Learning agent organisations: studies on collaborative modelling, performance management and learning capacities of networks of collaborative agents
Mulder, W.

Citation for published version (APA):
2 THEORETICAL BACKGROUND

This chapter summarizes a few theoretical concepts necessary to understand the context and mind-set behind the subject of the dissertation. The sections can be read independently of each other. Section 2.1 discusses some background of agent based systems and explains the various contexts in which agents are used within the scope of this dissertation. Section 2.2 is about learning in distributed environments. Section 2.3 explains the basics of grammar induction which is narrowed to inferring deterministic finite automata in section 2.4. Section 2.5 briefly addresses the notion of learning by compression. Section 2.6 elaborates on learning in dynamic environments and explains the concept of adaptive modelling. Section 2.7 completes the chapter with a brief section about self-organisation.

2.1 Agent based systems

In this dissertation the notion of agents is used in three different, but related, contexts:

- Modelling networked systems,
- Developing software systems using distributed software techniques,
- Distributed problem solving and design of collaborative learning mechanism.

2.1.1 Modelling

The concept of agents allows for abstractions and lets one describe systems in terms of how they interact and work. This can be done from a practical (Hillegersberg [68]) or an information theoretical (Crutchfield [32]) perspective.

Although there are commonalities in the definitions used, the focus of interest is different when agents are regarded from theoretical, mechanical, or technical points of view. In general autonomy is the most widely accepted characteristic. An agent is a system that is capable of autonomous action in its environment in order to meet its design objectives. (see definition in chapter 1)
In the field of robotics and artificial intelligence, agents are characterized as autonomous entities that have sensors and actuators (Russel and Norvig [113]). Examples are humans (having eyes as sensors, hands as actuators), robots (having cameras as sensors, wheels as actuators) but also business organisations (having humans both as sensors and actuators) and software agents (having input-interfaces as sensors and method calls on other agent as actuators). When used in practical applications, for example in the fields of logistics and planning, agents are often regarded from the perspective of communication and collaboration (Fox et. al [50], Rodriques et. al [111], Marik and McFarlane [91]).

Agent based models (Wooldridge [132]) are used to model complex real-world networked systems, making it easier to understand and improve those systems.

**Definition:** An *agent-based model* is a computational model of actions and interactions of autonomous entities created to assess their effects on the system as a whole.

Agent-based models can be used to simulate simultaneous operations in an attempt to re-create and predict the appearance of complex phenomena. In the 1990s, distributed computing technologies gave significant momentum to the use of agent-based models (Axelrod [12], Bonabeau [20], Shoham & Leyton-Brown [120]). Agent based models are commonly used for modelling decentralized systems in which the level and kind of interaction between the system elements varies from informative (Michaud et al. [94]), cooperative (Rodriques et. al [111]) to collaborative\(^4\) (Fox et. al [50]) kinds.

A related field is known as the area of complex adaptive systems (CAS). CAS are networks of entities (cells, populations of species, individuals, firms, nations) that interact with each other in a complex and continuous way (Dooley [39]). The control of a CAS tends to be highly dispersed and decentralized. Coherent behaviour in the system often arises from competition, cooperation or collaboration among the agents. The difference in the concepts of Multi Agent Systems (MAS), CAS and our notion of collaborative agent organisations is small: 1) a CAS does not necessarily need to consist of agents and 2) one can argue that a collaborative agent organisation can be considered as a MAS with a focus on the collaboration capabilities.

\(^4\) There is a difference between cooperation and collaboration. Cooperation is about individual contributions to a common goal, communicating orders and results. Collaborating is a more tightly way of cooperating; agents that collaborate, share information during the execution of their tasks in order to better align their processes with each other.
2.1.2 Engineering

In our work we use a technical agent framework as a basis for the design and implementation of our software systems that support the management of networked systems. Agents are widely used in the design and implementation of distributed systems (Caire [26], Luck [86]). Agent based software engineering is a common design and build paradigm (Jennings [77], Yolum [133]); systems are composed of pieces of autonomous software, forming the basis of large scale loosely coupled systems. There are many areas of application (Bellifemine et al. [16], Jennings & Wooldridge [76]). One of them relates to the field of network management (Di Marzo Serugendo [36], Foster [49], Michaud [94]).

The necessities for designing and building systems in complex, open environments by means of agents become clear when the application domain involves a number of distinct problem solving entities (or data sources) that are physically or logically distributed (in terms of their data, control, expertise, or resources) and which need to interact with one another in order to solve problems (Moonen [96]). Systems whose capabilities and parameters are likely to need to change over time, or are treated differently across multiple organisational domains, benefit from the scalability aspects of agent systems (Stone & Veloso [123]).

The state-of-the-art methodologies and notations for the development of agent-based systems was surveyed by Bauer [14]. Agent Oriented Software Engineering (Zambonelli et al. [134]) methods extend traditional design methodologies. It is common to use four engineering phases (Mes et al. [92]): decomposition of the system into functionalities, assignment of functionalities to agents, definition of interaction protocols and the design of the decision-making capabilities and action-criteria of the agents.

2.1.3 Distributed problem solving

Historically, agents are used in the area of distributed problem solving. They play an important role in the research fields of distributed artificial intelligence, distributed learning and distributed problem solving (Weiss and Dillenbourg [127]). Distributed problem solving involves the collective effort of multiple entities to combine their knowledge, information, and capabilities to develop solutions to problems that cannot be solved by a single entity (Durfee [41]).

Finding structures in network-events is a distributed learning problem in which multiple learners collaborate. A learner can be implemented in the form of a software agent. The agent acts as a problem-solver and that collaborates with other agents in order to develop solutions to the distributed learning problem. In chapter 3 we describe a distributed learning mechanism that uses agents that collaborate in a grammar induction task.
2.2 Learning in distributed environments

Learning systems are systems that automatically improve with experience (Mitchell [95]). Collaborative learning is regarded as a learning task that is carried out by a group of individual learners. Collaborative learning strategies and algorithms are applied in cases where constraints hamper the use of single learning methods. There are many strategies and algorithms that let a group of learners share hypotheses and results in order to fulfil a learning task (Cervone [27]). Some algorithms are designed to deal with communication- or data availability constraints in their environment whilst others are designed to improve an individual learning goal or the performance of a weak learning algorithm.

In cases where communication- and privacy constraints prevent environments from having all data accessible from a single place, learning tasks must be carried out by multiple learners. One kind of collaborative learning is therefore known as distributed learning (Lynch, [88]). Distributed algorithms are algorithms designed to run on hardware consisting of many interconnected processors. Pieces of a distributed algorithm run concurrently and independently, each with only a limited amount of information. Distributed algorithms are commonly applied using divide and conquer techniques; data is distributed to multiple intermediate structures and results are gathered into one output.

A second kind of collaborative learning is known as transfer learning (Haskell [61]). It refers to methods in which a learned model from one particular learner is used to improve or bias the learning of a second learner. This is also known as iterative learning (Kirby [80]), in which collaborative methods are designed such as to meet the constraint of a weak performance of a single, individual learning algorithm. The approach is based on Darwinian type evolutionary algorithms; new learners are generated by processes of mutation and recombination guided by hypotheses (Michalski et al. [93]). In successive time steps, each learner sees some data, forms a hypothesis about the process that has produced that data, and generates data to be fed into next generation learners.

Another popular and commonly known learning technique is reinforcement learning (Chalkiadakis et al. [28], Sutton and Barto [124], Bosoniu et al. [21]) which is an unsupervised learning method that is suitable for learning in unknown environments based on local experiences. A reinforcement learner learns by trial and error while getting rewards from its environment. Challenges in this field are to coordinate the learning behaviour of the learners. Other popular learning terms commonly heard in the field of collaborative learning are boosting and co-training (Blum [17]). Boosting refers to a general method of producing prediction rules by combining rough and moderately inaccurate weaker rules. Using a boosting algorithm, individual weak learners carry out learning tasks in sequential time steps.
Boosting algorithms are algorithms that run on top of other algorithms. Another technique commonly known is Co-training, a method for learning a classification task from a small set of labelled data and a large set of unlabeled data, using separate, but redundant, views on the data. From an agent-perspective, collaborative learning is a task that is carried out by agents in a multi agent system (Sen & Weiss [116]). There are three classes of learning mechanisms that make multi-agent learning different from single-agent learning (Weiss & Dillenbourg [127]). In the first class, multiplicated learning, multiple learners learn independently of each other; their interactions do not impact their individual learning processes. There may be interactions among the agents, but these interactions just provide input which may be used in the other agents' learning processes. The individual learners may use the same or a different learning algorithm. Each individual learner typically pursues its own learning goal without explicitly taking care of the other agents' learning goals and without being guided by the wish or intention to support the others in achieving their goals. (The learning goals may mutually supplement each other, but this is more an 'emerging side effect' than the essence of multiplicated learning).

The second class, divided-learning, is characterized by a single learning task that is divided among several agents. In order to achieve the desired learning effects, the division of tasks can either be based on decisions on the functional aspects of the algorithm or based on the characteristics of the data to be processed. The agents involved in divided learning have a shared overall learning goal. The division of the learning algorithm or task is typically done by the system designer and is not a part of the learning process itself. Interaction is required for putting together the results achieved by the different agents, but as in the case of multiplicated learning this interaction does only concern the input and output of the agents' learning activities.

The third class, interactive learning considers agents that interact during learning. In the case of interactive learning the interaction is a more dynamic activity that concerns the intermediate steps of the learning process. Interaction, is an essential part and ingredient of the learning process. An agent involved in interactive learning does not so much act as a generalist or a specialist, but as a 'regulator' who influences the learning path and as an 'integrator' who synthesizes the conflicting perspectives of the different agents involved in the learning process. Note that the three classes are not disjoint, but are of different complexity and partially subsume one another: divided learning includes elements of multiplied learning, and interactive learning includes elements of divided learning.

In multi agent systems, all three learning mechanisms may occur simultaneously on different levels and with different, but related, purposes. Our software agents share tasks as well as models. Their learning mechanisms can be classified both as multiplied learning as well as interactive learning.
2.3 Grammar induction

In our work we use grammar induction for modelling successive events that occur in networked systems. In fact, this means that we try to learn the grammatical rules that produce those structures. An extensive overview of the field of grammar induction and grammatical inference methods is provided by de la Higuera ([67]).

In a grammar induction process, a learning algorithm is used to obtain a set of rules, i.e. a grammar, that explains the structure of a given set of data. The aim is to learn from sample data, usually a list of sentences, an unknown grammar which explains this data. This grammar is captured in a model that can be used to verify whether unknown samples follow the rules of the grammar. The model takes sequences of symbols as input (sentences) and produces a binary classification as output, indicating whether that sentence belongs to the language or not.

The type of models that are commonly used to represent grammars and classify languages are automata.

2.3.1 Language theory and Automata

Formal Language Theory aims at characterizing the possibility of learning a certain class of languages given a specification of the conditions under which learning has to take place.

Formally, a language is defined as follows:

**Definition:** a **language** is defined as any subset of $\sum^*$, where $\sum$ is a fixed finite set of symbols, called the alphabet of the languages to be considered, and $\sum^*$ is the set of all finite strings of elements$^5$ of $\sum$.

**Definition:** a **grammar** is a set of rules that characterises a group of languages.

---

$^5$ In mathematical logic and computer science, the Kleene star (or Kleene operator or Kleene closure) is a unary operation, either on sets of strings or on sets of symbols or characters. The application of the Kleene star to a set $V$ is written as $V^*$. If $V$ is a set of symbols or characters then $V^*$ is the set of all strings over symbols in $V$, including the empty string.
Languages can be categorized based on their syntactic structures. In terms of the elements of a syntactic structure, a language is the collection of all strings of terminal symbols that can be generated from a given start symbol using so-called productions.

- **terminal symbols** $<t>$; the collection of words, or symbols in the language.
- **non-terminal symbols** $<nt>$ the collection of descriptive parts of speech of the language (sentence, noun, verb, adjective, …)
- **productions**; rules that describe how the non-terminals may be replaced by strings of terminals and non-terminals.
- **start symbol**; a non-terminal used to start every derivation.

To illustrate this, we look at a formal definition of the language of all fully parenthesized expressions that can be formed over the four arithmetic operations using a single letter A through Z as place holders for variables and numbers. A grammar that explains a construction like for example $((A + B)/(C - D))$ is then given by: $L = (<exp>, T, N, P)$ where

\[
T = \{A..Z, +, -, *, /\} \\
N = \{<exp>, <term>, <op>\}
\]

and $P$ is the set of so-called ‘productions’:

- $<exp> ::= <term> | ( <term><op><term>)$
- $<op> ::= + | - | * | /
- $<term> ::= <exp> | A | ... | Z$

Chomsky distinguished the syntactic complexity of languages by considering four broad categories of languages, which are distinguished by how complex their productions can be.

<table>
<thead>
<tr>
<th>Language type</th>
<th>Allowed productions</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Regular’</td>
<td>$&lt;nt&gt; ::= t &lt;nt&gt;$ or $&lt;nt&gt; ::= &lt;nt&gt; t$ for all non terminals $&lt;nt&gt;$ and all terminals $t$.</td>
<td>Finite</td>
</tr>
<tr>
<td>‘Context Free’</td>
<td>$&lt;nt&gt; ::= \text{any string of } t$ and $&lt;nt&gt;$</td>
<td>PushDown</td>
</tr>
<tr>
<td></td>
<td>This is a very rich category of languages. This category includes all programming languages, and the sample language described above.</td>
<td></td>
</tr>
<tr>
<td>‘Context Sensitive’</td>
<td>$\text{any string of } &lt;nt&gt; ::= \text{any other string of } &lt;nt&gt;$ and $t$ where the length of the string on the right is greater than or equal to that the string on the left.</td>
<td>Linear Bounded</td>
</tr>
<tr>
<td>‘Unrestricted’/‘Recursively enumerable’</td>
<td>$\text{any string of } &lt;nt&gt; ::= \text{any other string of } &lt;nt&gt;$ and $t$ with no restrictions on the lengths of the strings involved in the substitutions.</td>
<td>Turing Machines</td>
</tr>
</tbody>
</table>

Table 1, Chomsky hierarchy of languages
2.3.2 Language Learnability

In a landmark paper (Gold, [58]) the idea of identification in the limit is introduced as a paradigm to study language learning. At the beginning of the learning process a student and a teacher select a class of languages $L$. The teacher consequently selects an element $L_i \in L$ and starts to produce example sentences from $L_i$.

After each example the student is allowed to update his guess for the language the teacher has selected. We expect the teacher to be an informer for the language i.e. each sentence from $L_i$ will be produced in the limit. The class of languages $L$ is considered to be *identifiable in the limit from positive information* if the student can for each language in $L$ reach a stable guess in a finite amount of time on the basis of only positive examples.

2.3.3 Automata

An automaton (plural: automata) is a mathematical model that, given an input of symbols, jumps through a series of states according to a certain transition function. The transition function tells the automaton which state to go to next given a current state and a current symbol.

Automata theory is closely related to formal language theory. Automata are often classified by the class of formal languages they are able to recognize. In formal language theory one talks about an alphabet, which is set of symbols, denoted by $\Sigma$, strings (sequences of symbols) and languages, defined as a set of strings. Languages are categorized by grammars, covering the underlying rules, which can be represented by automata.

An overview of different kinds of automata that are used to identify languages over event sequences is given by Verwer. ([125]). In this work he elegantly stresses the fact that the behaviour, or working, of real life systems is usually described using descriptions of state and transitions. He notices the importance of timed-automata. These automata include the information about the time that a transition takes to bring a system from one state to another.

An automaton can be deterministic or non-deterministic. The difference between a deterministic automaton and a non-deterministic automaton is that a non-deterministic automaton can have multiple transitions for one particular symbol or may not have a transition for each symbol in the alphabet. In a deterministic automaton each state has a transition for every symbol in the alphabet. For practical reasons, transitions that are not allowed are not drawn, unless they explicitly denote a negative example. An non-deterministic automaton can be transformed to a deterministic automaton using a subset algorithm (Hopcroft [74]).
Such a step can be expensive in terms of computational complexity: the identification of sets of nodes on the original graph in the resulting deterministic graph implies an exponential computational complexity as there are $2^N$ sets of nodes in an original graph of $N$ nodes.

In our work, we model structures that can be represented by the class of regular languages. (see also section 2.4.3)
A specific subset within the class of regular languages is the set of finite languages. Finite languages are obviously regular as one can create a regular expression that is the union of every word in the language, and thus are regular.
Regular languages can be modelled by means of Deterministic Finite Automata (DFA). For every DFA there is a regular language and every regular language can be represented by a DFA.
For input of infinite length, one uses Büchi automata. Context Sensitive languages cannot be modelled by means of a DFA. Instead, pushdown automata (PDA) and Linear bounded automata (LBA) are used.

---

6 A Büchi automaton is the extension of finite state automata to infinite word lengths. A word is accepted if the automaton goes through some designated states infinitively often.

7 A Pushdown Automaton (PDA) is a finite automaton that can make use of a stack containing data. Every context-free grammar can be transformed into an equivalent pushdown automaton. The derivation process of the grammar is simulated in a leftmost way. Where the grammar rewrites a non-terminal, the PDA takes the topmost non-terminal from its stack and replaces it by the right-hand part of a grammatical rule (expand). Where the grammar generates a terminal symbol, the PDA reads a symbol from input when it is the topmost symbol on the stack (match). In a sense the stack of the PDA contains the unprocessed data of the grammar, corresponding to a pre-order traversal of a derivation tree. Pushdown automata differ from finite state machines in two ways: 1) They can use the top of the stack to decide which transition to take. 2) They can manipulate the stack as part of performing a transition.

8 A Linear Bounded Automaton (LBA) is a restricted form of a nondeterministic Turing machine. It possesses a tape made up of cells that can contain symbols from a finite alphabet, a head that can read from or write to one cell on the tape at a time and can be moved, and a finite number of states. It differs from a Turing machine in that while the tape is initially considered to have unbounded length, only a finite contiguous portion of the tape, whose length is a linear function of the length of the initial input, can be accessed by the read/write head. This limitation makes an LBA a more accurate model of computers that actually exist than a Turing machine in some respects. The only restriction placed on grammars for such languages is that no production maps a string to a shorter string. Thus no derivation of a string in a context-sensitive language can contain a sentential form longer than the string itself. Since there is a one-to-one correspondence between linear-bounded automata and such grammars, no more tape than that occupied by the original string is necessary for the string to be recognized by the automaton.
**Definition:** a Deterministic Finite Automaton $\mathcal{A}$ is defined as a 5-tuple $\mathcal{A} = \{Q, \Sigma, \delta, q_0, F\}$

Where

- $Q$ is a finite non-empty set of states,
- $\Sigma$ is a finite non-empty set of input symbols, input alphabet,
- $\delta: Q \times \Sigma \rightarrow Q$ the transition function,
- $q_0 \in Q$ the start state,
- $F \subseteq Q$ the final states or accepting states

Automata can be represented in the form of graphs. Figure 2.1 shows an example of a graph of a DFA. The nodes represent the state of the system, and the edges correspond with the transitions. The accepting states are denoted with a double-lined circle. One can see, by following the graph and concatenating the symbols of the edges that e.g. $ab$, $aaab$ and $aacbbb$ are strings that are accepted by the automaton. Examples of strings that are rejected (i.e. not accepted) are e.g. $aab$, $abb$, $acb$ and so on.

$Q = \{0,1,2,3,4\}$
$\Sigma = \{a, b, c\}$
$\delta = \{(0, a) \rightarrow 1; (0, c) \rightarrow 2; (1, a) \rightarrow 0; (1, b) \rightarrow 3; (2, b) \rightarrow 4; (4, b) \rightarrow 2\}$
$q_0 = \{0\}$
$F = \{3,4\}$

![Figure 2.1: Example of a DFA](attachment:dfa_example.png)
2.4 Learning Deterministic Finite Automata

DFA-inference, or DFA-induction is the process of creating a DFA from a set of examples, which can be classified both positive (accepted) as well as negative (not accepted) (Cicchello & Kremer [29]). In general, when learning a DFA from a set of examples, one starts by building a tree out of the sequences of the symbols that occur in the examples. When represented by means of a graph, every string in the dataset is a path in the graph. In the learning process, also known as the generalisation process, one tries to reduce the number of branches and leaves by means of merging states.

Figure 2.2 shows some examples of DFAs, created from three positive samples \textit{abcde}, \textit{abcbcd}, and \textit{abcbcd}.e.

The first DFA (1) shows the unfolded topology. This is called the maximum canonical automaton (MCA).

The second DFA (2) is the prefix-tree-acceptor (PTA) type. Starting from its first state (marked with a dot), transitions can be carried out sequentially ending in an accepting end state (denoted by a the double-lined circle).

The third DFA (3) in the figure shows a generalized version, i.e. it allows for the original samples to be accepted, but also other, “similar”, samples. As can be seen in this DFA, a generalization results in a DFA that contains symbol transitions that were not given in the original sample.

The fourth DFA (4) is also a generalized form, but here an extra constraint, called \textit{adjacency} is added. It forbids state transitions that result in neighbouring symbols which were not neighbours in any sentence of the sample. Compared to the third DFA, the adjacency constraint results in a model that does not accept the string \textit{ade}. In some cases this limited generalization results in more intuitive or practical models.

The most generalized form is shown in the fifth DFA (5), this form is called the Universal Automaton (UA).
2.4.1 Generalization by merging states

Since creating some DFA that is consistent with positive training data is trivial, namely the MCA or PTA, it is usual to add the constraint that the DFA should generalize to unseen test data. The process of making generalizations is called learning. By merging or clustering data using heuristics, the folding results in more generic DFAs, allowing unseen data to be classified as positive or negative. The smallest DFA that accepts all positive samples is of course the universal automaton, but this automaton has a poor generalization error\(^9\). We need a better criterion for smallest DFA. Given a finite positive data set representing an infinite regular language, the learning task is to find a DFA with minimum expected generalization error over the set of infinite regular languages consistent with the sample data. We use the principle of Minimum Description Length (MDL, see section 2.4.2) to identify such a DFA.

The process of learning a DFA starts with a sample, consisting of a set of sentences that are supposed to be accepted (or rejected) by the target DFA. First, a prefix tree...
acceptor (PTA) is built. This is the (tree-shaped) DFA that accepts exactly the sentences in the learning sample, and nothing else. Then, generalization (learning) takes place by merging states and making the resulting automaton deterministic again by applying further merges.

Figure 2.3 illustrates the process of merging two states. A merge of two states \( q \) and \( q' \) combines two states into one. This new state \( q'' \) has the incoming and outgoing transitions of both \( q \) and \( q' \). When a merge introduces a non-deterministic choice, i.e. when \( q'' \) is the source of two transitions with the same symbol, the target states of these are merged as well. Generalization takes place here.

Formally, a derived DFA as a result of merging states, is defined by means of partitions and quotient-DFAs (Cornuéjols & Miclet [31]):

An automaton \( \mathcal{A}/\pi \), called the quotient automaton, denotes the automaton derived from automaton \( \mathcal{A} \) with respect to the partition \( \pi \) of the set of states of \( \mathcal{A} \). It is obtained by merging states of \( \mathcal{A} \). The states belonging to the same subset, or block, \( B \) in \( \pi \) are merged in the quotient automaton. A particular quotient automaton belongs to a particular partition. During the process of generalizing automata, an algorithm (heuristic) is used to search through the set of possible partitions.
Figure 2.4 shows an automaton \( \mathcal{A} \) (left) and a derived quotient automaton \( \mathcal{A}/\pi \) (right) with \( \pi = \{\{0\}, \{1\}, \{2, 4\}, \{3, 5\}\} \)

Any accepting path in \( \mathcal{A} \) is also an accepting path in \( \mathcal{A}/\pi \).

**Definition:** Given a DFA \( \mathcal{A} \), we define \( L(\mathcal{A}) \) as the set of strings that belong to a language \( L \) described by \( \mathcal{A} \). If an automaton \( \mathcal{A}/\pi_{n+1} \) is derived from automaton \( \mathcal{A}/\pi_n \) by means of a partition, then \( L(\mathcal{A}/\pi_{n+1}) \subseteq L(\mathcal{A}/\pi_n) \).

**Definition:** A partition \( \pi \) of a set \( X \) is a set of nonempty subsets of \( X \) such that every element \( x \in X \) is in exactly one of these subsets. \( B(x, \pi) \subseteq X \) indicates the subset of the partition \( \pi \) of which \( x \) is an element.

**Definition:** Let \( \mathcal{A} = \{Q, \Sigma, \delta, q_0, F\} \) be a DFA.

The quotient automaton \( \mathcal{A}/\pi = \{Q', \Sigma, \delta', B(q_0, \pi), F'\} \) derived from \( \mathcal{A} \) on the basis of a partition \( \pi \) of \( Q \) is defined as follows:

\[
Q' = Q/\pi = \{B(q, \pi) | q \in Q\}
\]

\[
F' = \{B \in Q' | B \cap F \neq \emptyset\}
\]

\[
\delta': (Q' \times \Sigma) \to 2^{Q'} \forall B, B' \in Q', \forall a \in \Sigma, B' \in \delta'(B, a) \text{ iff } \exists q, q' \in Q, q \in B, q' \in B'
\]

and \( q' = \delta(q, a) \)

Using the notion of quotient automata we can decrease or increase the generality of the automaton and the associated language inclusion hierarchies by means of respectively splitting or merging states.
2.4.2 Minimum Description Length (MDL)

The 14th-century English logician William Occam stated that descriptions of the world, and models in general, should be as simple as possible. This principle, referred to as Occam’s razor (to cut off Plato’s beard of ideas) has had a decisive influence in the history of science (Domingos [38]). The principle of Occam’s Razor says that from a large set of models that all explain the same thing or behaviour, the least complex model is the best model to use. The Minimum Description Length principle (MDL) is a commonly known implementation of Occam’s way of selecting models. In the field of grammar induction methods MDL provides a useful solution to the model selection problem (Adriaans et al. [6]).

MDL is an information theoretic approach to inductive inference that originated in algorithmic coding theory (Rissanen [110]). MDL is based on the fact that any regularity in a given set of data can be used to compress the data, i.e. to describe it using fewer symbols than needed to describe the data as itself. Data is described in terms of a code as compressed by the model and the goal of model selection is to identify the model, from a set of candidate models, that permits the shortest description length (code) of the data (Grünwald [60], Myung [103]). Although MDL has evolved in the last decades, our work uses its original, so called two-part-code description.

**Definition:** The minimum description length principle (a.k.a. two-part code MDL): the best theory to explain a set of data is the one which minimizes the sum of the length in bits, of the description of the theory and the length in bits, of the data when encoded with the help of the theory.

Let $M_1, M_2, \ldots$ be a list of candidate models, the best model $M_l$ that describes the data $D$ is the one that minimizes the sum $L(M_l) + L(D|M_l)$ where $L(M_l)$ is the length, in bits, of the description of the model and $L(D|M_l)$ is the length in bits, of the description of the data when encoded with the help of the model.

In chapter 3 we will explain in detail how we use MDL as a model-selection criterion in the process of learning DFAs.

The principle of MDL tells us that, for a given set of models $M$ and a dataset $D$, we should try to find the model of combination of models in $M$ that compresses $D$ the most. The compression can be described in terms of the two-part-code: a description of a model (*model code*), and a description of the data in terms of the model (*data-to-model-code*). An intuitive example is a repeating data pattern. We only need to describe the pattern that is generated (*data-to-model-code*) and some information about the way the pattern is repeated (*model-code*). More general, data
that belongs to a particular language could be described in terms of the grammar (the model-code) of the language and a set of indexes corresponding to an enumeration of the sentences in the data (data-to-model-code). A commonly placed remark is that MDL strikes a balance between how complex a model/hypothesis is and how much it explains the data. This can be explained by means of fitting a dataset using an n-th degree polynomial (Grünwald [60]) and resembles the intuitive feeling about the DFA models we obtain after generalization.

### 2.4.3 Using DFAs for modelling successive network events

In general, modelling processes in networked systems can be done using different techniques and mechanisms. For example Petri Nets (Winskel & Nielsen [131]) are used to model the interaction between asynchronous and parallel processes. Sequential patterns by means of probabilistic models, e.g. N-grams (Manning & Schütze [90]) or Hidden Markov models (Ephraim & Merhav [43]). Learning such models is done by finding optimal probability distributions of state transitions given a number of the states. Typically for Markov models is that all information about the future is contained in the present state; one does not need to examine the past to determine the future.

In our work, we relate sequences of events and look for patterns in those sequences, while taking into account the history of paths in those patterns. We therefore use grammatical models.

The rules for regular languages (see table on page 19) are

\[
<nt> ::= t \\
<nt> ::= t<nt>
\]

In the case of monitoring network-events, the non-terminals \(<nt>\) correspond with the states of the system, the terminals \(<t>\) are the job-locations. E.g. when the system is at a certain state X, and a job goes from one location to another, the state of the system transforms to Y. The system has a limited number of states; 1) the number of network nodes is limited and 2) the job-paths are terminated either because the job gets executed or stays on a particular network node. A consequence of the Myhill–Nerode\(^{10}\) theorem is that a language L is regular if and only if the number of equivalence classes is finite. In our case, the equivalence classes correspond with the unique number of paths, which is a finite number. Furthermore, when looking at the so-called pumping lemmas for regular and context

---

\(^{10}\) In the theory of formal languages, the Myhill–Nerode theorem provides a necessary and sufficient condition for a language to be regular or not.
free languages\textsuperscript{11} that define repeating patterns for regular languages to be like $xy^iz$ and for context-free languages to be like $x^izy^i$, we can exclude the job-traffic patterns to be a context-free language: we don’t expect to model events that first show up $i$ times on node $x$, then at node $z$, followed by the same number of times on node $y$. Given the statements above, we consider DFA models as adequate for modelling job-handling behaviour. Although irregular behaviour can be modelled by Büchi automata because of their possible infinite loop situations, in practice we use fixed time-intervals in which the numbers of possible loops is finite.

2.5 Learning by compression

The previous sections explained that grammatical inference in general means to find laws or regularities underlying some given set of data. The grammar induction algorithms we use aim at constructing a grammar by means of incremental compression of the data set represented as a digraph or tree. The learning process takes the form of a guided incremental compression of the data set by means of merging or clustering of the nodes in the graph. The algorithms use heuristics to guide the model reduction.

Regularity is closely related to compression; the more one is able to compress (describe in a compact manner) a set of data, the more regularities one has found in it and thereby, the more one has learned about the data. Theoretically, optimal compression represents the optimal interpretation of the data. Practically such a optimal compression cannot be computed.

Until recently, the view of learning as algorithmic data compression did not seem to have much practical value. Although lots of learning algorithms in fact perform some kind of data compression, this was not a guiding principle of their design. Recently this perspective has changed. On one hand there are more applications of existing implementations of compression algorithms (Cilibrasi [30], Dalkilic [34]), on the other hand, there is a better understanding of the mathematics behind compression, such as the principle of MDL (Adriaans & Vitányi [7]) and the theory of Kolmogorov complexity (Li & Vitányi [85]).

\textsuperscript{11} A pumping lemma is a statement that any language of a given class can be "pumped". A language can be pumped if any sufficiently long string in the language can be broken into pieces and these pieces can be repeated to produce an even longer string in the language. Thus, if there is a pumping lemma for a given language class, any language in the class will contain an infinite set of strings all produced by a simple rule given by the pumping lemma.
2.6 Adaptive modelling

Collaborative agent organisations tend to be hard to manage because of their time dependent topologies, complex and chaotic behaviour. Complex systems (Boccaletti et al. [18], Newman et al. [105]) are systems with non-trivial topological features. Chaos is an intriguing feature of dynamic systems: changes applied to one agent influence the behaviour of others. In chaotic systems any uncertainty in the beginning produces rapidly escalating and compounding errors in the prediction of the system's future behaviour. (Devaney [35]). To be able to make long-term predictions of the behaviour of a chaotic system, the initial conditions must be known to an indefinite level of accuracy with respect to all possible states of that system, together with an evolution law that prescribes all future states.

Modelling (in) collaborative agent organisations might be compared to weather forecasting. In order to make long-term forecasts we must constantly update our weather models so that we can improve the forecast gradually and increase the reliability of our forecast to an acceptable level. Static modelling learns the structure of a system that is static in time. As more information about the system is gathered, the possibility that the learner approximates to an adequate model grows. Once a model that explains the behaviour of the system is learned, there is no need to change it in the future. Adaptive modelling (Adriaans [5]) is about learning the structure of a system that changes in time. There is a relation between the rate at which models are generated and the change rate of the system. The role of adaptive modelling techniques is of vital importance for modelling the behaviour of collaborative agent organisations.

In a distributed environment, adaptive modelling involves collaboration amongst the learning entities and their environment. For example, the modelling involves data retrieval and analysis over multiple datasets, sometimes even belonging to multiple domains. (Axelrod [12], Michaud [94], Hodík et al. [71], Mulder et al. [100]). Agents retrieve event-data and build models within their own domain. In order to obtain global models, the agents communicate and share their models. Instead of using learning techniques for modelling the behaviour of the whole network environment, such techniques are used for both modelling in local domains and combining those models into a global model. Chapter 3 discusses our collaborative modelling mechanism.

There are a number of general motivations for preferring adaptive modelling above static modelling:

- models often lag behind the actual network situation.
- information of the environment is incomplete. It is very hard to gather all the information systematically, at a single moment of time.
there are constraints on the availability and accessibility of monitoring data.
information is structured in domain-specific ways.
a gathering of all data at a single place is hardly feasible.

Some of these motivations involve the volatile aspects of the environment; they account for the need of an adaptive way of learning. Other motivations involve the multiple-domain aspects; they plea for sharing individually retrieved models.
Related to these motivations are the issues on centralised vs. decentralised modelling. Using a central controller for modelling and managing the network has many disadvantages: the controller usually needs current knowledge about the entire system, which implies communication links from every part of the system to the controller. This scales badly, and has the risk of a single failure that could bring the whole system down. On the other hand, decentralised modelling involves extra communication and data synchronisation efforts.

2.7 Self-organisation

Self-organisation\(^\text{12}\) refers to the effect of local rules in a set of interacting system elements, that cause a pattern or structure in the global behaviour of the set. (Heylighen [64]). In self-organizing phenomena the global behaviour of the set of elements seems to have no direct relation to the rules for local behaviour of the individual elements. Examples of self-organisation can be found in physics (e.g. the formation of snow crystals), biology (flocking birds) and computer science (autonomic computing)\(^\text{13}\). Self-organisation implies the creation of order in a system without a form of central coordination.

Self-organisation is typically associated with systems that maintain an entropy that is different from the environment for a certain period of time. Systems that have a high unchanging entropy are not interesting from a learning point of view. Since there is no structure, there is nothing to learn from. But self-organizing systems do contain structures, allowing the process of learning of models which, in turn, can be used to optimise processes.

The notion of self-organisation is found to be key to complex network management (Edmonds [42]). Complex networks are systems that consist of many

\(^{12}\) The term self-organisation was first proposed by the cyberneticist W. Ross Ashby in the 1940s and developed among others by his colleague Heinz von Förster. During the 1960s and 1970s, the idea was picked up by physicists and chemists studying phase transitions and other phenomena of spontaneous ordering of molecules and particles. These include Ilya Prigogine, who received a Nobel Prize for his investigation of self-organizing "dissipative structures"

\(^{13}\) http://www.research.ibm.com/autonomic, visited on 07-03-2010
interacting components. The global behaviour of these interacting components cannot be described only in terms of local behaviour-rules of their single parts. Unlike classical modelling, which attempts to describe a system’s behaviour in terms of deterministic behaviour of its components, the behaviour in complex systems cannot be modelled according to behavioural rules of their parts. As a result, modelling the behaviour of complex networks is a difficult task, certainly in cases where the topology is highly volatile. Self-organisation causes complex networks to tend to settle into recognizable behaviours, in spite of their intrinsic unpredictability. The coordination for the alignment of tasks and the order of the system is distributed and redundantly present in the system (Di Marzo Serugendo [36]). The lack of a centralized point of control reduces the risk of a single-point of failure that can reduce the performance of the system drastically.

One can say that self-organisation typifies complex systems in general; these systems organize themselves so as to better cope with various internal perturbations and conflicts (Holland [72]). This allows them to evolve and adapt to a constantly changing environment.

2.8 Summary

In this chapter we have discussed some theoretical background. We explained the concept of agents from three perspectives: modelling, engineering and distributed problem solving. All of them are important in our work: we use agents to model collaborative organisations, we develop software agents to support the process of performance management and we look at organisations that can be modelled in terms of agents.

We have presented the basics of automata and DFA induction. We also explained the concept of MDL, our strategy for generalizing DFAs. Together, the concept of collaborative learning and DFA inference are used in chapter 3 where we work out our collaborative learning mechanism and use it for modelling network events. We also explained the ideas behind the notion of learning by compression. On one hand this explains the ideas behind MDL, on the other hand it sets one’s mind towards more fundamental questions on learning in terms of self-organisation which we will encounter in chapter 6.