specific topic.

There exist several proposals, for publish/subscribe middleware, which are specially tailored to MANETS [32, e.g.]. Just like nearly any distributed system, publish/subscribe systems can be implemented in a centralized manner, i.e., with a central *message broker*, and a decentralized manner, without such a centralized broker. In a centralized design, each node registers itself at the broker and subsequently subscribes to particular events. When some event is emitted, the broker is notified and then notifies each subscriber. As we already argued, the centralized design is not well suited for mobile ad-hoc networks so MANET publish/subscribe middleware should aim at a decentralized design. A decentralized design, however, makes well-defined semantics more difficult to implement. A centralized broker could keep track of the nodes, their interests, which node already received which event, and it may even store a limit log of events to send them to new subscribers. Getting the same guarantees for a decentralized MANET middleware is hard to achieve, and to our knowledge, no MANET publish/subscribe system provides them.

A publish/subscribe system could provide at least part of the solution for the digital message board depending on the exact semantics of event distribution. However, no system provides the required log for workers which join the system later and need to be bought up to date since a log is not part of the publish/subscribe paradigm.

Other Proposals

Besides proposals for remote procedure call and publish/subscribe middleware there is a multitude of proposals using other paradigms, e.g., tuple spaces, partly especially tailored to MANETs (see [27] for an overview). This subsection does not claim to be comprehensive in any way but rather gave us a rough idea of what to expect from a MANET middleware, how an API could look like, and what other solutions already exist.

In this thesis, we target another approach which provides a belief-centric interface. Intuitively the fundamental idea is to distribute beliefs about, e.g., the messages which have been posted. The application programming interface then provides methods for introducing new facts and specifying a distribution strategy for evidence thereof. Before digging deeper into this, we, however, need to have a look at the other side of MANET middleware, i.e., the platform interface, which we cover in the remainder of this chapter.

2.3 MANET Model

We saw examples of high-level interfaces distributed systems middleware might provide. In this section, we have a closer look at the other side of middleware, i.e., the platform interface provided by the operating system. To understand which primitives an operating system might offer we need to have a look at different network protocols and how they interact with each other. Traditionally network protocols are classified into different layers according to the services they provide. In the following, we roughly sketch the functionality of the five-layers of the internet protocol stack based on [16, p. 49 ff.].

The lowest layer is the *physical layer*. A network protocol on the physical layer specifies how individual bits are transmitted on the actual transmission medium, e.g., in case of MANETs electromagnetic waves. Due to the shared medium used in MANETs, the physical layer protocol is a particular challenge on its own. It needs to account for concurrent accesses and manage medium arbitration without a central arbiter.

On top of the *physical layer* there is the *link layer*. A network protocol on the link layer uses a physical layer protocol and provides an interface for transmitting complete packets, called *frames*, between nodes which are directly connected by a link. In case of MANETs, the directly connected nodes are those which are in the range of each other.

Finally, on top of the link layer, there is the *network layer*, the *transport layer*, and the *application layer*. A protocol on the network layer provides support for routing a packet to nodes which are not directly connected. Transport layer protocols provide specific guarantees like reliability or ordered transmission. On top of them, application-specific protocols are used to provide application-specific services, e.g., instant messaging.

2.3.1 Network Primitives

The operating system exposes protocols on various layers to the middleware. For instance, the portable operating system interface (POSIX) standard [28] exposes the transport layer protocols TCP and UDP via *sockets* [28, cf.]. However, we do not want to go into further detail here but rather study the exposed primitives on an abstract level.

It has been argued that to be efficient MANET protocols based on the internet protocol stack must use some sort of a cross-layer architecture [7, cf.]. Most middleware which is specially tailored to MANETs is directly based on the link layer for maximal control and efficiency. Thus, we focus on the link layer primitives in the following and use them in the remainder of this thesis as a basis for our middleware considerations.

The IEEE 802.11 standard [12] defines protocols for the link and the physical layer. We assume that the physical, as well as the link layer, is already taken care of by an IEEE 802.11 like protocol. This means that we will not consider low-level details like medium arbitration, interference, or other properties underneath the link layer.

Definition 2.1 (MANET Link Layer Primitives)
A MANET link layer provides the following asynchronous and reliable communication primitives: A node may *send* a packet to another node in its range and a node may *broadcast* a packet to every node in its range. As a counterpart of these sending abilities of nodes, the network may *deliver* a packet to a node.

See Appendix B.1 for an implementation of those primitives without reliability using Linux RAW sockets, IEEE 802.11 [12] configured as independent basic service set (IBSS), and a custom ethernet protocol for protocol discrimination [13, see p. 13]. Protocol discrimination allows to simultaneously run multiple protocols on the link layer [13, cf.].

In practice, e.g., when using IEEE 802.11, packets are byte strings which are often restricted in length [13, cf.] and not transmitted reliably. We assume that an appropriate encoding of high-level structures to byte strings and vice versa exists which is known and used by all nodes. We further assume that link layer packets can be arbitrary in size. In practice, this can be archived by splitting them into multiple actual data link packets which are then reassembled by the receiver. As a result of these assumptions, arbitrary large data structures can be transmitted using the provided communication primitives. To ensure reliability acknowledgments and retransmissions on the link layer can be used. Hence, the primitives defined in 2.1 provide an adequate model of an IEEE 802.11 like protocol with some extensions feasible and justifiable in practice. We formally refine these primitives in the next section but let's first turn to the topology and node mobility models.

2.3.2 Topology and Mobility Model

The link layer primitives leave open various explications of models for the existence of links, i.e., models of the topology and mobility of nodes. Usually, nodes are assumed to be scattered on a plane and then connected to each other according to their radio range (see Figure 2.3). We use a model viewing the topology as an undirected finite graph.

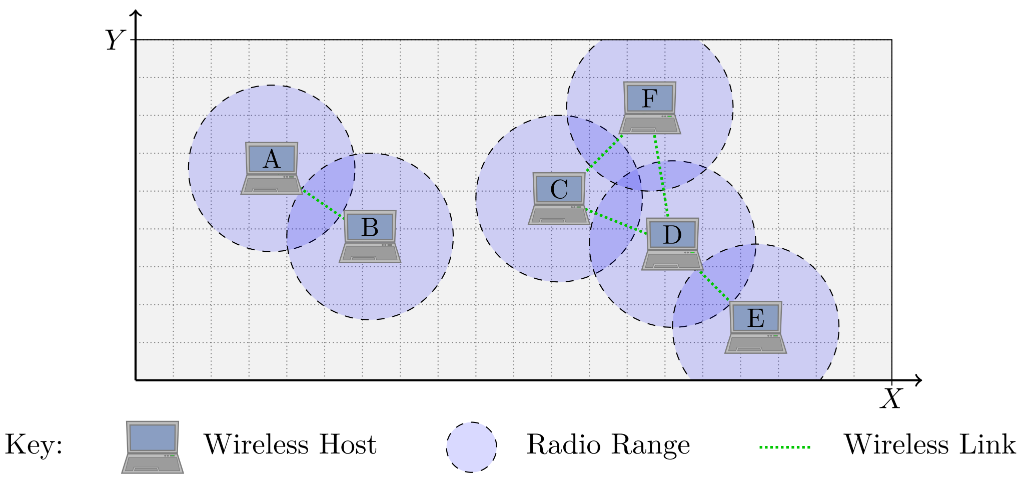

Figure 2.3: Example Topology with Two Segments

Definition 2.2 (MANET Topology)
A *MANET topology* is an undirected finite graph $G := \langle V, E\rangle$ with a set of *nodes* V and a set of *links* $E \subseteq V \times V$ between them. As the graph is undirected we require that $\langle v, v'\rangle \in E$ with $v, v' \in V$ if and only if $\langle v', v\rangle \in E$.

As a result of this model, links in the model are bidirectional. In practice wireless links might not conform to this restriction, however, routing in unidirectional networks is considered quite difficult [24, p. 6] and is thus out of the scope of this thesis. Furthermore, many existing routing protocols also require bidirectional links for their operation [23, 5, eg.], and since the transmission powers of wireless hosts in mobile ad-hoc networks are usually roughly equal, the assumption is realistic most of the time.

As Figure 2.3 shows a MANET might comprise multiple separated segments which we capture formally through the connected components of the topology graph.

Definition 2.3 (MANET Segment)
A *MANET segment* of MANET topology G is an undirected finite connected graph $G' := \langle V, E\rangle$ with a set of *nodes* V and a set of *links* $E \subseteq V \times V$ between them such that G' is a connected component of G. Let $seg(G)$ denote the set of all network segments, i.e., connected components, of MANET topology G.

Since communication between multiple segments is impossible we usually assume that networks are single-segment networks. Nevertheless, network partitions, i.e., the separation of one segment into multiple segments, as well as network unions respectively reunions, i.e., the fusion of multiple segments into one segment, happen in practice and should be handled gracefully by any decent network protocol, middleware, and application.

In case of the digital message board, nodes interested in the same topics are assumed to be physically close together because they work on the same physically located task, e.g., construction of the stage or catering. So, for example, consider the network shown in Figure 2.3. Nodes E and F could be responsible for the stage and nodes D and F for the catering. The nodes A and B could be temporarily separated from the other workers, e.g., because they need to get new parts for the stage from a nearby truck. Let's say that the worker corresponding to node A has found the needed part for the stage and returns to workers E and F. The application needs to cope with node A leaving the range of node B and entering the range of node E and F. It then needs to exchange those messages posted in the meanwhile in both original segments with all nodes in the newly formed segment which are interested and do not yet possess the respective messages.

As we have seen, one of the main characteristics of MANETs is their dynamically and continuously changing topology. This is caused by nodes moving around as well as entering or leaving the network. We capture such changes abstractly through MANET operations which take an existing topology and yield a new one.

Definition 2.4 (MANET Operations)
A *MANET operation* is a function from MANET topologies to MANET topologies.

There exist various, more detailed models for the mobility of MANET nodes [25, p. 327 ff.] which could be used to explicate this abstract view on the matter. Nevertheless, this abstract model is mighty enough to distinguish different kinds of operations or combinations of those which affect the topology in a significant way.

We introduce the following terminology for MANET operations: When a node *joins* the network, then this results in a new node within the topology. Analogously, when a node *leaves* the network, then this results in a node less in the topology. If a node moves around then, this may happen without any noticeable effect on our model at all if the topology is not changed. Otherwise, if moving affects the model, then wireless links are changed. Joining and moving can cause network *unions*, i.e., the union of multiple original segments into a single new segment. Leaving and moving can cause network *partitions*, i.e., a single original segment is divided into multiple new segments.

In this thesis, we do not make any assumptions about the specific operations that happen on the network other than fairness, i.e., that the network topology does not change with a frequency too high for the protocols to account for and that partitions do not happen between parts of the network which are required to be able to communicate.

Chapter 3

Open Multi-Agent Systems

Mobile ad-hoc networks are a specific form of *open multi-agent systems* (OMAS). Open multi-agent systems are systems comprising multiple communicating agents where agents might join or leave at runtime [2]. In case of MANETs, the agents are the nodes of the network communicating with an environment agent using the link layer primitives described as part of the MANET model in the previous chapter. In this chapter, we formally refine this model and its communication primitives. To this end, we introduce a formalism for open multi-agent systems based on previous work by Belardinelli, Grossi, and Lomuscio [2]. The presented formalism allows the adaption of knowledge-based programs [9, p. 253 ff.] to open multi-agent systems like MANETs which we cover in the next chapter.

3.1 The Formalism

The agents within open-multi agent systems need some means to communicate with each other. In general, there are multiple different ways to model communication in concurrent systems [1, p. 35 ff.]. Intuitively, if two agents communicate through an action, then one agent performs an active action while the other agent is passively affected by the action. For instance, if a node sends a packet via the network, then the node performs an active *send* action while the passive counterpart is accepted by the network. On the other hand, if the network delivers a message to a node, then the network performs the active action, and the node is passively affected. Moreover, actions may have different types, e.g., there is a *send* and a *broadcast* action type, which take parameters, like a source address and a message in case of a *broadcast* action. In addition, there may also be actions which are internal, i.e., executed by a single agent and not used for synchronization.

Formally we capture those intuitive considerations about actions as a means of communication by a notion of synchronized actions as introduced by Milner for the process calculus CCS in [21]. Please note that we do not use the complete process calculus CCS here, but merely import the idea of communication via synchronized actions.

Definition 3.1 (Parametric Actions)
Let $Act := \{ \alpha_1, \ldots, \alpha_n \}$ denote a finite set of *parametric action types* α_i. We distinguish between *active*, *passive*, and *internal* actions. Let $\vec{v} \in D^*$ be a parameter tuple from parameter domain D, then $\alpha!(\vec{v})$ denotes an active action, $\alpha?(\vec{v})$ denotes a passive action, and $\alpha(\vec{v})$ denotes an internal action for every action type α. Let $[Act]$ denote the set of all active, passive, and internal actions build form action types in Act, i.e.:

$$[Act] := \{ \alpha!(\vec{v}), \alpha?(\vec{v}), \alpha(\vec{v}) \mid \alpha \in Act \text{ and } \vec{v} \in D^* \}$$

We often use α to denote an arbitrary but fixed action from $[Act]$.

For the MANET link layer primitives we introduce three action types, *send*, *broadcast*, and *deliver*, corresponding to the respective communication primitives introduced in Definition 2.1. We require that the parameter domain D contains node addresses as well as packets which allows us to construct actions using the following patterns:

$$\mathit{send}(\mathit{src}, \mathit{dst}, \mathit{pkt}) \qquad \mathit{broadcast}(\mathit{src}, \mathit{pkt}) \qquad \mathit{deliver}(\mathit{src}, \mathit{dst}, \mathit{pkt})$$

For instance, the active action $\mathit{send}!(5, 8, \text{“Hello!”})$ sends, when performed, a packet with content “Hello!” from node 5 to node 8. After accepting the packet the network delivers it with $\mathit{deliver}!(5, 8, \text{“Hello!”})$ if and only if there actually is a link between nodes 5 and 8. The *broadcast* actions work analogously with the slight difference, that the network looks up all neighbors and then performs a sequence of delivery actions.

In case of the digital message board example, another action type is necessary, as nodes may not only communicate with each other but also have some internal message posting mechanism which is triggered by the user. Therefore, we introduce another action type *post* and instantiate it according to the pattern $\mathit{post}(\mathit{msg}, \mathit{topic})$, e.g., $\mathit{post}(\text{“Hello!”}, \text{“stage”})$ is supposed to post the message “Hello!” to the stage-topic by first internally modifying the local state and then causing a sequence of send or broadcast actions according to a network protocol which ensures that the requirements we introduced earlier are met.

Intuitively open multi-agent systems are composed of agents of different types which have specific abilities and behave in a specific way. For instance, in a MANET we may have two agent types, an environment agent type which accepts and delivers packets and an agent type for nodes. Although dividing the agents into two types, namely, one for the environment and one for the other agents, might be the most apparent separation there are other use-cases for agent types. For instance, consider that we would use the digital message board in a lecture, here we could introduce specific agent types for the lecturer and the students with different privileges and abilities.

Definition 3.2 (Agent Types)
Let $\mathcal{A} := \{ A_1, \dots, A_k \}$ denote a finite set of *agent types*. An agent type A is a tuple $\langle Q, \rho, \eta \rangle$ where Q is a possibly infinite set of *local states*, $\rho : Q \to 2^{[Act]}$ is a *protocol function*, and $\eta : Q \times [Act] \to Q_\oslash$ is a partial *effect function*. For every agent type A the effect function is required to be defined for every protocol compliant action, i.e.:

$$\forall q \in A.Q : \forall \alpha \in A.\rho(q) : A.\eta(q, \alpha) \neq \oslash$$

See Appendix A for the used notation.

An agent instance $[a]$ is a tuple $\langle A, q \rangle$ composed of an agent type A and a local state $q \in A.Q$. Let $[\mathcal{A}]$ denote the set of all agent instances of agent types from $\mathcal{A}$:

$$[\mathcal{A}] := \{ \langle A, q \rangle \mid A \in \mathcal{A} \text{ and } q \in A.Q \}$$

We introduce the following notation for the effect function $A.\eta$ of agent type A:

$$\langle A, q \rangle \xrightarrow{\alpha} \langle A, q' \rangle := A.\eta(q, \alpha) = q'$$

In case of the digital message board, we already made plausible that there should be at least two agent types. Since the specific explication of the local state space, the protocol function, and the effect function largely depends on the network protocol, we postpone their precise definition to Chapter 6 and instead continue with the cookie example which is sufficient to demonstrate the essential features of agent types.

In case of the cookie example, the agent may perform actions of four different types, *open*, *close*, *eat*, and *idle*, corresponding to opening the drawer, closing the drawer, eating a cookie, and doing nothing. Opening the drawer entails an observation, i.e., whether there is a cookie in the drawer or not. We model this observation by a parameter, i.e., the agent can perform $open!(0)$ if and only if the drawer is empty and $open!(1)$ if and only if the drawer contains a cookie. This is made possible by using an environment agent which provides the matching passive action based on the state of the drawer. To distinguish between the environment agent and the agent with the desire to eat cookies, we call the latter agent the *cookie monster*. One may see the nondeterministic performance of both $open!(0)$ and $open!(1)$ by the cookie monster as an experiment with the goal to learn something about the environment [21]. Figure 3.1 provides a graphical representation of the two agent types involved in the cookie example.

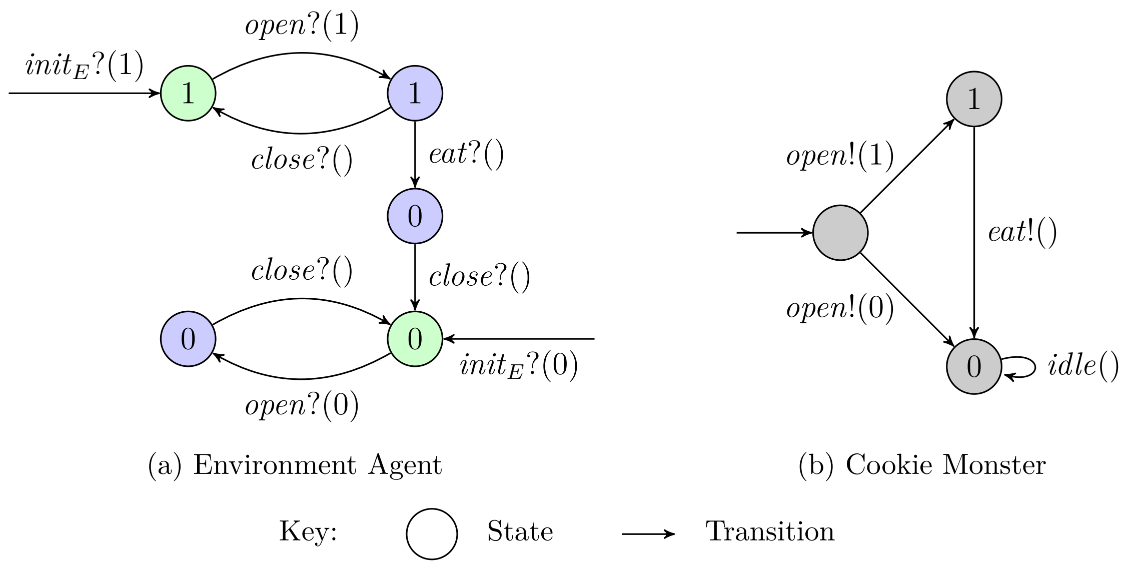

Figure 3.1: Cookie Example Agent Types

Initially, the environment agent may be in a state where there is either, a cookie in the drawer (state label “1”) or the drawer is empty (state label “0”). Depending on whether the drawer is currently opened (blue states) or closed (green states) and whether there is a cookie in the drawer or not, the environment accepts passive *open*, *close*, and *eat* actions. The cookie monster, of course, has the desire to eat a cookie and hence opens the drawer to learn whether there is a cookie in it or not. If there is a cookie in the drawer, then it performs an *eat* action. If it knows that the drawer is empty, i.e., either because it was empty initially or it already ate the cookie, it just does nothing.

For convenience, we also use pseudocode to capture protocol and effect functions as well as possible local states in a single succinct piece of code. See Code 3.3 for pseudocode representing the environment agent of the cookie example. The parts between *await* statements are executed by the effect function, and the current position within the code as well as all local variables and the call stack are remembered within the local state. When there is an *await* statement, then the agent waits for synchronization with another agent using one of the actions in the specified set which constitutes the protocol function.

Code 3.3 (Cookie Example: Environment Agent)

```
while true do
    if $opened \land cookie$ then
        $action :=$ await { $close?(), eat?()$ }
    else if $opened \land \neg cookie$ then
        $action :=$ await { $close?()$ }
    else if $\neg opened \land cookie$ then
        $action :=$ await { $open?(1)$ }
    else
        $action :=$ await { $open?(0)$ }
    switch $action$ do
        case $open?(\ldots)$
            $opened :=$ true
        case $close?()$
            $opened :=$ false
        case $eat?()$
            $cookie :=$ false
```

In contrast to traditional multi-agent systems, where agents cannot join or leave at runtime, we cannot use a fixed set of agents when formalizing open multi-agent systems. Instead, what is needed is some means to initialize new agents and destruct existing ones. In case of the digital message board, initialization allows new workers to be added to the system and destruction allows existing workers to leave the system. Also in the cookie example, the environment could either be initialized with or without a cookie in the drawer by means of a special $init_E$ action (see Figure 3.1a).

We use special actions for initialization as well as destruction of agents of specific types which not only allow to initialize agents of the respective type in the beginning but rather use the already existing synchronization mechanism to initialize new agents at runtime. For instance, the environment agent type in Figure 3.1a could be initialized by another agent which performs an active $init_E$ action with some parameter.

Definition 3.4 (Agent Initialization and Destruction)
Agents of a specific agent type A are initialized by means of actions of an *agent initialize* synchronized action type $init_A$ and destructed by means of an *agent die* internal action $die()$. Let ϵ denote the *uninitialized state*. We require $\epsilon \in A.Q$ for all agent types A. For every agent type A the protocol function $A.\rho$ needs to satisfy:

$$\emptyset \neq A.\rho(\epsilon) \subseteq \{\, \mathit{init}_A?(\vec{v}) \mid \vec{v} \in D^* \,\}$$

That is, it must be possible to initialize an agent of type A using $\mathit{init}_A!(\vec{v})$ with some parameters $\vec{v}$. If the *agent die* action $\mathit{die}()$ is protocol compliant, then the agent needs to be in the uninitialized state ϵ after destruction, i.e.:

$$\forall q \in A.Q : A.\eta(q, \mathit{die}()) \in \{\, \epsilon, \oslash \,\}$$

An agent must not be in the uninitialized state after every other action, i.e.:

$$\forall q \in A.Q, \alpha \in [\mathit{Act}], \alpha \neq \mathit{die}() : A.\eta(q, \alpha) = \oslash \vee A.\eta(q, \alpha) \neq \epsilon$$

Using those specific actions we can use the same mechanism to initialize new agents and to destroy them which we also use for other synchronized actions. To fully formalize synchronization between agents we first need to introduce a notion of global states. Intuitively the global state of a system is defined by the local states of all its agents. We capture this intuition through the following formalization.

Definition 3.5 (Global State)
Let $\mathcal{N} := \{\, a_1, a_2, \dots \,\}$ denote a possibly infinite set of *agent names*. A *global state* is a mapping $s : \mathcal{N} \to [\mathcal{A}]$ from agent names $a \in \mathcal{N}$ to agent instances $[a] \in [\mathcal{A}]$. The *active set* of a global state s is the set of agent names of initialized agents:

$$\mathit{active}(s) := \{\, a \in \mathcal{N} \mid s(a).q \neq \epsilon \,\}$$

A global state s is *finite* if and only if the set of initialized agent instances is finite, i.e., the set $\mathit{active}(s)$ is finite. We define the *extension* $[A]_s$ of an agent type A with respect to a global state s, i.e., the set of all names of agents of type A, by:

$$[A]_s := \{\, a \in \mathcal{N} \mid s(a).A = A \,\}$$

Let $[\mathcal{N}]$ denote the set of possible global states with agents with names from $\mathcal{N}$, i.e.:

$$[\mathcal{N}] := \{\, s : \mathcal{N} \to [\mathcal{A}] \,\}$$

A set $S \subseteq [\mathcal{N}]$ of global states is *agent type stable* if and only if each agent has a unique type across all states $s \in S$, that is:

$$\textit{type-stable}(S \subseteq [\mathcal{N}]) := \forall s, s' \in S, a \in \mathcal{N} : s(a).A = s'(a).A$$

Based on the local behavior of agents comprising a system we are now able to define a global transition relation which captures synchronization between agents.

Definition 3.6 (Transition Relation)
We define a *global transition relation* $\mathcal{T} \subseteq [\mathcal{N}] \times [\mathit{Act}] \times [\mathcal{N}]$ on global states by means of the following inference rules. For convenience we introduce the following notational shortcut for global state transitions:

$$s \xrightarrow{\alpha} s' := \langle s, \alpha, s' \rangle \in \mathcal{T}$$

Active and passive actions are executed in a synchronized fashion, i.e., there are two agents, one performs an active action and the other the passive counterpart. Thereby the local state transitions are recorded within the global successor state, and all other agents stay in their previous local state. Formally that is:

$$\frac{\begin{array}{lll} \exists a, a' \in \mathcal{N}, a \neq a' : & s(a) \xrightarrow{\alpha?(\vec{v})} s'(a) \quad s(a') \xrightarrow{\alpha!(\vec{v})} s'(a') & (3.6a) \\ \forall a'' \in \mathcal{N}, a \neq a'' \neq a' : & s(a'') = s'(a'') & (3.6b) \end{array}}{s \xrightarrow{\alpha(\vec{v})} s'}$$

Internal actions are executed by an individual agent without synchronization with other agents, i.e., they only affect the local state of the performing agent, and the local states of all other agents stay the same. Formally that is:

$$\frac{\begin{array}{lll} \exists a \in \mathcal{N} : & s(a) \xrightarrow{\alpha(\vec{v})} s'(a) & (3.6c) \\ \forall a' \in \mathcal{N}, a \neq a' : & s(a') = s'(a') & (3.6d) \end{array}}{s \xrightarrow{\alpha(\vec{v})} s'}$$

We define the set of successors of state s by $\mathit{post}(s)$, i.e.:

$$\mathit{post}(s) := \left\{ s' \in [\mathcal{N}] \;\middle|\; \exists \alpha \in [\mathit{Act}] : s \xrightarrow{\alpha} s' \right\}$$

Using those definitions, we are finally able to define formal models for open multi-agent systems where agents can communicate by means of synchronized actions and join or leave at runtime.

Definition 3.7 (Open Multi-Agent System)
Given a possibly infinite set $\mathcal{N}$ of agent names, we model an *open multi-agent system* $\mathcal{M}$ as a tuple $\langle \mathcal{N}, I \rangle$ where $\mathcal{N}$ is a set of agent names and $I \subseteq [\mathcal{N}]$ is an agent type stable set of finite *initial states* with $\forall s \in I : \mathit{active}(s) > 0$.

The key idea here is that the initial states already contain an adequate and possibly infinite number of existing but uninitialized agents such that they can be initialized by other agents using the special synchronized actions we introduced in Definition 3.4.

By requiring that $\forall s \in I : \mathit{active}(s) > 0$ we require that some of the agents are already initialized in the beginning. These agents could either be directly set to be in a specific state or initialized using their respective initialization action. By requiring that the initial states are finite, we ensure, that at every point in time only a finite amount of agents is initialized, as the transition relation ensures that one cannot jump from a finite state to an infinite state. Furthermore, type stability of the initial states ensures that there is a direct mapping between an agent name and an agent type which holds globally in all reachable states because the transition relation preserves agent type stability.

We now introduce some other definitions we need in the next chapter.

Definition 3.8 (Paths of the System [1, cf. p. 95])
A *finite path fragment* $\hat{\pi}$ from s_0 to s_n is a finite state sequence $s_0 s_1 \dots s_n \in [\mathcal{N}]^*$ such that for every $0 < i \leq n$ there exists an action $\alpha \in [\mathit{Act}]$ such that $s_{i-1} \xrightarrow{\alpha} s_i$, i.e.:

$$\mathit{finite\text{-}path}(\hat{\pi}) := \forall 0 < i \leq |\hat{\pi}| : \exists \alpha \in [\mathit{Act}] : \hat{\pi}[i-1] \xrightarrow{\alpha} \hat{\pi}[i]$$

An *infinite path fragment* π starting in s_0 is an infinite state sequence $s_0 s_1 \dots \in [\mathcal{N}]^\omega$ such that for every $0 < i$ there exists an action $\alpha \in [\mathit{Act}]$ such that $s_{i-1} \xrightarrow{\alpha} s_i$, i.e.:

$$\mathit{infinite\text{-}path}(\pi) := \forall 0 < i : \exists \alpha \in [\mathit{Act}] : \pi[i-1] \xrightarrow{\alpha} \pi[i]$$

We define the set of infinite path fragments starting in s and the set of infinite paths of $\mathcal{M}$ by:

$$\mathit{paths}(s) := \{\, \pi \in [\mathcal{N}]^\omega \mid \pi[0] = s \text{ and } \mathit{infinite\text{-}path}(\pi) \,\}$$

$$\mathit{paths}(\mathcal{M}) := \bigcup_{s \in \mathcal{M}.I} \mathit{paths}(s)$$

Notation 3.9 (Agent Type Extension)
Since every agent $a \in \mathcal{N}$ is required to have a unique type across all possible initial states $s \in I$ and the transition relation $\mathcal{T}$ preserves the agent type (see Definition 3.6) we can define the *extension of agent type A* with respect to OMAS $\mathcal{M}$ denoted by $[A]_{\mathcal{M}}$, or just $[A]$ if the reference OMAS is clear from the context, by:

$$[A]_{\mathcal{M}} := [A]_s \text{ for some } s \in \mathcal{M}.I$$

Definition 3.10 (Reachable States)
We define the set of *reachable states* of OMAS $\mathcal{M}$ by:

$$\mathit{reach}(\mathcal{M}) := \{\, s' \in [\mathcal{M}.\mathcal{N}] \mid \exists s \in \mathcal{M}.I : \exists \hat{\pi} \in [\mathcal{N}]^* : \mathit{finite\text{-}path}(s\hat{\pi}s') \,\}$$

An open multi-agent system $\mathcal{M}$ is *adequate* if and only if it is deadlock free, i.e., every reachable state has at least one successor state. For the remainder of this theses, we assume that all investigated systems are deadlock free.

Definition 3.11 (Deadlock Free)
An OMAS $\mathcal{M}$ is *deadlock free* or *reactive* if and only if every reachable state has at least one successor state. Formally that is:

$$\mathit{deadlock\text{-}free}(\mathcal{M}) := \forall s \in \mathit{reach}(\mathcal{M}) : \exists s' \in \mathit{reach}(\mathcal{M}), \alpha \in [\mathit{Act}] : s \xrightarrow{\alpha} s'$$

3.1.1 Comparison to Previous Work

Besides the logic part of [2], there are two main differences between the formalism for the system model. First, we use synchronized actions without fixed arity instead of joint actions with fixed arity. Second, we make no assumptions about the nature of local states, i.e., use black-box states, instead of the local states as database approach.

Synchronized Actions

Belardinelli et al. use joint actions in [2] as means of communication between agents. Joint actions are tuples of actions containing an action for each agent. They are executed all together atomically through a global transition relation, i.e., this transition relation takes a global state and an action for each agent and then returns a set of possible successor states. In principle both techniques could simulate each other, i.e., every system which can be modeled with one type of action can also be modeled with the other. However, they have an effect on the logic we introduce in the next chapter which is also based on [2]. We leave further investigation of this impact open to future work.

The difference that matters for us is that synchronized actions formally encapsulate an act of communication between precisely two agents and they can be seen as experiments, e.g., in the example above *open*!(0) is an experiment to find out whether the drawer is empty [21, cf.], as that action is only executed if its passive counterpart is enabled. This direct use for experimentation is not possible with joint actions, because they are executed no matter what other agents do, or in which state they are [2, cf. p. 3]. We use these kinds of experiments and encapsulation of a single act of communication in Chapter 6 when we introduce belief-based programming and take the observation of a synchronized action as evidence for something, e.g., *open*!(0) is evidence for an empty desk drawer. For joint actions, a similar approach does not make much sense, as a joint action is not a single act of communication but rather an atomically executed version of multiple communication acts being evidence for different propositions on their own. Whether and how a joint action can be disassembled into those individual communication acts depends on the global transition function and presumably would introduce much formal overhead and complexity. Therefore, we use synchronized actions here and in the remainder of the thesis.

The other difference of joint and synchronized actions is how they treat concurrency. Synchronized actions explicitly represent concurrency by interleaving of actions while joint actions execute all the actions atomically in parallel. This has an impact on necessary fairness constraints because with joint actions in every step each agent gets to perform an action while this is not the case for synchronized actions.

We leave the exact formal details of the differences between synchronized and joint actions open for future work and use synchronized actions for the presented reasons.

Back Box Local State

Belardinelli et al. assume more structure on the local states of agents in [2]. In particular, they represent an agent state as a finite set of database records and directly use them to define the semantics of their logic, i.e., a predicate $P(\vec{x})$ with parameters $\vec{x}$ holds if there exists an entry $\vec{x}$ on some agent in the database table corresponding to P. Instead of using such database instances as local states of the agent and then directly using these records to define an epistemic logic, we use black-box states which will later be interpreted by an interpretation function. This approach generalizes the database approach because database instances may still be taken to be local states and predicates can appropriately be defined as we see in the next chapter.

3.2 State-Based Programs

A program, e.g., implementing a network protocol, running on an agent can be disjoint in two parts. One part implements the effect of an action on the local state, and the other decides which actions to perform based on the current local state. The former part is formally captured within the effect function, while the latter is modeled by the protocol function. We already encountered a version of such programs in terms of the pseudocode used to specify agent types.

Within open multi-agent systems agents might have multiple options for an action. These options can be modeled explicitly by using nondeterminism or may be implicitly assumed. For example, in a MANET a node can send a packet containing the string "Hello World" or a packet containing the string "Hallo Welt". However, it cannot just arbitrarily do things, like directly communicating with nodes which are not in its range or move faster than the speed of light. To decide between these possible actions an agent must rely on the information contained in its local state. We now formally introduce a canonical notion of decision programs which are based on direct tests on the local state of an agent. In the next chapter, we generalize local decision programs to knowledge-based programs.

Definition 3.12 (Local Decision Programs [9, cf. p. 181])
A *local decision program* P for an agent type A is a set $P \subseteq T_L \times [Act]$ where T_L denotes a set of *local state tests* which can be evaluated using a local decision procedure:

$$decide_L : T_L \times A.Q \to \mathbb{B}$$

We require that $decide_L$ is *computable* by the agent. Given an appropriate decision procedure the program P induces a protocol function ρ_P:

$$\rho_P(q) := \{\, \alpha \mid \langle t, \alpha \rangle \in P : decide_L(t, q) = \top \,\}$$

We say that P *represents* the decisions of agent type A in the given OMAS if and only if $\rho_P = A.\rho$, i.e., the program yields the exact same actions as the protocol does. An agent type A in an OMAS *implements* the decision program if and only if the program represents the decisions of the agent.

Implementing local decision programs is straightforward, as assuming that the effect function is given, and the induced protocol function only includes actions which are actually possible for the agent, then we can just use the induced protocol function as the protocol function of an agent type constructed using the local state space, the existing effect function, and the induced protocol. Please note that such programs, although they are implementable, are not guaranteed to yield OMAS which are adequate, i.e., deadlock-free.

Chapter 4

Knowledge-Based Programming

The fundamental idea of knowledge-based programming is to utilize the knowledge about the system an agent posses to program its behavior. In the last chapter, we introduced local decision programs which allow defining the behavior of an agent based on local state tests. In this chapter, we extend local decision programs with knowledge tests, yielding a notion of knowledge-based programs [9, p. 253 ff.] for open multi-agent systems.

Knowledge-based programming is the first step towards the more general approach of belief-based programming. In contrast to beliefs, knowledge must not be false, i.e., everything an agent knows about the system is in fact true. Thus, belief-based programming can be understood as a generalization of knowledge-based programming allowing agents to act upon falsehoods. In Chapter 6, we show how to generalize knowledge-based programming to obtain a sensible and useful programming paradigm. This, however, requires the understanding of knowledge-based programming as introduced in this chapter.

We approach the subject matter of knowledge-based programming as follows. Prior to any application to and formalization within our formalism, we have to become clear about the nature of knowledge itself. Based upon these considerations about the nature of knowledge, we subsequently define a logic which includes knowledge (epistemic) and branching time operators based on [2]. The presented logic is used as a language for both, knowledge tests appearing in knowledge-based programs and as a means for the specification of epistemic and temporal correctness properties we want a system to satisfy.

The Nature of Knowledge

In this thesis we use the *possible worlds model* of knowledge [9, p. 15 ff.]. The intuition of this model of knowledge is, that an agent knows φ, if φ holds in all global states (or worlds) it has to consider possible. Which global states an agent has to consider possible depends on the information the agent possesses which is encoded in its local state. Since this model of knowledge is state-based we need to use a state-based logic as a basis. To apply this model of knowledge to our formalism, we thus use a variant of modal logic.

The possible worlds model entails some details regarding knowledge. First, as a result of the model, everything which is known is actually true, because it is the case in all states the system could be in as far as the agent knows and the actual state has to be in this set. Thus, the notion of knowledge satisfies the most central necessary condition of knowledge, namely, that falsehoods are not knowable. Furthermore, the objects of knowledge are propositions, i.e., sets of possible states (or worlds). As a result thereof, agents are logically omniscient, i.e., they know all logical truths, as those hold in every state, as well as all implications of their knowledge, as these hold in every state they consider possible as well because these are implications of what holds in those states [9, p. 31 ff.].

Formalizing Knowledge

Given the possible worlds model of knowledge — which is the standard model of epistemic logic and knowledge-based programming — we now formalize this model within our formalism. Intuitively, the set of states an agent has to consider possible is a subset of the reachable states of the system. The system cannot be in a state which is not reachable. Among all the reachable states an agent can only distinguish those classes of global states in which it has different local information, i.e., has distinct local states. If an agent has the same local state in two reachable global states, then it has to consider both states possible, when being in the respective local state. We formally capture this notion by equivalence classes of indistinguishable reachable global states.

Definition 4.1 (Indistinguishable Global States)
Based on the set of reachable states $\mathit{reach}(\mathcal{M})$ we define agent-relative equivalence classes $[s]^a_{\mathcal{M}}$ of global states $s \in \mathit{reach}(\mathcal{M})$ by:

$$[s]^a_{\mathcal{M}} := \{\, s' \in \mathit{reach}(\mathcal{M}) \mid s(a).q = s'(a).q \,\}$$

In global state s, an agent a has to consider all states $s' \in [s]^a_{\mathcal{M}}$ possible, as they are all indistinguishable given its local state, and they are all reachable. Thus, the system could be in any of these states, as far as agent a knows. Intuitively, agent a knows that φ, denoted by $K_a\varphi$, if and only if φ is true in all those indistinguishable states.

To make an example from the area of MANETs: If an agent has to consider two global states possible, one, where there is a link between two nodes and another, where this link does not exist, then it does neither know that there is a link, nor that there is no link, since either could be the case. On the contrary, if an agent has to consider only such states possible, where there is a link, then it knows, that there is a link.

Knowledge in MANETs

Regarding our example of a digital message board, what would be useful knowledge for the nodes? Let's start with something fundamental like the network topology of the MANET itself. To route messages through the network to everyone who is interested, the nodes need to (partially) know the network topology to determine routes. The topology is entirely given by the set of all links between the nodes. So, it would be useful, if every node would know whether there is a link between nodes v and v', then it could use, e.g., breadth-first search to determine the shortest path in hops to a specific node.

For a system to actually work with this knowledge, we have a temporal requirement, namely, that the nodes eventually know about those links they need to route packets. This might be a subset of the links, but for simplicity, we assume, that they eventually need to know all links. Please note, this somehow conflicts with the highly dynamic nature of MANETs, but we postpone this detail to Chapter 6.

To summarize, for knowledge-based programming in MANETs we would like nodes to be at least able to act upon knowledge about links, namely sending messages along a path they know exists, and we need that the nodes eventually obtain this knowledge which is a temporal and epistemic correctness property. So, if we look at this example, then we need a logic which allows expressing temporal and epistemic properties and some first-order quantification, e.g., *eventually* for all nodes and all links.

To this end, we introduce a first-order variant of an epistemic logic including temporal branching time operators in the next section. The logic not only provides a language for knowledge tests about the system including its temporal evolution, but it can also be used to specify epistemic and temporal properties like the one we have seen.

4.1 Epistemic Branching-Time Logic

Computation tree logic [4] (CTL) is a well known and studied branching-time logic. It is usually defined over a fixed set of atomic propositions [1, p. 317]. We already saw that we need quantification to express interesting properties about open multi-agent systems. Instead of using a fixed set of atomic propositions we borrow the concepts of predicates and quantification from first-order logic.

Definition 4.2 (Predicates)
Let $\mathcal{P} := \{ P_1, \ldots, P_m \}$ denote a finite set of *predicate symbols*. A *predicate interpretation* I over an *interpretation domain* U is a function $I : \mathcal{P} \to 2^{U^*}$ mapping predicate symbols P to a set of arguments which satisfy the predicate, i.e., $P(u_1, \ldots, u_n)$ with $u_i \in U$ for $1 \leq i \leq n$ is *satisfied* by I if and only if $\langle u_1, \ldots, u_n \rangle \in I(P)$. We call the values $u \in U$ *individuals*. We also refer to U as the *universe*.

For our example a predicate could be $\mathit{link}(v, v')$ which holds if and only if there is a wireless link between the nodes v and v'. To evaluate whether a predicate holds in a given global state or not, we assign an interpretation to each reachable state.

Definition 4.3 (Interpretation Assignment)
An *interpretation assignment* ν for OMAS $\mathcal{M}$ assigns a predicate interpretation to each reachable state $s \in \mathit{reach}(\mathcal{M})$ of an open multi-agent system $\mathcal{M}$, i.e.:

$$\nu : \mathit{reach}(\mathcal{M}) \to \mathcal{P} \to 2^{U^*}$$

In the case of *link* we can define an interpretation assignment by

$$\nu(s)(\mathit{link}) := s(e).q.G.E$$

where e is the agent name of the environment agent and $G.E$ is the set of edges of the network graph stored in its local state. As a result, $\langle v, v' \rangle \in \nu(s)(\mathit{link})$ holds if and only if there is a link in state s between nodes v and v'. Hence, $\mathit{link}(v, v')$ holds in state s if and only if in state s exists a link between nodes v and v'.

Based on these predicates we now inductively define a logic which allows building more powerful statements including temporal and epistemic expressions.

Definition 4.4 (CTLKx Syntax)
We define the syntax of CTLKx formulae φ inductively by means of the following abstract grammar where $P_i \in \mathcal{P}$ is a predicate symbol, $x \in \Sigma_X$ is a variable symbol, $c \in \Sigma_C$ is a constant symbol, and $\tau_1, \ldots, \tau_n$ is a variable amount of first-order terms:

$$\begin{aligned}
\varphi &::= P_i(\tau_1, \ldots, \tau_n) \mid \tau_1 \equiv \tau_2 \mid \exists x.\, \varphi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{A}\,\psi \mid \mathrm{E}\,\psi \mid \kappa && (\textit{state formula})\\
\kappa &::= K_a\varphi \mid K_x\varphi && (\textit{epistemic formula})\\
\tau &::= x \mid c && (\textit{first-order term})\\
\psi &::= X\varphi \mid \varphi_1\, U\, \varphi_2 && (\textit{path formula})
\end{aligned}$$

In addition we define the derived operators $\forall$, $\vee$, $\rightarrow$, and $\leftrightarrow$. The universal quantifier $\forall$ is defined as the dual of the existential quantifier, i.e., $\forall x.\, \varphi := \neg(\exists x.\, \neg\varphi)$, and the other boolean operators $\vee$, $\rightarrow$, and $\leftrightarrow$ are as usual derived from $\wedge$ and $\neg$ [1, see p. 232].

Since CTLKx is a derivate of CTL we also introduce the usual derived CTL operators [1, p. 318, p. 334] where "true" is a zero-ary predicate with $\forall s : \nu(s)(\text{true}) := \{\, \langle\rangle \,\}$:

$$\mathrm{AF}\,\varphi := A(\text{true}\, U \varphi) \qquad \mathrm{EF}\,\varphi := E(\text{true}\, U \varphi) \qquad \mathrm{EG}\,\varphi := \neg\mathrm{AF}\,\neg\varphi \qquad \mathrm{AG}\,\varphi := \neg\mathrm{EF}\,\neg\varphi$$

$$E(\varphi_1 W \varphi_2) := \neg A((\varphi_1 \wedge \neg\varphi_2) U (\neg\varphi_1 \wedge \neg\varphi_2))$$

$$A(\varphi_1 W \varphi_2) := \neg E((\varphi_1 \wedge \neg\varphi_2) U (\neg\varphi_1 \wedge \neg\varphi_2))$$

A CTLKx formula φ is *well-formed* if and only if it does not contain unbound variables, i.e., each occurrence of x and K_x is preceded by an existential quantifier $\exists x.$ in the syntax tree. Let the set of all well-formed CTLKx formulae φ be denoted by Φ. In the remainder, we implicitly assume that formulae are well-formed.

The CTLKx syntax is based on the syntax of FO-CTLK$_x$ as defined in [2, p. 3] with two main differences. First, the predicates do not have a fixed arity, and second, there is no distinction between agent variables and individual variables. Instead, we later introduce syntactic sugar to express quantification over agents of a specific type.

CTLKx Semantics

We now define the semantics of CTLKx formulae which are evaluated in the context of a given state of an OMAS and an assignment for variable and constant symbols.

Definition 4.5 (Variable and Constant Assignment)
A *variable and constant assignment* is a function $\sigma : \Sigma_X \uplus \Sigma_C \to U$ assigning to each variable symbol $x \in \Sigma_X$ and constant symbol $c \in \Sigma_C$ an individual from U. We say that,

x is *bound* to $u \in U$ by σ if and only if $\sigma(x) = u$. Let $\Sigma_C^X := \Sigma_X \uplus \Sigma_C$ denote the set of all variable and constant symbols. We assume without loss of generality, that Σ_C^X and U are disjoint. For convenience we further define $\sigma(u) := u$ for $u \in U$. Let $\Sigma \subseteq \left\{ \sigma : \Sigma_C^X \to U \right\}$ denote the set of all assignments such that all constant symbols are assigned to fixed values from U, i.e., $\forall \sigma, \sigma' \in \Sigma : \forall c \in \Sigma_C : \sigma(c) = \sigma'(c)$.

Definition 4.6 (CTLKx Semantics)
Given an OMAS $\mathcal{M}$, a reachable state $s \in \mathit{reach}(\mathcal{M})$, a variable and constant assignment $\sigma \in \Sigma$, and an interpretation assignment ν, we define a satisfaction relation $\models$ by:

$$\begin{array}{ll}
\langle \mathcal{M}, s, \sigma \rangle \models P_i(\tau_1, \dots, \tau_n) & \text{iff } \langle \sigma(\tau_1), \dots, \sigma(\tau_n) \rangle \in \nu(s)(P_i) \\
\langle \mathcal{M}, s, \sigma \rangle \models \tau_1 \equiv \tau_2 & \text{iff } \sigma(\tau_1) = \sigma(\tau_2) \\
\langle \mathcal{M}, s, \sigma \rangle \models \exists x.\, \varphi & \text{iff } \exists u \in U : \langle \mathcal{M}, s, \sigma[x \mapsto u] \rangle \models \varphi \\
\langle \mathcal{M}, s, \sigma \rangle \models \neg \varphi & \text{iff } \langle \mathcal{M}, s, \sigma \rangle \not\models \varphi \\
\langle \mathcal{M}, s, \sigma \rangle \models \varphi_1 \wedge \varphi_2 & \text{iff } \langle \mathcal{M}, s, \sigma \rangle \models \varphi_1 \text{ and } \langle \mathcal{M}, s, \sigma \rangle \models \varphi_2 \\
\langle \mathcal{M}, s, \sigma \rangle \models K_a \varphi & \text{iff } \forall s' \in [s]^a_{\mathcal{M}} : \langle \mathcal{M}, s', \sigma \rangle \models \varphi \\
\langle \mathcal{M}, s, \sigma \rangle \models K_x \varphi & \text{iff } \sigma(x) \in \mathcal{M}.\mathcal{N} \text{ and } \langle \mathcal{M}, s, \sigma \rangle \models K_{\sigma(x)} \varphi \\
\langle \mathcal{M}, s, \sigma \rangle \models A \psi & \text{iff } \forall \pi \in \mathit{paths}(s) : \langle \mathcal{M}, \pi, \sigma \rangle \models \psi \\
\langle \mathcal{M}, s, \sigma \rangle \models E \psi & \text{iff } \exists \pi \in \mathit{paths}(s) : \langle \mathcal{M}, \pi, \sigma \rangle \models \psi \\
\langle \mathcal{M}, \pi, \sigma \rangle \models X \varphi & \text{iff } \langle \mathcal{M}, \pi[1], \sigma \rangle \models \varphi \\
\langle \mathcal{M}, \pi, \sigma \rangle \models \varphi_1 U \varphi_2 & \text{iff } \exists j \geq 0 : \langle \mathcal{M}, \pi[j], \sigma \rangle \models \varphi_2 \text{ and } \forall 0 \leq i < j : \langle \mathcal{M}, \pi[i], \sigma \rangle \models \varphi_1
\end{array}$$

The CTLKx semantic is based on the semantic of FO-CTLK$_x$ as defined in [2, p. 4] with the main difference that predicates are freely definable on global states through an interpretation assignment. This allows expressing system properties which depend on local states of multiple agents. For instance, in a temperature sensor MANET, the property of the average temperature among all nodes is not a property defined via an individual node. In [2] predicates are defined via a union of agent local database instances, comparable to a union of agent local predicate interpretations. Hence, FO-CTLK$_x$ does not allow to define predicates based on multiple local states of multiple agents. In our formalism database instances can be encoded in the local states and predicate assignments can be created accordingly. Therefore, our formalism is more expressive regarding predicates.

Agent-Type Quantifiers

The other main difference between [2] and CTLKx is that CTLKx does not distinguish between agent and individual variables. Instead, we provide syntactic sugar to quantify over agents of a specific type and define $K_x\varphi$ to be not satisfied if x is bound to an individual which is not an agent-name. Based on that we define the following syntactic sugar for agent-type quantifiers which is made possible by the freely definable predicates.

Notation 4.7 (Agent-Type Quantifiers)
Assuming that the interpretation domain U contains all agent names, i.e., $\mathcal{N} \subseteq U$, we define an existential quantifier over agents of type A as syntactic sugar by:

$$\exists x : [A].\, \varphi := \exists x.\, \textit{is-agent}(x, A) \land \varphi$$

Whereby *is-agent*(x, A) is an auxiliary predicate which holds if and only if the value bound to x is an agent name for an agent of type A in the respective OMAS, i.e.:

$$\nu(s)(\textit{is-agent}) := \{\, \langle a, A \rangle \mid s(a).A = A \,\}$$

Analogously to the universal quantifier over individuals, we define an universal quantifier over agents of type A by the dual of the agent-type existential quantifier:

$$\forall x : [A].\varphi := \neg(\exists x : [A].\neg\varphi)$$

Before we have a look at useful examples, we prove that agent-type quantifiers have the intended semantics, i.e., that $\exists x : [A].\varphi$ holds if and only if there is an agent name which denotes an agent of type A and is bound to the variable x such that φ is satisfied.

Theorem 4.8 (Semantics of Agent-Type Quantifiers)
The agent-type quantifier $\exists x : [A].\varphi$ has the intended semantics, i.e.:

$$\langle \mathcal{M}, s, \sigma \rangle \vDash \exists x.\, \textit{is-agent}(x, A) \land \varphi \iff \exists a \in [A]_s : \langle \mathcal{M}, s, \sigma[x \mapsto a] \rangle \vDash \varphi \tag{4.8}$$

Proof If $\exists x.\textit{is-agent}(x, A) \land \varphi$, then there is an $u \in U$ such that *is-agent*(x, A) and φ are satisfied by assignment $\sigma[x \mapsto u]$. Since *is-agent*(x, A) is satisfied, x has to be bound to an u such that $s(u).A = A$, hence $u \in [A]_s$. Therefore, there exists an $a \in [A]_s$ such that φ is satisfied under the assignment $\sigma[x \mapsto a]$, i.e., $\exists a \in [A]_s : \langle \mathcal{M}, s, \sigma[x \mapsto a] \rangle \vDash \varphi$.

If there exists an agent name $a \in [A]_s$ such that φ is satisfied by assignment $\sigma[x \mapsto a]$, then *is-agent*(x, A) and φ are satisfied by assignment $\sigma[x \mapsto a]$. Hence, $\sigma[x \mapsto a]$ satisfies *is-agent*$(x, A) \land \varphi$ by Definition 4.6. Since $[A]_a \subseteq \mathcal{N} \subseteq U$, also $a \in U$. Therefore, there exists an $u \in U$ such that *is-agent*$(x, A) \land \varphi$ is satisfied by the assignment $\sigma[x \mapsto u]$. ■

The agent-type quantifiers come in handy when we need to express properties like that every node in the MANET knows about every wireless link. Formally we express this by

$$\forall x : [N].\, \forall v.\, \forall v'.\, \mathit{link}(v, v') \rightarrow K_x \mathit{link}(v, v')$$

where N is the agent-type of nodes. If we combine this with the temporal operator for all-paths eventually, AF , we can express the property that every node eventually knows all the links between all nodes:

$$\mathrm{AF}\, \forall x : [N].\forall v.\forall v'.\mathit{link}(v, v') \rightarrow K_x \mathit{link}(v, v')$$

We now formally incorporated implicit knowledge in terms of the possible worlds model into the formalism. Please note, this does not mean, that an agent can explicitly compute what it implicitly knows given its local state. As we show in the next chapter knowledge in open multi-agent systems is undecidable in general.

The main goal of this chapter is to introduce a notion of knowledge-based programs for open multi-agent systems by extending local decision programs with knowledge tests based on CTLKx formulae. We now have all the basic tools needed to define knowledge-based programs for open multi-agent systems based on the presented logic.

4.2 Knowledge-Based Programs

Knowledge-based programs have been introduced in [9]. The fundamental idea is to extend the local state tests of regular decision programs with additional knowledge tests which test the implicit knowledge of an agent. We extend the approach of knowledge-based programs to our formalism and by that to open multi-agent systems.

The original notion of knowledge-based programs used a notion of agent-relative knowledge as a basis for decisions. Although we have introduced a notion of agent-relative knowledge for open multi-agent systems — constituted by the agent-relative K_a operator — we cannot use it directly for knowledge-based programs because programs must be agent-type relative for open multi-agent systems. Thus, we adapt the notion of agent-relative knowledge programs to agent-type relative knowledge programs. To this end, we introduce knowledge sets which allow us to introduce a notion of agent-type relative knowledge.

Knowledge Sets

We start with an agent-relative knowledge set which, as we show, adequately captures the notion of agent-relative knowledge.

Definition 4.9 (Agent-Relative Knowledge Set)
We define the *agent-relative knowledge set* $\mathcal{K}^a_{\mathcal{M}}(q) \subseteq \Phi$ of agent a in state q, i.e., the set of all formulas $\varphi \in \Phi$ the agent a knows when it is in state q, by:

$$\mathcal{K}^a_{\mathcal{M}}(q) := \{\, \varphi \in \Phi \mid \forall s \in \mathit{reach}(\mathcal{M}), s(a).q = q : \langle \mathcal{M}, s, \sigma \rangle \vDash \varphi \,\}$$

Please keep in mind, that Φ by Definition 4.4 contains only well-formed formulae, i.e., formulae without free variables. By Definition 4.5 every $\sigma \in \Sigma$ assigns the same individuals to constants. Hence, we can use an arbitrary $\sigma \in \Sigma$ here, because the variable assignments do not matter and the constant assignments are identical across all $\sigma \in \Sigma$.

Theorem 4.10 (Agent-Relative Knowledge Set)
The agent-relative knowledge set adequately captures agent-relative knowledge, i.e.:

$$\varphi \in \mathcal{K}^a_{\mathcal{M}}(s(a).q) \iff \langle \mathcal{M}, s, \sigma \rangle \vDash K_a\varphi$$

Proof by equivalence transformation:

$$\begin{aligned}
& \varphi \in \mathcal{K}^a_{\mathcal{M}}(s(a).q) \\
\overset{(4.9)}{\iff} \; & \forall s' \in \mathit{reach}(\mathcal{M}), s'(a).q = s(a).q : \langle \mathcal{M}, s', \sigma \rangle \vDash \varphi \\
\overset{(4.1)}{\iff} \; & \forall s' \in [s]^a_{\mathcal{M}} : \langle \mathcal{M}, s', \sigma \rangle \vDash \varphi \\
\overset{(4.6)}{\iff} \; & \langle \mathcal{M}, s, \sigma \rangle \vDash K_a\varphi \qquad \blacksquare
\end{aligned}$$

Based on agent-relative knowledge sets, we define an agent-type relative knowledge set, for each agent type A and local state $q \in A.Q$.

Definition 4.11 (Agent-Type-Relative Knowledge Set)
Given an agent type A and a local state $q \in A.Q$, we define the *agent-type relative knowledge set* $\mathcal{K}^A_{\mathcal{M}}(q)$ of agent-type A in state q by:

$$\mathcal{K}^A_{\mathcal{M}}(q) := \bigcap_{a \in [A]_{\mathcal{M}}} \mathcal{K}^a_{\mathcal{M}}(q)$$

Intuitively this means, that the agent-type relative knowledge set comprises those formulae that are known by every agent of type A in state q. To further investigate the relation between agent-relative and agent-type relative knowledge we define a notion of *self-knowledge*, i.e., under which circumstances an agent knows which agent it is.

Definition 4.12 (Agent Self-Knowledge)
An agent a of type A knows that it is agent a of type A in local state q if and only if no other agent a' of type A ever reaches local state q. Formally:

$$\textit{know-self}_{\mathcal{M}}(q, a) := \forall a' \in [A]_{\mathcal{M}}, a' \neq a : \forall s \in \textit{reach}(\mathcal{M}) : s(a').q \neq q$$

Intuitively this means, if agent a is in state q it knows, that it must be agent a as no other agent of its type can be in state q. This is an adequate extension of the possible worlds model, as it captures which concrete agents an agent has to consider possible for itself. For traditional multi-agent systems, there exists only one agent of each type, and thus agents implicitly know which agents they are as they know their type. We now prove that if an agent knows which agent it is, then the agent-type relative knowledge set is identical with the agent-relative knowledge set (see Theorem 4.14).

Lemma 4.13 (Unreachable State Knowledge Set)
If a local state q is never reached by an agent a, then $K^a_{\mathcal{M}}(q) = \Phi$. Formally:

$$(\forall s \in \textit{reach}(\mathcal{M}) : s(a).q \neq q) \Longrightarrow K^a_{\mathcal{M}}(q) = \Phi$$

Proof If $(\forall s \in \textit{reach}(\mathcal{M}) : s(a).q \neq q)$, then there is no $s \in \textit{reach}(\mathcal{M})$ such that $s(a).q = q$. If there is no $s \in \textit{reach}(\mathcal{M})$ such that $s(a).q = q$, then $\forall s \in \textit{reach}(\mathcal{M}), s(a).q = q : \langle \mathcal{M}, s, \sigma \rangle \vDash \varphi$ holds for every $\varphi \in \Phi$. Thus, $\mathcal{K}^a_{\mathcal{M}}(q) = \Phi$, by Definition 4.9. ■

Theorem 4.14 (Agent-Type-Relative and Agent-Relative Knowledge)
If an agent a of type A knows that it is agent a in state q, then the agent-type relative knowledge in q is identical to the agent-relative knowledge of a in q. Formally:

$$\textit{know-self}_{\mathcal{M}}(q, a) \Longrightarrow \mathcal{K}^A_{\mathcal{M}}(q) = \mathcal{K}^a_{\mathcal{M}}(q)$$

Proof If $\textit{know-self}(q, a)$, then $\forall a' \in [A], a' \neq a : \mathcal{K}_{a'}(q) = \Phi$ by Definition 4.12 and Lemma 4.13. Therefore, $\mathcal{K}_A(q) = \mathcal{K}_a(q)$ by Definition 4.11 and 4.9. ■

Theorem 4.14 allows us to capture the original notion of agent-relative knowledge programs [9, see p. 253 ff.] by explicitly requiring that the agents know who they are. Hence, agent-type relative knowledge programs for open multi-agent systems are an extension of knowledge-based programs in traditional multi-agent systems where this assumption is implicitly present. Before we define knowledge-based programs for open multi-agent systems, we define knowledge tests which are used by knowledge-based programs in addition to local state tests.

Definition 4.15 (Knowledge Tests)
We define the syntax of knowledge tests by the following abstract grammar

$$k := \neg k \mid k_1 \wedge k_2 \mid K\varphi$$

where $\varphi \in \Phi$ is a CTLKx formula. The other boolean connectives are defined as usual. Let T_K denote the set of knowledge tests.

Given a knowledge test k, a state q, and a function $\mathcal{K}$ for a knowledge set, we define the semantics of knowledge test k recursively through the following evaluation function:

$$\llbracket \neg k \rrbracket_K(\mathcal{K}, q) := \begin{cases} \top & \text{iff } \llbracket k \rrbracket(\mathcal{K}, q) = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$\llbracket k_1 \wedge k_2 \rrbracket_K(\mathcal{K}, q) := \begin{cases} \top & \text{iff } \llbracket k_1 \rrbracket(\mathcal{K}, q) = \llbracket k_2 \rrbracket(\mathcal{K}, q) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$\llbracket K\varphi \rrbracket_K(\mathcal{K}, q) := \begin{cases} \top & \text{iff } \varphi \in \mathcal{K}(q) \\ \bot & \text{otherwise} \end{cases}$$

Based on knowledge tests we define knowledge-based programs as an extension of local decision programs.

Definition 4.16 (Knowledge-Based Programs)
An *knowledge-based program* P_K [9, cf. p. 253 ff.] for an agent type A is a set $P_K \subseteq T_L \times T_K \times [Act]$ where T_L again denotes a set of local state tests and T_K denotes the set of knowledge tests. A knowledge-based program P_K induces the following protocol function:

$$\rho_{P_K}(q) := \left\{ \alpha \;\middle|\; \langle t_l, k, \alpha \rangle \in P : \mathit{decide}_L(t_l, q) = \top \text{ and } \llbracket k \rrbracket_K\left(\mathcal{K}^A_{\mathcal{M}}, q\right) = \top \right\}$$

Just like for local decision programs we say that P_K *represents* the decisions of agent type A in the given OMAS if and only if $\rho_{P_K} = A.\rho$, i.e., the program yields the exact same actions as the protocol does. An agent-type A in an OMAS *implements* a knowledge-based program if and only if the program represents the decisions of the agent.

Let's take a step back from the definitions and reconsider the cookie example, how could a knowledge-based program for the cookie monster look like? While explaining the original program, we already used the term "knows", namely, that the cookie monster eats the cookie if it knows that there is a cookie. With the formalism of knowledge-based programs we can now obtain a knowledge-based program (see Code 4.17) which exactly captures this intuitive description of the behavior in terms of knowledge and is implemented by the cookie-eating OMAS introduced in the last chapter (see Figure 3.1).

Code 4.17 (Cookie Monster Knowledge-Based Program)

```
1: if $\neg K(\mathit{cookie}()) \wedge \neg K(\neg \mathit{cookie}())$ then { $\mathit{open}!(1), \mathit{open}!(0)$ }
2: if $K \mathit{cookie}()$ then { $\mathit{eat}!()$ }
3: if $K \neg \mathit{cookie}()$ then { $\mathit{idle}()$ }
```

The first line of Code 4.17 checks whether the agent knows that there is a cookie or that there is no cookie in the drawer. If the agent does neither know that there is a cookie, nor that there is no cookie, then it opens the drawer. As a result of opening the drawer the agent knows whether there is a cookie or not, as the observation, i.e., $\mathit{open}!(1)$ or $\mathit{open}!(0)$, is stored in the local state (see Figure 3.1b). If the agent now knows, that there is a cookie, then it eats the cookie. If it knows, that there is no cookie, then it idles.

Unlike local decision programs which can be implemented straightforwardly, knowledge-based programs require a more sophisticated synthesis step, i.e., searching for an OMAS which implements a program for a specific agent [30]. This is mainly due to an infinite regress, i.e., what agents know in the agent-type relative sense in a certain local state depends on their actions, and their actions are defined utilizing their knowledge.

Implementability

Consider the knowledge based-program Code 4.18 and assume the drawer is initially opened and the agent knows that there is a cookie in it.

Code 4.18 (Unimplementable Knowledge-Based Program)

```
1: if $K(\mathrm{AG}\, \mathit{cookie}())$ then { $\mathit{eat}!()$ }
2: if $\neg K(\mathrm{AG}\, \mathit{cookie}())$ then { $\tau$ }
```

The two knowledge tests are exhaustive and disjoint, i.e., exactly one of them must evaluate to true. If the agent itself does not eat the cookie, then no other agent does. Hence, if the agent does not execute *eat*!(), AG *cookie*() holds and the agent knows that it holds, as it knows, initially that there is a cookie in the drawer. Hence, $K(\mathrm{AG}\, \mathit{cookie}())$ would hold, if the agent would not decide to eat the cookie, but, if $K(\mathrm{AG}\, \mathit{cookie}())$ holds, the agent decides to eat the cookie. This means, we can never execute *eat*!(). But neither can we not execute *eat*!() because then we would have to execute *eat*!(). This leads up to a vicious infinite regress, i.e., knowledge depends on the actions taken and the actions taken depend on the knowledge without any hope to eventually well-ground the knowledge or actions. The knowledge-based program Code 4.18 is thus not implementable. Whether a program has an actual implementation is often not obvious or even undecidable [19] and makes it thus very hard for a programmer to program something. This makes knowledge-based programs unsuitable as a basis for a general MANET middleware, although proving a useful notion of knowledge-based programming.

4.3 Algorithmic Knowledge Programs

Another weaker explication of knowledge-based programming are algorithmic knowledge programs [9, p. 402 ff.]. Algorithmic knowledge programs are based on an algorithmic inference capacity which allows computing explicit agent-type relative knowledge given the local state of an agent.

Definition 4.19 (Algorithmic Inference Capacity)
An *algorithmic inference capacity* for an agent type A is a function from CTLKx formulae and local states to a three-valued truth domain [9, cf. p. 394 f.], i.e.:

$$\Lambda^A_{\mathcal{M}} : \Phi \times A.Q \to \{ \top, ?, \bot \}$$

We define the *extension* $\left[\Lambda^A_{\mathcal{M}}\right](q)$ of an algorithmic inference capacity in state q as the set of formulae which are known according to the inference capacity, i.e.:

$$\left[\Lambda^A_{\mathcal{M}}\right](q) := \left\{ \varphi \in \Phi \;\middle|\; \Lambda^A_{\mathcal{M}}(\varphi, q) = \top \right\}$$

Definition 4.20 (Soundness of Inference Capacity [9, cf. p. 397])
An inference capacity is *sound* if and only if every formula which is known according to the inference capacity in state q is actually known in state q, i.e.:

$$\forall q \in A.Q : \forall \varphi \in \Phi : \quad \begin{array}{l} \Lambda^A_{\mathcal{M}}(\varphi, q) = \top \Longrightarrow \varphi \in \mathcal{K}^A_{\mathcal{M}}(q) \\ \Lambda^A_{\mathcal{M}}(\varphi, q) = \bot \Longrightarrow \varphi \notin \mathcal{K}^A_{\mathcal{M}}(q) \end{array}$$

Definition 4.21 (Completeness of Inference Capacity [9, cf. p. 397])
An inference capacity is *complete* with respect to a subset $\Psi \subseteq \Phi$ of CTLKx formulae if and only if:

$$\forall q \in A.Q : \forall \varphi \in \Psi : \Lambda^A_{\mathcal{M}}(\varphi, q) \in \{ \top, \bot \}$$

An inference capacity is *complete* if and only if it is complete with respect to Φ.

Explicit knowledge is more fine-grained than implicit knowledge. It is defined on a pure syntactical level, and it is thus technically possible that logically equivalent formulae evaluate differently. Algorithmic knowledge provides a solution for the logical omniscience problem, i.e., that agents are omniscient assuming the possible worlds model [9, cf. p. 333], but in practice, they are not most of the time. An algorithmic inference capacity which is both sound and complete, however, does resemble implicit knowledge completely.

Based on algorithmic knowledge we define algorithmic knowledge programs.

Definition 4.22 (Algorithmic Knowledge Program)
An *algorithmic knowledge program* P_{KA} is a knowledge-based program with different semantics for the induced protocol, namely that knowledge tests are evaluated using the algorithmic inference capacity instead of agent-type relative knowledge sets, i.e.:

$$\rho_{P_{KA}}(q) := \left\{ \alpha \;\middle|\; \langle t_l, k, \alpha \rangle \in P : \mathit{decide}_L(t_l, q) = \top \text{ and } [\![k]\!]_K\left(\left[\Lambda^A_{\mathcal{M}}\right], q\right) = \top \right\}$$

If we write down an algorithmic knowledge program, then finding an algorithm which is both sound and complete with respect to all occurring knowledge queries is as hard as the original problem of implementing knowledge-based programs. Also, programs based on algorithmic knowledge might not be implementable in the sense that there exist such algorithms, as algorithmic knowledge produced by a sound and complete inference capacity is essentially implicit knowledge leading up to the same potentially vicious infinite regress. Thus, the problem merely shifts to the knowledge deduction algorithm, and we are back to square one. We now face the challenge that knowledge-based programs are too strong, in the sense that they are based on knowledge about the same system in which the actions get executed leading to a possibly vicious infinite regress, and for algorithmic knowledge programs we need adequacy criteria weaker than soundness and completeness but still strong enough to be useful. Otherwise, we get the same problems as for knowledge-based programs for algorithmic knowledge programs.

4.4 Towards a MANET Middleware

While this chapter provides the first major contribution of this thesis, namely a notion of knowledge-based programs for open multi-agent systems, it also raises two further challenges for the general applicability of knowledge-based programs.

We have seen in this chapter that, although the idea of knowledge-based programs is very appealing and the first step in the right direction towards belief-based programming, they have a major shortcoming. Not every program is implementable, and it could be challenging or even impossible for the programmer to see whether the current program she writes will be implementable. Moreover, the knowledge model itself is too strong. Very much in MANETs is not knowable within the possible worlds model of knowledge. For instance, unpredictable and not observable topology changes make it impossible to know anything about the topology, as the agents cannot update their local states synchronously to topology changes and hence, an agent has to consider many topologies possible. As a matter of fact, an agent cannot know anything about the topology if one does not restrict topology changes to a very limiting model.

In the next two chapters, we tackle those two shortcomings by weakening knowledge-based programs. In the first step, we tackle the first shortcoming arriving at something very similar to knowledge-based programs but guaranteed to be implementable. In the second step, we weaken the knowledge aspect itself by introducing beliefs. This makes it finally possible to act upon beliefs about the topology and can be explicated in useful ways although, unlike knowledge, beliefs are not required to be true.

Chapter 5

The Nomological Framework

Not all knowledge-based programs as introduced in the last chapter are implementable [30] and it might not be obvious or even undecidable [20] whether they are or not. This makes them unsuitable as a basis for a general MANET middleware. In this chapter, we, therefore, weaken the notion of knowledge-based programs such that the resulting notion of knowledge-based programming guarantees implementability of all programs which adhere to some very basic criteria. The resulting *nomological framework* also serves as a general basis for belief-based programming as well as for normative adequacy criteria for beliefs and thus plays a central role in this thesis.

We begin this chapter with a brief investigation of the fundamental problem which allows for unimplementable knowledge-based programs. Subsequently, we introduce the *nomological framework*, explain how it circumvents this fundamental problem.

Knowledge-based programs require a synthesis step, where we search for a system which implements them [19]. This is required because knowledge tests are evaluated within the same system in which the actions are performed [8]. Which actions are performed depends on the knowledge which on the other hand depends on the actions performed. Another notion of knowledge-based programs has been introduced in [15] and [26] where the knowledge is assumed to be explicitly stored in the local state making those programs more like our local decision programs [8, cf. p. 7].

The key problem which leads to unimplementable knowledge-based programs is — as we already saw in the last chapter — that knowledge tests are evaluated within the same system in which the actions are performed leading to a possibly vicious infinite regress.

The fundamental idea of this chapter is to distinguish two views on a system, both described using the formalism for open multi-agent systems introduced in Chapter 3. The *nomological view* and the *actual view* captured by the *nomological system* and *acutal system* respectively. Knowledge tests are evaluated on the nomological view, i.e., in the nomological system, while actions are performed on the actual view, i.e., in the actual system. This circumvents the infinite regress problem by separating the system in which knowledge tests are evaluated from the system in which actions are performed. Hence, it circumvents the fundamental problem of knowledge-based programs. In the remainder, we have a closer look at both views and how they are related to each other.

5.1 Nomological System

A nomological system is an open multi-agent system which describes general laws. These include the options for action available to agents utilizing nondeterminism and the *nomological effects* of those actions on agents.

The word *nomological* refers to basic physical laws or rules of reasoning and originates from the Greek word *nomos* for *law* and *logos* for *reason*. It is often used in philosophy to distinguish between different kinds of modalities, e.g., $\varphi \vee \neg\varphi$ is a logical necessity while that nothing can move faster than the speed of light is a nomological necessity [10, cf. p. 5], i.e., necessary by the laws of physics. These nomological necessities restrict the way the actual world can evolve, e.g., it is not possible that an object moves faster than the speed of light in the actual world.

Just like the laws of physics restrict our actions and determine their effects, the nomological system restricts the options for action of agents within an OMAS and determines at least partially their effect. We refer to the agent types of the nomological system as nomological agent types. We refer to the actions an agent can perform as the *nomological actions* or *options* and to their effect as the *nomological effect*. Just like humans have the innate ability to learn, we require that agents have some kind of innate nomological abilities, that is, actions they are capable of performing and corresponding nomological effects of those actions. These abilities must be strong enough such that they can be used to do what the agents are supposed to do.

Freedom of choice, i.e., that an agent can choose between multiple actions is modeled by means of *nomological nondeterminism*, i.e., nondeterminism in the nomological system which is resolved by choosing an action in the actual system. This gives us a first idea of how we obtain the actual system. We use a program which uses knowledge an agent posses about the nomological system to resolve the nomological nondeterminism, i.e., decides which of the available options for action to perform.

To make an example of how this could look like: Let's assume you have two options for action, you can either turn on a stove with a pot of water on it, or you cannot do so. If your turn on the stove, the water will eventually boil. This effect is nomological, it is done to the water by the laws of physics, and you cannot influence it. You can, although, decide whether you initially want to turn on the stove or not. It works just the same within the nomological framework, agents cannot influence nomological effects, but they can decide between several nomological possibilities for action explicitly encoded in the nomological system by means of nomological nondeterminism.

Agents act within the frame of the nomological system. Maybe, we would like to move faster than the speed of light, but that is just nomologically impossible, so we cannot do it. We have to stick to options which are nomologically given. The same holds for the agents in a MANET, maybe a node wants to send a message to a node within another MANET segment, but that is just nomologically impossible. An adequate nomological MANET model would capture these possibilities for action of the nodes.

Nomologically enforced actions are actions the agent has no control over, they are basic reactions triggered by interacting with other agents. They are comparable to, e.g., the blink reflex of human beings which bypasses the human brain completely and just immediately reacts to an external stimulus. The same is true for nomologically enforced actions which cannot be prevented using a decision program running on the agent, as they are directly triggered and bypass any decision procedure.

Figure 5.1 shows a nomological cookie-eating model. The agent can decide to do nothing in each state, the agent can decide to open the drawer in the initial state, and the agent can decide to eat the cookie, after opening the drawer and seeing that there is a cookie. This intuitively encodes all nomologically possible actions.

It is crucial to note here that the agents need some means to know which actions are nomologically possible to choose among them which directly leads to knowledge, e.g., in Figure 5.1 the agent may only eat the cookie if there actually is a cookie. If one wants to prevent this kind of knowledge leakage, then every action has to be possible in every state. Nevertheless, an action could still fail to synchronize, e.g., because the environment does not let an agent eat a cookie which is not there. This would allow an agent to do experiments to learn something about the environment but somehow contradicts the idea that the nomological options are actual options for action. Another way to deal with

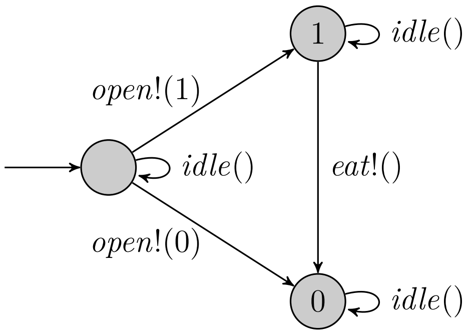

Figure 5.1: Nomological Cookie Monster Agent Type

knowledge leakage is to map actions like eating a cookie which is not there to error (sink) states, which would require agents to learn somehow which actions they actually can perform, or ignoring them altogether.

5.2 Nomological Knowledge

The acquisition of knowledge is also a nomologically enforced effect. We cannot decide what we know. Rather it is given by how the world works. We can although decide which actions to perform based on our knowledge which might indirectly modify our knowledge. So, agents do not get to decide what they know and what they do not know, at least not directly. They can, however, decide to perform actions selected among the nomological options for action. Performing certain actions leads to an implicit effect on their knowledge. Nomological knowledge is the knowledge agents have about the nomological system. Let $\mathcal{M}_N$ denote a nomological OMAS. We can now formally define agent-relative and agent-type relative nomological knowledge by means of knowledge sets on $\mathcal{M}_N$.

Definition 5.1 (Nomological Knowledge Sets)
We define *agent relative nomological knowledge* of agent a by $\mathcal{K}^a_{\mathcal{M}_N}$ and *agent-type relative nomological knowledge* of agents of agent type A by $\mathcal{K}^A_{\mathcal{M}_N}$.

Based on this knowledge set definition we define nomological knowledge programs.

Definition 5.2 (Nomological Knowledge Programs)
Let $P_N \subseteq T_L \times T_K \times [Act]$ be a *nomological knowledge program.* Given an additional state space Q for storing non nomological state, nomological knowledge programs induce a

protocol function for *augmented states* $\langle q_n, q\rangle \in A.Q \times Q$:

$$\rho_{P_N}(\langle q_n, q\rangle) := \left\{ \alpha \;\middle|\; \langle t_l, k, \alpha\rangle \in P_N : \mathit{decide}_L(t_l, \langle q_n, q\rangle) = \top \text{ and } [\![k]\!]_K\Big(\mathcal{K}^A_{\mathcal{M}_N}, q_n\Big) = \top \right\}$$

We call P_N *valid* for nomological agent type A if and only if the actions chosen are a subset of the nomological options of A and a superset of the nomologically enforced actions, formally encoded by means of an *enforcement protocol* ρ_E, i.e.:

$$\forall q \in Q : \rho_E(q_n) \subseteq \rho_{P_N}(\langle q_n, q\rangle) \subseteq A.\rho(q_n)$$

If P_N is valid for A we obtain an *augmented agent type* A' with

$$\begin{aligned} A'.Q &:= (A.Q \times Q) \cup \{\, \epsilon \,\} \\ A'.\eta(\langle q_n, q\rangle, \alpha) &:= \langle A.\eta(q_n, \alpha), \eta(\langle q_n, q\rangle, \alpha)\rangle \\ A'.\rho(\langle q_n, q\rangle) &:= \rho_{P_N}(q_n, q) \end{aligned}$$

where $\eta : A.Q \times Q \to Q$ is an additional effect function for the augmented states. The augmented agent type needs to fulfill Definition 3.2 and 3.4. For simplicity we treat the uninitialized state ϵ and $\langle \epsilon, \epsilon\rangle$ equivalently. If at least one of the effect functions that make up $A'.\eta$ is undefined for the given parameters, then $A'.\eta$ is also undefined. We can now *implement* P_N by swapping the original agent type A in $\mathcal{M}_N$ with the augmented agent type A' in all initial states of the nomological system yielding a new system $\mathcal{M}$. Formally we replace each initial state $s \in \mathcal{M}_N.I$ with a state s' such that

$$s'(a) := \begin{cases} \langle A', \langle s(a), q_0\rangle\rangle & \text{iff } s(a).A = A \\ s(a) & \text{otherwise} \end{cases}$$

where $q_0 \in Q$ is either ϵ, if $s(a) = \epsilon$, or some other arbitrary state from Q.

In Definition 5.2 we use the nomological knowledge to decide which actions to take and then derive a new OMAS by replacing the protocol of the respective compatible agent type. Since the program is required to be valid implementing nomological knowledge programs can be seen as reducing nomological nondeterminism, i.e., the resulting system contains less nondeterminism than the original nomological OMAS.

Programs based on nomological knowledge can do more than deciding which actions to take. Since the knowledge is also available when updating the augmented state (see Definition 5.2), we can use knowledge to compute this state and then use local state tests based on these computation results. We see how we can utilize this extended notion and use knowledge respectively beliefs to compute successor states in the next chapter.

According to the nomological notion of knowledge an agent does not know what it will do in the future. This makes sense since what it will do might depend on what others do which again might depend on what the agent itself does. So, this could again lead to a possibly vicious infinite regress which we prevent by not making any assumptions about the future actions of an agent. However, an agent might be able to resolve some nondeterminism in the nomological system by carefully analyzing its own nomological knowledge program and those of others and check whether some decisions only depend on already determined knowledge. Furthermore, after fixing a nomological knowledge program for a specific agent type, we get a new nomological system which could be used as a basis for other agents, i.e., technically we could fix various decisions of the agents which are already determined in an iterative process and by that resolve nondeterminism potentially strengthening knowledge in each step. We leave further investigation of this open to future work.

Depending on the particular scenario, it might make sense to restrict the options for action to active actions and enforce that an agent accepts all nomologically possible passive counterparts for actions of other agents. For instance, for MANETs, an agent might choose which packets to send, but it cannot decide which packets to accept.

Definition 5.3 (Actual System)
Given valid nomological knowledge programs for a set of nomological agent types $\mathbb{A} \subseteq \mathcal{A}$ we build the *actual system* by transforming all the agent types appropriately such that the resulting system implements the nomological knowledge programs for each agent type.

The main difference between nomological knowledge programs and knowledge-based programs is that they are not evaluated within the nomological system but rather used to *strengthen* the nomological system, i.e., resolve nondeterminism. This means, that what is nomologically knowable is not influenced by the actions taken. Which tackles the main problem of knowledge-based programs, namely the possibly vicious infinite regress.

If we again recap Code 4.18, then we can now simply execute the program without having any difficulties. AG $\mathit{cookie}()$ just does not hold even if the agent does never decide to eat the cookie because in the nomological system there always exists a path such that eventually the cookie has been eaten, since the agent could decide to do so in each step. This means that $K\mathrm{AG}\,\mathit{cookie}()$ evaluates to false and the agent just does nothing, which does not influence wether AG $\mathit{cookie}()$ holds in the nomological system.

So, Definition 5.2 provides well-defined semantics for every valid nomological knowledge program. The semantics are not arbitrary but weaker than those of knowledge-based programs. In contrast to knowledge-based programs, formulae known nomologically may be false in the actual system. For instance, if an agent decides to never eat the cookie in the actual system, then there exists no path in the actual system such that the cookie is eventually eaten. However, such a path exists in the nomological system.

To be able to implement a nomological knowledge program we need to be able to decide nomological knowledge for the respective agent type. However, in general, both nomological and actual knowledge are undecidable.

Theorem 5.4 (Undecidability of Knowledge)
Whether $\varphi \in \mathcal{K}^{a}_{\mathcal{M}}(q)$ is undecidable in general.

Proof Consider an arbitrary but fixed turing machine TM. Construct an agent which given TM and an input I simulates TM on I. If TM halts, then the agent makes a transition into a special state h. We define a predicate H as follows:

$$H(s) := \begin{cases} \{\, \langle \text{TM} \rangle \,\} & s(a).q = h \\ \emptyset & \text{otherwise} \end{cases}$$

Now AF $H(TM)$ holds if and only if TM eventually halts. The agent knows right after initialization whether TM halts or not, as this is determined. However, this knowledge is surely not decidable since as it would decide the halting problem. ■

Due to the possibly infinite state space of agent types, the formalism itself is Turing complete. It is possible to capture the content of the tape of a Turing machine in the local states of an agent type. The effect function then allows capturing the transition relation of the Turing machine. This allows an agent to simulate the Turing machine.

Corollary 5.5
It follows from Theorem 5.4 that the model checking problem for OMAS is undecidable, as deciding whether $\langle \mathcal{M}_H, s, \sigma \rangle \vDash \text{AF}\, H(TM)$, where $\mathcal{M}_H$ denotes the OMAS constructed in Theorem 5.4, is undecidable in general.

As a result of Theorem 5.4 we cannot directly compute the implementation for all valid nomological knowledge programs although there exists an implementation of every valid program. Computing the implementation as specified in Definition 5.2 requires us to decide nomological knowledge. However, if both the reachable state space and the universe is finite, then we can adapt the satisfaction set based CTL model checking algorithm [1, see p. 341 ff.] to automatically compute nomological knowledge by augmenting states with additional information for satisfaction assignments. We also think that this is possible for an infinite universe when imposing that the satisfaction assignments for each state are finitely representable and this finite representation allows for the needed operations on satisfaction assignments when computing satisfaction sets.

Nomological knowledge programs can be seen as an explication of algorithmic knowledge programs, but instead of requiring sound and completeness regarding the actual system we require sound and completeness regarding the nomological system, and we can use model checking algorithms to decide this knowledge in the finite case.

Nomological knowledge programs, however, raise the question how the nomological effect function has to be designed such that the agents gather sufficient knowledge. If we can come up with some correctness criteria for the nomological knowledge of agents, e.g., that they eventually know φ if φ globally holds, then finding a nomological effect function can be seen as a synthesis problem, i.e., coming up with a nomological state space and effect function which satisfies the correctness specification.

Chapter 6

Belief-Based Programming

Knowledge-based programming, in general, has a serious shortcoming. It only accounts for those propositions and corresponding formulae an agent actually knows. In mobile ad-hoc networks and other highly dynamic and decentralized systems, however, most of the time very little about the system is knowable. That is why we use beliefs to generalize the approach of knowledge-based programming in this chapter. In contrast to knowledge, beliefs may be incorrect and thus allow an agent to act upon falsehoods. Nevertheless, to be useful beliefs certainly must not be arbitrary.

We begin this chapter with a brief motivation of belief-based programming by showing that existing MANET protocols already rely on some informal notion of beliefs. The goal of this chapter and the whole concept of belief-based programming is to capture this notion within a formal framework and thereby enable us to reason about such beliefs, specify correctness criteria for them, and use them for programming. To this end, we introduce a formalism which allows to model beliefs as well as their revision and subsequently investigate various normative requirements for beliefs. Based on that framework we further generalize knowledge-based to belief-based programming. Finally, we demonstrate the usefulness of our approach using the example of an interest-based routing protocol inspired by the digital message board example. We conclude this chapter with a rough sketch of a general belief-centric middleware which also gives an outlook on future work.

To get a better intuition for the limitations of knowledge-based programming, let us start with a concrete example of a proposition which we would like to use in MANET protocols but is not knowable. Most nontrivial network protocols for MANETs need to route packets at some point. For instance, in case of the digital message board, we need to route new messages to everyone who is interested. While naive broadcast-based flooding of a network with a packet might suffice to deliver the packet, this is certainly not an efficient use of network resources. Hence, an efficient routing protocol has to take at least partial aspects of the topology of the underlying network into account. An example of useful knowledge regarding the network topology would be, whether there is a link between two nodes v and v' which could then be used as part of a route. However, due to frequent and unpredictable changes in the network topology, it is in general not knowable whether there is a link between two nodes v and v'. The topology in MANETs does change without the nodes immediately noticing. Hence, nodes have no chance to update their local state on link changes and as a result, cannot distinguish between global states where these links exist and states where they do not. According to the possible worlds model of knowledge, they, therefore, can generally not know whether there is a link or not.

Many common routing protocols for mobile ad-hoc networks have some kind of mechanism to either discover new routes on-demand [23, eg.] or to proactively distribute routing tables in the network [5, eg.]. Both types of protocols have in common that routing information might be outdated, i.e., does not match the current topology of the network and contains routes with broken links. Thus, both types of protocols already rely on some informal notion of beliefs — they use the routing information they believe is correct to route packets even though it might not be correct. If they become aware of a broken route, they revise their beliefs accordingly and no longer use this route although it could be the case that in the meantime the route exists again. Further, these beliefs cannot be characterized as knowledge in any way, neither as knowing that a route exists nor as not knowing that a route does not exist. The goal of this chapter is to provide a formal framework not only to describe such beliefs but also study normative criteria for them.

6.1 Belief Theories

So, how to model beliefs of agents within a system? Let's start by listing some properties of beliefs we would like to capture with the formal model. We do so, using the example of MANET routing protocols and network links introduced above.

The first most obvious or even defining property of beliefs is that beliefs in contrast to knowledge might be incorrect. While this is the property of beliefs which makes them attractive for highly dynamic systems it also raises an important question: If truth is not an adequacy criterion for beliefs then, how can we distinguish adequate beliefs which are useful from those which are not? After introducing a descriptive theory of beliefs in this section, we extensively study normative criteria in the next section.

Another very interesting property of beliefs is that they may come in different certainties or degrees. For example, we are usually more certain about the current weather at our current location than about the weather far away. In case of MANET routing protocols, a node might analogously be more certain about links nearby than links far away. This is reasonable since information about existing or broken links takes more time to travel to a node the further away this node is from those links in the topology. Thus, in the meantime, the information about those links could already be outdated.

Moreover, beliefs are inherent to local states and unlike knowledge cannot be defined without interpreting the internal structure of them. For example, whether a node believes that there is a link between two nodes, is not a relation between a black-box local state and the system but rather explicitly encoded in the local state itself, e.g., in the form of a routing table. While it made sense to take sets of indistinguishable global states as the objects of knowledge, this raises the question what the objects of beliefs are. Instead of sets of indistinguishable global states, we chose CTLKx formulae to be the objects of beliefs and study the implications of this choice later in this chapter.

The formal model of beliefs needs to capture these characteristics. Instead of directly coupling beliefs with agents or their types we introduce a more general approach. A *belief theory* comprises a set of *belief states* which encode particular beliefs and can be interpreted with a *belief inference capacity*. A belief inference capacity assigns to each belief state and CTLKx formula a belief degree from some *belief domain*. To revise beliefs in response to new information, a belief theory provides, similar to agent types, a *belief effect function* which takes a belief state and new information in the form of an observed action and maps them to a successor belief state encoding the revised beliefs.

Definition 6.1 (Belief Theory)
A *belief theory* $\mathcal{T}_B$ is a quadruple $\langle Q_B, \eta_B, \Gamma, \mathcal{B} \rangle$ comprising a set of *belief states* Q_B, a *belief effect function* $\eta_B : Q_B \times [Act] \to Q_B$, a *belief inference capacity* $\Gamma : \Phi \times Q_B \to \mathcal{B}$, and a strictly partially ordered *belief domain* $\mathcal{B}$. We refer to the elements $b \in \mathcal{B}$ of the belief domain as *belief degrees* and to the partial order on $\mathcal{B}$ by $\succ$. A belief theory *interprets* an agent type A if and only if $Q_B = A.Q$ and $\eta_B = A.\eta$.

Please note that belief theories are a very general tool to study beliefs. We may use them to ascribe beliefs to agents by defining a belief theory interpreting the respective agent type or as an independent theory as part of an agent type enabling the agents to act upon beliefs. In both cases we refer in the following to the belief state of agent a in global state

s of some OMAS $\mathcal{M}$ as $s(a).q_B$. In the former case, the agent's belief state is identical to the full local state of the agent, i.e., $s(a).q_B = s(a).q$, while in the latter case the belief state is just part of the agent's local state. Further, we omit the $s(a)$ and refer to the belief state merely by q_B if the global state s is irrelevant.

6.1.1 Belief Domains

The belief degrees a belief domain $\mathcal{B}$ provides together with the partial order $\succ$ on them enable us to distinguish between different levels of certainty. An agent in belief state q_B is *more certain* that φ than that ψ if and only if $\Gamma(\varphi, q_B) \succ \Gamma(\psi, q_B)$. In this thesis, we use a five-valued, totally ordered belief domain providing two degrees of belief and disbelief respectively and one belief degree to express indifference.

Definition 6.2 (Belief Domain)
We define the five-valued totally ordered belief domain $\mathcal{B}_5$ by:

$$\mathcal{B}_5 := \{ \bot, \downarrow, ?, \uparrow, \top \} \quad \text{with} \quad \top \succ \uparrow \succ ? \succ \downarrow \succ \bot$$

In the remainder of this thesis, we implicitly assume, except otherwise stated, that the belief domain of a belief theory is this five-valued belief domain $\mathcal{B}_5$. Although there are other possibilities to explicate belief domains, this belief domain already provides enough belief degrees for many interesting cases like MANET routing protocols.

Definition 6.3 (Terminology for Belief Theories)
We further define the following useful terminology for belief theories based on $\mathcal{B}_5$. An agent *a is certain of* φ in belief state q_B if and only if $\Gamma(\varphi, q_B) = \top$. An agent *a believes* φ in belief state q_B if and only if $\Gamma(\varphi, q_B) \in \{ \uparrow, \top \}$. An agent *a disbelieves* φ in belief state q_B if and only if $\Gamma(\varphi, q_B) \in \{ \downarrow, \bot \}$. An agent a is *indifferent* of φ in belief state q_B if and only if $\Gamma(\varphi, q_B) = ?$ and an agent *a knows* φ in global state s if and only if it believes φ and φ is actually true, i.e., $\Gamma(\varphi, s(a).q_B) \in \{ \uparrow, \top \}$ and $\langle \mathcal{M}, s, \sigma \rangle \vDash \varphi$.

It is crucial to note that Definition 6.3 provides us with a different notion of knowledge than the possible worlds model. While Definition 6.3 still fulfills the necessary condition that knowledge must not be false, it understands knowledge as true beliefs. This notion of knowledge explicitly decouples knowledge from certainty, i.e., agents may be certain of φ without knowing that φ and agents may know φ without being certain of it.

6.1.2 Evidence

A useful concept when it comes to beliefs and their revision is evidence. For example, imagine a MANET routing protocol using heartbeats on links to check, whether those links still exist. While a received heartbeat should be evidence that the link is still up, a missing heartbeat should be evidence that the link is broken. Within our formal model, agents perceive the system through the lens of a sequence of observations in terms of actions. Hence, evidence comes in the form of these observations.

Definition 6.4 (Observations)
An *atomic observation* is an action $\alpha \in [Act]$. Based on atomic observations we define *finite observation fragments* $\hat{\varrho}$ as finite sequences of atomic observations, i.e., $\hat{\varrho} \in [Act]^*$, and analogously *infinite observation traces* ϱ by $\varrho \in [Act]^\omega$. We usually treat atomic observations as finite observation fragments of length one.

In general, it is important to distinguish the descriptive model of evidence, i.e., what is evidence given a specific belief theory and belief state, from normative requirements, i.e., what should be evidence. While the former is belief-theory and belief-state relative, the latter is a universal, although system relative, normative criterion.

Intuitively, a finite observation fragment $\hat{\varrho}$ is evidence for a formula φ relative to belief state q_B if and only if it strengthens the belief in φ, i.e., starting in belief state q_B and after processing $\hat{\varrho}$ with the belief effect function the degree of belief that φ is greater compared to the original belief degree assigned to φ in belief state q_B.

Definition 6.5 (Evidence)
A finite observation fragment $\hat{\varrho} = \alpha_1\alpha_2\ldots\alpha_n$ is *evidence* for φ relative to belief state $q_B \in Q_B$ and belief theory $\langle Q_B, \eta_B, \Gamma, \mathcal{B}\rangle$ if and only if:

$$\Gamma(\varphi, q_B) \prec \Gamma(\varphi, \eta_B(\eta_B(\ldots\eta_B(q_B, \alpha_1)\ldots, \alpha_{n-1}), \alpha_n))$$

The same observation fragment is *counter-evidence* for φ if and only if:

$$\Gamma(\varphi, q_B) \succ \Gamma(\varphi, \eta_B(\eta_B(\ldots\eta_B(q_B, \alpha_1)\ldots, \alpha_{n-1}), \alpha_n))$$

In case that the belief degrees assigned to φ before and after processing a finite observation fragment $\hat{\varrho}$ are incomparable using $\succ$, we are unable to tell whether the agent is more certain that φ before or after the observation of $\hat{\varrho}$. Hence, we can neither classify the observation as evidence nor as counter-evidence in such cases. Furthermore, for infinite observation traces the concept of evidence makes no sense because an agent never is in a state where it observed an infinite trace of actions.

The concept of evidence is beneficial when analyzing belief theories. It allows us to reason about the information carried by individual communication acts between two agents and its effects on the belief state. For example, we may study the role of the ordering of (atomic) observations, or whether observations are idempotent, i.e., multiple (consecutive) identical observations have no effects on the beliefs beyond the initial observation.

6.2 Normative Requirements

To be actually useful beliefs certainly must not be arbitrary. Imagine a network protocol which takes a missing heartbeat as evidence that there is a link, and a received heartbeat as evidence that there is not. If routing a packet succeeds based on such inadequate beliefs, then this is pure luck. Therefore, it is important to have normative requirements separating good and useful belief theories from those that are arbitrary and useless.

In this section, we study various normative requirements for belief theories based on the descriptive framework introduced in the previous section. Whether a belief theory leads to adequate and useful beliefs can usually not be judged in isolation but rather with respect to a specific system, an agent type, and the needs of the respective agent type.

6.2.1 Consistency Criteria

Let's start with consistency criteria. Consistency criteria check whether the beliefs together with the disbeliefs are compatible, e.g., with an OMAS. Hence, they only make sense if the belief theory is *qualitative*, that is, it allows to partition all CTLKx formulae in three sets, those formulae which are believed, those which are disbelieved, and those which are neither [11]. Let $B^+(q_B)$ and $B^-(q_B)$ denote the set of believed and disbelieved formulae in q_B respectively. For $\mathcal{B}_5$ we define those sets according to Definition 6.3 by:

$$B^+(q_B) := \{\, \varphi \in \Phi \mid \Gamma(\varphi, q_B) \in \{\uparrow, \top\}\,\}$$
$$B^-(q_B) := \{\, \varphi \in \Phi \mid \Gamma(\varphi, q_B) \in \{\downarrow, \bot\}\,\}$$

The most basic and minimal consistency criterion is that beliefs and disbeliefs are *logically consistent*, i.e., there is some OMAS $\mathcal{M}$, context $\langle \mathcal{M}, s, \sigma \rangle$, and interpretation assignment ν such that all believed formulae are satisfied, but no disbelieved formula is.

Definition 6.6 (Logical Belief Consistency)
A belief theory is *logically consistent* if and only if for all belief states $q_B \in Q_B$ there exists an OMAS $\mathcal{M}$, context $\langle \mathcal{M}, s, \sigma \rangle$, and interpretation assignment ν such that:

$$\langle \mathcal{M}, s, \sigma \rangle \vDash \left(\bigwedge_{\varphi \in B^+(q_B)} \varphi \right) \wedge \left(\bigwedge_{\varphi \in B^-(q_B)} \neg\varphi \right)$$

For instance, believing both $\varphi \wedge \psi$ and $\neg\varphi \vee \neg\psi$ as well as believing $\varphi \wedge \psi$ but disbelieving $\neg(\neg\varphi \vee \neg\psi)$ is logically inconsistent for all $\varphi, \psi \in \Phi$. In both cases we can easily prove by contradiction that the logical consistency criterion is not met.

In contrast to knowledge, beliefs can be logically inconsistent. This is due to the fact that we have chosen CTLKx formulae instead of sets of states to be the objects of beliefs. If we would have chosen non-empty sets of global states as the objects of beliefs and based on that defined whether a formula is believed analogously to knowledge as truth in all those states, then logically inconsistent beliefs would not have been possible, as it is logically impossible that logically inconsistent formulae hold in any of those states. In addition, the beliefs of agents would have been closed with respect to the logic, i.e., agents would believe all logical truths and implications of their beliefs.

Logical consistency is a very basic criterion and does not take into account what is possible with respect to a specific system model. For instance, that there is a link between nodes v and v' but no link between v' and v, is logically consistent. However, it is not possible with respect to a MANET model which guarantees bidirectional links.

Definition 6.7 (System Consistency)
A belief theory is consistent with an OMAS $\mathcal{M}$ if and only if for every belief state $q_B \in Q_B$ there is a reachable state $s \in \mathit{reach}(\mathcal{M})$ of the system which is compatible with the belief inference function, i.e., for every CTLKx formula $\varphi \in \Phi$ it holds that:

$$(\varphi \in B^+(q_B) \implies \langle \mathcal{M}, s, \sigma \rangle \vDash \varphi) \text{ and } (\varphi \in B^-(q_B) \implies \langle \mathcal{M}, s, \sigma \rangle \nvDash \varphi)$$

Intuitively Definition 6.7 states that what is believed and disbelieved in belief state q_B altogether is possible within the system, i.e., there is a reachable state such that every believed formula is satisfied in that state and every disbelieved formula is not. For instance, with respect to a MANET model which guarantees that links are bidirectional, believing $\mathit{link}(v, v')$ and disbelieving $\mathit{link}(v', v)$ is inconsistent with the system because there is no state in its model such that $\mathit{link}(v, v')$ is satisfied but $\mathit{link}(v', v)$ is not.

Belief theories which are consistent with respect to a specific system are also logically consistent as Definition 6.6 is met by the respective model. Further, they allow the definition of a modal belief operator which closes the belief theory logically and with respect to the system. We define a set of states which are compatible with a belief state by:

$$B^s(q_B) := \left\{ s \in \mathit{reach}(\mathcal{M}) \;\middle|\; \langle \mathcal{M}, s, \sigma \rangle \vDash \left(\bigwedge_{\varphi \in B^+(q_B)} \varphi \right) \wedge \left(\bigwedge_{\varphi \in B^-(q_B)} \neg\varphi \right) \right\}$$

Definition 6.7 ensures that this set is non-empty for every belief state and therefore can be used as a basis for a modal belief operator which is defined as follows:

$$\langle \mathcal{M}, s, \sigma \rangle \vDash \mathcal{B}_a\, \varphi \quad \text{iff} \quad \forall s' \in B^s(s(a).q_B) : \langle \mathcal{M}, s', \sigma \rangle \vDash \varphi$$

We may use this definition to automatically close a belief theory logically and with respect to a specific system using a model checking algorithm for the system.

Another useful consistency criterion, called *epistemic consistency*, is that the beliefs of an agent are consistent with its implicit knowledge, i.e., an agent should not believe φ if it implicitly knows $\neg\varphi$ and analogously an agent should not disbelieve φ if it implicitly knows φ. In the following, we define a slightly modified version of this intuition, which is relative to an agent type, i.e., based on agent-type relative knowledge. This is reasonable since belief theories are usually considered relative to an agent type.

Definition 6.8 (Epistemic Consistency)
A belief theory is consistent with the knowledge of agent type A in an OMAS $\mathcal{M}$ if and only if the agent-type relative knowledge in every local state $q \in A.Q$ is compatible with the belief inference function, i.e., for every CTLKx formulae $\varphi \in \Phi$ it holds that:

$$\left(\varphi \in B^+(q.q_B) \Longrightarrow \neg\varphi \notin \mathcal{K}^A_{\mathcal{M}}(q)\right) \text{ and } \left(\varphi \in B^-(q.q_B) \Longrightarrow \varphi \notin K^A_{\mathcal{M}}(q)\right)$$

Where $q.q_B$ denotes the belief state portion of local state q.

It is crucial to note that consistency criteria are safety properties, as they are trivially fulfilled by a belief inference capacity yielding ? for every formula φ.

6.2.2 Completness Criteria

One way to ensure that something is believed or disbelieved is to define completeness criteria. Completeness criteria may depend on the epistemic needs of an agent. For instance, an agent running a routing protocol needs beliefs about links.

Definition 6.9 (Completeness Criteria)
An belief inference capacity is *complete* with respect to a subset $\Psi \subseteq \Phi$ of CTLKx formulae relative to a subset $B \subseteq \mathcal{B}$ of belief degrees if and only if:

$$\forall q_B \in Q_B : \forall \varphi \in \Psi : \Gamma(\varphi, q_B) \in B$$

Completeness criteria have the disadvantage that they do not account for the temporal evolution of a system. Sometimes it takes some time until a belief or disbelief is justified, and in the meantime, the agent should be indifferent. For instance, imagine a routing protocol which discovers new routes. Until route discovery is complete, it makes neither sense to believe that a route exists nor that it does not. Instead, the agent should be indifferent until it is justified in believing either.

Furthermore, merely having some arbitrary but consistent beliefs is not useful. However, completeness criteria do not ensure that beliefs are grounded in facts. For instance, an agent can consistently believe in the existence of arbitrary links, since the existence of arbitrary links is consistent with the MANET model and further it cannot know that some of those links do not exist if they do not. Therefore, we study normative requirements that try to ground beliefs in facts about the system in the next two subsections.

6.2.3 Temporal Criteria

Temporal criteria are based on the temporal evolution of a system. They ensure that something good is eventually believed. Like completeness criteria, they depend on the epistemic needs of an agent and are not meant as universal criteria. Furthermore, temporal criteria may be applicable only to a subset of CLTKx formulae that have certain properties with respect to the system, e.g., that they hold eventually.

The first class of temporal criteria, we introduce, is suited for those formulae which hold for all paths globally when holding in some state. An example of such a formula is whether a message m has been posted to the digital message board. If a message has been posted, then it stays posted forever as messages cannot be removed.

Definition 6.10 (Weak-Eventually Globally Criteria)
We assume that φ is a formula which holds for all paths globally when holding in some state, i.e., $\mathrm{AG}\,(\varphi \rightarrow \mathrm{AG}\,\varphi)$. An agent a should not believe that φ until φ actually holds and the agent should eventually believe φ if φ holds, i.e.:

$$\mathrm{A}\,(\neg B(a, \varphi)\,W \varphi) \wedge \mathrm{AG}\,(\varphi \rightarrow \mathrm{AF}\,B(a, \varphi))$$

Where $B(a, \varphi)$ is a binary predicate taking agent names and CTLKx formulae as arguments which holds in a global state s if and only if $\varphi \in B^+(s(a).q_B)$. For convenience we allow variables to leak into the CTLKx formulae passed as arguments to B.

In case of the digital message board we can now use such a criterion to express the property that every node eventually believes that a message has been posted if and only if the respective message has actually been posted:

$$\forall x : [N].\ \forall m.\ \mathrm{A}\,(\neg B(x, \mathit{posted}(m))\,W \mathit{posted}(m)) \wedge \mathrm{AG}\,(\mathit{posted}(m) \rightarrow \mathrm{AF}\,B(x, \mathit{posted}(m)))$$

Further, this property can easily be extended to also account for topics:

$$\begin{aligned}&\forall x : [N].\ \forall t.\ \forall m.\ \mathit{interested}(x, t) \wedge \mathit{belongs}(m, t) \rightarrow \\ &\qquad \mathrm{A}\,(\neg B(x, \mathit{posted}(m))\,W \mathit{posted}(m)) \wedge \mathrm{AG}\,(\mathit{posted}(m) \rightarrow \mathrm{AF}\,B(x, \mathit{posted}(m)))\end{aligned}$$

Where $\mathit{interested}(x, t)$ checks whether agent x is interested in topic t and $\mathit{belongs}(m, t)$ checks whether message m belongs to topic t. If a message belongs to a topic of interest, then the respective node should eventually believe that the message has been posted if and only if it has actually been posted.

Although this criterion might sound useful at first glance, beliefs are not necessary for formulae which fulfill the condition $\mathrm{AG}\,(\varphi \rightarrow \mathrm{AG}\,\varphi)$. If φ is the case forever if φ is the case, then φ is in principle also implicitly knowable.

The condition $\mathrm{AG}\,(\varphi \rightarrow \mathrm{AG}\,\varphi)$ is, however, not satisfied by formulae about the network topology in a MANET because their truth value can change. Therefore, we cannot apply such criteria to our motivating example of links in a mobile ad-hoc network. However, formulae like these are the very reason why we introduced the belief-based approach in the first place. So, how can we specify correctness criteria for them?

All those network protocols which use beliefs about the topology have in common that they require some kind of stability of the network. The topology must not change too fast for the protocol to accommodate. In other words, the topology is required to stay partially stable for long enough periods such that the believed routes are actually correct and packets delivered via them reach their destination. We capture these stability requirements in the general case for belief theories with the following requirement.

Definition 6.11 (Belief Stability Requirements)
If the system is stable for a formula φ, then the qualitative beliefs about φ should eventually correctly reflect the state of the system, i.e., for all paths $\pi \in \mathit{paths}(\mathcal{M})$:

$$(\exists i : \forall i' \geq i : \langle \mathcal{M}, \pi[i'], \sigma \rangle \vDash \varphi) \implies (\exists j \geq i : \forall j' \geq j : \varphi \in B^+(\pi[j'](a).q_B))$$
$$(\exists i : \forall i' \geq i : \langle \mathcal{M}, \pi[i'], \sigma \rangle \nvDash \varphi) \implies (\exists j \geq i : \forall j' \geq j : \varphi \in B^-(\pi[j'](a).q_B))$$

It is further possible to modify Definition 6.11 to require correct beliefs after a certain, finite amount of discrete time steps. Please note that during the process of stabilization, a belief stability requirement gives us no guarantees about the beliefs at all.

We do not spell this out formally, but stability is precisely the requirement needed for MANET network protocols. If the parts of the topology necessary for routing stay stable long enough, then the information about routes encoded in the local state by means of routing tables or something similar should be correct. As a result thereof, when using those routes, the packets reach their destinations. Stability requirements ensure that beliefs eventually converge towards knowledge in the sense of true beliefs.

6.2.4 Causality Criteria

The criteria introduced so far merely provide means to analyze existing belief theories but give us no hint how to revise beliefs in response to observations. On the other hand, if we design belief theories we have something in mind. For instance, when updating a belief state in response to a received heartbeat such that it encodes the belief that there is a link, we do so for a reason and not just because it satisfies some of those criteria presented above. The question is, what is this reason and how to formalize it?

Intuitively, in case of a received heartbeat, we update the belief state accordingly because there is a causal relationship between the reception of a heartbeat and the existence of a link. Taking inspiration from David Lewis' counterfactual theory of causation [17] we capture this relation using a counterfactual conditional. The reception of a heartbeat was caused by the existence of a link because the agent would not have received the heartbeat if there would not have been the link. When updating belief states in response to the heartbeat, we also assume that the cause of the reception is still present and we are justified in doing so because we assume stability of the topology. So, stability together with the causal relationship can give us a full formal justification why a received heartbeat should indeed be evidence for the existence of the respective link.

In general, we can use the counterfactual analysis to ascribe implicit information about the evolution of the system to observations, which provides together with other properties of the system, like stability, a theory of how we should design belief theories. The ultimate goal, which is out of the scope of this thesis, is to automatically synthesize belief theories based on such properties and counterfactual relationships.

When working with a counterfactual conditional, the tense of the antecedent is essential for the information carried by the observation. For instance, the counterfactual conditional that an agent would not have observed α if φ would not have been the case, ascribes implicit knowledge about the past to α, namely, that φ was eventually the case. On the other hand, the counterfactual conditional that an agent would not have observed α if φ will not be the case, ascribes implicit knowledge about the future to α, namely, that φ will eventually be the case. Belief theories can be understood as means which take the implicit knowledge coming with observations and transform them with the help of other properties of the system, like stability, into useful assumptions, i.e., beliefs, about the present. These beliefs can then be used to decide which actions to perform.

Recap, the reason why we used synchronized actions instead of joint actions for our formalism was that they allow a clear separation of individual communication acts (see Section 3.1.1) which enables this analysis as well as the concept of evidence.

This subsection aimed to give us an intuition for the design of belief theories useful in the remainder of this thesis. We leave out the formal details of causality and counterfactual conditionals and instead turn directly to belief-based programs.

6.3 Belief-Based Programs

The fundamental idea of belief-based programming is to utilize beliefs to decide which actions to perform. Thus, the basis of every belief-based program is a belief theory which should produce adequate and useful beliefs needed by the respective program. Similar to knowledge tests we define belief tests which are used by belief-based programs.

Definition 6.12 (Belief Tests)
Given a belief domain $\mathcal{B}$ we define *belief tests* by means of the following abstract grammar where $b \in \mathcal{B}$ is a belief degree and $\varphi \in \Phi$ is a CTLKx formula:

$$\beta ::= \neg\beta \mid \beta_1 \wedge \beta_2 \mid \mathfrak{b}_1 \equiv \mathfrak{b}_2 \mid \mathfrak{b}_1 \prec \mathfrak{b}_2 \qquad \textit{(belief test)}$$
$$\mathfrak{b} ::= b \mid B\,\varphi \qquad \textit{(belief term)}$$

Given a belief theory $\langle Q_B, \eta_B, \Gamma, \mathcal{B}\rangle$ and a belief state $q_B \in Q_B$ we evaluate belief terms $\mathfrak{b}$ to belief degrees $b \in \mathcal{B}$ using the following evaluation function:

$$\begin{aligned} \mathit{eval}_B(b, \Gamma, q_B) &:= b \\ \mathit{eval}_B(B\,\varphi, \Gamma, q_B) &:= \Gamma(\varphi, q_B) \end{aligned}$$

Similar to knowledge tests we define truth-value semantics for belief tests:

$$[\![\mathfrak{b}_1 \equiv \mathfrak{b}_2]\!]_B(\Gamma, q_B) := \begin{cases} \top & \text{iff } \mathit{eval}_B(\mathfrak{b}_1, \Gamma, q_B) = \mathit{eval}_B(\mathfrak{b}_2, \Gamma, q_B) \\ \bot & \text{otherwise} \end{cases}$$
$$[\![\mathfrak{b}_1 \prec \mathfrak{b}_2]\!]_B(\Gamma, q_B) := \begin{cases} \top & \text{iff } \mathit{eval}_B(\mathfrak{b}_1, \Gamma, q_B) \prec \mathit{eval}_B(\mathfrak{b}_2, \Gamma, q_B) \\ \bot & \text{otherwise} \end{cases}$$

The truth-value semantics for $\neg\beta$ and $\beta_1 \wedge \beta_2$ are defined analogously to knowledge tests and the other boolean connectives are defined as usual. In addition we define a notion of *greater-equal* on belief terms as syntactic sugar by:

$$\mathfrak{b}_1 \preccurlyeq \mathfrak{b}_2 := \mathfrak{b}_1 \equiv \mathfrak{b}_2 \vee \mathfrak{b}_1 \prec \mathfrak{b}_2$$

For $\mathcal{B}_5$-belief theories we further define the following syntactic sugar for belief tests:

$$\begin{aligned} B\,\varphi &:= B\,\varphi \succcurlyeq \uparrow \\ K\varphi &:= B\,\varphi \equiv \top \end{aligned}$$

Please note that the syntactic sugar for $\mathcal{B}_5$-belief theories is unambiguous as the belief term $B\,\varphi$ must occur as an operand of $\equiv$ or $\prec$ but the belief test $B\,\varphi$ must not.

Let T_B denote the set of all belief tests.

In general there are multiple ways to use belief theories and tests for programming. A belief theory can be used as part of the local states of an agent type. That is, the agent saves the belief state as part of the local state and updates it using the belief effect function with every observation. However, this notion of belief-based programming has a few disadvantages because it directly couples the belief theory with an agent type and a program. As a result, whether a belief theory is adequate and fulfills certain normative requirements with respect to the system or the agent type depends on the behavior of the program. Also, there is no way of enforcing certain actions, e.g., a heartbeat, which ensure correctness of beliefs. Further, the direct coupling potentially leads to infinite regress problems similar to those we saw for knowledge-based programs as the beliefs of an adequate theory depend on the behavior of the program which depends on those beliefs.

To prevent those problems right from the beginning, we use a different approach. Instead of directly coupling a belief theory with an agent type and a program, we explicate the nomological framework with belief theories. That is, we use a nomological agent type and interpret its nomological states with a belief theory. We then use this belief theory and the respective nomological agent type to derive agent types for the actual system analogously to Definition 5.2 using a belief-based program.

Definition 6.13 (Belief-Based Programs)
Since the nomological states are belief states of the respective belief theory, belief-based programs are an explication of the nomological framework. Let $P_B \subseteq T_L \times T_B \times [Act]$ be a *belief-based program* inducing the following protocol function:

$$\rho_{P_B}(\langle q_B, q\rangle) := \{\, \alpha \mid \langle t_l, \beta, \alpha\rangle \in P_N : \mathit{decide}_L(t_l, \langle q_B, q\rangle) = \top \text{ and } [\![\beta]\!]_B(\Gamma, q_B) = \top \,\}$$

Belief-based programs are implemented analogously to nomological knowledge programs (see Definition 5.2) by constructing a new agent type using the nomological agent type and a program, and then replace the original agent type with the new one.

Decoupling the actual program and the belief theory allows us to analyze and prove adequacy criteria for the respective belief theory with respect to the nomological system and agent type, independent of the behavior of the program. With enforcement protocols, we are also able to construct nomological agent types and corresponding belief theories which satisfy normative requirements independent of the behavior of the program. With the larger goal of a general belief-centric middleware in mind, this is essential since it gives every belief-based program well-defined semantics by yielding guarantees about the adequacy of beliefs which hold no matter what the application does.

Cookie Example Revisited

Before we present a full example using the introduced approach for a MANET routing protocol inspired by the digital message board example, we give a brief intuition for belief-based programs using the cookie example. We capture the original cookie example by means of the belief-based program Code 6.14.

Code 6.14 (Cookie-Eating Belief-Based Program)

```
1: if $K(\mathit{desire}(\mathit{eat\text{-}cookie})) \wedge B(\mathit{cookie}())$ then { $\mathit{open}!(1), \mathit{open}!(0)$ }
2: if $K(\mathit{desire}(\mathit{eat\text{-}cookie}) \wedge \mathit{cookie}())$ then { $\mathit{eat}!()$ }
3: if $K(\neg\mathit{desire}(\mathit{eat\text{-}cookie})) \vee \neg B(\mathit{cookie}())$ then { $\mathit{idle}()$ }
```

Intuitively Code 6.14 encodes the following behavior. Whenever the agent is certain that it has the desire to eat cookies and it believes that there is a cookie in the drawer, then it opens the drawer. As a result, it learns whether there is a cookie in the drawer. If the agent is now certain that it has the desire to eat and that there is a cookie in the drawer, then it eats the cookie. If the agent is certain that it has no desire to eat a cookie or does not believe that there is a cookie in the drawer it just does nothing.

6.4 Interest-Based Routing

So, how to apply belief-based programming to real problems from the area of distributed systems? In this section, we show the usefulness of our approach by building a routing protocol for MANETs inspired by the example of the digital message board. To this end, we proceed as follows: We first describe the protocol informally to get an intuition of how it works and what abilities we need the nomological system to provide. Based on these considerations we then design a nomological system and a belief theory which we use as the basis for the belief-based program implementing the protocol. Along the way, we informally argue for certain correctness properties of the nomological system, the belief theory, and the protocol as well as its practical feasibility.

Informal Description

The network protocol we implement provides unreliable, many-to-many, interest-based routing. That is, nodes can announce their interest in certain topics, and then messages belonging to a specific topic are routed to every node interested in the respective topic. The protocol is unreliable in a sense that it does not ensure that every node gets all messages belonging to its topics of interest in the face of network reconfiguration.

The routing protocol is based on *zones*. A *k-zone* around a node v is a subgraph of the topology containing all nodes reachable from v with at most k hops. We assume that all nodes interested in a specific topic are connected via their zones. Imagine there are three nodes v_1, v_2, and v_3, all interested in topic t. In this case, each node needs to have at least one other node within its zone. For instance, while it is valid that the nodes v_1 and v_2 are in each others zone, and node v_2 and v_3 are in each others zone, but not v_1 and v_3 are not in each others zone, it is not valid, that v_1 and v_2 are in each others zone, but neither of them is in the zone of v_3 essentially isolating v_3. Please note that in general it is not sufficient that every node has at least one other node within its zone. For instance, imagine that there is another node v_4, then it is not valid, that v_1 and v_2 are in each others zone, and v_3 and v_4 are in each others zone, but neither v_1 nor v_2 is in the zone of v_3 or v_4 because all interested nodes need not be connected by their zones.

This assumption is reasonable, especially in case of the digital message board example, since topics are usually either local, i.e., related to physically located circumstances like a lecture, or global. In the local case, the nodes interested in the topic are near each other, and in the global case, the nodes interested are most likely spread evenly across the topology since the global topics are not related to physical circumstances.

The idea of the network protocol is to pass messages from zone to zone. That is, if a node gets a new message it looks which nodes in its own zone are interested in that message. It then computes an efficient tree to send the message to all those nodes and sends the message along the tree. On reception, the other nodes repeat the same process. As a result, the message diffuses through the network to all interested nodes.

So, a node needs adequate beliefs about its zone. It needs information about the topology of its zone to compute the routing trees for messages and it needs information about which of the nodes in its zone are interested in which topics.

6.4.1 Nomological Model

Belief-based programs are based on the nomological framework (see Definition 6.13). So, the first step towards a belief-based program implementing the protocol is to design an appropriate nomological model including an environment agent and a nomological agent type for nodes with sufficient abilities together with a belief theory such that we can implement the protocol. In particular, this means that the nomological model needs to ensure that the agents develop adequate beliefs about their zone.

Environment Agent

We begin with the environment agent. Instead of imposing fairness constraints on the system, we build a fair scheduler right into the environment agent. The scheduler also allows for a global, discrete notion of time enabling timeouts and message delays. We use timeouts to distribute routing information periodically. The environment implements a discrete event simulator [25, p. 2 ff.]. We queue discrete events, called *commands*, ordered according to their execution time, using a discrete timestamp, in a priority queue.

Code 6.15 (Environment Agent)

```
 1: while $queue \neq \emptyset$ do
 2:     $cmd, time := queue.pop()$
 3:     switch $cmd$ do
 4:         case $send?(src, dst, msg)$
 5:             $queue.schedule(time, deliver(src, dst, msg))$
 6:         case $broadcast?(src, msg)$
 7:             for all $dst \in \{ v' \in G.V \mid \langle src, v' \rangle \in G.E \}$ do
 8:                 $queue.schedule(time, deliver(src, dst, msg))$
 9:         case $notify(node, handle)$
10:             await $notify!(node, handle)$
11:             ACCEPT_COMMANDS($node$)
12:         case $deliver(src, dst, msg)$
13:             if $\langle src, dst \rangle \in G.E$ then
14:                 await $deliver!(src, dst, msg)$
15:                 ACCEPT_COMMANDS($dst$)
16:     MODIFY_NETWORK()
```

In each iteration of the main event loop, we first pop the next command together with its timestamp from the queue (line 2) and subsequently execute the respective command. If the command tells us to send a packet (line 4), then we schedule a *delivery* command to be executed immediately (line 5). If the command tells us to broadcast a packet (line 6), then we schedule a *delivery* command to be executed immediately (line 8) for each node within the range of the source node (line 7). If the command tells us to notify a node that a timeout expired (line 9), then we notify the node by means of an active *notify* action (line 10). Afterwards, we accept new commands from that node (line 11). If the command tells us to deliver a packet (line 12) and there is a link between the source and the destination node (line 13), then we deliver the packet by means of an active *delivery* action (line 14). Afterwards, we accept new commands from the destination node (line 15). After the execution of a command we allow the network to reconfigure (line 16).

The procedure MODIFY_NETWORK modifies the topology graph of the network in some unspecified way. Part of this procedure is to initialize new nodes, destroy existing ones, and move them around in the network. Since the exact details are not essential to the routing protocol, we leave them out.

The procedure ACCEPT_COMMANDS (see Code 6.16) accepts new commands to be scheduled in the command queue of the environment after the delivery of a packet or the expiration of a timeout. To this end, we introduce a *done*() action which tells the environment that the respective agent has no more commands to schedule and is ready to pass the control back to the scheduler. The procedure ACCEPT_COMMANDS accepts *send* and *broadcast* actions (line 4) and schedules them for execution with a message delay ΔT_t (line 7). It further accepts *timeout* actions (line 4) which register a timeout at the scheduler with a custom delay *delta* and some handle *handle* (line 9) which is passed back to the agent later (see Code 6.15) and is used to identify the timeout.

Code 6.16 (Accept Commands from Nodes)

```
1: procedure ACCEPT_COMMANDS($node$)
2:     $action :=$ null
3:     while $action \neq done?()$ do
4:         $action :=$ await { $send?(\ldots), broadcast?(\ldots), timeout?(\ldots), done?()$ }
5:         switch $action$ do
6:             case $send?(\ldots)$ or $broadcast?(\ldots)$
7:                 $queue.schedule(time + \Delta T_t, action)$
8:             case $timeout?(handle, delta)$
9:                 $queue.schedule(time + delta, notify(node, handle))$
```

Theorem 6.17 (Discrete Time Progression)
We require that the packet delay ΔT_t as well as timeout delays *delta* are greater than zero and that an agent only schedules a finite number of commands in each round. As a result, the discrete time used by the scheduler (*time* in Code 6.15) increases monotonically after a finite number of discrete time steps of the system.

Node Agents

For both active actions performed by the environment agent, i.e., *deliver*! and *notify*!, we define a handler Code 6.18 and Code 6.19 respectively. We use guards to ensure that the respective handlers only accept *delivery* and *notify* actions intended for the respective node. Please note that we assume here that each node has a unique address *self.id*. A guard directly translates to the protocol of the nomological agent type and thereby enforces that the belief-based program can, e.g., only accept packets intended for the respective node. We further need to enforce some actions allowing a nomological agent type to aggregate adequate beliefs about its zone. To this end, each node floods its zone with a packet containing information about its neighbors and its interests.

Code 6.18 (Packet Delivery Handler)

```
Guard: $dst = self.id$
procedure $deliver?(src, dst, msg)$
    switch $msg$ do
        case $hello(seq, node, ttl, neighbors, interests)$
            if $seq \geq zone[node].seq$ then
                if $seq > zone[node].seq$ then
                    $zone[node].ttl := 0$
                    $zone[node].neighbors := neighbors$
                    $zone[node].interests := interests$
                    if $node = src$ then
                        $self.neighbors.add(node)$
                    else
                        $self.neighbors.remove(node)$
                    await $timeout!(\langle node, seq \rangle, \Delta T_d)$
                if $ttl > zone[node].ttl$ then
                    $zone[node].ttl = ttl$
                    await $broadcast!(hello(seq, node, ttl - 1, neighbors, interests))$
        case $message(\ldots)$
            HANDLE_MSG$(message(\ldots))$
    await $done!()$
```

The procedure *deliver*? (see Code 6.18) handles a *deliver* action performed by the environment. We allow two sorts of messages, messages processed by the belief-based program, denoted by *message*(...), and *hello* messages which are nomologically enforced. Each *hello* message comprises a sequence number *seq*, a node id *node*, a time to live *ttl*, a set of node ids *neighbors*, and a set of topics *interests*. Sequence numbers are required to be unique for each message and monotonically increasing with respect to the node with id *node*. The idea of *hello* messages is that each node periodically floods its zone with them and thereby distributes information about its own neighbors and its interests within its zone. If a node receives a *hello* message (line 3), it first checks whether it has already seen a newer *hello* message from the respective node (line 4). If this is the case, then it ignores the message altogether. If it has not seen a newer message, it checks whether it has already seen the respective message (line 5). If the message is completely new, then it stores the information carried by the message about the node *node* in its local state (lines 6-8). If the respective node is a direct neighbor (line 9) then it adds the node to its own neighbors (line 10), if not it removes it from the set of its neighbors[1] (line 12). We further register a timeout (line 13) which removes the node from the zone and the neighbor set (see Code 6.19) after a certain amount of time if no new *hello* message is received. If the time to live of the message is greater than the previously seen time to live (line 14), then the node rebroadcasts the message with a decreased time to live (line 16). The time to live is decreased to ensure that a *hello* message is only distributed within a zone of a node.

Code 6.19 (Timeout Event Handler)

```
Guard: $node = self.id$
 1: procedure $notify?(node, handle)$
 2:     switch $handle$ do
 3:         case $\langle node, seq \rangle$
 4:             if $zone[node].seq = seq$ then
 5:                 delete $zone[node]$
 6:                 $self.neighbors.remove(node)$
 7:         case $TICK$
 8:             $self.seq = self.seq + 1$
 9:             await $timeout!(TICK, \Delta T_h)$
10:             await $broadcast!(hello(self.seq, self.id, k, self.neighbors, self.interests))$
11:     await $done!()$
```

When a node is notified about a timeout the procedure *notify?* handles the timeout (see Code 6.19). Depending on the handle associated with the timeout the node either removes a node from its zone (lines 3-6) or broadcasts a *hello* message containing its neighbors and interests (line 10) and subsequently reschedules the *TICK* timeout (line 9). The *TICK*

[1] Please note that we assume here that removing an element from a set which is not an element of the set succeeds without an error and leaves the set untouched.

timeout ensures with the rebroadcasting code of Code 6.18 that each node floods its zone periodically with information about its direct neighbors and its own interests.

So, the nomological agent type of nodes does the following. It floods its own zone periodically with information about its neighbors and its interests. Since zone membership is reflexive, it also receives such information about each node in its zone. It stores this information and deletes it after a certain amount ΔT_d of discrete time if it does not hear again from the respective node. Intuitively, the information stored in the local state variable *zone* constitute beliefs about the zone which we need to interpret with a belief theory.

6.4.2 Belief Theory

Based on the nomological states of nodes we define a belief theory interpreting those states. This belief theory is then used as a basis for the belief-based program implementing the network protocol. Based on the stored information about the direct neighbors of each node within the zone, we define the belief inference capacity for the *link* predicate by:

$$\Gamma(\mathit{link}(v, v'), q_B) := \begin{cases} \uparrow & \text{iff } v \in \mathit{zone}[v'].\mathit{neighbors} \text{ or } v' \in \mathit{zone}[v].\mathit{neighbors} \\ \downarrow & \text{otherwise} \end{cases}$$

While this definition is *nomologically consistent*, i.e., consistent with the nomological system, the following alternative definition would not be nomologically consistent:

$$\Gamma'(\mathit{link}(v, v'), q_B) := \begin{cases} \uparrow & \text{iff } v \in \mathit{zone}[v'].\mathit{neighbors} \\ \downarrow & \text{otherwise} \end{cases}$$

According to our MANET model links are bidirectional (see Section 2.3) and the nomological network protocol does not ensure that the information stored in the neighborhood sets are consistent. Therefore, it is possible that $v \in \mathit{zone}[v'].\mathit{neighbors}$ but not $v' \in \mathit{zone}[v].\mathit{neighbors}$. In this case, the former definition nevertheless provides beliefs which are consistent with the system while the latter definition does not.

We further define the belief inference capacity for the *interest* predicate based on the stored information about the interests of nodes within the zone:

$$\Gamma(\mathit{interested}(v, \mathit{topic}), q_B) := \begin{cases} \uparrow & \text{iff } \mathit{topic} \in \mathit{zone}[v].\mathit{interests} \\ \downarrow & \text{otherwise} \end{cases}$$

For complex formulae the belief inference function for $\varphi \wedge \psi$ is recursively defined by taking the minimum of the belief degrees assigned to φ and ψ and the belief inference function for $\neg\varphi$ is recursively defined by swapping belief and disbelief degrees for φ. Further, every other predicate and formulae with temporal or epistemic operators are assigned the indifferent belief degree.

6.4.3 Belief-Based Protocol

Based on the nomological system and the belief theory defined above we are now able to write belief-based programs that use beliefs about the zone of a node. To construct the routing tree, we use the belief-based procedure Code 6.20. We first query which nodes are interested in a certain topic (line 2) and subsequently what links exist within the zone (line 3). Then we use the links to construct a tree to those nodes (line 4).

Code 6.20 (Belief-Based Routing-Tree Construction)

```
1: procedure CONSTRUCT_TREE($topic$)
2:     $nodes := \{ v \in U \mid B\ interested(v, topic) \}$
3:     $E := \{ \langle v, v' \rangle \in U^2 \mid B\ link(v, v') \}$
4:     return ROUTING_TREE($nodes$, $E$)
```

Code 6.21 (Message Handler)

```
procedure HANDLE_MSG($message(id, tree, topic, content)$)
    if $id \notin seen\_ids$ then
        $seen\_ids.add(id)$
        if $topic \in self.interests$ then
            $new\_tree :=$ CONSTRUCT_TREE($topic$)
            await $broadcast!(message(id, new\_tree, topic, content)$
        if SUCCSSORS($self.id$, $tree$) $\neq \emptyset$ then
            await $broadcast!(message(id, tree, topic, content))$
```

Whenever a node receives a packet which is supposed to be handled by the belief-based program, the procedure HANDLE_MSG (see Code 6.21) is executed. Please note that this does not match the exact formal definition of belief-based programs where the execution of the belief-based program and the nomological system is done in parallel (see Definition 6.13). So strictly speaking a message is delivered to the nomological part and the belief-based part but each part reacts solely to those messages that are intended to be processed by the respective part. The procedure HANDLE_MSG first checks whether the message has already been seen (line 2). If this is not the case, then it adds its id to the set of seen message ids (line 3). We assume that every message has a unique global id. According to the protocol, a node needs to rebroadcast a message in its own zone if the node is interested in the respective topic itself. If the node is interested in the topic (line 4), it constructs a new routing tree using its beliefs about its zone (line 5) and then rebroadcasts the along this tree (line 6). If the node is part of the routing tree of the original message and has successors in this tree (line 7) it, in addition, needs to rebroadcast the message along the existing tree (line 8). Since we store the ids of seen messages, we only deal once with each message.

6.5 Belief-Centric Middleware

As we saw in the last section, belief-based programs can be used to write network protocols for MANETs. However, this approach required the manual design of a nomological system and a corresponding belief theory. Our initial goal was to derive a general belief-centric MANET middleware. In this section, we briefly sketch how a belief-centric middleware could look like. Building an actual belief-centric middleware is out of the scope of this thesis.

The key idea of a belief-centric middleware is to provide basic building blocks for building nomological open multi-agent systems and restrict their composition in such a way that belief theories can be synthesized based on the needs of belief-based programs and the properties of the system such that certain adequacy criteria are met.

A belief-centric middleware naturally addresses the challenges we presented in Section 2.1. The approach is inherently decentralized and able to deal with the dynamic nature of MANETs. In addition, we can use belief degrees to account for the lack of a central authority and trustworthiness of nodes. For instance, we could assign trust degrees to observations which influence how belief degrees are updated, i.e., a less trustworthy observation results in more uncertainty of the revised beliefs.

Chapter 7

Conclusion

We began this thesis with the idea to use beliefs to generalize the concept of knowledge-based programming and thereby make it useful for highly dynamic systems like mobile ad-hoc networks. We introduced a formalism for open multi-agent system which models communication between agents using synchronized actions. Based on this formalism we adapted knowledge-based programs to open multi-agent systems. We saw that the traditional notion of agent-relative knowledge-based programs is not suited for open multi-agent systems. We, therefore, introduced agent-type relative knowledge and built knowledge-based programs for open multi-agent systems based on it. To tackle the infinite regress problem arising from the definition of knowledge-based programs and making them unsuitable as a basis of a general MANET middleware we introduced the nomological framework which separates the system from which the knowledge is derived from the system in which the actions are performed. Finally, we introduced a formal model of beliefs which enables us to capture the informal notion of beliefs already used in existing MANET protocols. We studied based on this framework various correctness criteria for beliefs and, among other things, saw why existing MANET protocols are justified in taking specific observations as evidence for existing routes. We then used the belief framework to explicate the nomological framework and introduced belief-based programs which provide well-defined semantics independent of the program making them suitable as a basis of a general MANET middleware. We showed the usefulness of our approach by means of an example network protocol and sketched a general belief centric middleware their details we left open for future work.

Appendix A

Notation

Notation A.1 (Disjoint Set Union)
Given two disjoint sets A and B, $\uplus$ denotes the disjoint union:

$$A \uplus B := A \cup B$$

Notation A.2 (Named Tuple Components)
To refer to individual components of a tuple, we use the name introduced by the definition, i.e., let $t := \langle a, \ldots, z \rangle$ be a tuple with components a to z, then we refer to component a by "$t.a$". Furthermore, we omit the index when we refer to components of a tuple introduced with an index, i.e., let $t' := \langle a_1, \ldots, z_1 \rangle$ be a tuple with components a_1 to z_1, then we refer to component a_1 by "$t'.a$".

Notation A.3 (Variant Functions)
Given a function $f\colon A \to B$ then $f[a \mapsto b]$ for $a \in A$ and $b \in B$ is an a-variant of f defined by:

$$f[a \mapsto b](x) := \begin{cases} b & \text{iff } x = a \\ f(x) & \text{otherwise} \end{cases}$$

Notation A.4 (Partial Functions)
Let $\oslash$ denote an unique value representing *undefined*. To describe partial functions $f\colon A \to B$ as total functions, we use the notation $f\colon A \to B_{\oslash}$ which is equivalent to $f\colon A \to (B \mathbin{\dot\cup} \{\,\oslash\,\})$:

$$f\colon A \to B_{\oslash} \equiv f\colon A \to (B \mathbin{\dot\cup} \{\,\oslash\,\})$$

$$Dom(f) := \{\, a \in A \mid f(a) \neq \oslash \,\}$$

Notation A.5 (Power Set)
Let A denote a set. The *power set* of A denoted by 2^A is defined by:

$$2^A := \{\, B \mid B \subseteq A \,\}$$

Appendix B

Code

B.1 Linux IEEE 802.11 Link Layer Primitives

The following code provides the MANET communication primitives on top of IEEE 802.11 IBSS on Linux using Unix RAW sockets. It specifies its own ethernet protocol to distinguish between different protocols running on top of the data link layer.

```python
#!/usr/bin/env python3
# -*- coding:utf-8 -*-

import socket
import struct

from subprocess import check_call


WIFI_SSID = 'BeliefNetwork'  # ad-hoc network SSID
WIFI_FREQUENCY = 2417  # ad-hoc network frequency in MHz

ETHERNET_PROTOCOL = 0x0700  # non-standard ethernet protocol number
BROADCAST_ADDRESS = b'\xff' * 6  # ethernet broadcast address

_ethernet_frame = struct.Struct('! 6s 6s H')  # ethernet frame header format


def _make_frame(source, destination, payload):
    """ Make an ethernet frame. """
    header = _ethernet_frame.pack(destination, source, ETHERNET_PROTOCOL)
    return header + payload


def format_mac(mac_address):
    """ Returns the colon separated hex representation. """
    return ':'.join(f'{byte:02x}' for byte in mac_address)


def configure(interface):
    """ Configure the WLAN interface appropriately. """
    # disable network-manager for all WIFI interfaces
    check_call(['nmcli', 'r', 'wifi', 'off'])
    # unblock the WLAN interface
    check_call(['rfkill', 'unblock', 'wlan'])
    # shut network link down
    check_call(['ip', 'link', 'set', interface, 'down'])
    # reconfigure the WLAN interface to IBSS
    check_call(['iw', interface, 'set', 'type', 'ibss'])
    # set up the network link
    check_call(['ip', 'link', 'set', interface, 'up'])
    # join the IBSS network
    check_call(['iw', interface, 'ibss', 'join', WIFI_SSID, str(WIFI_FREQUENCY)])


class DataLink:
    """ Interface to the 802.11 data link layer. """

    def __init__(self, interface):
        self.interface = interface
        protocol = socket.ntohs(ETHERNET_PROTOCOL)
        self.socket = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, protocol)
        self.socket.bind((interface, 0))
        self.address = self.socket.getsockname()[4]

    def broadcast(self, payload):
        """ Broadcast to all nodes within range. """
        frame = _make_frame(self.address, BROADCAST_ADDRESS, payload)
        self.socket.send(frame)

    def send(self, address, payload):
        """ Send a packet to a specific node in range. """
        frame = _make_frame(self.address, address, payload)
        self.socket.send(frame)

    def recv(self):
        """ Receive data sent by other nodes in range. """
        data, address = self.socket.recvfrom(4096)
        header = data[:_ethernet_frame.size]
        payload = data[_ethernet_frame.size:]
        destination, source, protocol = _ethernet_frame.unpack(header)
        return destination, source, payload
```

List of Definitions

Remarks

The used icons originate from OpenClipart and are released into public domain.

Base Station `https://openclipart.org/detail/171413/wireless-router`

Wireless Host `https://openclipart.org/detail/171417/laptop`

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 978-0-262-02649-9.

[2] Francesco Belardinelli, Davide Grossi, and Alessio Lomuscio. "Finite Abstractions for the Verification of Epistemic Properties in Open Multi-agent Systems." In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 854–860. ISBN: 978-1-57735-738-4.

[3] Philip A. Bernstein. "Middleware: A Model for Distributed System Services." In: *Commun. ACM* 39.2 (Feb. 1996), pp. 86–98. ISSN: 0001-0782.

[4] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic." In: *Logic of Programs, Workshop*. London, UK, UK: Springer-Verlag, 1982, pp. 52–71. ISBN: 3-540-11212-X.

[5] T. Clausen and P. Jacquet. *Optimized Link State Routing Protocol (OLSR)*. RFC 3626. RFC Editor, Oct. 2003.

[6] Marco Conti et al. "Cross-Layering in Mobile Ad Hoc Network Design." In: *Computer* 37.2 (Feb. 2004), pp. 48–51. ISSN: 0018-9162.

[7] M. Conti et al. "Cross-layering in mobile ad hoc network design." In: *Computer* 37.2 (Feb. 2004), pp. 48–51. ISSN: 0018-9162.

[8] Ronald Fagin et al. "Knowledge-based programs." In: *Distributed Computing* 10.4 (July 1997), pp. 199–225. ISSN: 1432-0452.

[9] Ronald Fagin et al. *Reasoning About Knowledge*. Cambridge, MA, USA: MIT Press, 1995. ISBN: 0262562006.

[10] Tamar Szabó Gendler and John Hawthorne. *Conceivability and Possibility*. Oxford University Press, 2002.

[11] Franz Huber. “Formal Representations of Belief.” In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Spring 2016. Metaphysics Research Lab, Stanford University, 2016.

[12] “IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.” In: *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)* (Dec. 2016), pp. 1–3534.

[13] “IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture.” In: *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)* (June 2014), pp. 1–74.

[14] Alan Kaminsky and Hans-Peter Bischof. “Many-to-Many Invocation: A New Object Oriented Paradigm for Ad Hoc Collaborative Systems.” In: *Companion of the 17th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’02. Seattle, Washington: ACM, 2002, pp. 72–73. ISBN: 1-58113-626-9.

[15] Reino Kurki-Suonio. “Towards Programming with Knowledge Expressions.” In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’86. St. Petersburg Beach, Florida: ACM, 1986, pp. 140–149.

[16] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (6th Edition)*. 6th. Pearson, 2012. ISBN: 0132856204, 9780132856201.

[17] David Lewis. “Causation.” In: *Journal of Philosophy* 70.17 (1973), pp. 556–567.

[18] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. “MCMAS: an open-source model checker for the verification of multi-agent systems.” In: *International Journal on Software Tools for Technology Transfer* 19.1 (2017), pp. 9–30. ISSN: 1433-2787.

[19] Ron van der Meyden. “Finite state implementations of knowledge-based programs.” In: *Foundations of Software Technology and Theoretical Computer Science: 16th Conference Hyderabad, India, December 18–20, 1996 Proceedings*. Ed. by V. Chandru and V. Vinay. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 262–273. ISBN: 978-3-540-49631-1.

[20] Ron van der Meyden. “Knowledge Based Programs: On the Complexity of Perfect Recall in Finite Environments.” In: *Proceedings of the 6th Conference on Theoretical Aspects of Rationality and Knowledge*. TARK ’96. The Netherlands: Morgan Kaufmann Publishers Inc., 1996, pp. 31–49. ISBN: 1-55860-417-9.

[21] R. Milner. *A Calculus of Communicating Systems.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982. ISBN: 0387102353.

[22] Bruce Jay Nelson. “Remote Procedure Call.” AAI8204168. PhD thesis. Pittsburgh, PA, USA, 1981.

[23] C. Perkins, E. Belding-Royer, and S. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing.* RFC 3561. RFC Editor, July 2003.

[24] Charles E. Perkins, ed. *Ad Hoc Networking.* Addison-Wesley Professional, 2001.

[25] George F. Riley and Thomas R. Henderson. *Modeling and Tools for Network Simulation.* Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-12331-3.

[26] Yoav Shoham. “Agent-oriented Programming.” In: *Artif. Intell.* 60.1 (Mar. 1993), pp. 51–92. ISSN: 0004-3702.

[27] Eduardo da Silva and Luiz Carlos P Albini. “Middleware Proposals for Mobile Ad Hoc Networks.” In: *Journal of Network and Computer Applications* 43 (2014), pp. 103–120.

[28] “Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7.” In: *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)* (Sept. 2016), pp. 1–3957.

[29] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition).* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[30] Moshe Y. Vardi. “Implementing Knowledge-Based Programs.” In: *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 15–30. ISBN: 1-55860-417-0.

[31] R. Jay Wallace. “Practical Reason.” In: *The Stanford Encyclopedia of Philosophy.* Ed. by Edward N. Zalta. Summer 2014. Metaphysics Research Lab, Stanford University, 2014.

[32] Sanghyun Yoo, Jin Hyun Son, and Myoung Ho Kim. “A Scalable Publish/Subscribe System for Large Mobile Ad Hoc Networks.” In: *J. Syst. Softw.* 82.7 (July 2009), pp. 1152–1162. ISSN: 0164-1212.