MAKING MULTIAGENT SYSTEMS
MORE RELIABLE

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2005

By
Yingqian Zhang
School of Computer Science
Contents

Abstract ................................................................................................................. 8
Declaration ............................................................................................................. 9
Copyright .................................................................................................................. 10
Acknowledgements .................................................................................................. 11

1 Introduction .......................................................................................................... 13
   1.1 Monitoring Multiagent Systems ................................................................. 14
   1.2 Multiagent Survivability ............................................................................... 14
   1.3 Objectives ...................................................................................................... 15
   1.4 Thesis Structure ........................................................................................... 15

I Monitoring Multiagent Systems ............................................................................. 17

2 Motivation and Background .................................................................................... 18
   2.1 Agents and Multiagent Systems ................................................................. 18
   2.2 Debugging and Monitoring in Multiagent Systems ...................................... 22

3 Monitoring Agents using Planning ........................................................................ 25
   3.1 Introduction .................................................................................................. 25
   3.2 Architecture of the Monitoring System ...................................................... 26
   3.3 Message Flow in a Multiagent System ......................................................... 27
   3.4 Modelling Agent Behaviour via Planning .................................................. 31
   3.5 Agent monitor ................................................................................................ 37
   3.6 Properties ...................................................................................................... 44
   3.7 Implementation within IMPACT .................................................................... 54
List of Tables

6.1 A difference table generated by the deployment agent . . . . . . . . 88
6.2 Experiment 1: CPU and Network time of three algorithms with varying problem size when dps are uniformly distributed . . . . . 103
6.3 Experiment 2: CPU and Network time of three algorithms with varying problem size when dps vary dramatically . . . . . . . 103
7.1 Experiment 1: Approximation ratio using different algorithms with setting S1 ....................................................... 142
7.2 Experiment 2: Approximation ratio of different algorithms with setting S2 ....................................................... 143
7.3 Experiment 4: Approximation ratio using different algorithms with setting s4 ....................................................... 147
7.4 Experiment 5: Approximation ratio with setting S1 but using larger space ratio (3-4) ....................................................... 147
# List of Figures

3.1 The overall architecture of the monitoring system. 27
3.2 The Gofish post office system. 29
3.3 Message protocol of the simplified Gofish system. 30
3.4 An agent in IMPACT 55
3.5 Actions for monitor 57
3.6 AgentDE Program 58
3.7 AgentDE Connect Library screen 59
3.8 IMPACT agent monitor 60
5.1 Centralised Architecture for MAS Survivability 81
5.2 Distributed Architecture for MAS Survivability 81
6.1 Disconnection probabilities of the nodes 104
6.2 Experiment 3: survivability of deployment using algorithm ASA1 and ASA3. 105
6.3 Experiment 4: CPU time and network time with varying agent density. 106
6.4 Experiment 5: survivability of deployment using algorithm 1 and algorithm 3 with changing agent density. 107
6.5 Experiment 6: survivabilities of deployments as space density varies 108
7.1 Graph representation of the Set-covering transformation. 120
7.2 Experiment 1: computation time using different algorithms with setting S1 142
7.3 Experiment 3: approximation ratio using different algorithms with setting S3 144
7.4 Experiment 3: computation time using different algorithms with setting S3 145
7.5 Experiment 4: computation time using different algorithms with setting S4 .................................................. 146

7.6 Experiment 5: computation time with setting S1 but using larger space ratio (3-4) .................................................. 148

7.7 Experiment 6: Computation time and survivability returned by SF3 with varying $\alpha$. .................................................. 149
Abstract

Recent years have seen an increasing number of multiagent systems (MAS) developed within various agent platforms and frameworks in applications. Consequently we see the growing need to build more reliable multiagent systems. This thesis approaches this need in two parts: (1) *Part I: Monitoring Multiagent Systems*; and (2) *Part II: Survivability of Multiagent Systems*.

The first part aims at *verification of the implementation of a multiagent system and the collaboration of agents*. Verification of a MAS is difficult especially when the complete specification of agents is missing. Instead, in our approach, with the aid of the *message protocol of agents*, a debugging tool has been developed to verify whether a MAS is correctly designed and coded, and to monitor online behaviours of agents. We showed the properties of the proposed method. A running example has been implemented within the agent platform *IMPACT* and the planner *DLV*. The proposed debugging tool works for legacy multiagent systems and is not bound to any particular agent platform or planning system.

The aim of the second part is to *maximally ensure that a multiagent system is robust and resilient against failures in dynamic environment*. Based on the idea of agent replication, we built a *probabilistic survivability model*. Our approaches to multiagent survivability have been discussed in two chapters: one introduced *distributed models and algorithms* which can re-deploy agents when there is a need to re-evaluate the survivability of the MAS; and the other proposed various *centralised algorithms* to compute the survivability of a given agent deployment. With these *centralised algorithms*, we are able to measure the quality of an agent deployment and thus to guide agent replication. With the *distributed approach*, we can achieve more adaptive multiagent deployments that respond to the changes in the environment. We have implemented and tested the proposed algorithms and reported on the advantages and disadvantages of them in different environmental settings.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.
Copyright

Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the Author and lodged in the John Rylands University Library of Manchester. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the head of School of Computer Science.
Acknowledgements

During my PhD studies, I have been fortunate enough to work with many wonderful people, and I am delighted to be able to acknowledge them in my thesis.

First of all, a million thanks go to my supervisor Professor Jürgen Dix—the “Doktorvater” of mine—for always being there for taking care of my research matters as well as my personal problems. Thanks Jürgen for guiding me on the track at the beginning of my research, for providing me lots of chances to attend various summer schools, workshops, and conferences, for encouraging me to work with different researchers, for telling me to take the time and cheering me up when I feel frustrated and upset, and for always opening his door to me. It was really lucky for me to have him be my supervisor. I would also like to thank his wife, Vera Dix, for her friendship, for her warm encouragement, and for many wonderful time spending in her cozy house.

My sincere thanks go to my second supervisor, Dr. Renate Schmidt. I really appreciate her great support during my last year in Manchester—I felt that I were not alone there. I also thank her for her help on my research and many good suggestions on my writing-up. My advisor, Dr David Rydeheard, always expresses deep concern for me; thank you very much.

My research benefitted a lot from working with many other researchers out of Manchester. My thesis consists of two parts, which came out from two projects working with Professor V.S. Subrahmanian from University of Maryland, Professor Sarit Kraus from Bar-Ilan University, and Professor Thomas Eiter from Vienna University of Technology. I would like to thank Professor V.S. Subrahmanian for his help and generosity: thanks for giving me the chance to work on the project of multiagent survivability; thanks for the financial support during my stays in Maryland—I really enjoyed my time there; and thanks for guiding me on the research, even though he was extremely busy on his new baby—unfortunately for me but very fortunately for him. I would like to express my grateful thanks
to Professor Sarit Kraus. I have learned a lot from her serious attitude to research, her hard working and her deep insights into research problems. Thanks also go to Efrat Manisterski, for her cleverness and her help on approximations. At the very beginning of my PhD studies, I was fortunate enough to work with Professor Thomas Eiter in Vienna. I would like to thank him for his kind support and guidance. Thanks Michael Fink and Axel for being very comfortable collaborators.

Finally, I would like to thank the friends I have made these years. I had great time in Manchester with Liwen, Xiaoxia, Jun, Morna, Jenny, Li Lei and the members of Formal Method group. Thanks Airu, Zhang Mei, Ah Chiu, Annie, Justin, for their help in Maryland. Thanks Peter, Andreas, Wojtek, and Juan for being there in Clausthal. I thank you all for your witty and encouraging presence at all times.

My special thanks to Junqiu for his unconditional love, understanding and support. I wish to express my gratitude to my family: my mum, my dad, and my sister. No matter happiness or unhappiness happened these years, you always trust me and give me the strength to go through all ups and downs. Therefore, I dedicate this thesis to you.
Chapter 1

Introduction

Multiagent systems (MAS for short) have been recognised as the key technologies for tackling complex, realistic problems [Wei99, Woo02, LMP03]. In recent years, work on developing multiagent systems has significantly progressed. We have seen an increasing number of MAS developed within various agent platforms and frameworks in applications, ranging from auctions for critical commodities like electricity to monitoring of nuclear plants and computer networks.

Nevertheless, there are still many challenges for the development and deployment of multiagent systems. High possibility of failure on the developed systems has become one of obstacles to the take-up of agent techniques for large-scale, real-world applications [LMP03]. Hence, we see that there is a growing need to investigate the issue of how to make multiagent systems more reliable.

The difficulty in building reliable MAS applications can arise at all stages of the MAS development, such as: the design of the MAS may not completely reflect the developer’s intentions; the implementation of the MAS may be incorrect; the behaviours of the agents may go wrong; or the MAS could easily crash when facing malicious attacks or system failures.

It is certainly impossible to discuss and tackle all potential failures in this thesis. Instead, we contribute in investigating two important problems to build reliable multiagent systems: (1) how to verify that multiagent systems are correctly developed and work properly? and (2) how to ensure multiagent applications are robust and resilient against failures?
1.1 Monitoring Multiagent Systems

The first problem (1) usually arises during the implementation of a multiagent system, where a major problem is to verify that agents are coded correctly and they collaborate well to reach the goal.

It is well known that verification is impossible in general especially when details of the agents are missing. An alternative approach is through extensive testing, where large number of scenarios are created and simulated by the developed multiagent system in order to ensure the agents are bug-free and with appropriate behaviours. Clearly, such testing approach is expensive and inefficient. Debugging tools designed specifically for multiagent systems are highly desirable.

1.2 Multiagent Survivability

The second problem (2) arises when a set of agents are deployed over a dynamic network, where a major problem is the deployed multiagent applications may easily crash due to external events.

The ability of multiagent systems to provide services in the presence of attacks and failures is known as the survivability of multiagent systems. As multiagent applications heavily rely on the collaboration of agents, the failure of one of the agents may bring the whole system down. Thus, replicating agents on different hosting nodes is a natural and efficient way to ensure the survivability of multiagent systems.

In most of existing replication approaches, it is common that there is one special algorithm in the network which is in charge of the replication. However, the approaches of this kind are centralised—in another word, even though the agents in the MAS may be distributed across the network, the algorithm itself resides on a single node. Therefore, if the node hosting the algorithm goes down, then the whole agent system is compromised. Moreover, most of approaches are static—once agent replication is done, they forget the survivability issue afterwards. It is desirable to continuously monitor how well the MAS is surviving and to make the replication respond to the changes in the environment. Furthermore, we need efficient algorithms to measure the performance of the replication so as to guide the replication method.
1.3 Objectives

Motivated by the needs and the difficulties listed in Sections 1.1 and 1.2, the objectives of this thesis are as follows:

Objective 1: To develop a debugging technique to verify whether a multiagent system is correctly designed and implemented, and to monitor online behaviours of collaborative agents to reach some certain goals.

Objective 2: To develop distributed approaches which ensure the survivability of a multiagent system and are capable of adapting to the changing environment.

Objective 3: To develop efficient survivability algorithms which are able to measure and compute the survivability of a given multiagent system, and thus to guide replications in the multiagent system.

1.4 Thesis Structure

Our work is presented in two parts: monitoring multiagent systems and survivability of multiagent systems, linking two problems mentioned at the beginning of this chapter. The remaining of this thesis is structured as follows.

Part I: Monitoring Multiagent Systems (Part I of this thesis aims to achieve Objective 1).

Chapter 2: Motivation and Background. This chapter gives a brief introduction to agents and multiagent systems. It also shows the motivation of our work on monitoring agents.

Chapter 3: Monitoring Agents using Planning. This chapter presents our approach to the debugging of multiagent systems by monitoring agents. It gives technical details of our approach as well as the implementation—a running example demonstrates the proposed method.

Chapter 4: Discussion and Future Work. Chapter 4 overviews and discusses the existing techniques used for debugging multiagent systems and monitoring the behaviours of agents. Suggestions for future work are also given in this chapter.
Part II: Survivability of Multiagent Systems (Part II of this thesis achieves Objective 2 and Objective 3).

Chapter 5: Motivation and Background. This chapter contains a brief background on survivability and replication techniques in multiagent systems, which then leads to the motivation of our work.

Chapter 6: Agent-Oriented Survivability Models. This chapter aims at the Objective 2. It introduces an agent-oriented (or distributed) approach to adaptively ensure the survivability of multiagent systems. An extensive empirical analysis of the proposed algorithms is also contained in this chapter.

Chapter 7: Centralised Survivability Algorithms. This chapter aims at Objective 3. It proposes two exact algorithms to compute the survivability of a multiagent system. We show that computing survivability is intractable, thus various heuristics are introduced for fast computation. A set of experimental results report on the advantages and disadvantages of different heuristics in different environmental settings.

Chapter 8: Discussion and Future Work. Replication based approaches to improving survivability of multiagent systems are discussed in this chapter. We also suggest some interesting future works, as the extensions of our approach.

Finally we conclude with:

Chapter 9: Conclusion. This chapter makes a final remark on the whole thesis and shows the contributions of our work.

Appendix A: It gives the specification of the extended Gofish multiagent system, which is mentioned in Chapter 3.
Part I

Monitoring Multiagent Systems
Chapter 2

Motivation and Background

Part 1 of this thesis aims at the Objective 1 presented in Chapter 1. The objective is to verify whether a multiagent system is correctly developed and to monitor online behaviours of collaborative agents. Before formally presenting our approach in Chapter 3, in this chapter, we first give a brief introduction to agents and multiagent systems (Section 2.1). We then show the motivation of our work on monitoring agents in Section 2.2.

2.1 Agents and Multiagent Systems

Agent technology has been attracting lots of interests in recent years. In this section, we give a short introduction to agent technology. We focus on definitions and characteristics of agents and multiagent systems.

Notation of Agency

There are various definitions of what an agent is. According to the definition given by Wooldridge and Jennings [WJ95], an agent is a computer system that acts autonomously in its environment in order to reach certain goals. Distinguished from the object, an agent is considered to be intelligent, with the following properties [WJ95]:

- Autonomous: An agent operates without direct intervention of others, and has some kind of control over its actions and internal state.

- Reactive: An agent reacts to changes in the environment at certain times to reach its goals.
• **Proactive**: An agent takes the initiative being goal-directed.

• **Social**: An agent interacts with others to reach the goals.

Agents situate and behave in the *environment*. Thus, the environment heavily affects the decision making process of an agent. Russell and Norvig [RN95] discuss and classify the environment properties as follows:

• **Accessible vs. inaccessible**: An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment’s state. In an accessible environment, the agent needs not maintain an internal state to keep track of the world.

• **Deterministic vs. non-deterministic**: A deterministic environment is one in which any action has a single guaranteed effect, that is, there is no uncertainty about the state that will result from performing an action. However, if the environment is inaccessible, it may appear to be nondeterministic to the agent. Non-deterministic environments present more challenging problems for the agent designer.

• **Episodic vs. non-episodic**: In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios. Episodic environments are simpler because the agent can decide what action to perform based only on the current episode. It need not reason about the interactions between this and future episodes.

• **Static vs. dynamic**: A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. A dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent’s control.

• **Discrete vs. continuous**: An environment is discrete if there are a fixed, finite number of actions and percepts in it.

Obviously different types of environment may require different designs of agent program. The most complex environment where an agent can be situated is one which is inaccessible, non-episodic, dynamic, and continuous.

When facing some difficult situations, the capability of a single agent is often limited by its knowledge and its computation resources. Such situations could be,
2.1. AGENTS AND MULTIAGENT SYSTEMS

for example, when the agent situates in a complex environment like the internet, or when the problem the agent aims at tackling is large-scale and heterogeneous in nature. Because of the limitations of a single agent, more and more attentions have been paid on the research of multiagent systems (MAS for short).

A *multiagent system* can be defined as a loosely coupled network of agents that interact to solve problems that are beyond the individual capabilities or knowledge of each agent [DL90]. Sycara [Syc99] outlines the main characteristics of a multiagent system: (1) each agent has *incomplete* information or capabilities for solving the problem; (2) there is no global system control in a multiagent system; (3) data is decentralised, and (4) computation is asynchronous.

Compared with a single agent, a multiagent system enjoys many potential advantages with the following capabilities [Syc99, JSW98]:

- A multiagent system can solve problems that are difficult for a centralised agent. For example, a single agent may not be able to deal with a large problem because of resource limitations. Furthermore, there is the risk of having one centralised agent system, which could be a performance bottleneck.

- A multiagent system allows for the interconnection and interoperation of multiple existing legacy systems. Different legacy systems can be incorporated into an agent system, so that they can be exploited by other pieces of softwares.

- A multiagent system provides solutions to the problems that can naturally be regarded as a society of autonomous interacting agents who may have different goals. It provides solutions in situations where expertise and information sources are distributed.

- A multiagent system may enhance the performance of computational efficiency, reliability, responsiveness, and flexibility, since it allows parallel computation and cooperation among agents who have various capabilities and responsibilities.

Multiagent systems provide many potential advantages, however, there remain lots of difficult challenges especially for realistic, large, distributed and open multiagent systems—which represent the long-term future of agent technology. In complex multiagent systems, other than regular problem solving agents, other
specially designed agents, *middle agents*, may provide assistance. Researchers have identified many different types of middle agents. For instance, in a *brokered multiagent system* [DSW97a], *matchmakers* are agents that maintain a continually updated repository of information about regular agents currently in the system. *Blackboard agents* can collect requests. And *brokers* are agents that are capable of accepting tasks from requesting agents and assigning them to others. These agents provide an effective means for mediating the interactions between agents in an open system—internet. In the context of fault tolerant multiagent systems, middle agents are also responsible for detecting failures and recovering from failures. They are sometimes called *sentinels* (see Chapter 4).

**Collaboration in Multiagent Systems**

Since multiagent systems have capabilities to tackle more complex problems than single agents, we are more interested in problems related to *multiagent systems*. As we described earlier, a multiagent system can be considered as a society of different kinds of agents. Thus in multiagent systems, interaction between agents has become one of the key technologies. Agents in the same system can be either *cooperative* or *self-interested*. In a multiagent system where agents are cooperative, agents work *cooperatively* towards a common goal of the system. While in a self-interested multiagent system, if agents represent individuals or organisations who have their own interests, agents are then assumed to act based on their own goals, possibly at the expense of others, which may cause conflicts between agents. *Negotiation* is considered as a method to solve such problems in self-interested systems.

A lot of progress has been made on research of interactions between agents in multiagent systems, ranging from the early simple but effective *Contract Net Protocol* introduced by Smith [Smi80] and many variations proposed thereafter, to more recent work with market mechanisms in coordination [JSW98].

In addition to multiagent collaboration, a lot of other issues on multiagent systems have been addressed, such as multiagent learning, agent communication languages, multiagent planning, ontologies and so on. Furthermore, agent researchers also draw on ideas from many disciplines outside of Artificial Intelligence, including, for example, biology, social science, economics, logic, and philosophy. We refer to [HS98, Woo02, LMP03, JSW98, Wei99, AGE, UMB, MAS] for more detailed and comprehensive discussions on agent technologies.
Applications of Agent Technology

Applications of agents and multiagent systems can be divided into three broad categories [LMP03]:

1. *Assistant agents*, who take part in gathering information or executing trans-
actions on behalf of human beings. This type of application is usually
performed by a *single agent*;

2. *Multiagent decision systems*, where agents belong to one single system must
work together to make some joint decisions based on their knowledge and
capability;

3. *Multiagent simulation systems*, where agents are used for representing the
real world components, and the multiagent systems are models to simulate
the real-world environments.

Different from the first type of application, the second and the third types
are *multiagent* based applications, which are proven to be more widely used and
developed than single agent systems. For example, over the past years, multi-
agent systems have been successfully applied to many real-world applications,
such as manufacturing, telecommunication systems, air traffic control, electronic
commerce, medical care, and information filtering and gathering. In addition,
multiagent systems have been used to simulate economies, societies and biologi-
cal environments.

We are not going to further outline the agent applications developed in various
domains for various purposes. We refer to [LMP03, BDDS05] for more descrip-
tions of the application areas of agent based systems.

2.2 Debugging and Monitoring in Multiagent Systems

In contrast to numerous efforts on the development of a wide range of topics in
agent based systems, relatively little work has been done on the issue of *debugging*
and *monitoring* in multiagent systems. As more and more multiagent systems
have been developed and deployed for real-world applications and simulations, it
becomes clear that there is an increasing need to developing debugging techniques
for multiagent systems.
The traditional debugging techniques for distributed systems can help multiagent debugging: unfortunately the help is quite limited. As discussed in Section 2.1, multiagent systems have some specific characteristics compared with traditional distributed systems and they are inherently complex due to agents themselves, the environments which they reside in, and the interactions between agents. For instance, the environment where a traditional distributed system situates is usually closed and static. To the contrary, we often assume the dynamic and nondeterministic environment for a multiagent application. The individual agents are autonomous and they may act in complicated and sophisticated ways. Furthermore, interactions between agents in multiagent systems are more complex than those in distributed systems, which are usually just simple communication protocols. All of these complexities of multiagent systems make the traditional debugging tools somehow ineffective. The debugging tools designed specifically for multiagent systems are highly desirable.

Current debugging tools for multiagent systems mostly use the techniques of information gathering and visualisation to present a graph which demonstrates the system behaviour to the developers [BHS04, NNLC99, LA95, GPD94]. The limitation of such visualisation based approaches is that the developers are usually provided so much information that it is difficult for them to filter the useful information and draw conclusions on the problems in the system. Some other current approaches to monitoring multiagent coordination are based on plan-recognition, by Huber [HD95], Tambe [Tam97], Intille and Bobick [IB99], Devaney and Ram [DR98], Kaminka et al. [KT00]. In this plan recognition approach, an agent’s intentions—goals and plans, beliefs or future actions are inferred through observations of another agent’s ongoing behaviour. These approaches mainly aim to inferring team or sub-team plans and future actions of agents. The multiagent debugging issue is not addressed. We will give more in-depth discussions on these approaches in Chapter 4.

As we stated before, a major problem that agent developers may face with many multiagent platforms is to verify that a set of implemented agents collaborate well in order to reach a certain goal, for example, in supply chain management systems. It is well known that verification is impossible in general especially when details of the agents are missing in heterogeneous environments. As shown in our approach in the next chapter, in contrast to verification, we point out that monitoring a multiagent system does not require a complete specification of
the behaviour of the particular agents. Rather, we adopt a more general, and in practice much more realistic view: We do not have access to the entire internal state of each single autonomous agent, but we are able to observe the communication between agents of the system. By means of its communication capabilities, an agent can potentially control another agent. Our aim is to draw conclusions about the state of a multiagent system by monitoring the message protocol.

In the next chapter, we will detail our approach and illustrate it with an example derived from an implemented agent system.
Chapter 3

Monitoring Agents using Planning

In a collaborative multiagent system (MAS), autonomous agents work together in order to reach a common goal. In the work presented in this chapter, our aim is to monitor some aspects of the behaviours of the agents in order to detect inconsistencies and help to debug the whole agent system.

Section 3.2 gives an overall architecture of the monitoring system. Section 3.3 describes the basic agent framework that we build upon and presents a (simplified version of a) multiagent system in the postal services domain. Section 3.4 describes how to model the intended behaviour of a multiagent system as an abstract planning problem, and instantiates this for our example system using the action language $K$. Our approach to agent monitoring is then discussed in Section 3.5; some fundamental properties are investigated in Section 3.6. Section 3.7 introduces a running example to demonstrate the proposed method.

3.1 Introduction

In this chapter, we present a monitoring approach which aids in automatically detecting that agents do not collaborate as intended. In the spirit of Popper’s principle of falsification, it aims at refuting from (possibly incomplete) information at hand that an agent system works properly, rather than proving its correctness. In our approach, agent collaboration is described at an abstract level, and the single steps in runs of the system are examined to see whether the agents behave “reasonably”, that is, we check whether the agents’ behaviours are
“compatible” with a sequence of steps for reaching a goal.

Therefore, even if the internal structure of some agents is unknown, we may get hold of the messages exchanged among them. A given message protocol allows us to draw conclusions about the correctness of the agent collaboration. Our monitoring approach is based on this fact and involves the following steps:

1. The intended collaborative behaviour of the agents is modelled as a planning problem. More precisely, knowledge about the actions performed by the agents and their effects is formalised in an action theory, $T$, which can be reasoned about to automatically construct plans as sequences of actions to reach a given goal.

2. From the action theory $T$ and the collaborative goal $G$, a set of intended plans, $I$-Plans, for reaching the goal $G$ is generated via a planner.

3. The observed agent behaviour, more specifically, the message actions from a message log, is then compared to the plans in $I$-Plans.

4. In case an incompatibility is detected, an error is flagged to the developer (or user, respectively), pinpointing the last action causing the failure so that further failure recover steps might be taken.

Steps (2)–(4) can be done by a special monitoring agent called monitor, which is added to the agent system in order to provide supports both during testing, and in the operational phase of the system.

We now start with the overall architecture of the monitoring system.

### 3.2 Architecture of the Monitoring System

The overall design of the monitoring system is depicted in Figure 3.1. There are four main elements in the figure: a multiagent system, an agent log, an agent monitor, and the user. We give a brief description of them as follows.

1. The multiagent system represents the system being debugged, where agents can be coded and implemented with any languages. We assume that agents are cooperative by exchanging messages in order to reach a common goal.
2. *The agent log*, which is provided by the agent system or can be added to the agent system easily if not existing, maintains a record of exchanged messages between agents.

3. *Agent monitor* is a special monitoring agent, who is introduced into the standard multiagent system for the monitoring purpose. Agent monitor holds a set of intended plans *I-Plans* and examines the interactions of agents against intended plans. Agent monitor generates an error report once any error has been detected.

4. *The user* could be an agent developer or a computer, which receives the error report generated by agent monitor. Further actions such as correcting errors are then carried out by the user.

In the following, we detail our approach and illustrate it with an example derived from an implemented agent system.

### 3.3 Message Flow in a Multiagent System

In this section, we present the basic agent framework. We consider multiagent systems consisting of a finite set \( A = \{a_1, \ldots, a_n\} \) of collaborating agents \( a_i \). Although agents may perform a number of different internal actions, we assume that only one action is externally observable, namely an action called `send_msg(m)`, which allows an agent to send a message, \( m \), to another agent in the system. Every `send_msg` action is given a timestamp and recorded in a message log file containing the history of messages sent. The following definitions do not assume a
sophisticated messaging framework and apply to almost any multiagent systems. Thus, our framework is not bound to a particular MAS.

**Definition 3.3.1 (Message, \( \mathcal{M}_{\text{log}} \) file).** A message is a quadruple \( m = \langle s, r, c, d \rangle \), where \( s, r \in A \) are the identifiers of the sending and the receiving agents, respectively; \( c \in C \) is from a finite set \( C \) of message commands; \( d \) is a list of constants representing the message data. A message log file is an ordered sequence \( \mathcal{M}_{\text{log}} = t_1.m_1,t_2.m_2, \ldots ,t_k.m_k \) of messages \( m_i \) with timestamps \( t_i \), where \( t_i \leq t_{i+1}, i < k \).

The set \( C \) constitutes a set of message performatives specifying the intended meaning of a message. In other words, it is the type of a message according to speech act theory: the illocutionary force of an utterance. These commands may range from \texttt{ask/tell} primitives to application specific commands fixed during system specification.

Often, an agent \( a_i \) will not send every kind of message, but use a message repertoire \( C_i \subseteq C \). Moreover, only particular agents might be message recipients (allowing for simplified formats). Given that the repertoires \( C_i \) are pairwise disjoint and each message type \( c \) has a unique recipient, we will use \( m = \langle c, d \rangle \) in place of \( m = \langle s, r, c, d \rangle \).

Finally, we assume each action in the agent system has a fixed time bound within which the next action should happen, that is, a timeout for each action.

**Definition 3.3.2 (Action timeout).** The action timeout is a mapping from actions in the multiagent system to non-negative numbers, \( td : m \rightarrow D \), which specifies the bound on the time within which the next action should happen.

Action timeout \( td \) allows to determine from the message log file \( \mathcal{M}_{\text{log}} \) whether the multiagent system is stuck or it is still idle. Note that according to different example scenarios, the timeout \( td(m) \) may depend on time taken by previous actions.

**Gofish Post Office**

We consider an example multiagent system called *Gofish Post Office* for postal services. Its goal is to improve postal services by mail tracking, customer notifications, and advanced quality control. The following scenario is our running example:
3.3. MESSAGE FLOW IN A MULTIAGENT SYSTEM

Example 1 (Example scenario). Pat drops a package, $p_1$, for a friend, Sue, at the post office. In the evening, Sue is informed by Pat through a phone call that a package has been sent. The next day, Sue decides to pick up the package herself at the post office on her way to work. Unfortunately, the clerk has to tell her that the package is already on a truck on its way to her home.

In the Gofish post office, a package’s life cycle events are defined as below:

- **dropOff**: It describes the package being dropped off to a Gofish mail collection point.

- **distCenter**: It refers to the arrival of the package at the distribution centre where is closest to the destination.

- **pickup**: It refers to the pickup of the package by the customer at the distribution centre.

- **truck**: It refers to the loading of the package into a truck at the distribution centre. The truck will deliver the package to the destination.

- **delivery**: It refers to the delivery of the package to the destination.

The overall design of the Gofish multiagent system is depicted in Figure 3.2. An *event dispatcher agent* ($\text{disp}$) processes incoming package drop off messages and communicates relevant external events to an *event management agent* ($\text{em}$). The *event management agent* maintains an event database. Information about packages is stored in a package database manipulated by a *package agent* ($\text{pa}$). The *notification agent* ($\text{notify}$) notifies customers about package status and expected delivery time, for which it has direct query access to the statistics database.
It obtains the phone number by interaction with the package agent. Finally, a zip agent (zip) informs responsible managers, which are stored in a manager database, about zip codes not being well served. Interaction in Gofish is asynchronous and is done via message exchanges between agents.

**Example 2 (Simple Gofish).** To keep things simple and illustrative, we restrict the Gofish MAS to the package agent, \( pa \), the event management agent, \( em \), and the event dispatcher agent, \( disp \); thus, \( A = \{ pa, em, disp \} \). The messages concerning agent notify will be discussed in the extended version of the example at the end of Section 3.5. Figure 3.3 illustrates the message protocol in the simplified Gofish multiagent system.

Agent \( disp \) informs agent \( em \) about a package (identified by a unique identifier) being dropped off at the post office, its arrival at the distribution centre, its loading on a truck, its successful delivery, or when a recipient shows up at the distribution centre to pick up the package by herself: \( C_{\text{disp}} = \{ \text{dropOff}, \text{distCenter}, \text{truck}, \text{delivery}, \text{pickup} \} \). Agent \( em \) instructs agent \( pa \) to add a package to the package database after the drop off, as well as to update the delivery time after delivery or customer pickup: \( C_{\text{em}} = \{ \text{addPkg}, \text{setDelivTime} \} \). After updating the delivery time, the package agent provides package information relevant for statistics to the event manager agent: \( C_{\text{pa}} = \{ \text{statisticInfo} \} \).

We use this simple Gofish multiagent system to illustrate our approach in this chapter.\(^1\)

\(^1\)For the full specification of the Gofish MAS as a planning problem in \( \text{DLV}^K \) notation, we refer to Appendix A.
3.4 Modelling Agent Behaviour via Planning

The previous section presented a simplified Gofish post office MAS and its message protocol. The Gofish agents are communicating via messages. Consequently, if we want to monitor the collaborative behaviour of the other agents without full insight in the actual internal states of the other agents all we can do is to gather information about the message flow.

We now discuss how to formalise the intended collaborative behaviour of agents as an action theory for planning that encodes a legal message flow. In it, actions correspond to messages and fluents represent assumptions about the current state of the world.

Under suitable encodings, we could use planning formalisms like STRIPS [Nil71], PDDL [GHK+98] or HTN [EHN94] based planners to model simple agent environments. In fact, HTN planning has recently been incorporated in a MAS [DMANZ02] and formulated as action theories in logic programming [DKN02]. Another powerful language suitable for modelling control knowledge and plans for agents is Golog [LRL+97]. However, due to its high expressive power (loop, conditionals) automated plan generation is limited in this formalism. In this section, we first give a generic formulation of our approach, independent of a particular planning mechanism. Then, we instantiate this high-level description using the action language $\mathcal{K}$ [EFL04, EFL+03b]. While our approach does not rely on $\mathcal{K}$, we have chosen it because of its declarative nature and its capabilities of dealing with incomplete knowledge and nondeterminism.

Modelling Intended Behaviour of a MAS

Our approach to formalise the intended collaborative behaviour of a MAS consisting of agents $A = \{a_1, \ldots, a_n\}$ as a planning problem $\mathcal{P}$ comprises three steps:

**Step 1: Actions ($\text{Act}$).** We declare a set of actions such that for each message $m = (s, r, c, d)$ in our domain, we have $c(s, r, d) \in \text{Act}$ (see Def. 3.3.1). Again, if the message repertoires $C_i$ are pairwise disjoint and each message type $c$ has a unique recipient, we simply write $c(d)$. These actions might have effects on the states of the agents involved and will change the properties that hold on them.
3.4. MODELLING AGENT BEHAVIOUR VIA PLANNING

**Step 2: Fluents (Fl).** We define properties, *fluents*, of the “world” that are used to describe action effects. We distinguish between the sets of *internal* fluents, $F_{la}$, of a particular agent $a$ and *external* fluents, $F_{lext}$, which cover properties not related to specific agents. These fluents are often closely related to the message performatives $C_i$ of the agents.

**Step 3: Theory (T) and Goal (G).** Using the fluents and actions from above, we state various axioms about the collaborative behaviour of the agents as a *planning theory* $T$. The axioms describe how the various actions change the state and under which assumptions they are executable. Finally, we state the ultimate *Goal* $G$ (in the running scenario: to deliver the package) suitable for the chosen planning formalism.

We end up with a *planning problem* $P = \langle Act, Fl, T, G \rangle$, where $Fl = \bigcup_{a \in A} F_{la} \cup F_{lext}$, whose solutions are plans denoted by $P$-Plans. Note that the precise formulation of these notions depends on the underlying planning formalism. For example, in HTN planning one has to specify *operators* and *methods* and their effects (this is closely related to actions $Act$ and fluents $Fl$ above), as well as a domain description and a task list (which corresponds to theory $T$ and goal $G$ above); we refer to [DKN02] for a full discussion.

The above description is a generic formulation suitable for many planning frameworks. We shall consider planning at an abstract level in these frameworks in Section 3.6. In the remainder of this section, we turn to the particular planning framework built around the language $\mathcal{K}$.

**Using Action Language $\mathcal{K}$**

In this section, we instantiate the planning problem $P$ described above to a problem $P^\mathcal{K}$ formulated in the action language $\mathcal{K}$. Rather than giving a detailed review of the language $\mathcal{K}$ and its planning framework, we describe here for space reasons only the key features and refer to [EFL+04, EFL+03b] for further details.

The language $\mathcal{K}$ (where $\mathcal{K}$ stands for planning with knowledge states) is a member of a family of logic-based action languages in the area of knowledge representation and reasoning. These languages aim at providing a flexible, declarative formalism for reasoning about actions and their effects, on which planning

---

2Internal fluents especially can describe private values which might be inaccessible by an external observer.
systems might be built. Prominent languages in this family are the languages $A$ [GL93] and $C$ [GL98]. Compared with these languages $K$ is closer to logic programming than to classical logic, since it includes respective features (e.g., default negation and strong negation). In a nutshell, $K$ offers the following distinguishing features:

- **handling incomplete knowledge:** for a fluent $f$, in a state neither $f$ nor $\neg f$ may be known.
- **nondeterministic effects:** actions may have multiple possible outcomes.
- **optimistic and secure (conformant) planning:** construction of “credulous” plans or “sceptical” plans which work in all cases.
- **parallel actions:** more than one action may be executed simultaneously.

In $K$, an action domain is defined by the static background knowledge $BK$, which specifies a finite set of static facts through a non-monotonic logic program in a function-free first-order language, and a dynamic action description, $AD$. Actions and fluents, $p$, are defined by declarations of the form

$$p(X) \text{ requires } bk_1(Y_1), \ldots, bk_m(Y_m)$$

where $X = X_1, \ldots, X_n$ is a list of parameters, each of which must be typed by some predicates $bk_1, \ldots, bk_m$, which are defined in the background knowledge $BK$. In addition, to specify action executions and effects, $K$ allows to state axioms of the following forms:

1. Rule (1) says that if $\alpha$ is known to be true in the current state and $\beta$ is true in the previous state, then fluent $f$ is known to be true in the current state. Both the if part and the the after can be empty, which means that they are true.
2. Rule (2) simulates *nondeterministic* effects: it means that fluent $f$ is either true or false if $\alpha$ is true in the current state and $\beta$ is true in the previous state.

3. Rule (3) models *inertia* of a fluent $f$: it is a macro for $\text{caused } f \text{ if not } \lnot . f \text{ after } f$, where $\text{not}$ is default negation and $\lnot$ is strong negation. It means that fluent $f$ holds in the current state if it is known to be true in the previous state unless $\lnot . f$ has been explicitly derived.

4. Rule (4) says that an action $a$ is *executable* if $\beta$ holds. The if part can be empty, which means that the action $a$ is always executable.

5. Rule (5) expresses that execution of an action $a$ is *forbidden* if $\beta$ is true. In case of conflicts, *nonexecutable $a$* overrides *executable $a$*.

A goal $G$ is a conjunction of ground fluent literals. A plan for a goal is a sequence of actions leading from the initial state to a state where all fluent literals in the goal are true.

A planning problem $\mathcal{P}^{K}$ in $\mathcal{K}$ can formalised as a tuple $\langle \text{Act, Fl, T, G} \rangle$, where $\text{Act}$ defines the actions, $\text{Fl}$ the fluents, $\text{T}$ comprises background knowledge $BK$ and all axioms (of the sorts introduced above), and $G$ is the goal.

The semantics of $\mathcal{K}$ is defined through transitions $t = \langle s, A, s' \rangle$ from states $s$ to states $s'$ by simultaneous execution of an action $A$, where a state $s$ is any consistent set of ground fluent literals. Roughly, the action description yields a non-monotonic logic program which computes the possible successor states $s'$ from $s$ and $A$ in its models. For a full discussion of action language $\mathcal{K}$, we refer to [EFL+04, EFL+03b].

After defining state transitions, we now describe plans as suitable sequences of state transitions leading from an initial state to a state which satisfies a predefined goal.

**Definition 3.4.1 (Optimistic plan).** A legal transition sequence is any initial state $s_0$ or sequence $t_1, \ldots, t_n$ of transitions $t_i = \langle s_{i-1}, A_i, s_i \rangle$, $i \in \{1, \ldots, n\}$, starting in an initial state $s_0$. An (optimistic) plan for goal $G$ is $P = \langle \rangle$, or the projection $P = \langle A_1, \ldots, A_n \rangle$ of a legal transition sequence, such that $G$ holds in $s_0$, or $s_n$, i.e., $G \subseteq s_0$, or $G \subseteq s_n$, respectively.

\footnote{Note that in $\mathcal{K}$ states are not “total”, that is, a fluent $f$ can be neither true nor false in a state.}
We might be in a situation where we do not exactly know the current state of the world: we only know a set of states we might be in. An optimistic plan is then a sequence of actions projected from a trajectory which reaches the goal. The term “optimistic” should stress the credulous view—executing actions according to an optimistic plan does not guarantee to always reach the goal due to incomplete information and possible alternative transitions.

As the execution of an optimistic plan does not promise that the goal will be reached, we now give the notion of secure plans (or conformant plans)\cite{GB96} as follows.

**Definition 3.4.2 (Secure plan).** An optimistic plan \(\langle A_1, \ldots, A_n \rangle\) is a secure plan, if regardless of the initial state \(s_0\) and the outcomes of the actions, the steps of the plan will always be executable one after the other and reach the goal. Thus, a secure plan is a sequence of actions projected from all legal transition sequences from the current set of states to the goal states.

An operational prototype of a planning system for \(\mathcal{K}, \text{DLV}^\mathcal{K}\), built as frontend on top of the \text{DLV} system \cite{EFLP00}, is available at http://www.dbai.tuwien.ac.at/proj/dlv/. The definitions in the following example are in \text{DLV}^\mathcal{K} notation.

**Example 3 (Simple Gofish continued).** In the Gofish example, the following \(\mathcal{K}\) actions (corresponding to the possible messages) and fluents are defined:

\[
\text{actions: } \begin{cases}
dropOff(P) \text{ requires pkg}(P) \\
addPkg(P) \text{ requires pkg}(P) \\
distCenter(P) \text{ requires pkg}(P) \\
truck(P) \text{ requires pkg}(P) \\
delivery(P) \text{ requires pkg}(P) \\
pickup(P) \text{ requires pkg}(P) \\
setDelivTime(P) \text{ requires pkg}(P)
\end{cases} \quad \text{Act}
\]

\[
\text{fluents: } \begin{cases}
pkgAt(P, Loc) \text{ requires pkg}(P), loc(Loc) \\
delivered(P) \text{ requires pkg}(P) \\
recipAtHome(P) \text{ requires pkg}(P) \\
added(P) \text{ requires pkg}(P) \\
delivTimeSet(P) \text{ requires pkg}(P)
\end{cases} \quad \text{Fl}
\]

The first three external fluents describe the current location of a package, whether it has been successfully delivered, and, whether its recipient is at home, respectively. The last two fluents are internal fluents about the state of agent pa.
3.4. MODELLING AGENT BEHAVIOUR VIA PLANNING

describing whether the package has already been added to the package database and whether the delivery time has been set properly, respectively.

A possible package represented by a generic $p_1$ and its location form the background knowledge represented by the set of facts $BK=\{pkg(p_1), loc(drop), loc(dist), loc(truck)\}$. Now we specify further axioms for $T$ (in DLV$^K$ notation) as follows:

- **initially**: `recipAtHome(p_1)`.
- **always**: `noConcurrency`.
- **inertial**: `pkgAt(P, Loc)`, `delivered(P)`, `recipAtHome(P)`, `added(P)`.
- **executable**: `dropOff(P) if not added(P)`.
- **caused**: `pkgAt(P, drop) after dropOff(P)`.
- **nonexecutable**: `dropOff(P) if pkgAt(P, drop)`.
- **executable**: `addPkg(P) if pkgAt(P, drop), not added(P)`.
- **caused**: `added(P) after addPkg(P)`.
- **executable**: `distCenter(P) if added(P), pkgAt(P, drop)`.
- **caused**: `pkgAt(P, dist) after distCenter(P)`.
- **caused**: `-pkgAt(P, drop) after distCenter(P)`.
- **executable**: `truck(P) if pkgAt(P, dist), not delivered(P)`.
- **caused**: `pkgAt(P, truck) after truck(P)`.
- **caused**: `-pkgAt(P, dist) after truck(P)`.
- **executable**: `delivery(P) if pkgAt(P, truck), not delivered(P)`.
- **caused**: `delivered(P) after delivery(P), recipAtHome(P)`.
- **executable**: `setDelivTime(P, DTime) if delivered(P)`.
- **caused**: `delivTimeSet(P) after setDelivTime(P)`.
- **executable**: `pickup(P) if pkgAt(P, dist), not delivered(P)`.
- **executable**: `pickup(P) if pkgAt(P, truck), not delivered(P)`.
- **caused**: `delivered(P) after pkgAt(P, dist), pickup(P)`.
- **total**: `recipAtHome(P) after pickup(P)`.

Most of the theory is self-explanatory. The recipient is at home initially. The keyword `noConcurrency` specifies that concurrent actions are disallowed. An important aspect is modelled by the final `total` statement. It expresses uncertainty
whether after a pickup attempt at the distribution centre, the recipient will be back home, in particular in time before the truck arrives to deliver the package, if the truck is already on the way. Finally, the goal is \( G = \text{delivTimeSet}(p_1) \).

The following (optimistic) plans reach \( G \): the goal \( \text{delivTimeSet}(p_1) \) that the package \( p_1 \) is delivered:

\[
\begin{align*}
P_1 &= \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{truck}(p_1); \\
    &\quad \text{pickup}(p_1); \text{delivery}(p_1); \text{setDelivTime}(p_1) \rangle \\
P_2 &= \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{truck}(p_1); \\
    &\quad \text{delivery}(p_1); \text{setDelivTime}(p_1) \rangle \\
P_3 &= \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{pickup}(p_1); \text{setDelivTime}(p_1) \rangle
\end{align*}
\]

In \( P_1 \), the recipient shows up at the distribution centre after the package is loaded on the truck and the truck is on its way. In \( P_2 \), the package is successfully delivered before the recipient comes to pick it up herself, whereas in \( P_3 \), she picks up the package before it has been loaded on the truck. Each of these plans might work out fine, but also might fail.

**Example 4 (Running scenario).** We assume the following entries in the message log:

\[
\begin{align*}
\mathcal{M}_{\log} &= 0: \langle \text{disp}, \text{em}, \text{dropOff}, p_1 \rangle, \\
      &\quad 5: \langle \text{em}, \text{pa}, \text{addPkg}, p_1 \rangle, \\
      &\quad 13: \langle \text{disp}, \text{em}, \text{distCenter}, p_1 \rangle, \\
      &\quad 19: \langle \text{disp}, \text{em}, \text{truck}, p_1 \rangle, \\
      &\quad 20: \langle \text{disp}, \text{em}, \text{pickup}, p_1 \rangle.
\end{align*}
\]

According to the message history in \( \mathcal{M}_{\log} \), we can see that plan \( P_2 \) is infeasible, as well as \( P_3 \) since the package can not be handed over to Sue at the distribution centre. Thus, only \( P_1 \) remains for successful task completion.

### 3.5 Agent monitor

The overall aim of adding a monitoring agent (monitor) is to aid in debugging a given MAS. We can distinguish between two principal types of errors: (1) **design errors**, and (2) **implementation (or coding) errors**:

- The first type means that the model of the system is wrong (that is, the MAS behaves correctly to the model of the designer of the MAS, but this model is faulty and does not yield the desired result in the application).
• The second type points to more mundane mistakes in the actual code of the agents: the code does not implement the formal model of the system—the actions are not implemented correctly.

Note that often it is very difficult, if not impossible at all, to distinguish between design and implementation errors. But even before the system is deployed, the planning problem $\mathcal{P}$ can be given to a planner and thus the overall existence of a solution can be checked. If there is no solution, this is clearly a design error and the monitoring agent can pinpoint where exactly the planning fails (assuming the underlying planner has this ability). If there are solutions, the agent designer can check them and thus critically examine the intended model.

However, for most applications the bugs in the system become apparent only at runtime. Our proposed monitoring agent has the following structure.

Definition 3.5.1 (Structure of agent monitor). The agent monitor loops through the following steps:

1. Read and parse the message log $\mathcal{M}_{log}$. If the message log is empty ($\mathcal{M}_{log} = \emptyset$), the set of all possible plans for $\mathcal{P}$ may be cached for later reuse.

2. Check whether an action timeout has occurred.

3. If this is not the case, compute the current intended plans (according to the planning problem description and additional information from the designer) compatible with the actions as executed by the multiagent system.

4. If no compatible plans survive, or the system is no more idle, then inform the agent designer about this situation.

5. Sleep for some pre-specified time.

We now elaborate more deeply on these tasks.

Checking MAS behaviour: monitor continually keeps track of the messages sent between the agents. They are stored in the message log, $\mathcal{M}_{log}$, which is accessible by monitor. Thus for monitor, the behaviour of the MAS is completely determined by $\mathcal{M}_{log}$. We think this is a realistic abstraction from internal agent states. Rather than describing all the details of each agent (which might be unknown, e.g. if legacy agents are involved), the kinds of messages sent by an agent can be chosen so as to give a declarative high-level view of it. In the simplified
The desired collaborative MAS behaviour is formalised as a planning problem $\mathcal{P}$ (in language $\mathcal{K}$, as shown in Section 3.4). Thus, even before the MAS is in operation, problem $\mathcal{P}$ can be fed into a planner which computes potential plans to reach a goal. Agent \textit{monitor} is exactly doing that.

In general, not all solutions of the planning problem, $\mathcal{P}$-Plans, may be admissible, as constraints may apply (derived from the intended collaborative behaviour).\textsuperscript{4} For instance, some actions ought to be taken in a fixed order, or actions may be penalised with costs whose sum must stay within a limit. We thus distinguish a set $I$-Plans($\mathcal{P}$) $\subseteq$ $\mathcal{P}$-Plans as \textit{intended plans} (of the MAS designer).

It is perfectly possible that the original problem has successful plans, yet after some actions executed by the MAS, these plans are no longer valid. This is the interesting case for the agent designer since it clearly shows that something has gone wrong: \textit{monitor} can pinpoint the precise place indicating which messages have caused the plan to collapse. Because these messages are related to actions executed by the agents, information about them will help to debug the MAS. In general, it is difficult to decide whether the faulty behaviour is due to a coding or design error. However, the information given by \textit{monitor} will aid the agent designer in detecting the real cause.

\textbf{Messages from \textit{monitor}:} Agent \textit{monitor} continually checks and compares the actions taken so far for compatibility with all current plans. Once a situation has arisen in which no successful plan exists (detected by the planner employed), \textit{monitor} writes a message into a separate file containing (1) the first action that caused the MAS to go into a state where the goal is unreachable, (2) the sequence of actions taken up to this action, and (3) all the possible plans before the action in 1) was executed (these are all plans compatible with the MAS behaviour up to it).

In the above description, we made heavily use of the notion of a \textit{compatible} plan. Before giving a formal definition, we consider our running scenario. In \textit{Gofish}, all three plans $P_1$, $P_2$, $P_3$ generated from the initial problem coincide on the first three steps: dropOff($p_1$), addPkg($p_1$), and distCenter($p_1$).

\textsuperscript{4}This might depend on the capabilities of the underlying planning formalism to model constraints such as state axioms, cost bounds, or optimality with respect to resource consumption etc.
Example 5 (Running scenario (coding error)). Suppose on a preliminary run of our scenario, $M_{\log}$ shows that $m_1 = \text{dropOff}(p_1)$. This is compatible with each plan $P_i, i \in \{1, 2, 3\}$. Next, $m_2 = \text{distCenter}(p_1)$. This is incompatible with each plan; monitor detects this and gives a warning. Inspection of the actual code may show that the command of adding the package to the database is wrong. While this doesn't result in a livelock (the MAS is still idle), the database was not updated. Informed by monitor, this flaw is detected at this stage already.

After correction of this coding error, the MAS may be started again and another error shows up as follows:

Example 6 (Running scenario (design error)). Instead of waiting at home (as in the “standard” plan $P_2$), Sue shows up at the distribution centre and makes a pickup attempt. This “external” event may have been unforeseen by the designer (problematic events could also arise from MAS actions). We can expect this in many agent scenarios: we have no complete knowledge about the world; unexpected events may happen; and, action effects may not fully determine the next state.

Only plan $P_1$ remains to reach the goal. However, there is no guarantee of success, if Sue is not back home in time. This situation can be easily captured in the framework of $[EFL+04, EFL+03b]$, where we have the notion of a secure plan (see Definition 3.4.2). In secure plans, no matter of the initial state and the outcomes of the actions, the steps of the plan will always be executable one after the other and reach the goal, that is, in all trajectories. As can be easily seen, $P_2$ and $P_3$ are secure plans, while $P_1$ is not secure. Thus, a design error is detected, if delivering the package must be guaranteed under all circumstances.

Based on a generic planning problem $P$, we now define compatible plans as follows. Let $\mathcal{P}$-Plans denote the set of all plans for $P$.

**Definition 3.5.2 ($M_{\log}$ compatible plans).** Let the planning problem $P$ model the intended behaviour of a MAS, which is given by a set $I$-Plans$(P) \subseteq \mathcal{P}$-Plans. Then, for any message log $M_{\log} = t_1, m_1, \ldots, t_k, m_k$, we denote by $C$-Plans$(P, M_{\log}, n), n \geq 0$, the set of plans from $I$-Plans$(P)$ which comply on the first $n$ steps with the actions $m_1, \ldots, m_n$, i.e., for a plan $P \in I$-Plans$(P)$ and $P = \langle A_1, \ldots, A_n, \ldots \rangle$, $A_i = c_i \ (c_i \in m_i)$ holds for $i \leq n$.

In a planning framework with different notions of plans, $\mathcal{P}$-Plans is assumed to comprise the most liberal notion of plan. For example, in $\text{DLV}^K$, the planner
for $\mathcal{K}$, optimistic and secure plans can be computed for any problem $\mathcal{P}^\mathcal{K}$, and $\mathcal{P}$-Plans would consist of all optimistic plans.

**Definition 3.5.3 (Culprit($M_{\log}, \mathcal{P}$)).** Let $t_n.m_n$ be the first entry of $M_{\log}$ such that either (i) $C$-Plans($\mathcal{P}, M_{\log}, n$) = $\emptyset$ or (ii) a timeout is detected. Then, Culprit($M_{\log}, \mathcal{P}$) is the pair $\langle t_n.m_n, \text{idle} \rangle$ if (i) applies and $\langle t_n.m_n, \text{timeout} \rangle$ (resp. $\langle \text{timeout} \rangle$ if $M_{\log}$ is empty) otherwise.

Initially, $M_{\log}$ is empty and thus $C$-Plans($\mathcal{P}, M_{\log}, 0$) = $I$-Plans($\mathcal{P}$). As more and more actions are executed by the MAS, they are recorded in $M_{\log}$ and the set $C$-Plans($\mathcal{P}$) shrinks. Agent monitor can thus check at any point in time whether $C$-Plans($\mathcal{P}, M_{\log}, n$) is empty or not. Whenever this happens, Culprit($M_{\log}, \mathcal{P}$) is computed and pinpoints the problematic action.

**Example 7 (Running scenario).** Under guaranteed delivery (i.e., under secure planning), agent monitor writes $\text{Culprit}(M_{\log}, \mathcal{P}) = \langle 20.m_5, \text{idle} \rangle$ (the pickup($p_1$) message) in a file, and thus clearly points to a situation missed in the MAS design. Note that there are also situations where everything is fine; if pickup would not occur, agent monitor would not detect a problem at this stage.

In order to monitor the agents messages we then have to decide whether the current course of messages is compatible with a valid plan for the respective planning problem. Desiderata include:

- being able to detect where an error occurred in case of divergence or

- preventing possible errors in advance.

**Example 8 (Simple Gofish extended).** We now consider the extension of the previous simple Gofish example by adding a customer notification service. That is, the Gofish postal service performs mail tracking in order to be able to notify customers about the status of mail delivery. Each recipient of a package is notified about the arrival of the package at the distribution centre and when the package has been loaded on a truck for delivery.

The realisation of the notification service in our MAS brings the notification agent (notify) into play. The notification agent is informed by the event management agent (em) about the arrival of a package, as well as its loading on a truck. In both cases agent notify contacts the package agent (pa) in order to obtain the required customer information. For simplicity, we subsume both
messages—from em to notify and from notify to pa—into a single message getRecipInfo which is parameterised by the corresponding event dist or truck, i.e., \( C_{\text{notify}} = \{ \text{getRecipInfo} \} \). The package agent replies to the requests of notify with the corresponding information to notify the customer, e.g., the email address of the recipient in case of dist and the phone number in case of truck. Thus now, \( C_{\text{pa}} = \{ \text{recipInfo} \} \).

In order to reflect this extension in our model, we add the following actions for the newly introduced messages and another fluent:

**actions:**
- getRecipInfo\((P, \text{Loc})\) requires pkg\((P)\), loc\((\text{Loc})\).
- recipInfo\((P, \text{Loc})\) requires pkg\((P)\), loc\((\text{Loc})\).

**fluents:**
- informed\((P, \text{Loc})\) requires pkg\((P)\), loc\((\text{Loc})\).

The new external fluent informed captures the state of the customer concerning her knowledge about the package status. The effects and executability conditions of the new actions are defined as follows:

- inertial informed\((P, \text{Loc})\).
- inertial -informed\((P, \text{Loc})\).
- executable getRecipInfo\((P, \text{dist})\) if pkgAt\((P, \text{dist})\).
- executable getRecipInfo\((P, \text{truck})\) if pkgAt\((P, \text{truck})\).
- caused -informed\((P, \text{Loc})\) after getRecipInfo\((P, \text{Loc})\).
- nonexecutable getRecipInfo\((P, \text{Loc})\) if informed\((P, \text{Loc})\).
- nonexecutable getRecipInfo\((P, \text{Loc})\) if -informed\((P, \text{Loc})\).
- executable recipInfo\((P, \text{Loc})\) if pkgAt\((P, \text{Loc})\), -informed\((P, \text{Loc})\).
- caused informed\((P, \text{Loc})\) after pkgAt\((P, \text{Loc})\), recipInfo\((P, \text{Loc})\).
- nonexecutable recipInfo\((P, \text{Loc})\) if informed\((P, \text{Loc})\).

Furthermore, we modify the axioms for the delivery and the pickup action:

- executable delivery\((P)\) if pkgAt\((P, \text{truck})\), informed\((P, \text{truck})\), notdelivered\((P)\).
- executable pickup\((P)\) if pkgAt\((P, \text{dist})\), notinformed\((P, \text{truck})\), notdelivered\((P)\).

The customer must be informed about the package being delivered by a truck before delivery, thus she will no longer show up at the distribution centre for picking up the package.
In general, because of the “nondeterminism” of the external event of a customer showing up at a distribution centre for picking up a package, we will now obtain more plans that reach the goal $G = \text{delivTimeSet}(p_i)$. For example, the following (optimistic) plans are variants of the plan $P_1$:

$$P_{1,1} = \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{truck}(p_1); \text{getRecipInfo}(p_1, \text{truck}); \text{pickup}(p_1); \text{recipInfo}(p_1, \text{truck}); \text{delivery}(p_1); \text{setDelivTime}(p_1) \rangle$$

$$P_{1,2} = \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{truck}(p_1); \text{pickup}(p_1); \text{getRecipInfo}(p_1, \text{truck}); \text{recipInfo}(p_1, \text{truck}); \text{delivery}(p_1); \text{setDelivTime}(p_1) \rangle$$

Note that still the customer may show up at the distribution centre after the package has been loaded on a truck. However, this can no longer be the case after the customer has been notified. Moreover, it is assumed that the customer notification takes place while or shortly after loading the package on the truck.

**Example 9 (Running scenario).** Consider a run of our scenario in the extended MAS and suppose the following sequence of messages in the message log:

$M_{log} = 0\langle \text{disp, em, dropOff, } p_1 \rangle, 5\langle \text{em, pa, addPkg, } p_1 \rangle,$

$13\langle \text{disp, em, distCenter, } p_1 \rangle, 14\langle \text{em, pa, getRecipInfo, } (p_1, \text{dist}) \rangle,$

$15\langle \text{pa, em, recipInfo, } (p_1, \text{dist}) \rangle, 19\langle \text{disp, em, truck, } p_1 \rangle,$

$20\langle \text{em, pa, getRecipInfo, } (p_1, \text{truck}) \rangle, 22\langle \text{pa, em, recipInfo, } (p_1, \text{truck}) \rangle.$

Then, everything is fine even under secure planning, i.e. guaranteed delivery, since pickup cannot occur after Sue has been notified that her package has been loaded on truck for delivery. That is, $M_{log}$ is compatible with the secure plan

$$P_{2,1} = \langle \text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{truck}(p_1); \text{getRecipInfo}(p_1, \text{truck}); \text{recipInfo}(p_1, \text{truck}); \text{delivery}(p_1); \text{setDelivTime}(p_1) \rangle.$$  

However, if Sue had shown up at the distribution centre before notification, i.e. $m_8 = \langle \text{pickup, } p_1 \rangle$ say again at time 21, the package would no longer be guaranteed delivered on time, and monitor would again write $\text{Culprit}(M_{log}, P) = \langle 21, m_8, \text{idle} \rangle$ to a file to indicate the problematic situation.

To sum up, by extending our postal service with customer notification we reduced the probability of unsuccessful deliveries but we did not achieve guaranteed
delivery. We remark, however, that in our example setting this could easily be obtained by disallowing customer pickups.

3.6 Properties

In this section, we take a closer look at our agent monitoring approach and show that it has some desirable properties. To this end, we shall need some preliminary definitions in order to make the notions we have used above formally more precise.

As for the underlying planning framework, referred to as $\mathcal{PF}$, we assume that the basic element for the semantics of plan execution is given by trajectories in the planning world formed by state transitions, similar to the semantics of the planning language $\mathcal{K}$. That is, we assume that there is a set of possible world states, $S_{\mathcal{PF}}$ (where world states $s$ are described e.g. by fluents) and a set of actions $A_{\mathcal{PF}}$ in $\mathcal{PF}$, as well as a set $I_{\mathcal{PF}} \subseteq S_{\mathcal{PF}}$ of initial states. Furthermore, there is a partial, multi-valued transition function $tr_{\mathcal{PF}} : S_{\mathcal{PF}} \times A_{\mathcal{PF}} \rightarrow 2^{S_{\mathcal{PF}}}$ which assigns a set of possible successor states $tr_{\mathcal{PF}}(s, A)$ to a state $s \in S_{\mathcal{PF}}$ and an action $a \in A_{\mathcal{PF}}$ to be executed in $s$; the transition $tr_{\mathcal{PF}}$ might be undefined, however, or no successor state may exist.

**Definition 3.6.1.** A trajectory $T$ in $\mathcal{PF}$ is a sequence $s_0, a_1, s_1, \ldots, a_n, s_n$ of states $s_i \in S_{\mathcal{PF}}$ and actions $a_i \subseteq A_{\mathcal{PF}}$, $n \geq 0$, such that $s_0 \in I_{\mathcal{PF}}$ and $s_i \in tr_{\mathcal{PF}}(s_{i-1}, a_i)$, for every $i \in \{1, \ldots, n\}$.

We view plans for reaching a goal $G$ in $\mathcal{PF}$, which in general is some constraint on the desired states, from a semantical perspective, as structures corresponding to the trajectories in the planning world which are compatible with them.\(^5\)

More formally,

**Definition 3.6.2.** Any plan $P$ in $\mathcal{PF}$ is an object which has associated with it a nonempty set of trajectories in $\mathcal{PF}$, $Sem(P)$, such that $G$ holds in state $s_n$ for each $T \in Sem(P)$ where $T = s_0, a_1, s_1, \ldots, a_n, s_n$.

By way of illustration, in the $\mathcal{K}$ planning framework, the transition function $tr_{\mathcal{K}}$ is implicit by the definition of state transitions, that is, $tr_{\mathcal{K}}(s, A)$ is defined for a state $s$ and a set of actions $A$ (which we can view as a single compound

---

\(^5\)Note the terms for descriptions depend on the planning languages applied, here sometimes we informally use the terminology of language $\mathcal{K}$ in order to present the general, underlying planning framework.
3.6. PROPERTIES

action) if and only if, in the terminology of $K$, $A$ is executable with respect to $s$ and $\text{tr}_K(s, A) = \{s' \mid \langle s, A, s' \rangle\}$ is a state transition} in this case. An optimistic plan $P$ in $K$ for the goal $G$ is then semantically characterised by the condition that there is a sequence of actions, $\langle A_1, \ldots, A_n \rangle$, $n \geq 0$, such that (i) each $T \in \text{Sem}(P)$ is of form $s_0, A_1, s_1, \ldots, A_n, s_n$ and (ii) each trajectory $s_0, A_1, s_1, \ldots, A_n, s_n$ which establishes the goal $G$ is in $\text{Sem}(P)$. Furthermore, a secure plan $P$ is in $K$ an optimistic plan which satisfies in addition that (iii) for each trajectory $T' = s_0, A_1, s_1, \ldots, A_m, s_m$ with $m \leq n$, the goal $G$ is established if $m = n$, and $\text{tr}(s_m, A_{m+1})$ is defined and nonempty otherwise.

As for the multiagent system $M$ in question, we assume that its collaborative behaviour is governed by some strategy, $S$. We take here also a pure semantical view and project $S$ to the set of possible runs which might be observed during execution (in particular, by the agent tester). Formally, a run is defined as follows.

**Definition 3.6.3.** Given a MAS $M$, we assume there is an underlying set of possible system states, $S_M$. A run is a sequence $R = S^0, m_1, S^1, \ldots, m_k, S^k$, $k \geq 0$, of (global) system states $S^i$ and messages $m_j$, where $S^0$ is the initial state (respectively, from the set of possible initial states in case of indeterminism).

Informally, upon message $m_i$, the system transits from state $S^{i-1}$ to state $S^i$; here, we abstract from concrete time points when messages are sent. The collaboration goal of a MAS is a success-predicate on global states. We denote the set of possible runs under obedience of strategy $S$ by $\text{Runs}(S)$, which is assumed to be nonempty. Each such run $R$ must establish the collaboration goal, that is, $S^k$ must satisfy it given $R = S^0, m_1, S^1, \ldots, m_k, S^k$.

In order to account for the case that we do not know the precise collaboration strategy $S$ adopted by $M$ (e.g., this could be negotiated in a startup phase), we model the intended behaviour by a nonempty set $\mathcal{IS}(M)$ of possible strategies $S$. A run $R$ is admissible, if it is possible for some $S \in \mathcal{IS}(M)$, i.e., $R \in \bigcup_{S \in \mathcal{IS}(M)} \text{Runs}(S)$.

The planning framework, $\mathcal{PF}$, and the MAS $M$ are linked by the basic Modelling Assumption that runs in $M$ gracefully correspond to trajectories in $\mathcal{PF}$ and vice versa such that $\mathcal{PF}$ models evolutions in $M$, and that the planning goal corresponds to the collaboration goal. This is made precise as follows.

**Modelling Assumption:**
1. There is a one-to-one correspondence, \( f \), between messages \( m \) and actions \( a \), \( f(m) = a \), and a correspondence, \( g \), (not necessarily one-to-one) between states \( S \in S_M \) in the agent system and states \( s \in S_{\mathcal{PF}} \) in the planning framework, \( S \xrightarrow{g} s \), such that the initial states in \( M \) and \( \mathcal{PF} \) correspond to each other;

2. there is a fixed planning goal, \( G \), defining a planning problem, \( \mathcal{P} \), such that the states in \( \mathcal{PF} \) fulfilling \( G \) and the states in \( M \) establishing the collaboration goal correspond to each other; and

3. the correspondence homomorphically extends to transitions in runs in \( M \) and transitions in trajectories in \( \mathcal{PF} \), respectively. That is, for any \( S^{i-1}, m_i, S^i \) in a run, we have that \( g(S^{i-1}), f(m_i), g(S^i) \) is part of a trajectory in \( \mathcal{PF} \), and conversely, for any \( s_{i-1}, a_i, s_i \) in a trajectory, \( g^{-1}(s_{i-1}), f^{-1}(a_i), g^{-1}(s_i) \) is part of a run of \( M \).

Conditions 2 and 3 of the Modelling Assumption aim at allowing abstraction in the encoding of the multiagent system in the planning formalism; note that no one-to-one correspondence between trajectories and runs is requested. For example, fluents in the multiagent system might be disregarded in the planning formulation, such that states in the multiagent system with different fluents values correspond to the same state in the planning world. On the other hand, the planning formulation might include fluents which do not correspond to fluents in the multiagent system and whose value is immaterial for the intended monitoring task. These fluents can be projected away, leading to a possible many-to-one correspondence from states in the planning world to states of the multiagent system. We emphasise that some assumption on the correspondence between, on the one hand, states and runs in the multiagent system and, on the other hand, states and trajectories is mandatory for proving meaningful results.

We shall denote the solutions (plans) for the planning problem \( \mathcal{P} \) by \( \mathcal{P}\text{-Plans} \). Furthermore, we shall occasionally simply write \( m \xrightarrow{*} A, S \xrightarrow{*} s, \) and \( R \xrightarrow{*} T \) for appropriate objects corresponding via \( g \) and \( f \).

The correspondence \( \equiv \) induces a notion of similarity \( \simeq_{\equiv} \) (for short, \( \simeq \)) among runs by \( R \simeq_{\equiv} R' \) if and only if there is some trajectory \( T \) in \( \mathcal{PF} \) such that \( R \Rightarrow T \) and \( R' \Rightarrow T \). In order to get meaningful results, we assume that collaboration strategies \( \mathcal{S} \) are closed under similarity \( \simeq \); that is, whenever \( R \in \text{Runs}(\mathcal{S}) \) and \( R \simeq R' \), then also \( R' \in \text{Runs}(\mathcal{S}) \) holds. As a consequence, each trajectory corresponds either only to admissible runs or to no admissible run.
After these preliminary definitions, our first result concerns the soundness of the monitoring approach. Let us say that a plan \( P \in \mathcal{P}\mathcal{F} \) models a strategy \( S \), iff each run \( R \in \text{Runs}(S) \) corresponds to some trajectory \( T \in \text{Sem}(P) \) and vice versa, and that a set \( MP = \{ P_i \mid i \in I \} \) of plans models a set of strategies \( SS = \{ S_i \mid i \in I \} \), if each \( P_i \) models \( S_i \).

**Theorem 3.6.4 (Soundness).** Suppose that the set \( I\text{-Plans}(P) \subseteq \mathcal{P}\text{-Plans} \) of intended plans for the planning problem \( P \) in \( \mathcal{P}\mathcal{F} \) models the intended collaborative behaviour of the MAS \( M, IS(M) \). Let \( M_{\mathcal{L}og} \) be a message log. Then, \( M \) is implemented incorrectly (that is, there exists the coding error or design error) if \( \text{Culprit}(M_{\mathcal{L}og}, P) \) exists.

**Proof.** Let \( R = S^{0}, m_1, S^{1}, \ldots, m_k, S^{k} \) be the run which produces \( M_{\mathcal{L}og} = t_1 ; m_1, \ldots, t_k ; m_k \). Consider the two different types of \( \text{Culprit}(M_{\mathcal{L}og}, P) \). Suppose first that it is of form \( \langle t_n ; m_n, \text{timeout} \rangle \) or \( \langle \text{timeout} \rangle \). Then, a timeout has been detected and we have \( n = k \) or \( k = 0 \), respectively. This means that either \( R \) has terminated or that \( M \) is stuck. The monitor agent expects, supported by a trajectory \( T \in \text{Sem}(P) \) for some \( P \in I\text{-Plans}(P) \) with the property that the prefix \( T' = S^{0}, a_1, s_1, \ldots, a_n, s_n \) of \( T \) and the prefix \( R' = S^{0}, m_1, S^{1}, \ldots, m_n, S^{m} \) of \( R \) satisfy \( R' \models T' \). Hence, \( R \notin \bigcup_{S \in IS(M)} \text{Runs}(S) \). Hence, \( M \) is not implemented correctly.

Suppose next that \( \text{Culprit}(M_{\mathcal{L}og}, P) = \langle t_n ; m_n, \text{idle} \rangle \). Then, we have \( 0 < n \leq k \).

By the definition of culprit, we have that \( C\text{-Plans}(P, M_{\mathcal{L}og}, n) = \emptyset \). This means that there is no trajectory \( T = s_0, a_1, s_1, \ldots, a_k, s_k' \) in the planning framework \( \mathcal{P}\mathcal{F} \) such that \( T \in \text{Sem}(P) \) for some \( P \in I\text{-Plans}(P) \) with the property that the prefix \( T' = s_0, a_1, s_1, \ldots, a_n, s_n \) of \( T \) and the prefix \( R' = S^{0}, m_1, S^{1}, \ldots, m_n, S^{m} \) of \( R \) satisfy \( R' \models T' \). Hence, \( R \notin \bigcup_{S \in IS(M)} \text{Runs}(S) \), which again means that \( M \) is not implemented correctly. This proves soundness.

The soundness result of the monitoring approach can be generalised to a setting in which the intended collaborative behaviour \( IS(M) \) of the agents is not exactly modelled by some intended plans in the planning framework, but just cautiously approximated. This is in particular useful if the strategies governing the collaborative behaviour in the MAS \( M \) amount to an expressive notion of plans.
3.6 PROPERTIES

For example, the MAS might employ a conditional plan $P^M$ [PS92] in which depending on conditions $c_1, \ldots, c_k$ on the current state, suitable actions $a_1, \ldots, a_k$ are executed, respectively, and a similar strategy is recursively applied on each case. Conditional plans are very important, since sensing information (observations from the world) can be suitably respected. They are more liberal than secure plans, which do not allow for branching on conditions.

However, the planning framework $PF$ which we employ for agent monitoring might not be capable of conditional planning, such that we cannot model $P^M$ as a respective intended plan; for example, the $K$ planning framework does not support conditional planning. Despite this obstacle, we might employ the planning framework $PF$ fruitfully for error detection as follows.

**Definition 3.6.5.** Given a planning framework $PF$ and a MAS $M$, we say that a set $CP$ of plans in $PF$ covers the intended collaborative behaviour of $M$, $IS(M)$, if for each run $R \in \bigcup_{S \in IS(M)} \text{Runs}(S)$, there exists some plan $P \in CP$ and trajectory $T \in \text{Sem}(P)$ such that $R \equiv T$.

Agent monitor then uses in Step 3 of its procedure from Definition 3.5.1 the cover $CP$ instead of the intended plans $I\text{-Plans}(P)$. We then write $C\text{-Plans}(CP, M_{\log}, n)$, $\text{Culprit}(M_{\log}, CP)$ etc.

We have the following result:

**Theorem 3.6.6 (Soundness of Covering).** Suppose the planning problem $P$ in $PF$ is such that its accepted solutions, $CP \subseteq P\text{-Plans}$, cover the intended collaborative behaviour of the MAS $M$, given by $IS(M)$. Let $M_{\log}$ be a message log. Then, the MAS is implemented incorrectly if $\text{Culprit}(M_{\log}, CP)$ exists.

**Proof.** The proof is similar to the proof of Theorem 3.6.4. Let $R = S^0, m_1, S^1, \ldots, m_k, S^k$ be the run which produces $M_{\log} = t_1:m_1, \ldots, t_k:m_k$. Consider the two different types of $\text{Culprit}(M_{\log}, CP)$. Suppose first that it is of form $\langle t_n:m_n, \text{timeout} \rangle$ or $\langle \text{timeout} \rangle$. Then, a time-out has been detected and we have $n = k$ or $k = 0$, respectively. This means that either $R$ has terminated or that $M$ is stuck. The monitor agent expects, supported by a trajectory $T \in \text{Sem}(P)$ for some $P \in C\text{-Plans}(CP, M_{\log}, n)$ ($\neq \emptyset$), such that $T$ is compatible with the messages $m_1, \ldots, m_n$ in $M_{\log}$, that the execution of $M$ continues, i.e., some message $m_{n+1}$ follows. Hence in both cases (whether $M$ is terminated or stuck), $R \not\in \bigcup_{S \in IS(M)} \text{Runs}(S)$. Hence, $M$ is not implemented correctly.
3.6. PROPER TIES

Suppose next that \( \text{Culprit}(M_{\log}, CP) = \langle t_n, m_n, \text{idle} \rangle \). Then, we have \( 0 < n \leq k \). By the definition of culprit, we have that \( \text{C-Plans}(CP, M_{\log}, n) = \emptyset \). This means that there is no trajectory \( T = s_0, a_1, s_1, \ldots, a_k, s_k \) in the planning framework \( \mathcal{PF} \) such that \( T \in \text{Sem}(P) \) for some \( P \in CP \) with the property that the prefix \( T' = s_0, a_1, s_1, \ldots, a_n, s_n \) of \( T \) and the prefix \( R' = S_0, m_1, S_1, \ldots, m_n, S_n \) of \( R \) satisfy \( R' \equiv T' \). Hence, \( R \notin \bigcup_{S \in IS(M)} \text{Runs}(S) \), which again means that \( M \) is not implemented correctly. This proves soundness.

As an immediate corollary, we obtain soundness of agent monitor via optimistic plans in the \( \mathcal{K} \) planning framework:

**Corollary 3.6.7 (Soundness of \( \mathcal{P}^\mathcal{K} \) Cover).** Let \( \mathcal{P}^\mathcal{K} \) be a \( \mathcal{K} \) planning problem, such that the set \( OP \subseteq \mathcal{P}^\mathcal{K} \)-plans of optimistic plans covers the intended collaborative behaviour of the MAS \( M \). Let \( M_{\log} \) be a message log. Then, MAS is implemented incorrectly if \( \text{Culprit}(M_{\log}, OP) \) exists.

In particular, if nothing is known about the collaboration strategy of \( M \), the set \( OP \) might safely be set to \( \mathcal{P}^\mathcal{K} \)-Plans, i.e., all optimistic plans. Then, any behaviour will be covered, including intended behaviour governed by a conditional plan, or by a more restrictive secure plan.

As for completeness of the monitoring method, there is clearly no converse of the soundness result for covers \( CP \) of the intended behaviour in general, since \( CP \) might include a plan \( P \) which has an associated trajectory that masks an inadmissible run of the MAS \( M \); this is the price to pay for overestimating the intended behaviour.

On the other hand, if all trajectories of plans in the cover \( CP \) correspond to admissible runs of the MAS, then the cover allows to unveil an incorrect MAS implementation, provided certain conditions are met.

As for a general completeness result, we adopt the following assertions. The first is that successful runs can not grow arbitrarily long, i.e., they have a (known) upper bound on their length. It is important to note that all admissible runs should be bounded. If there is an unbounded admissible run \( R \in \bigcup_{S \in IS(M)} \text{Runs}(S) \), then according to the Modelling Assumption there must exist some unbounded trajectory \( T \in \text{Sem}(P) \) for some plans \( P \in CP \) such that \( R \equiv T \). Thus at some time \( t \), it is possible that there is a trajectory \( T' = R' \) and \( R' \notin \bigcup_{S \in IS(M)} \text{Runs}(S) \) such that \( T'(t) = T(t) \). Thus no completeness result is possible if some admissible runs are unbounded. The second assertion concerns the evolution of the
3.6. PROPERTIES

MAS with respect to the particular mechanism of message logging we build on, which does not foresee recording state information about $M$. From the messages in $\mathcal{M}_{\text{log}} = t_1:m_1, \ldots, t_k:m_k$ alone, it is in general impossible to infer the state of the agent system $M$. We thus assert for $M$ the property that any runs $R = S^0, m_1, S^1, \ldots, m_k, S^k$ and $\bar{R} = \bar{S}^0, m_1, \bar{S}^1, \ldots, m_k, \bar{S}^k$ are similar, i.e., $R \simeq \bar{R}$ holds; we say that $M$ has one-way logging. For example, one-way logging is guaranteed in agent systems with deterministic message effects and a single initial state.

Let us call a cover $CP$ for $\mathcal{I}\mathcal{S}(M)$ exact, if for each $P \in CP$ and each $T \in \text{Sem}(P)$, there exists some strategy $S \in \mathcal{I}\mathcal{S}(M)$ and run $R \in \text{Runs}(S)$ such that $T$ corresponds to $R$.

**Theorem 3.6.8 (Completeness).** Let the planning problem $\mathcal{P}$ in $\mathcal{P}\mathcal{F}$ be such that the set $CP \subseteq \mathcal{P}$-plans of selected plans exactly covers the intended collaborative behaviour of a MAS $M$, given by $\mathcal{I}\mathcal{S}(M)$, where all admissible runs are bounded. If $M$ has one-way logging and is implemented incorrectly, then either (i) $CP = \emptyset$ or (ii) there is some message log $\mathcal{M}_{\text{log}}$ such $\text{Culprit}(\mathcal{M}_{\text{log}}, \mathcal{P})$ exists.

**Proof.** Suppose that $M$ is incorrectly implemented. That is, the intended collaborative behaviour is violated, and there is a run $R = S^0, m_1, S^1, \ldots, m_k, S^k$ witnessing this fact, i.e., $R \notin \bigcup_{S \in \mathcal{I}\mathcal{S}(M)} \text{Runs}(S)$. If $k$ exceeds the length bound, then any trajectory $T$ such that $R \rightleftharpoons T$ exceeds the length bound as well, and thus $\text{C-Plans}(\mathcal{P}, \mathcal{M}_{\text{log}}, k) = \emptyset$ must hold; hence, $\text{Culprit}(\mathcal{M}_{\text{log}}, \mathcal{P})$ exists in this case. Thus, for the rest assume that $k$ is within the limit. Let $\mathcal{M}_{\text{log}} = t_1:m_1, \ldots, t_k:m_k$ be the message log produced by $R$; notice that because of one-way logging, $\mathcal{M}_{\text{log}}$ is produced only by runs $R'$ such that $R' \simeq R$. Towards a contradiction, suppose that $CP \neq \emptyset$ and $\text{Culprit}(\mathcal{M}_{\text{log}}, \mathcal{P})$ is not found by agent monitor. Thus, there is no time-out detected (and thus $M$ is not judged terminated or stuck), and there must exist some trajectories $T \in \text{Sem}(P)$ for some plans $P \in CP$ of form $T = s_0, m_1, s_1, \ldots, m_k, s_k$. By the Modelling Assumption, there exists some runs $R'$ of $M$ such that $R' \rightleftharpoons T$. Since $CP$ is an exact cover, at least one $R'$ among them is admissible, i.e., $R' \in \text{Runs}(S)$ for some $S \in \mathcal{I}\mathcal{S}(M)$. Since the correspondence $f$ between action sets and messages is one-to-one and $M$ has one-way logging, it follows that $R' \simeq R$. However, by closure of strategies under $\simeq$, it follows $R \in \text{Runs}(S)$ and thus $R \in \bigcup_{S \in \mathcal{I}\mathcal{S}(M)} \text{Runs}(S)$, which is a contradiction. □
In particular, the above theorem holds if CP models the intended behaviour \( IS(M) \) (i.e., \( CP = I\text{-Plans}(P) \)). In (i), we can conclude a design error, while in (ii) a design or coding error may be present. Again, we obtain an easy corollary for the \( K \) planning framework:

**Corollary 3.6.9 (Completeness of Exact \( P^K \) Cover).** Let \( P^K \) be a \( K \) planning problem, such that the set \( OP \subseteq P^K \)-plans of optimistic plans exactly covers the intended collaborative behaviour of the MAS \( M \). If \( M \) has one-way logging and is implemented incorrectly, then either (i) \( CP = \emptyset \) or (ii) there is some message log \( M_{log} \) such \( Culprit(M_{log}, P) \) exists.

Notice that Theorem 3.6.8 allows us to detect incorrectness despite a mismatch of the structure of strategies \( S \) used in \( M \) and the structures of plans supported in \( PF \).

We may dispose some of the assertions if the strategies used in the MAS satisfy certain properties. An example is the case in which the collaborative behaviour is governed by a \textit{conformant strategy} \( S \), which means that \( S \) semantically corresponds to a conformant (i.e., secure) plan; that is, starting from any possible initial state, the same sequence \( m_1, m_2, \ldots, m_n \) of messages is expected to appear (and always lead to success), regardless of how the global state evolves. If we model the intended behaviour by secure plans in the planning framework, then we can drop the one-way logging assertion. We formulate the result here for the particular formalism \( K \), for which we described secure plans more detailed after Definition 3.6.2 above. In fact, it turns out that exact covers are tantamount to secure plans.

**Lemma 3.6.10.** Let \( P^K \) be a \( K \) planning problem, such that the set \( CP \subseteq P^K \)-plans of optimistic plans exactly covers the intended collaborative behaviour \( IS(M) \) of \( M \). Suppose that each \( S \) in \( IS(M) \) is conformant. Then, each plan \( P \in CP \) is secure.

**Proof.** Indeed, suppose \( P \in CP \) is not secure. Every trajectory \( T \in Sem(P) \) corresponds to some admissible run \( R \) of a strategy \( S \in IS(M) \), and some such \( R \) and \( S \) must exist. On the other hand, insecurity of \( P = \langle a_1, a_2, \ldots, a_n \rangle \) implies that there exists some trajectory \( T' = s_0, a_1, s_1, \ldots, a_m, s_n \) in \( P^K \) violating condition (iii) for a secure plan above. That is, either \( m = n \) and the goal is not established, or \( m < n \) and \( tr(s_m, a_{m+1}) \) is either undefined or empty. By the Modelling Assumption, \( T' \) corresponds to a run \( R' \) in \( M \) which does not establish
the collaboration goal at termination of $M$ or that $M$ is stuck. Since $R'$ and $R$
have the same message sequence $m_1, \ldots, m_k$ (where $f(m_i) = a_i$, $1 \leq i \leq k$), this
$R'$ compromises that $S$ is conformant. This is a contradiction. □

**Theorem 3.6.11 (Completeness of $\mathcal{P}^K$ for Secure Strategies).** Let $\mathcal{P}^K$
be a $K$ planning problem, such that the set $CP \subseteq \mathcal{P}^K$-plans of optimistic plans
exactly covers the intended collaborative behaviour $\mathcal{IS}(M)$ of $M$. Suppose that
$M$ has bounded runs and a conformant collaboration strategy, $S_M$. Then, if $M$
is implemented incorrectly, there is some message log $M_{\log}$ such that either (i)
$CP = \emptyset$, or (ii) $\text{Culprit}(M_{\log}, CP)$ exists.

**Proof.** The argument is similar to that in the proof of Theorem 3.6.8. In arguing
towards a contradiction, rather than concluding that $R \simeq R'$ must hold,
we use that $R$ must correspond to some trajectory $T' = s'_0, a_1, s'_1, \ldots, a_k, s'_k$,
where $a_i = f(m_i)$, $i \in \{1, \ldots, k\}$, which does not reach the planning goal
$G$. This trajectory means that the plan $P = \langle a_1, \ldots, a_n \rangle$ given by the traject-
ory $T = s_0, a_1, s_1, \ldots, a_n, s_n$, is not secure. However, this is a contradiction to
Lemma 3.6.10. □

For example, in our running scenario, a design error is detected for conformant
plans as MAS collaborative behaviour formalism, if we use secure plans in $K$. The
culprit vanishes if we move to a cover which contains in addition the (non-secure)
plan $P_1$, since it is compatible with $M_{\log}$.

We can deploy the $K$ planner also for checking $\text{C-Plans}(\mathcal{P}, M_{\log}, n) \neq \emptyset$
or whether $\text{Culprit}(M_{\log}, \mathcal{P})$ exists. In particular, if the intended behaviour is ex-
pressed by optimistic/secure plans in $K$, deciding $\text{C-Plans}(\mathcal{P}^K, M_{\log}, n) \neq \emptyset$ is
tantamount to optimistic/secure plan existence; the plan prefix given by $M_{\log}$,
$n$ can easily be encoded in the planning problem itself by adding corresponding
constraints.

Let $M_{\log} = t_1:m_1, t_2:m_2, \ldots, t_n:m_n$, and let $\mathcal{P}^K$ be a $K$ planning problem
modelling the MAS $M$ at hand. Let $\mathcal{P}^K_{M_{\log}}$ be the problem obtained from $\mathcal{P}^K$
by adding the following to the $K$ program (cf. [EFL+04] for details on the semantics):

Initially: step$_0$.

Always:

caused false after not $m_1$, step$_0$.
caused step$_1$ after $m_1$.
caused false after not $m_2$, step$_1$.
caused step$_2$ after $m_2$. 

```
where \texttt{step}_0, \ldots, \texttt{step}_{n-1} are newly added propositional fluents. Intuitively, this modified planning encoding enforces the “execution” of the messages in \( M_{\log} \) and only plans which comply with these messages are computed. We therefore obtain the following result:

**Proposition 3.6.12.** Suppose the set \( OP \subseteq \mathcal{P}^K \)-plans of optimistic plans for \( \mathcal{P}^K \) (respectively, the set \( SP \subseteq \mathcal{P}^K \)-plans of secure plans) covers \( \mathcal{I} \mathcal{S}(M) \). Let \( M_{\log} = t_1.m_1, t_2.m_2, \ldots, t_n.m_n \) be a message log. Then, \( \text{C-Plans}(\mathcal{P}^K, M_{\log}, n) \neq \emptyset \) iff \( \mathcal{P}^K_{M_{\log}} \) has an optimistic (resp. secure) plan.

As easily seen, the encoding \( \mathcal{P}^K_{M_{\log}} \) can therefore be used to check the existence of \( \text{Culprit}(M_{\log}, \mathcal{P}^K) \) of form \( \langle t_n.m_n, \text{idle} \rangle \), while in general we cannot use a planner for detecting a timeout in the MAS.

This encoding is not restricted to our particular formalism. For instance, the computation of plans compatible with a prefix \( M_{\log} \) can be achieved in any planning formalism which allows for a similar modelling of domains such that certain actions can be fixed. Thus, we can apply planning also to check whether \( \text{C-Plans}(\mathcal{P}^K, M_{\log}, n) \neq \emptyset \) or whether a \( \text{Culprit}(M_{\log}, \mathcal{P}^K) \) of form \( \langle t_n.m_n, \text{idle} \rangle \) exists.

As for complexity, we mention that in expressive planning formalisms like \( K \), deciding whether \( \text{C-Plans}(\mathcal{P}, M_{\log}, n) \neq \emptyset \) or \( \text{Culprit}(M_{\log}, \mathcal{P}) \) exists from \( \mathcal{P}, M_{\log} \) and \( n \) is NP-hard in general, which is inherited from the expressive planning language; the corresponding complexity results and a discussion of complexity issues can be found in [EFL+04]. We remark that, like for satisfiability (SAT), NP-hardness (or even worse, if secure plans are required) is a theoretical worst-case measure. Nevertheless, solutions for many instances can be found quickly, especially if only optimistic planning is required. Moreover, there are problem classes which are polynomial time solvable and for which \( \mathcal{DLV}^K \) is guaranteed to compute plans in polynomial time. This highly depends on the requirements of intended plans and how complicated the corresponding planning problem gets.

For small domains, where the number of plans is moderate, \( I\text{-Plans} \) (or \( C\text{-Plans} \), resp.) might be computed offline or simply be cached such that checking against \( M_{\log} \) becomes simple.
3.7 Implementation within IMPACT

We presented the idea of monitoring agents using planning above. In this section, in order to demonstrate the proposed approach, we show a running example which was implemented within the multiagent platform IMPACT and the planning framework DLVκ.

A multiagent framework: IMPACT

We first briefly describe the IMPACT system. The IMPACT project (Interactive Maryland Platform for Agents Collaborating Together) aims at providing a powerful system to create and deploy agents (see the project homepage). IMPACT is based on logical notions and semantically well understood, therefore we can develop and test our methods and ideas within it. In addition, IMPACT is general enough, so it is possible for us to extend our approach to other multiagent systems.

Before explaining the main features of IMPACT agents, we start with a short description of the general architecture of IMPACT. IMPACT agents and IMPACT servers are distributed over the network. An IMPACT server, which can be replicated or mirrored, consists of registration server, yellow pages server, thesaurus, types server and user interfaces. In IMPACT, agents communicate with other agents (or server) through the network. An agent can not only send out (and receive) messages from other agents, but they can also ask the server to find out about services that other agents offer.

The core part of an IMPACT agent includes a set of data type definitions, API function calls, a message type and API functions for messaging, integrity constraints, action constraints, a set of actions, an agent program and a notion of concurrency. To get an overview of IMPACT, here are some of the most important features (see Figure 3.4):

- **Code Calls(cc):** Code Calls is a special mechanism in IMPACT, which is used for accessing different types of data. A code call executes an API function and returns as output a set of objects $d_1, \ldots, d_n$ of the appropriate output type. Code call atoms, denoted $in(X, cc)$, are logical atoms layering on top of cc. X is either a variable symbol or and object of the output

---

3.7. IMPLEMENTATION WITHIN IMPACT

Figure 3.4: An agent in IMPACT

- **Actions α**: Each agent has a set of associated actions $\alpha$. Agents act in their environment according to their *agent program* and a well-defined *semantics* $\text{Sem}$ determining which of the actions the agent should execute.

- **Agent Program**: An agent program is a set of rules, which specifies what actions the agent must take and regulates the execution of these actions.

- **State $\mathcal{O}$**: At any time, the state of an agent is the set of all objects the agent currently has. The state of an agent may be changed by executing an action.

- **IN/OUT Messages**: Each agent also has an associated module that manages incoming and outgoing messages.

- **Function conc**: A notion of concurrency $\text{conc}$ is a function that takes a set of actions to be executed concurrently as input, together with a *state*, type of $\text{cc}$. Executing code call atoms return boolean values. They build a connection between the data type in the software and the logical atoms in IMPACT. In this way, IMPACT agents can be built on top of arbitrary software code (*Legacy Data*).
and outputs a single actions to be executed.

Each agent continually undergoes the following cycle:

1. The agent receives a set of new messages sent by other agents. This changes the state of the agent.

2. The change may trigger some rules in the agent’s associated Agent Program. Based on its semantics and its state, the agent determines for each action its status (permitted, obliged, forbidden, . . .). The agent ends up with a set of status atoms.

3. Based on a notion of concurrency, the agent determines the actions that can be executed. The agent’s state may change as a consequence of the performance of such actions. In addition, the message box of other agents may also change.

**Implementation of IMPACT agents**

To demonstrate the proposed method, a running example has been implemented. In our implementation, the Gofish multiagent system and agent monitor are developed within IMPACT. Here we briefly introduced the implementation using the example of agent monitor.

The IMPACT system consists of five major software components to support the development and deployment of IMPACT agents.

- *Agent Development Environment:* Agent developers can easily build and test agents within the IMPACT Agent Development Environment (AgentDE for short). The AgentDE provides a network accessible, easy-to-use graphical user interface through which an agent developer can specify the date types, functions, actions, notion of concurrency and agent program of an agent, compile and then test if they work properly.

When we developed agent monitor within IMPACT, we must explicitly define its parameters via the AgentDE. Figure 3.5 provides an overview of the AgentDE. As shown in the figure, there are 8 specified actions that agent monitor can execute. Figure 3.6 shows one rule (with Do in the head) defined in the agent program of monitor.
After defining these parameters, the agent developer may start testing the agent. The AgentDE performs compile-time checks. Pressing “Test Definition” in Figure 3.6 triggers the test. After the test phase is completed, status sets are generated and executed.

- **IMPACT Connection**: The IMPACT connection library allows IMPACT agents to access third party platforms. The developer can define a connection alias and specific parameters for the target connection in the AgentDE Connection specification dialog. Figure 3.7 shows the AgentDE interface with the accepted DLV\(^K\) planner connection definition. When a connection is established, IMPACT can execute code call over the data source and process the returned requests.

- **IMPACT Server**: The IMPACT Server provides various services that are required by a group of agents as a whole. It supports registration services, which is used to specify the services provided by the agent; yellow pages server, where IMPACT agents can find the desired services by other agents; type server, which maintains a set of class hierarchies containing information about different data types used by different agents; thesaurus server, which
3.7. IMPLEMENTATION WITHIN IMPACT

Figure 3.6: AgentDE Program

receives requests when new agent services are being registered and when the IMPACT Yellow Pages Server is looking for agents providing a service. Figure 3.5 shows the yellow page information of the agent monitor, where we can find its registered name “MonitorGofish_New” and its description “Monitoring the Gofish MAS”.

• Agent Roost: An agent roost is a location where a set of deployed agents resides. An agent roost serves as a duty officer since it manages all messages for this set of agents. Initially, all agents are inactive. When one of these agents receives a message, the agent roost includes it in this agent’s message box and lets it run. If an agent sends out a message to another internal agent (i.e., an agent who is managed by the same roost), this message can be delivered by the roost in the same way. If the message is addressed to an external agent, the roost first contacts the IMPACT server to determine the location of the target agent. It then routes the message to the appropriate roost, which will pass it to the specified agent. Figure 3.8 is a screenshot of the agent roost, where locates 12 agents and depicts the moment when the agent “MonitorGofish_New” is active.
• **Agent Log**: The agent log allows an agent developer to maintain a record of agent communication and agent actions. The log supports log queries by content or time, and action browse, playback of video, text and image message objects. It can be used for many purposes such as record keeping, usage statistics, and it is essential for monitoring system performance and debugging.

### Running Examples

As we described earlier, the *Gofish* MAS is a multiagent system to simulate *Gofish* Post Office. We add a monitoring agent `monitor` to aid debugging the given *Gofish* MAS. We already showed above that agents have been implemented within *IMPACT*. The planner $\mathcal{DLV}^K$ can be accessed by the agent `monitor` via a new connection module which has been created in *IMPACT* (see Figure 3.7). In this way, before the *Gofish* MAS operates, we feed the *Gofish* planning problem $\mathcal{P}_{\text{Gofish}}$ into `monitor`, which then calls $\mathcal{DLV}^K$ to compute all potential plans including both secure and optimistic plans.

Agent `monitor` exploits the feded planning problem $\mathcal{P}_{\text{Gofish}}$ to generate all
possible plans to reach a goal. monitor then continually checks and compares the messages sent between agents with all plans. Once it detects any incompatibility, monitor generates an error file and reports to the designer.

Now we are ready to run the system in order to demonstrate the proposed method. Two types of errors, design errors and implementation (coding) errors can be distinguished as shown in the following two examples.

Example 10 (Running scenario: detecting design error). The Gofish post office guarantees package delivery within 24 hours from dropOff. Consider the case that Sue receives an email at time 16 informed that her package \( p_1=0x00fe6206c.1 \) has arrived at the distribution centre. Sue decides to pick up the package herself. Unfortunately, when she reaches the distribution centre at time 20, the clerk tells her that the package has been loaded on the truck at time 19 and it is now on the way to her home.

Because of the guaranteed delivery requirement, agent monitor computes secure plans for the purpose of monitoring:
3.7. IMPLEMENTATION WITHIN IMPACT

\[ P_1 = (\text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{truck}(p_1); \text{getRecipInfo}(p_1, \text{truck}); \text{recipInfo}(p_1, \text{truck}); \text{delivery}(p_1); \text{setDelivTime}(p_1)). \]

\[ P_2 = (\text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{pickup}(p_1); \text{setDelivTime}(p_1)). \]

In the above scenario, instead of waiting at home, Sue shows up at the distribution centre and made a pickup attempt. This “external” event may have been unforeseen by the designer. Thus after the action “pickup”, a design error is immediately detected by monitor. Agent monitor writes an error log file, and clearly points to a situation missed in the design of Gofish multiagent system:

Problematic action:
20: \text{pickup}(0x00f6206c.1), \text{idle}

Actions executed:
0: \text{dropOff}(0x00f6206c.1); 5: \text{addPkg}(0x00f6206c.1);
13: \text{distCenter}(0x00f6206c.1); 15: \text{getRecipInfo}(0x00f6206c.1, \text{dist});
16: \text{recipInfo}(0x00f6206c.1, \text{dist}); 19: \text{truck}(0x00f6206c.1);

Possible plans before problematic action:
⟨\text{dropOff}(p_1); \text{addPkg}(p_1); \text{distCenter}(p_1); \text{getRecipInfo}(p_1, \text{dist}); \text{recipInfo}(p_1, \text{dist}); \text{truck}(p_1); \text{getRecipInfo}(p_1, \text{truck}); \text{recipInfo}(p_1, \text{truck}); \text{delivery}(p_1); \text{setDelivTime}(p_1)⟩

The following example illustrates how agent monitor works if there is a coding error in the Gofish system.

**Example 11 (Running scenario: detecting coding error).** Suppose the message log of the Gofish MAS shows that after “\text{dropOff}”, a message “\text{distCenter}” is followed. However in the actual code, “\text{addpkg}” should be executed in order to update the database. Although this does not result in a livelock, Monitor detects this coding error and generates the error log file.
Problematic action:
13: distCenter(0x025da2198.1), idle

Actions executed:
0: dropOff(0x025da2198.1);

Possible plans before problematic action:
⟨dropOff(p1); addPkg(p1); distCenter(p1); getRecipInfo(p1, dist);
recipInfo(p1, dist); truck(p1); getRecipInfo(p1, truck);
recipInfo(p1, truck); delivery(p1); setDelivTime(p1)⟩
⟨dropOff(p1); addPkg(p1); distCenter(p1); getRecipInfo(p1, dist);
recipInfo(p1, dist); pickup(p1); setDelivTime(p1)⟩

We set up a project homepage,\(^7\) which provides information about: (1) Gofish Multiagent system and its specification as a planning problem; (2) agent monitor; and in addition, (3) two animations to show the implementation of Gofish MAS and agent monitor within IMPACT and to demonstrate the use of the proposed idea to detect two different types of errors that may appear in Gofish MAS.

In addition, we refer to [SBD+00] for the details of IMPACT and [EFL+03b] for the details of DLV^K.

### 3.8 Conclusion

In this chapter, we introduced a monitoring approach which can be used for offline debugging and online monitoring for a multiagent system, based on monitoring their message exchange using planning methods. This can be seen as a very useful debugging tool for detecting coding and design errors. We also presented some soundness and completeness results for our approach, and touched upon its complexity.

Our approach works for arbitrary agent systems and can be tailored to any planning formalism that is able to express the collaborative behaviour of the MAS. We have briefly discussed and implemented how to couple a specific planner, DLV^K, which is based on the language K, to a particular MAS platform, IMPACT. A webpage for further information and detailed documentation has been set up (see footnote 7).

\(^7\) [http://www.in.tu-clausthal.de/∼yzhang/monitoring.html](http://www.in.tu-clausthal.de/~yzhang/monitoring.html)
Among the benefits of this approach are the following:

- It allows to deal with collaboration behaviour regardless of the implementation languages used for single agents.
- It works for legacy multiagent systems, thus is able to preserve the privacy of the systems.
- Depending on the planner used, different kinds of plans such as *optimal* plans, *secure* (or *conformant*) plans might be considered, reflecting different agent attitudes and collaboration objectives.
- Changes in the agent messaging by the system designer may be transparently incorporated to the action theory $T$, without further need to adjust the monitoring process.
- Furthermore, the action theory $T$ forms a formal system specification, which may be reasoned about and used in other contexts.
- As a by-product, the method may also be used for automatic *protocol generation*, that is, determining the messages needed and their order, in a (simple) collaboration.
Chapter 4

Discussion and Future Work

In the previous chapter, we introduced a planning based approach, which assisted agent developers in (1) debugging a multiagent system, and (2) online monitoring of the behaviours of the agents. In this chapter, we will discuss our approach by reviewing related works about these two issues.

Section 4.1 and Section 4.2 overview the techniques for debugging multiagent systems and monitoring behaviours of agents respectively. Suggestions for valuable future work can be found in Section 4.3.

4.1 Debugging Multiagent Systems

In this section, we review some approaches developed for addressing the debugging issue in multiagent systems.

Botia et al. [BHS04] propose an analysis tool to assisting in debugging both functional and coordination related errors in multiagent systems. More specifically, they introduce a spy agent who keeps track of all the interactions being performed in a run of the multiagent system, and stores them in a relational database. The relational database clusters these interactions by using categorical clustering technique, and provides a complete graph of all the agents and their interaction in a run. When an error is produced during the run, the tool generates general statistics in a window where it shows the information of the conversations, such as the number of total conversations, the number of conversations which did not follow the interaction protocols. The developers then can find the error by checking the detailed information of agents and the conversation history. The key
technique in their approach is to firstly gather information, cluster or filter information, and finally present the users with a graphical depiction of the behaviours of agents. Similar approaches can be also found in [NNLC99, LA95, GPD94]. We call them visualisation approaches.

These visualisation approaches look as if they provide an intuitive way to explain what is happening to the agents. Unfortunately, this is not true—it is difficult for a developer to understand what is really happening in the system due to the large number of presented information. One major drawback of such a visualisation approach is that it relies on the agent developers to interpret the gathered information correctly and determine the problematic agents or actions. Furthermore, this kind of approach is highly system dependent and may need a large amount of work if it is applied to previously developed multiagent systems. For example, the method of [BHS04] only works for multiagent systems which are developed within the JADE platform (Java Agents Development Environment). Since our approach is independent of any particular agent systems, it can be employed to debug legacy multiagent systems, no matter what languages were used in the system implementations.

Some other approaches, which are closer to our idea, make use of design documents to aid run time error detection and debugging. Design documents could be good tools for multiagent system debugging since they are produced in the pre-implementation stages of system development and reveal the intentions of the developers that how the agents in the system behave. Poutakidis et al. [PPW02, PPW03] adopt the design of interaction protocols as a debugging tool. They introduce a method to monitor the messages exchanged between agents against interaction protocols. A debugging agent is added to the standard multiagent system to assist in the debugging. They assume that the agent-based unified modelling language (AUML for short) is the language used for specifying agent communications (or protocols) by the system designer. The debugging agent then translate the AUML specification into Petri nets. Therefore, before the conversation of agents starts, the debugging agent has a library of known protocols. The precise notations and formal semantics of Petri nets make it easier than AUML to being examined the properties. Thus, during the run of the system, by checking the messages exchanged between agents against Petri nets, the debugging agent could detect different types of errors, for example unexpected

\[^{1}\]AUML was developed by FIPA (see http://www.auml.org/).
or unknown messages. The debugging agent reports to the user any error that it might flag.

Their idea of debugging is similar to ours—we all make use of interaction protocols and translate them into formal semantics to monitor the exchanged messages between agents. The difference is that they translate the protocols into Petri net for monitoring, while we generate plans from the protocols. The advantage of their approach is that they provide a possible systematic method to automatically convert the design specification into monitoring components (by Petri nets), assuming AUML is the specification language of agent communication protocols. In our approach, we rely on agent developers giving the plan specification based on the communication protocol of agents. However, their approach suffers from some drawbacks compared to ours.

- There are many AUML constructs which are not addressed by the translation. For instance, because internal states of an agent are unknown by other agents, it is difficult for the debugging agent to know when preconditions become true.

- There may exist different interpretations of an ambiguous interaction. When such interaction is translated to Petri nets, some unnecessary assumptions may need to be make, which may not be originally intended by agent developers.

- It is difficult to formalise the cardinality of messages by Petri nets. For the moment, they allow conversations between only two agents.

- Since the copies of any messages sent by agents are sent to the debugging agent, this additional message delivery may affect the system’s behaviour especially if the system is highly time dependent.

- They focus on debugging agent online communication, while we tackle the problems that may arise both in online agent collaboration and in offline coding or design.

Our method has no problem to specify the communications of agents as a planning problem, as long as agents are cooperative and working together to reach a common goal. With the use of proper planning languages, our approach can deal with nondetermination and incomplete information, which makes our
method more applicable than theirs. Furthermore, it is clear that planning is more suitable for specifying the collaboration of agents than Petri nets.

4.2 Plan Based Monitoring

In contrast to research on plan generation, there has been relatively little work on the use of plans to debug a multiagent system and to monitor the execution of agents. In this section, we review some plan based approaches to execution monitoring of a single agent system and a distributed multiagent system respectively.

Monitoring in Single-Agent Systems

Interleaving monitoring with plan execution by a single agent has been addressed in the context of single agent environment by Giacomo et al. in [GRS98]. They present a situation calculus-based account of execution monitoring for robot programs written in Golog. A situation calculus specification is given for the behaviour of Golog programs. Combined with the interpretation of Golog programs, an execution monitor detects the discrepancy after each execution of a primitive action. Once a discrepancy is found, the execution monitor checks whether it is relevant in the current state, that is, whether preconditions of the next action still hold with the effect of an exogenous action. If this exogenous action does matter, a recover mechanism will be invoked. The method of recovering is based on planning. A new plan (or program) is computed whose execution will make things right by way of leading the current state to the desired situation, had the exogenous action not occurred. In their work, declarative representations have been proposed for the entire process of plan-execution, plan-monitoring and plan-recovery. Similar to our method, the approach is completely formal and capable of monitoring arbitrary programs. The authors have addressed the problem of recovering from failure, which is not included in our system for the moment. However, their approach must know in advance all exogenous events in order to specify appropriate relevance checks and recover mechanisms. In addition, they do not explore in-depth how to properly define Relevant and Recover. The framework was later expanded in [Sou99] by introducing tracing and backtracking into the process of online monitoring of Golog programs.
4.2. PLAN BASED MONITORING

To enable generation of plans in dynamic environments, Veloso et al. [VPC98] introduce Rationale-Based Monitoring based on the idea of planning as decision making. Rationale-based monitors encode the features that are associated with each planning decision. The method is used for sensing relevant (or potentially relevant) features of the world that likely affect the plan. Moreover, it investigates the balance between sensitivity to changes in the world and stability of the plans. Although this approach provides the planner opportunities to optimise the plans in a dynamic environment during plan generation, as opposed to our approach, they have not studied the issue of execution monitoring.

As the methods mentioned above address the problem of a single agent acting in an uncertain environment, the techniques focus on monitoring of environment and verifying plans. While our approach could be directly applied to single agent domains, these approaches need extra work in order to handle monitoring the collaboration of multiple agents.

Monitoring in Multiagent Systems

Teamwork monitoring has been recognised as a crucial problem in multiagent coordination. Jennings proposes two foundations of multiagent coordination in [Jen93]: commitments and conventions. Agents make commitments, and conventions are a means of monitoring of the commitments. The monitoring rules, that is, what kind of information is monitored and the way how to perform monitoring, are decided by conventions. Jennings illustrates the method by some examples, but does not investigate how to select such conventions. Different from his idea, our approach avoids the problem of monitoring selectivity.

Myers [Mye99] introduces a continuous planning and execution framework (CPEF). The system’s central component is a plan manager, which directs the processes of plan-generation, plan-execution, plan-monitoring, and plan-repair. Monitoring of the environment is carried out at all time during plan generation and execution. Furthermore, execution is tracked by the plan manager by comparing reports of individual action outcomes with the temporal ordering relationships of actions. Several types of event-response rules have been concerned: (1) failure monitors encode suitable responses to potential failures during plan execution, (2) knowledge monitors detect the availability of information required for decision making, and (3) assumption monitors check whether assumptions that a given plan relies on still hold. The idea of assumption monitors helps
early detection of potential problems before any failure occurs, which can also be achieved in our system with a different approach. Based upon CPEP, Wilkins et al. present a system in [WLB03], where the execution monitoring of agent teams is performed based on communicating state information among team members that could be any combination of humans or machines. Humans make the final decision, therefore, even if unreliable communications exist, the monitoring performance may not be degraded much with the help of human experience.

Hagg [Hag96] uses a control system, called *sentinels*, to monitor agent communication, to select various problem solving methods when necessary, and to exclude faulty agents in the system. The approach allows the system developer to program the functionality of the MAS and then add on a control system. The sentinels introduced in this paper is a kind of middle agent who is centralised.

In [KD99], Klein et al. propose a domain independent service to handling of exceptions in agent systems. This service can be viewed as a “coordination doctor”, which predefines several typical abnormal situations that may arise in the system. An exception handling institution then monitors agent behaviour, diagnoses problematic situations and takes recovery actions. The exception handling process is carried out by several collaborative agents.

Another interesting monitoring approach in multiagent coordination is based on *plan-recognition*, by Huber [HD95], Tambe [Tam96], Intille and Bobick [IB99], Devaney and Ram [DR98], Kaminka et al. [KPT01, KT00]. In this approach, an agent’s intentions (goals and plans), beliefs or future actions are inferred through observations of another agent’s ongoing behaviour.

Devaney and Ram [DR98] describes the plan recognition problem in a complex multiagent domain involving hundreds of agents acting over large space and time scales. They use pattern matching to recognise team tactics in military operations. The team-plan library stores several strategic patterns which the system needs to recognise during the military operation. In order to make computation efficient, they utilise representations of agent-pair relationships for team behaviour recognition.

Intille and Bobick [IB99] construct a probabilistic framework that can represent and recognise complex actions based on visual evidence. Complex multiagent action is inferred using a multiagent belief network. The network integrates the likelihood values generated by several visual goal networks at each time and
returns the likelihood that a given action has been observed. The network explicitly represents the logical and temporal relationships between agents, and its structure is similar to a naive Bayesian classifier network structure, reflecting the temporal structure of a particular complex action. The approach relies on all coordination constraints among the agents. Once an agent fails, it may not be able to recognise the plans.

Another line of work has been pursued by Kaminka et al. [KPT01, KT00], who developed the OVERSEER monitoring system building upon work on multiagent plan-recognition in [IB99, Tam96]. The authors address the problem of many geographically distributed team members collaborating in a dynamic environment. The system employs plan recognition to infer the current state of agents based on the observed messages exchanged between them. The basic component is a probabilistic plan-recognition algorithm which underlies the monitoring of a single agent and runs separately for each agent. This algorithm is built under a Markovian assumption and allows linear-time inference. To monitor multiple agents, social knowledge—relationships and interactions among agents—is utilised for better predicting the behaviour of team members and detect coordination failures. The OVERSEER system supports reasoning about uncertainty and time, and allows to answer queries related to the likelihood of current and future team plans.

While our objective is (1) to debug offline an implemented MAS, and (2) to monitor online the collaboration of multiple agents, the plan-recognition approaches described above mainly aim to inferring (sub-)team plans and future actions of agents. The MAS debugging issue is not addressed. Furthermore, we point out that our method might be used in the MAS design phase to support protocol generation, that is, to determine at design time the messages needed and their order, for a (simple) agent collaboration. More precisely, possible plans $P = \langle m_1, \ldots, m_k \rangle$ for a goal encode sequences of messages $m_1, \ldots, m_k$ that are exchanged in this order in a successful cooperation achieving the goal. The agent developer may select one of the possible plans, e.g. according to optimality criteria such as least cost, $P^*$, and program the individual agents to obey the corresponding protocol. In subsequent monitoring and testing, $P^*$ is then the (single) intended plan.

However, plan recognition is suitable for various situations: if communication is not possible; if agents exchanging messages are not reliable; or if communications must be secure. It significantly differs from our approach in the following
4.2. PLAN BASED MONITORING

points:

1. If a multiagent system has already been deployed, or if it consists of legacy code, the plan-recognition approach can do monitoring without modifications on the deployed system. Our method entirely relies on an agent message log file.

2. The algorithms developed in [KT00] and [DR98] have low computational complexity. Especially the former is a linear-time plan recognition algorithm.

3. Our model is not yet capable of reasoning about uncertainty, time and space.

4. In some tasks, agents do not frequently communicate with others during task execution. In addition, communication is not always reliable and messages may be incorrect or get lost.

We believe the first three points can be taken into account in our framework.

1. It should not be too difficult to add an agent actions log file explicitly for a given multiagent system.

2. While the developed algorithms are of linear complexity, the whole framework needs to deal with uncertainty or probabilistic reasoning which can be very expensive. Although our approach is NP-hard in the worst case, we did not encounter any difficulties in the scenarios we have dealt with.

3. IMPACT does not yet have implemented capabilities for dealing with probabilistic, temporal and spatial reasoning, but such extensions have been developed and are currently being implemented [SBD+00, DKS01, DKS05].

Compared with the approaches described above, our method enjoys some advantages as the following:

- Our method can be more easily extended to do plan repair than the methods above. Kaminka et al. [KPT02] introduce the idea of dealing with failure actions.
• The approach we have chosen includes protocol generation in a very intuitive sense relying on the underlying planner while the cited approaches model agent behaviour at an abstract level which can not be used to derive intended message protocols directly.

• Since ascertaining the intentions and beliefs of the other agents will result in uncertainty with respect to that information, some powerful means of reasoning under uncertainty are required for the plan recognition method.

4.3 Future Work

Of course, our approach is not the final one—there are still several issues that remain to be addressed in further work. One issue concerns the modelling of multiagent systems in a planning framework. This seems to be particularly important for complex multiagent systems, since modelling such a system is not easy in general and requires a thought-through methodology. Another issue is scalability of our approach. The example which we have considered in our work is of moderate size, and for larger multiagent systems, the planning tasks will become more difficult to solve. But as we know, planning can be abstracted to any level of granularity (for example, Hierarchical Task Network (HTN) planning which decomposes a task into a set of sub-tasks (see [GNT04])). Therefore, for the large problem, we can model agents’ behaviours at properly higher level. In this way, our approach is feasible for large multiagent systems.

There are also several extensions to our approach. We mention just some of the possible future research in this direction:

(1) Cost based planning: Can the goal still be reached with a certain bound on the overall costs, given that actions which the agents take have costs assigned? And, what is the optimal cost and how does the corresponding behaviour look like? This would allow us to assess the quality of an actual agents behaviour and to select cost-effective strategies. To keep the exposition simple, we have omitted that DLV$^K$ is also capable of computing admissible plans (plans within a cost bound) and, moreover, optimal plans over optimistic and secure plans, respecting that each action has certain declared costs [EFL+03a]. For instance, in the GoFISH example we might prefer plans where the customer picks up the package herself, which
is cheaper than sending a truck. Thus, in the realisation of our approach, also economic behaviour of agents in a MAS under cost aspects can be easily monitored, such as obedience to smallest number of message exchanges or least total communication cost.

(2) Dynamic planning: We assumed an a priori chosen collaboration plan for $\mathcal{M}_{\text{log}}$ compatibility. This implies $\text{C-Plans}(\mathcal{P}, \mathcal{M}_{\text{log}}, n') \subseteq \text{C-Plans}(\mathcal{P}, \mathcal{M}_{\text{log}}, n)$, for all $n' \geq n \geq 0$. However, this no longer holds if the plan may be dynamically revised. Checking $\mathcal{M}_{\text{log}}$ compatibility then amounts to a new planning problem whose initial states are the states reached after the actions in $\mathcal{M}_{\text{log}}$.

Agent monitor can also be designed to redo the planning by updating the planning problem and taking into account the changes in the environment. For example after the occurrence of the pickup($p_1$) action, a new plan might be to phone the truck on the mobile and make sure that the delivery is scheduled until after the customer is back home.

(3) At the beginning of monitoring, all potentially interesting plans for the goal are generated, and they can be cached for later reuse. We have shown the advantages of this method. However, if a very large number of intended plans exists up front, the method may become infeasible. In this case, we might just check, similar to above, whether from the states possibly reached by the actions in $\mathcal{M}_{\text{log}}$, the goal can still be established.
Part II

Survivability of Multiagent Systems
Chapter 5

Motivation and Background

Part 1 of this thesis introduced a monitoring-based approach to making a multiagent system more reliable by tackling potential design or coding errors. In Part 2 of this thesis, the question of how to make agent systems more reliable is addressed at a more general and higher level—we introduce and investigate the notion of survivability into multiagent systems.

As we mentioned in Chapter 1, Part 2 of this thesis aims at Objective 2 and Objective 3. Before presenting our approaches to these two objectives in Chapter 6 and 7 respectively, in this chapter, we show some background and the motivation of our work on multiagent survivability.

5.1 Notation of Survivability

The past few years have seen significant efforts on the development of realistic multiagent systems. In contrast, the actual number of deployed systems is remarkably small. One of the reasons that impedes the deployment of multiagent systems in real-world situations is the brittleness of the deployed systems—even if a multiagent application is well designed and correctly programmed, it can crash easily due to various external events, such as system crashes, shortage of system resources, communication faults, or malicious attacks. Such failures and attacks are not uncommon in dynamic, resource bounded, or hostile environments. Especially when research in multiagent systems has moved slowly from closed multiagent systems to the systems which will require cross corporate boundaries, and in the long-term future, to open multiagent systems [LMP03], we see the increasing need of ensuring the survival of the deployed multiagent systems.
The notation *survivability* was introduced as a means of protecting critical systems. Like many terms in computer science that have not yet matured, several notions of survivability have appeared within recent years. In earlier work, Ellison et al. [EFL+99] define survivability as *the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents*, where the term *mission* is a set of high-level goals or requirements, and the terms *attack, failure, and accident* could include all potentially damaging events. Thereafter, there are some other variations of defining survivability. Knight et al. [KSEW00, KS03] give a more precise definition of survivability based on specification: *a system is survivable if it complies with its survivability specification*. Their definition requires a complete, well designed survivability specification for the system.

Survivability of multiagent systems has been recently investigated by the UltraLog project, which is a DARPA-sponsored program where 15 to 20 universities and companies have contributed [ULT, BG03, BBH+04, HKB04, GLG+04]. UltraLog aims at ensuring the survivability of military logistics applications which are deployed on a large-scale distributed multiagent systems in dynamic and hostile environments. By *survivability*, they mean *the extent to which the system maintains utility in the face of stress*. In their scenario, they consider attacks or failures that may degrade the network resource or the computation resource of the system. A set of *measures of performance* have been used to determine the *overall success* of the system, which include, for instance, performance, availability, and integrity. They apply a hierarchy of control loops to guide survivability. Because their survivability solutions are built on the Cougaar (Cognitive Agent Architecture) framework [COU], to apply their approach to survivability, one must also develop the multiagent applications on Cougaar and its UltraLog extensions and tools.

As an emerging discipline, survivability builds on related topics, for instance, fault tolerance, security, reliability, performance, verification and testing. There are multiple dimensions to the survivability of different applications. Thus, the precise definition of the goals and the measurements of whether or not an application is “success” or “survival” highly depends on the applications. When we study the survivability of multiagent systems, we keep the following considerations in mind:

---

**5.1. NOTATION OF SURVIVABILITY**

---
• Firstly, we are not going to investigate all of the issues related to survivability. We put the survivability issue in the context of multiagent systems, and we would like to give a general and intuitive way to define and measure the survivability of multiagent applications.

• Secondly, unlike the approach of UltraLog whose applicability is limited by agent platform, we want to develop a survivability solution which could be widely and easily applied to ensuring the survival of multiagent applications, independently of agent development platform.

5.2 Our Definition

We are interested in the scenarios where multiagent systems are deployed over resource bounded, dynamic networks. By resource bounded networks, we mean that each node in the networks has only limited resource available to the agents. By dynamic networks, we mean the networks where the communication between nodes may be unreliable, or the nodes themselves have some probabilities to get disconnected sometime in the future. Such failures on the nodes make a great impact on the survival of the multiagent system.

As the multiagent applications heavily rely on the collaboration of agents, the failure of one of the agents (caused by the failure of the hosting nodes) may bring the whole system down. Thus, replicating agents on different hosting nodes is a natural and efficient way to ensure the survivability of the multiagent systems. In this way, if a node hosting one agent goes down, a copy of this agent residing on another network location will still be functioning. Replication techniques have been widely developed in distributed computing systems [GS97, WPS+00], and further in multiagent systems [MSBG01, MBS03, FD02, KCL00, DSW97b, SSCJ98] (more discussions on these approaches can be found in Chapter 8).

Replication techniques provide a possible way to enhancing survival of the multiagent systems. Therefore, we borrow the idea of replication when developing survivability solutions. Consequently, we define the multiagent survivability as below.

Definition 5.2.1. In our definition, the survival of a multiagent system means that at any time, at least one replica of each agent in the system must be available on at least one node in the network. Furthermore, the degree of the survival of
5.3 A CENTRALISED SURVIVABILITY MODEL

A multiagent system is measured by the probability of with which the multiagent system will survive.

From now on, when we say “survivability of a multiagent system”, we mean “survivability of a MAS deployment”. It is important to point out that it is the services (or applications) which are provided by the multiagent system that must survive, not any particular system or network component. More explanations on the definition and its assumptions will be shown in Chapter 6.

We now present the replication-based survivability model of Kraus et al. [KST03] in the following.

5.3 A Centralised Survivability Model

As our survivability mechanism extends the work of Kraus et al. [KST03], we give a brief description of their survivability model.

They assume that a multiagent system \( \mathcal{M} \) is a finite set of agents and that for \( \mathcal{M} \) to function, at least one copy of each agent \( a \in \mathcal{M} \) must be accessible. They further assume the existence of a network \( \mathcal{N} = (V, E) \) where \( V \) is the set of nodes in the network and \( E = V \times V \). Each node has an associated disconnect probability (dp for short) that denotes the probability that that node will “go down” within a finite window of time. A deployment \( \mu \) of the system \( \mathcal{M} \) takes a node as input, and tells us which agents in \( \mathcal{M} \) are located at that node.

Kraus et al. defined the survivability of a deployment under the assumption that we are completely ignorant about the dependencies between node failures. We quickly recapitulate the survivability algorithm (named COD) in [KST03].

Suppose we use \( FN \) to denote the set \( \{(V', V' \times V') \mid V' \subseteq V\} \) of all future networks that can arise when some subset of nodes gets disconnected. For a possible future network \( Ne' \in FN \), we say that \( \mu \) is a deployment w.r.t. \( Ne' \) if for each agent \( a \in \mathcal{M} \), there exists a node \( n' \in Ne' \) such that \( a \in \mu(n') \). \( COD \) computes the survivability of \( \mu \) by returning the result of the following linear program.

\[
\text{minimize } \sum_{Ne' \in FN \land (\mu \text{ is a deployment w.r.t. } Ne')} p_{Ne'}
\]
subject to \(\text{CONS}(dp, V)\):

\[
1 - dp(n) = \sum_{N_e' \in FN \land n \in N_e'} p_{N_e'} \quad \text{for each } n \in V
\]

\[
1 = \sum_{N_e' \in FN} p_{N_e'}
\]

\[
0 \leq p_{N_e'} \quad \text{for any } N_e' \in FN \quad (5.1)
\]

\(\text{CONS}(dp, V)\) contains an instance of the first constraint above for each node \(n \in V\) plus the second and third constraints shown above. The variables \(p_{N_e'}\) denote the probability that a future network \(N_e'\) will arise. The first constraint above says that the probability that the node \(n\) will survive equals the sum of the probabilities of all future networks in which \(n\) is still connected. The second constraint says that the space of possible future networks is exhaustive. The last constraint says that each probability is non-negative.

Note that \(\text{CONS}(dp, V)\) is enormous because the number of variables in this linear programming is \textit{exponential} in the number of nodes. [KST03] shows that computing survivability of a deployment \(\mu\) is intractable.

### 5.4 Our Approach

While the uses of replication techniques by [MSBG01, MBS03, FD02, KCL00, DSW97b, SSCJ98, KST03] bring an efficient way to enhancing the survivability of multiagent systems, for our purpose they have a number of shortcomings.

1. The first shortcoming is that most of the approaches are \textit{centralised}. In other words, even though the agents in the multiagent system are distributed across the network, the control algorithm—the one maintains replication—resides on a single node. Thus, if the node hosting the control algorithm goes down along with all nodes containing some agent in the system, then the system is compromised.

2. The second shortcoming is that the replication methods do not \textit{adapt to changes} that affect the survival of the multiagent system. Most algorithms assume that once the replication algorithm tells us where to replicate agents, we just replicate the agents at the appropriate nodes and then ignore survivability issues altogether. It is clearly desirable to continuously (or at
least regularly) monitor how well the system is “surviving” and to respond to changes in this quantity by redeploying the agent replicas to appropriate new locations.

3. The third shortcoming is that these approaches do not address the important issues of the selection of replication—which agents to be replicated, where and when to replicate agents, and how many replicas to be made. It is often infeasible to replicate every agent in the multiagent systems because replication is costly, and more importantly, for lots of MAS applications, the computation resource in the hosting nodes is limited. Therefore, replication should be applied optimally in order to ensure the maximal survivability of the MAS. The COD algorithm of Kraus et al. has touched this selection issue a bit, however, their algorithm is computationally intractable.

Part 2 of this thesis aims at better approaches which are motivated by the idea of agent replication, yet tackle the above shortcomings. Recall that our objectives in Part 2 are:

**Objective 2:** To develop *distributed survivability models* which maximise the survivability of a multiagent system and are capable of adapting to the changing environment—tackling the shortcomings 1 and 2.

**Objective 3:** To develop *efficient centralised survivability algorithms* that are able to measure and compute the survivability of a given multiagent system, and thus to guide replication in the multiagent system.

We now give a brief overview of our approaches to multiagent survivability below.

**Two Architectures for Multiagent Survivability**

We investigate two types of architectures for multiagent survivability in Part 2 of this thesis in order to fulfill two objectives.

Figures 5.1 and 5.2 depict the *centralised* architecture and the *distributed (or agent-oriented)* architecture respectively for the sake of multiagent survivability. As shown in both figures, the network consists of a set of nodes. We assume that a multiagent system is deployed over the network of hosting nodes. In the centralised architecture in Figure 5.1, a special *survivability algorithm*, who is
in charge of the survivability of the multiagent system, locates on one node $n_2$. While in the distributed architecture, a special survivability agent, equipped with some survivability algorithm, is functioning on one or more nodes in the network. These special agents automatically collaborate to increase the survivability of the system.

These two types of survivability approaches will be discussed in the next two chapters:

**Chapter 6:** In this chapter, the distributed approach (Figure 5.2) will be elaborated to achieve objective 2. We introduce agent-oriented models that are completely distributed and can redeploy agents when there is a need to re-evaluate the survivability of agents due to external events. The focus of our
work is on the development of the special survivability agents’ behaviours—how to redeploy regular agents. In addition, the locations of the special survivability agents should be decided. We show that the proposed distributed algorithms can be built on top of any centralised survivability algorithm.

Chapter 7: Objective 3 is accomplished by the centralised approach (Figure 5.2) in this chapter, where the focus of our work is to develop centralised survivability algorithms to compute the survivability of a given multiagent system. We first propose the expensive, exact algorithms, then we introduce various fast polynomial-time heuristics. We design various scenarios to investigate the performance of the proposed algorithms.

Now we are ready to present our agent-oriented approach to multiagent survivability.
Chapter 6

Agent-Oriented Survivability Models

As we have shown in the previous chapter, most of existing methods to ensure survivability of multiagent systems are centralised and not adaptive enough: first, the survivability of the MAS is questionable if the node hosting the survivability algorithm goes down; second, no mechanism exists to re-deploy the MASs when external events trigger a re-evaluation of the survivability of the MAS. This chapter introduces an agent-oriented (or distributed) approach to ensure the survivability of multiagent systems. Our approach aims to tackle the above two problems.

After introduction, we describe three distributed algorithms (DSA for short), which can be built on top of any centralised survivability algorithm (CSA). A set of experiments assessing the efficiency of the proposed algorithms is then reported, followed by a conclusion.

6.1 Introduction

To date, most approaches to ensuring survivability of MASs are based on the idea of replicating or cloning agents so that if a node hosting that agent goes down, a copy of the agent residing on another network location will still be functioning. As shown in Chapter 5, existing replication based approaches suffer from two major flaws.

- The first major flaw is that the survivability algorithms themselves are centralised. In other words, even though the agents in the MAS may themselves
be distributed across the network, the survivability algorithm itself resides on a single node. Thus, if the node hosting the survivability algorithm goes down, the system is compromised. This way of “attacking” the MAS can be easily accomplished by a competent hacker.

- The second major flaw is that the survivability algorithms do not adapt to changes that affect the survivability of the MAS. Most algorithms assume that once the survivability algorithm tells us where to replicate agents, we just replicate the agents at the appropriate nodes and then ignore survivability issues altogether. It is clearly desirable to continuously (or at least regularly) monitor how well the MAS is “surviving” and to respond to changes in the level of survivability by redeploying the agent replicas to appropriate new locations.

In this chapter, we present three distributed algorithms to ensure that a multiagent system will survive with maximal probability. These algorithms extend centralised algorithms for survivability such as those developed by Kraus et al. [KST03] but are completely distributed and are adaptive in the sense that they can dynamically adapt to changes in the probability with which nodes will survive. In addition, the algorithms are shown to achieve a new deployment that preserves whatever properties the centralised algorithm has. If our distributed algorithms were built on top of existing centralised survivability algorithm, for instance COD of Kraus et al. which computes deployments of MASs that maximise the probability of survival of the deployment, then the resulting deployments created by our system would also maximise probability of survival. In this chapter, we incorporate COD into our distributed framework.

We have developed a prototype implementation of our algorithms and conducted detailed experiments to assess how good the algorithms are from three points of view:

- what is the CPU time taken to find a deployment,
- what is the amount of network time used for redeploying agents, and
- what is the survivability of deployments.
6.2 Assumptions

First, we present some assumptions our approach is based upon. We assume that an agent is a program that provides one or more services. Our framework for survivability is independent of the specific agent programming language used to program the agent. In addition, we assume that a multiagent application \( M \) is a finite set of agents. We will develop the concept of a deployment agent introduced in the next section that can be used by the MAS to ensure its own survivability.

We assume that there exists a network \( N = (V, E) \), where \( V \) is the set of nodes in the network and \( E = V \times V \) shows that \( N \) is a fully connected overlay network.\(^1\) In addition, we assume that each node \( n \in V \) has some memory, denoted \( space(n) \), that it makes available for hosting agents in a given multiagent system. We also use \( space(a) \) to denote the space requirements of an agent \( a \). If \( A \) is a set of agents, \( space(A) \) is used to denote \( \sum_{a \in A} space(a) \).

When a set \( D \) of nodes in a network \( N = (V, E) \) goes down, the resulting network is the graph \( N' = (V - D, E - \{(v_1, v_2) \mid (v_1, v_2) \in E \text{ and either } v_1 \in D \text{ or } v_2 \in D \}) \).

We define the deployment of a multiagent system \( M \) w.r.t. a network \( N \) as follows.

**Definition 6.2.1.** Given a network \( N = (V, E) \) and a MAS \( M = \{a_1, \ldots, a_n\} \), a deployment is a mapping \( \mu : V \rightarrow 2^M \) such that for all \( 1 \leq i \leq n \), there exists a \( v \in V \) such that \( a_i \in \mu(v) \).

Suppose \( \mu \) is a deployment of \( \{a_1, \ldots, a_n\} \) w.r.t. a network \( N \) and suppose \( N' = (V', E') \) is the resulting network when some set \( D \) of nodes goes down. \( \mu \) survives the loss of \( D \) iff the restriction \( \mu' \) of \( \mu \) to \( V - D \) is a deployment w.r.t. \( N' \). Intuitively, this definition says that a MAS survives when a set of nodes goes down if and only if at least one copy of each agent in the MAS is still present on at least one node that did not go down.

Our problem in this chapter is how to make a multiagent application adapt to the changes triggered by external events in order to ensure the maximum survivability of the MAS. The following example is one scenario to illustrate our problem.

---

\(^1\)This is a reasonable assumption as it does not require full connectivity of the underlying physical network (just that all nodes in the physical network are reachable—perhaps through multiple physical links—from all other nodes).
Example 12. Suppose $V = \{n_1, n_2, n_3, n_4\}$ and $M = \{a, b, c, d\}$. The current deployment is given by:

\[
\mu_{\text{old}}(n_1) = \{a\}, \mu_{\text{old}}(n_2) = \{b, d\}, \\
\mu_{\text{old}}(n_3) = \{a, b\}, \mu_{\text{old}}(n_4) = \{b, c\}.
\]

Suppose the system administrator of node $n_4$ announces that this node will go down in an hour in order to perform an urgent maintenance task. It is easy to see that $\mu_{\text{old}}$ will not survive this event as agent $c$ will not be present in any of the nodes. Thus, $\mu_{\text{old}}$ should be changed: a copy of agent $c$ should be deployed on one of the nodes $n_1, \ldots, n_3$ and additional copies of $b$ may also be deployed in $n_1, \ldots, n_3$. Space restrictions of these nodes may lead to additional changes in the deployment, e.g., a copy of $d$ may be moved from node $n_2$ to node $n_3$.

6.3 Distributed Multiagent Survivability

In this section, we present three alternative algorithms to ensure the survivability of a MAS. As mentioned above, we assume that our distributed survivability algorithms (or DSA) are built on top of some arbitrary, but fixed centralised survivability algorithm CSA. Several such algorithms exist such as those in [KST03]. We now describe each of these three algorithms. It is important to note that all copies of the deployment agent perform the actions here, not just one copy. If only one copy performed the computations, then we would just have a centralised algorithm.

As mentioned earlier, our algorithms will use a special deployment agent ($da$ for short) which will ensure the survivability of the multiagent system. The deployment agent $da$ is added to the original MAS as an additional survivability agent.

**Definition 6.3.1.** Suppose $\mathcal{M}$ is a multiagent application. The survivability enhancement of $\mathcal{M}$, denoted $\mathcal{M}^*$, is the set $\mathcal{M} \cup \{da\}$ where $da$ is a special agent called deployment agent.

The rest of this section focuses on three distributed algorithms ASA1, ASA2, and ASA3. These algorithms give different methods to design and implement the deployment agent $da$ according to the following two aspects:

- the location of $da$: In algorithm ASA1, the deployment agent $da$ is added to each node of the network. While in ASA2 and ASA3, $da$ is added to
some nodes in the network based upon the centralised algorithm.

- the behaviour of da: In algorithms ASA1 and ASA3, the deployment agent da moves the agents in the MAS based on our proposed algorithm. While in algorithm ASA2, the behaviour of da is guided by a random based algorithm.

We discuss three algorithms in details next.

The ASA1 Algorithm

The ASA1 algorithm deploys a copy of da in every node of the network. We make the following assumptions about da: (i) da knows the current deployment, (ii) whenever a new deployment needs to be computed, da is triggered, (iii) da is built on top of any arbitrary centralised survivability algorithm CSA.

As da is located in each node, we will assume that for any $n \in V$, $\text{space}(n)$ is the available memory on $n$ excluding the space for da.

Whenever the da agents are notified that a new deployment needs to be computed, each copy of the da agent performs the following steps:

1. It examines the current deployment $\mu_{old}$;
2. Once da is told to redeploy by an external process, it uses the CSA algorithm to compute a new deployment $\mu_{new}$;
3. da stores the difference between $\mu_{old}$ and $\mu_{new}$ in a special data structure called a difference table (dif for short). The difference table dif has the following schema:

   - Node (string): the node identifier for all nodes in the network;
   - Deploy (set of string): a set of agents that are currently located in the node according to the old deployment $\mu_{old}$;
   - Insrt (set of string): a set of agents that are presently not located in the node but need to be allocated according to the new deployment $\mu_{new}$;
   - Remv (set of string): a set of agents that are presently located in the node but need to be deleted from it according to the new deployment $\mu_{new}$;
6.3. DISTRIBUTED MULTIAGENT SURVIVABILITY

Table 6.1: A difference table generated by the deployment agent

<table>
<thead>
<tr>
<th>Node</th>
<th>Insrt</th>
<th>Remv</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td>n_1</td>
<td>b</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>n_2</td>
<td>c</td>
<td>d</td>
<td>b, d</td>
</tr>
<tr>
<td>n_3</td>
<td>d</td>
<td>b</td>
<td>a, b</td>
</tr>
<tr>
<td>n_4</td>
<td>d</td>
<td>b, c</td>
<td>b, c</td>
</tr>
</tbody>
</table>

4. Each copy of da at each node looks at its Insrt and Remv columns and makes a decision on how to delete and/or add agents from its node.

Notice that at any given instance in time, all the deployment agents on all nodes have the same difference table. Our key task is to design Step 4. Before doing this, we present an example of a difference table.

Example 13. Consider the MAS and $\mu_{old}$ of Example 12. Consider a new deployment: $\mu_{new}(n_1) = \{a, b\}$, $\mu_{new}(n_2) = \{b, c\}$, $\mu_{new}(n_3) = \{a, d\}$, and $\mu_{new}(n_4) = \{d\}$. In this case, the difference table between $\mu_{old}$ and $\mu_{new}$ is given by Table 6.1.

Adding agents to the nodes or deleting agents from the nodes can be performed according to the difference table. However, these operations should be handled very carefully as there are two constraints that must be satisfied during the whole re-deployment process.

- Space constraint: While the add or deletion operations are being performed, the space constraint on each node in the network must be satisfied. For example, before we add an agent $a$ to the node $n$, we must make sure there is enough space in the node $n$ for locating $a$.

- Replication constraint: at any point in time during step (4), there must exist at least one copy for each agent in the MAS $a \in M$ within the network.

Example 14. We want to note that Step 4 is not as simple as the readers may think from the text. To see why Step 4 is not easy to implement, consider the difference table in Table 6.1. One may be tempted to say that we can implement the insertions and deletions as follows: (i) Insert agent $c$ on node $n_2$. (ii) Delete agent $d$ from node $n_2$. Notice however that we can insert $c$ on $n_2$ only if there is enough space on $n_2$ to accommodate agents $b, c, d$ simultaneously (as otherwise the host node $n_2$ may reject the insertion of agent $c$) for space violations. Alternatively, one may be tempted to first delete agent $d$ from node $n_2$ to free space to
6.3. DISTRIBUTED MULTIAGENT SURVIVABILITY

insert agent \( c \)—but this means that agent \( d \) has disappeared from all nodes and is hence lost for ever.

Before presenting our algorithm for deleting and adding agents, we first present a few definitions of concepts that will be used in the algorithm.

**Definition 6.3.2.** An agent \( a \) can be safely deleted from node \( n \) (denoted by \( \text{safeDel}(a, n) \)) if the number of copies of agent \( a \) in the Deploy column of the difference table is larger than the number of copies of the agent \( a \) in the Remv column.

When an agent can be safely deleted, we are guaranteed that at least one copy of the agent is present elsewhere on the network. In our running example (Table 6.1), the only agent that can be safely deleted is agent \( b \) at node \( n_3 \).

We use \( \text{Insrt}(n) \), \( \text{Remv}(n) \) and \( \text{Deploy}(n) \) to denote the insert list, the remove list and the deploy list of a node \( n \in V \) in the difference table. The implementation of the deployment agent \( da \) in ASA1 algorithm is based on a set of logical rules governing the operations of \( da \). We first present these rules before describing the algorithm in detail. The rules use the following action predicates (the predicates representing actions are used in much the same way as in Kowalski’s and Green’s formulations of planning, cf. [Nil98]).

- **ADD(a, n):** Add agent \( a \in \mathcal{M} \) to node \( n \in V \).
- **DEL(A, n):** Delete a set of agents \( A \subseteq \mathcal{M} \) from node \( n \in V \).
- **SWITCH(A, n, A', n')**: Switch two sets of agents \( A \subseteq \mathcal{M} \) and \( A' \subseteq \mathcal{M} \) that are located on nodes \( n \) and \( n' \) respectively.
- **remdif(A, L, n) and insdif(A, L, n)**: Suppose \( A \) is a set of agents, \( L \) is a string \( L \in \{\text{Remv}, \text{Insrt}, \text{Deploy}\} \), and \( n \) is a node. \( \text{remdif}(A, L, n) \) removes all the nodes in the \( L \)-list of node \( n \) in the difference table. Likewise, \( \text{insdif}(A, L, n) \) inserts all the nodes in \( A \) into the \( L \) list of node \( n \)'s entry in the difference table.

Note that \( \text{Insrt}(n) \) represents the \( \text{Insrt} \) field of node \( n \) in the difference table. It specifies what new agents must be inserted into node \( n \). In contrast, \( \text{insdif}(A, \text{Insrt}, n) \) specifies that \( \text{Insrt}(n) \) must be updated to \( \text{Insrt}(n) \cup A \), i.e. it refers to an update of the difference table itself. In the example of Table 1,
remdif(\{b\}, Deploy, n_2) causes the deploy field associated with n_2 to be reset to just \{d\} instead of \{b,d\}.

We now introduce the rules governing the execution of these actions.

Rule 1. The first rule says that if A is a set of agents each of which can be safely deleted from node n, then A can be removed from node n.

\[ \text{DEL}(A, n) \leftarrow (\forall a \in A) \text{safeDel}(a, n) \]

Rule 2. This rule says that if a set A of agents is deleted from node n, we need to update the difference table by removing A from the remove and deploy lists of node n.

\[ \text{remdif}(A, \text{Remv}, n) \land \text{remdif}(A, \text{Deploy}, n) \leftarrow \text{DEL}(A, n) \]

Rule 3. This rule says that an agent a can be added to node n if there is sufficient space on node n to accommodate a’s memory needs.

\[ \text{ADD}(a, n) \leftarrow (\text{space}(n) - \text{space}(\text{Deploy}(n)) \geq \text{space}(a)) \]

Rule 4. If agent a is added to node n, we must remove its id from the insert column and add it to the deploy column of node n.

\[ \text{remdif}(\{a\}, \text{Insrt}, n) \land \text{insdif}(\{a\}, \text{Deploy}(n)) \leftarrow \text{ADD}(a, n) \]

Rule 5. This rule says that two sets of agents, A deployed on node n and A' on node n', can be switched if (1) A' is a subset of the insert set on node n as well as A' is in the deleted set of node n'; (2) A is a subset of the remove set on node n and it is also in the added list of node n'; (3) furthermore, the switch action has an auxiliary function \text{CHKSWITCH}(A, n, A', n'), which checks if the space constraints on switching A and A' between n and n' are satisfied.

\[ \text{SWITCH}(A, n, A', n') \leftarrow \]
\[ A' \subseteq \text{Remv}(n') \land A' \subseteq \text{Insrt}(n) \land A \subseteq \text{Remv}(n) \land A \subseteq \text{Insrt}(n') \land \]
\[ \text{CHKSWITCH}(A, n, A', n'). \]
\[ \text{CHKSWITCH}(A, n, A', n') \leftarrow \]
\[ (\text{space}(n) - \text{space}(\text{Deploy}(n)) + \text{space}(A) \geq \text{space}(A')) \land \]
\[ (\text{space}(n') - \text{space}(\text{Deploy}(n')) + \text{space}(A') \geq \text{space}(A)). \]

SWITCH(A, n, A', n') performs appropriate ADD and DEL actions on agents at the appropriate nodes.

\[ (\forall a \in A')\text{ADD}(a, n) \land (\forall a \in A)\text{ADD}(a, n') \land \text{DEL}(A', n') \land \text{DEL}(A, n) \leftarrow \text{SWITCH}(A, n, A', n') \]
Rule 6. This rule says that when the switch action $\text{SWITCH}(A, n, A', n')$ is performed, we must update the difference table $\text{dif}$ accordingly.

\[
\text{remdif}(A, \text{Remv}, n) \land \text{remdif}(A', \text{Remv}, n') \land \text{remdif}(A', \text{Insrt}, n) \\
\land \text{remdif}(A, \text{Insrt}, n') \land \text{remdif}(A, \text{Deploy}, n) \land \text{insdif}(A', \text{Deploy}, n')
\]

\[\leftarrow \text{SWITCH}(A, n, A', n').\]

Rule 7. The rules below deal with the case where there is no agent that can be safely deleted from node $n$ (the case shown in rule 1) and there is no currently available space for adding an agent (as described in rule 3) and there is no direct switch that could be performed (the case of rule 5). That is, when more than two nodes are involved in a switch, we need the following rules.

\[
\text{SWITCH}(A, n, A', n') \leftarrow \\
A' \subseteq \text{Remv}(n') \land A \subseteq \text{Remv}(n) \land (\exists B \subseteq A')B \subseteq \text{Insrt}(n) \land \\
\text{CHKSWITCH}(A, n, A', n').
\]

\[\forall a \in A')\text{ADD}(a, n) \land (\forall a \in A)\text{ADD}(a, n') \land \text{DEL}(A', n) \land \text{DEL}(A, n)
\]

\[\leftarrow \text{SWITCH}(A, n, A', n').\]

When switching $A$ and $A'$, if we move an agent $b$ to a node where $b$ is not the desired agent in the new deployment, we should delete $b$ from that node in the future process, that is, we should add $b$ to the delete list of the node.

\[
\text{remdif}(A, \text{Remv}, n) \land \text{remdif}(A', \text{Remv}, n') \land \text{remdif}(A', \text{Deploy}, n') \land \\
\text{insdif}(A', \text{Deploy}, n') \land \text{remdif}(A', \text{Deploy}, n') \land \text{insdif}(A, \text{Deploy}, n')
\]

\[\leftarrow \text{SWITCH}(A, n, A', n').\]

\[
\text{insdif}(\{b\}, \text{Remv}, n) \leftarrow (\forall b \in A')b \notin \text{Insrt}(n) \land \text{SWITCH}(A, n, A', n').
\]

\[
\text{remdif}(\{b\}, \text{Insrt}, n) \leftarrow (\forall b \in A')b \notin \text{Insrt}(n) \land \text{SWITCH}(A, n, A', n').
\]

\[
\text{insdif}(\{b\}, \text{Remv}, n') \leftarrow (\forall b \in A)b \notin \text{Insrt}(n') \land \text{SWITCH}(A, n, A', n').
\]

\[
\text{remdif}(\{b\}, \text{Insrt}, n') \leftarrow (\forall b \in A)b \notin \text{Insrt}(n') \land \text{SWITCH}(A, n, A', n').
\]

Our algorithm ASA1 to redeploy a multiagent application is based on the above set of rules. ASA1 takes a network $\mathcal{N}$, a multiagent application $\mathcal{M}$, and a difference table $\text{dif}$ as inputs. The algorithm changes the deployment of $\mathcal{M}$ in the network $\mathcal{N}$.
Algorithm 1. ASA1(\(N, M, \text{dif}\))

(* Inputs: (1) a network \(N = (V, E)\) *)
(* (2) a multiagent application \(M\) *)
(* (3) a current difference table \(\text{dif}\) *)

1. \(\text{flag}_1 = \text{true}\)

2. while \(\text{flag}_1\) do

   • if (for all \(n \in V\), \(\text{Remv}(n) = \emptyset\) and \(\text{Insrt}(n) = \emptyset\))
     then \(\text{flag}_1 = \text{false}\);

   • else, do

     (1) \(\text{flag}_2 = \text{true}\), \(\text{flag}_3 = \text{true}\)

     (2) while \(\text{flag}_2\), do

        (a) \(\text{flag}_2 = \text{false}\)

        (b) for each \(n \in V\), do

           A. \(A = \text{Remv}(n)\)

           B. if \(A \neq \emptyset\), then

                \((\text{dif}, \text{flag}_2) = \text{DEL}(A, n, \text{dif}, \text{flag}_2)\) (* delete agent *)

        (c) for each \(n \in V\) do

           A. \(A = \text{Insrt}(n)\)

           B. if \(A \neq \emptyset\), then

                \((\text{dif}, \text{flag}_2) = \text{ADD}(A, n, \text{dif}, \text{flag}_2)\) (* add agents *)

     (3) for each \(n \in V\), do

        if \(\text{flag}_3\), then

        i. \(A = \text{Insrt}(n)\)

        ii. if \(A \neq \emptyset\), then \((\text{dif}, \text{flag}_3) = \text{SWITCH}(A, M, n, \text{dif}, \text{flag}_3)\)

           (* switch agents *)

There are three auxiliary functions in Algorithm 1 ASA1: \text{DEL}, \text{ADD} and \text{SWITCH}. The function \text{DEL} is implemented based on Rules 1 and 2 above. The function \text{DEL}(A, n, \text{dif}, \text{flag}) receives as input: (1) a set of agents \(A\), (2) a node \(n\) (3) a current difference table \(\text{dif}\) and (4) a flag. For each agent in \(A\), the algorithm checks if it can safely delete the agent; if so it deletes the agent from \(n\) and updates the \(\text{dif}\) table. It returns the updated \(\text{dif}\) table and sets the flag to be true if any agent was deleted.
Algorithm 2. DEL($A, n, \text{dif}, \text{flag}$)

1. for each agent $a \in A$, do
   
   if $\text{safeDel}(a, n) = \text{true}$, then
   
   (a) remove agent $a$ from node $n$
   
   (b) $\text{Remv}(n) = \text{Remv}(n) \setminus \{a\}$, $\text{Deploy}(n) = \text{Deploy}(n) \setminus \{a\}$
   
   (c) $\text{flag} = \text{true}$

2. return dif and flag

The function ADD is based on Rules 3 and 4. It receives as an input (1) a set of agents $A$, (2) a node $n$, (3) the current difference table dif, and (4) a flag. For each agent $a \in A$ if there is enough space on $n$ to deploy $a$, it adds $a$ to $n$, updates the dif table and changes $\text{flag}$ to indicate an agent has been added to $n$. It returns the updated difference table dif and the flag. The following algorithm shows how it works.

Algorithm 3. ADD($A, n, \text{dif}, \text{flag}$)

1. for each agent $a \in A$, do
   
   if $\text{space}(n) - \text{space}(\text{Deploy}(n)) \geq \text{space}(a)$, then
   
   i. add agent $a$ to node $n$
   
   ii. $\text{Insrt}(n) = \text{Insrt}(n) \setminus \{a\}$, $\text{Deploy}(n) = \text{Deploy}(n) \cup \{a\}$
   
   iii. $\text{flag} = \text{true}$

2. return dif and flag

The SWITCH function is based upon Rules 5–7. SWITCH uses a subroutine called CHKSWITCH. This function checks to see if any space overflows occur when exchanging a set $A$ of agents current on node $n$ with a set of agents $A'$ currently on node $n'$. If no space overflow occurs, it returns true—otherwise it returns false.

Algorithm 4. SWITCH($R, M, n, \text{dif}, \text{flag}$)

(* Input: (1) a set of agents $R$ *)

(* (2) multiagent application $M$ *)

(* (3) node id $n$ *)

(* (4) current difference table dif *)

(* (5) flag *)

(* Output (1) updated difference table dif *)

(* (2) flag *)
1. for each agent \( a \in R \), if (flag), do

   - if there exists a set \( A \subseteq Remv(n) \) and a set \( A' \subseteq Remv(n'), n' \in V \) such that,
     - \( a \in A' \), and
     - \( CHKSWITCH(A, n, A', n', \text{dif}) = \text{true} \)
   - then, do
     (a) switch \( A \) and \( A' \) between nodes \( n \) and \( n' \)
     (b) \( Remv(n) = Remv(n) \setminus A, Remv(n') = Remv(n') \setminus A' \)
     (c) \( \text{Deploy}(n) = \text{Deploy}(n) \cup A' \setminus A, \text{Deploy}(n') = \text{Deploy}(n') \cup A \setminus A' \)
     (d) for each \( b \in A' \), do
       if \( b \notin \text{Insrt}(n) \), then \( Remv(n) = Remv(n) \cup \{b\} \)
       else, \( \text{Insrt}(n) = \text{Insrt}(n) \setminus \{b\} \)
     (e) for each \( b \in A \), do
       if \( b \notin \text{Insrt}(n') \), then \( Remv(n') = Remv(n') \cup \{b\} \)
       else, \( \text{Insrt}(n') = \text{Insrt}(n') \setminus \{b\} \)
     (f) update \( \text{dif} \)
     (g) flag = false

2. return \( \text{dif} \) and \( \text{flag} \)

Algorithm 5. \( CHKSWITCH(A, n, A', n', \text{dif}) \)

1. set \( \text{result} = \text{false} \)

2. if \( \text{space}(n) - \text{space}(\text{Deploy}(n)) + \text{space}(A) \geq \text{space}(A') \) and
   \( \text{space}(n') - \text{space}(\text{Deploy}(n')) + \text{space}(A') \geq \text{space}(A) \)
   then, \( \text{result} = \text{true} \)

3. return \( \text{result} \)

The following lemmas are needed to prove that ASA1 is correct.

**Lemma 6.3.3.** Each execution of the actions DEL, ADD, and SWITCH always results in a decrease on the number of agents in column Remv or Insrt.

**Proof.** Rules 2 and 4 clearly show that the actions DEL(\( A, n \)) and ADD(\( a, n \)) remove agents from Remv(\( n \)) and Insrt(\( n \)) respectively. Now consider the SWITCH action. When switching agents between two nodes only (Rule 5), SWITCH(\( A, n, A', n' \)) removes \( A \) from Remv(\( n \)), \( A \) from Insrt(\( n' \)), \( A' \) from Remv(\( n' \)), and \( A' \) from
In the case of Rule 7, where more than two nodes are involved in the switch, action \( \text{SWITCH}(A, n, A', n') \) adds at most \( A + A' - 1 \) agents in \( \text{Remv}(n) \) and \( \text{Remv}(n') \), while removing at least \( A + A' + 1 \) agents from \( \text{Remv} \) and \( \text{Insrt} \) of node \( n \) and \( n' \). This shows that performing each action must reduce the number of agents in the \( \text{Remv} \) or \( \text{Insrt} \) columns.

\[ \text{Lemma 6.3.4. In each iteration of the while loop shown in Step 2 of algorithm AS1, at least one action (DEL, ADD, or SWITCH) must be executed.} \]

\[ \text{Proof. In Step (b) of the while loop (2), all possible deletions } \text{DEL} \text{ will be performed if agents in } \text{Remv} \text{ can be safely deleted according to Rule 1. In the loop of Step (c), all possible additions } \text{ADD} \text{ will be done based on constraints shown in Rule 3. Even if no action is performed in the while loop (2), in Step (3), according to Rule 5 and Rule 7, there must exist two sets of agents on two different nodes such that action } \text{SWITCH}(A, n, A', n') \text{ can be performed. This shows that for each while loop in Step 2, at least one action on agents will be executed.} \]

\[ \text{Theorem 6.3.5 (Correctness). Suppose the rules (Rule 1–7) are applied according to the order listed. Then the sequence of actions performed by Algorithm AS1 is the one performed by the rules. AS1 always terminates.} \]

\[ \text{Proof. In AS1, the execution of actions is determined by the rules. With the assumption that the actions of the rules are taken according to their order, actions executed by the algorithm are those entailed by the rules.} \]

\[ \text{In Algorithm AS1, the while loop of Step (2) makes sure that no more agents can be safely deleted and no more agents can be added to the nodes, i.e. } \text{flag}_2 = \text{false}. \text{ Thus, the loop in Step (2) terminates after some iterations.} \]

\[ \text{For each execution of the while loop in Step 2, according to Lemma 6.3.4, at least one action must be executed on some agent at some nodes. Moreover each action must reduce the size of } \text{Remv}(n) \text{ or } \text{Insrt}(n) \text{ as explained in Lemma 6.3.3. Thus the size of } \text{Remv} \text{ and } \text{Insrt} \text{ decreases monotonically with each iteration of the while loop. Therefore the algorithm must reach a step where for all } n \text{ in the network, } \text{Remv}(n) = \emptyset \text{ and } \text{Insrt}(n) = \emptyset, \text{ which sets } \text{flag}_1 \text{ to false, and the algorithm terminates.} \]

\[ \text{A corollary is immediately obtained for the algorithm AS1.} \]

\[ \text{Corollary 6.3.6. Algorithm AS1 is guaranteed to move the agents } A \in \mathcal{M} \text{ to the new locations on the network such that the survival of the multiagent system } \mathcal{M} \text{ is ensured.} \]
Example 15. Consider the network and the deployment of Example 12. Suppose each node in the network can store a deployment agent \( da \) and two regular agents. Suppose that the deployment agents were triggered and suppose they computed a new deployment as specified in Example 13. Then, each copy of \( da \) computes the difference table as listed in Table 6.1. According to algorithm \( ASA1 \), \( b \) is first deleted from node \( n_3 \) by \( da \) located on that node and \( b \) and \( c \) are deleted from node \( n_4 \) by its deployment agent. \( d \) is not deleted in the first round because it is not safe to delete it at that stage. \( b \) is then inserted into node \( n_1 \) (copied from \( n_2 \)) and \( d \) is inserted into node \( n_3 \) and \( n_4 \). \( d \) is then removed from \( n_2 \), and finally \( c \) is inserted into node \( n_2 \).

The ASA2 Algorithm

In this algorithm the deployment agent \( da \) is not located at each node. Instead, we add \( da \) to a multiagent system \( M \) to get an updated multiagent system \( M^* \) and apply the centralised algorithm on \( M^* \). This returns a new deployment \( \mu_{\text{new}} \) which is then executed by the deployment agent. In this case, the programming of \( da \) is somewhat different from the programming of it in \( ASA1 \) because there is no guarantee that each node has a copy of \( da \).

Algorithm \( ASA2 \) assumes that each agent has a mobility capability, i.e., it can obtain a movement instruction from a \( da \) and perform it. In addition, each agent can delete itself. Also, all agents in \( M \) as well as \( da \) satisfy the condition that whenever it receives a message from any \( da \) to move to another location, it does so. After performing the move, it sends a message to all deployment agents saying it has moved.

Once \( \mu_{\text{new}} \) is computed by CSA, each copy of \( da \) executes an algorithm called \( DELETECOPY \) that deletes all but one copy of all agents in \( M \). All copies of \( da \) send messages to the agent copies to be deleted telling them to delete themselves. \( da \) copies create a plan to move or copy the one remaining copy of each agent to the nodes specified by \( \mu_{\text{new}} \). Note that all copies of \( da \) perform the same actions at the same time.

**Algorithm 6. ASA2** \((\mathcal{N}, \mathcal{M}^*, \mu_{\text{old}}, \mu_{\text{new}})\)

\[
\begin{array}{ll}
(* & \text{Input:} \ (1) \ \text{network} \ \mathcal{N} = (V, E) \ (*) \\
(* & \text{ } \ (2) \ \text{multiagent application} \ \mathcal{M}^* \ (*) \\
(* & \text{ } \ (3) \ \text{current deployment} \ \mu_{\text{old}} \ (*) \\
(* & \text{ } \ (4) \ \text{new deployment} \ \mu_{\text{new}} \ (*)
\end{array}
\]
1. for each $a \in M^*$, do
   \[\text{DELETECOPY}(a, \mu_{old}); \quad (* \text{call function DELETECOPY} *)\]
2. $\text{flag} = \text{true}$;
3. while $\text{flag}$, do
   • if for all $n \in V$, $\mu_{old}(n) = \mu_{new}(n)$, then
     \[\text{flag} = \text{false}\]
   • else, do
     (a) $\text{flag2} = \text{false}$, $\text{flag3} = \text{true}$
     (b) for all $n \in V$, do
       i. $A = \mu_{new}(n)$
       ii. $\text{flag2} = \text{ADDCOPY}(A, \mu_{old}, \mu_{new}, n)$ \quad (* \text{call function ADDCOPY} *)
     (c) if ($\text{flag2} = \text{false}$), then
       for each $n \in V$, do
       if $\text{flag3}$, then
       i. $A = \mu_{old}(n)$
       ii. $\text{flag3} = \text{SWITCHCOPY}(A, \mu_{old}, \mu_{new})$ \quad (* \text{call function SWITCHCOPY} *)

Algorithm ASA2 includes a function $\text{DELETECOPY}(a, \mu_{old})$ which uses a deterministic algorithm to delete all but one copy of each agent. The delete actions are performed via a lexicographic order on nodes. It is important that all copies of $\text{da}$ use the same $\text{DELETECOPY}$ algorithm so that they all agree on what nodes each agent should be deleted from. The function updates the deployment of agents.

Likewise, algorithm $\text{ADDCOPY}(a, \mu_{old}, \mu_{new}, n)$ receives as inputs (1) an agent $a$, (2) an old deployment $\mu_{old}$, (3) a new deployment $\mu_{new}$, and (4) a node $n$. It adds a copy of agent $a$ to node $n$ if there is space on node $n$ and if the new deployment $\mu_{new}$ requires $a$ to be in $n$—it does this by asking a node currently hosting $a$ to clone and move such a copy of $n$. The function $\text{ADDCOPY}$ changes the deployment of agents.

The inputs of function $\text{SWITCHCOPY}$ are a set of agents, an old deployment and a new deployment. It returns a flag. Details of function $\text{SWITCHCOPY}$ in ASA2 are shown below.
Algorithm 7. SWITCHCOPY($A, \mu_{old}, \mu_{new}$)

(* Input: (1) a set agents $A$ *)

(* (2) old deployment $\mu_{old}$ *)

(* (3) new deployment $\mu_{new}$ *)

(* Output: (1) flag *)

1. for each agent $a \in A$, do

   • if there exists a set $A'$ on $n'$ such that
     (a) $a \in \mu_{old}(n')$ (* agent $a$ currently locates on node $n'$ *)
     (b) $\text{space}(n) - \sum_{a \in \mu_{old}(n')} \text{space}(a) + \text{space}(A) \geq \text{space}(A')$ (* if there is enough space on node $n$ for the agent set $A'$ *)
     (c) $\text{space}(n') - \sum_{a' \in \mu_{old}(n')} \text{space}(a') + \text{space}(A') \geq \text{space}(A)$ (* if there is enough space on node $n'$ for the agent set $A$ *)

   • then, do

     (a) switch $A$ and $A'$ between nodes $n$ and $n'$ and update $\mu_{old}$;
     (b) flag = false;

2. return flag

The following example illustrates how ASA2 redeploys the agents to the new locations in the network.

Example 16. Suppose nodes $n_1$ and $n_3$ of the network of Example 12 can store a da agent and two other agents. Suppose $n_2$ and $n_4$ can store only two regular agents. First, agents $a, b$ and da are removed from node $n_3$. Then, agent $b$ is removed from node $n_4$. The deployment agent da in node $n_1$ is responsible for all these deletions and for further updates. It also updates $\mu_{old}$ accordingly. $b$ is then added to node $n_1$, and $d$ and da are added to nodes $n_3$ and $n_4$. Only then is $d$ deleted from $n_2$. c is then added to $n_2$ and then deleted from $n_4$.

ASA2 can also ensure that the multiagent application is redeployed according to the new deployment. The proof is similar to the one done for ASA1.

The ASA3 Algorithm

Just as in algorithm ASA2, the deployment agent used in Algorithm ASA3 is not located on each node. Instead it is treated just like any other agent and deployed
using the CSA. However, the procedure to decide the order of deletion and adding the copies of agents to nodes is that of algorithm ASA1.

In addition, we assume that each agent in the MAS $a \in \mathcal{M}$ has a mobility capability and each agent $a \in \mathcal{M}$ is augmented with the following rules:

When any deployment agent $\text{da}$ sends it a message to move to a new location, it does so. After executing the move, it informs $\text{da}$ replicas that it has moved.

We further assume no other move operations in the agent that interfere with the above rules. The behaviour of the deployment agent is as follows.

Originally, the deployment agent $\text{da}$ is deployed (along with other agents) using the CSA algorithm. When the survivability of one or more nodes changes, each $\text{da}$ computes the difference table (as in the ASA1). Each $\text{da}$ then sends a message to all agents that can be safely deleted (including, possibly a deployment agent $\text{da}$) telling them to delete themselves and send a message just when they are about to finish the operation. After this, they send “move” or “exchange” messages to agents one at a time. When they get an acknowledgment that the move has been performed, they send a move message to the next agent, and so on until they are done. Note that while in Algorithm ASA1, agents can be moved or copied to other nodes simultaneously, in algorithm ASA3, this is done sequentially.

Algorithm ASA3 is guaranteed to re-deploy agents according to the new deployment. The proof is similar to the one done for ASA1.

\section*{6.4 Implementation and Experimental Results}

We developed a prototype implementation of all the above algorithms and tested them on a Linux PC. We first came out an experiment plan, where we decided how to measure the performance of three algorithms, what experiment environments should be considered—different algorithms may work differently for different environmental settings, and what kind of data should be used in experiments. After carefully designing the experiments, we implemented all of the algorithms
and conducted a large number of experiments with various settings and measurements. The details of the implementation and the experimental results are shown below.

We measured the performance of three algorithms in terms of:

1. **the survivability** of the deployment found by ASA1 and ASA3: note ASA2 has the same deployment survivability with ASA3. The survivability of the deployment is effected by the size of the deployment agents.

2. **the network time**: the time taken for communication between agents at different nodes as well as moving or copying an agent from one node to another node on the network.

3. **the CPU time**: the computation time of the algorithms.

The performance of the algorithms is evaluated as a function of:

1. **the problem size**: the sum of the numbers of agents and nodes. We also vary the size ratio of \( da \), that is, the ratio of size of \( da \) to the average size of agents.

2. **the agent density**: the ratio of number of agents to the number of nodes.

3. **the space density**: the ratio of sum of memory requirement of agents to the total memory available on nodes.

As to the agent sizes in the experiments, we used a sample of 31 existing agents (taken from existing IMPACT applications) to determine the distribution of agent sizes (in the range of 0 to 250 KB): it is with \( \frac{3}{31} \) probability to be size between \([150k, 300k]\); with \( \frac{8}{31} \) probability to be size between \([50k, 150k]\); and with \( \frac{20}{31} \) probability to be size between \((0k, 50k)\). We randomly generated the size of nodes and the disconnect probabilities of nodes in the network.

The way to generate deployments of agents is based on the idea of the knapsack problem [KST03]. It works as follows:

We first sort the nodes in ascending order according to their disconnect probabilities. Then we place agents on the sorted nodes starting from the node with the lowest dp. We put as many agents as possible on this node, then go to the nodes with the second lowest dp and so on.
We ran experiments with varying network bandwidths—we only report on experiments where the bandwidth was 100 KB/s (this is twice the bandwidth of a dial-in model, but much smaller than the bandwidth of broadband connections that may exceed 100 MB/s). The centralised survivability algorithm we used for our experiments was COD described in Chapter 5.

A brief workflow of the experiment is given as follows:

1. run ./prog \(N_n, N_a, BW, exp\);
   (* running the program with the command line arguments; \(N_n\) and \(N_a\) specify the number of nodes and agents; \(BW\) gives the average network bandwidth, and \(exp = 1, \ldots, 12\) defines which experiment to run. *)

2. generate\_environ(\(N_n, N_a\));
   (* function which randomly generates nodes’ space \(mem_n\), memory requirements of agents \(mem_a\), and memory requirement for the deployment agent \(mem_d\), where \(mem_d\) must be smaller than the minimum value of \(mem_n\) and smaller than the average size of agents *)

3. \(dp = getdp(N_n)\);
   (* generate the disconnect probability of each node according to different experiments *)

4. \(N = build\_network(N_n, mem_n, dp), A = build\_mas(N_a, mem_a)\);
   (* build a network and a multiagent system *)

5. \(\mu = COD(N_n, N_a, dp, mem_a, mem_n)\);
   (* get the first deployment by the centralised survivability algorithm *)

6. if ASA1, then (* if it is algorithm ASA1 *)
   
   for each \(n \in N\), do
   \(mem_n = mem_n - mem_d\); (* the available space on each node is reduced by subtracting the size of \(da\) *)

7. if(ASA2 or ASA3), then
   
   • \(A = A \cup da\); (* add deployment agent to the MAS *)
   • \(N_a = N_a + 1\). (* increase the number of agents *)

8. \(dp' = getdp(N_n)\); (* randomly change some nodes’ disconnect probabilities *)
9. $\mu' = COD(N_n, N_a, dp', mem_a, mem_n)$;
   (* get the new deployment by the centralised survivability algorithm *)

10. call function $\text{algo}(\mu', \mu, N', A, mem_a, mem_n, mem_a, mem_n)$.
    (* run the algorithm ASA1, ASA2, or ASA3 *)

In order to investigate the effects of the various environment settings on the performance of different algorithms, we ran the experiments with various settings, more precisely:

- Experiments 1–3 were carried out with varying problem size;
- Experiments 4–5 were carried out with varying agent density;
- Experiment 6 was done with varying space density.

We explain each experiment and report the experiment results next.

**Experiments with different problem size**

**Experiment 1.** In this set of experiments, we investigate the performance of three algorithms as problem size varies. The problem size refers to the sum of the number of nodes and the number of agents involved in the MAS deployment.

We enlarge the problem size from 9, 18 to 27.

The experiment settings are as follows.

- The size ratio is set to 0.2.
- The size ratio of the deployment agent $da$ is 0.1.
- The disconnect probabilities are generated randomly. The average value is around 0.2.

Table 6.2 shows the effect of problem size on the CPU time required by the algorithms as well as the network time required to move the agents to the new locations. The markings such as $n : 5, a : 4$ refer to a multiagent application of 4 agents which are deployed over 5 nodes in the network.

We made the following observations:
6.4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>problem size</th>
<th>n5, a4</th>
<th>n10, a8</th>
<th>n15, a12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASA1</td>
<td>ASA2</td>
<td>ASA3</td>
</tr>
<tr>
<td>CPU time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>microsecond</td>
<td>11</td>
<td>89</td>
<td>2</td>
</tr>
<tr>
<td>network time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>millisecond</td>
<td>0</td>
<td>6330</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2: Experiment 1: CPU and Network time of three algorithms with varying problem size when dps are uniformly distributed

1. CPU Time: ASA1 and ASA3 outperform ASA2 w.r.t CPU time. ASA1 and ASA3 are more or less incomparable. If the problem size is small, ASA1 and ASA3 have very close performance, however when the total number of agents and nodes is 18 and 27, ASA1 is more efficient than ASA3. The difference of time taken on network among ASA1, ASA2 and ASA3 is obvious. The time difference increases as the problem size is enlarged.

2. Network Time: again, ASA1 and ASA3 outperform ASA2. As the problem size gets larger, ASA1 outperforms ASA3.

**Experiment 2.** In this set of experiments, we investigate the performance of three algorithms as problem size varies when the disconnect probabilities of nodes vary dramatically. The experiment settings are similar to those in Experiment 1 except for the distribution of disconnection probability of nodes. Here, over 80% of nodes have low survival probability—the values are under 0.1, while the remaining 20% of nodes have a higher disconnect probability—the values are above 0.9. The distribution samples of dps are shown in Figure 6.1.

As shown in Table 6.3, similar to the results of Experiment 1, the computation and network time of ASA1 is much less than that of ASA2 and ASA3.

<table>
<thead>
<tr>
<th>problem size</th>
<th>n5, a4</th>
<th>n10, a8</th>
<th>n15, a12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ASA1</td>
<td>ASA2</td>
<td>ASA3</td>
</tr>
<tr>
<td>CPU time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>microsecond</td>
<td>12</td>
<td>89</td>
<td>3</td>
</tr>
<tr>
<td>network time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>millisecond</td>
<td>0</td>
<td>5980</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.3: Experiment 2: CPU and Network time of three algorithms with varying problem size when dps vary dramatically
Figure 6.1: Disconnection probabilities of the nodes

**Experiment 3.** In Experiment 3, we want to compare the survivability of agents with algorithms ASA1 and ASA3. The settings are similar with those in Experiment 1 except:

- we vary the size ratio of the deployment agent $da$ from 0.1, 0.3 to 0.8;
- the survival probabilities of nodes are randomly generated from 0.3 to 0.95 in steps of 0.05 with standard deviation 0.1;
- the x-axis is the mean of survival probability of nodes, varying from 0.3 to 0.95;
- the y-axis represents survivability of deployments.

Results are shown in Figure 6.2, where three figures represent different size of $da$ used in the experiment. In all figures, the survivability of agents decreases when the problem size increases. ASA1 and ASA3 have the same deployment performance if the problem size is small—the number of nodes is 5 and the number of agents is 4. However when problem size becomes larger, the survivability of the deployments identified by ASA3 becomes higher than the survivability of deployments identified by ASA1.

In addition, the results demonstrate the effect of $da$ agents on survivability of deployment. As the size of deployment agent increases, ASA3 gives higher survivability than ASA1. Especially when the ratio reaches to 0.8, ASA3 can
almost always finding better deployment than ASA1. Compared with ASA2 and ASA3, ASA1 deploy more da agents in the network, and hence, the amount of available space for adding regular agents is decreased.

Experiments with different agent density

Experiment 4. In Experiment 4, settings are similar to Experiment 1 but we vary agent density and keep the sum of number of agents and nodes unchanged. The number of nodes and agents vary from 15 and 3, 10 and 8, to 9 and 9, respectively.
Figure 6.3: Experiment 4: CPU time and network time with varying agent density.

Figure 6.3 demonstrates time taken by three algorithms as agent density enlarges. As shown in the figure, more time is needed on computation and communication between agents as the agent density increases. ASA1 shows its advantage of computation efficiency, especially compared to ASA2. In addition, ASA1 is faster than ASA3 in terms of CPU time when agent density becomes larger.

**Experiment 5.** To investigate the effects of different agent density on deployment feasibility, in Experiment 4, we did a set of experiments with ASA1 and ASA3 as agent density varies. We fixed the size ratio of da by 0.1.

Experimental results are shown in Figure 6.4. The results are quite similar with those in Experiment 3 where the problem size is changing. The survivability of agents decreases as the agent density increases. ASA1 and ASA3 have the
6.4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Figure 6.4: Experiment 5: survivability of deployment using algorithm 1 and algorithm 3 with changing agent density.

same deployment performance if the agent density is small. It is easy to see the advantage of ASA3 of finding good deployments over ASA1.

Experiments with different space density

Experiment 6. The ratio between the total resource requirement of agents and the total available resource on all nodes may affect the performance of three algorithms, especially on the deployments that different algorithms identify. Thus, in Experiment 6, we repeat Experiment 2 but vary space density from 0.1, 0.2, to 0.4. The number of nodes and agents is 10 and 8 respectively.

Figure 6.5 includes a set of curves representing survivabilities of deployments found by algorithm 1, 2 and 3 as space density varies. From the graphs, we notice that when available space on nodes becomes smaller and smaller, the survivability of deployment decreases steadily. Moreover, the influence of increasing space density is greater on ASA1 than on ASA3. This is because when the total available space on each node goes to a small number, the memory requirement of a deployment becomes crucial to find a feasible deployment. Thus when memory requirement of each da is comparable to that of regular agent, or when the space density is large, ASA3 turns out to be better method to get deployments which have higher survivability.
6.4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Summary of experiments

Based on the experiments that we carried out, we made the following conclusions:

- The above experiments imply that ASA1 is preferable to both ASA2 and ASA3 as far as time is concerned. In terms of network time, although the basic movement algorithm of ASA1 and ASA3 is the same, in ASA3, as there is no deployment agent at every node, da must send “move” message to agent one at a time, furthermore, da should wait for the acknowledgment from the agent before sending message to another agent. In contrast, in ASA1, da at each node is able to carry out actions “move” at the same time based on difference table. That reduces the whole network time especially when the agent density is small.

- In terms of survivability of deployments identified, if a multiagent application involves small number of agents and nodes, or, if the size of da is small compared to average size of agents in the application, both ASA1 and ASA3 can be considered since they have very close performance. But if the MAS applications consist of a large number of agents or nodes, or there is a great resource demand for the agents in the application, we would prefer ASA3 to ASA1.

- In addition, the network bandwidth plays an important role when we choose which algorithm to apply. With slow bandwidth, ASA1 is preferable to ASA3
because it is much fast. However, if the bandwidth in applications is high, say 100MB, the time difference taken by ASA1, ASA2 and ASA3 could be in the order of microseconds. Thus we may prefer ASA3 since it results in deployments whose survivability is relatively higher than that given by ASA1.

6.5 Conclusion

In this chapter, we presented three distributed algorithms DSA to ensure the maximal survivability of a multiagent deployment. Our algorithms can handle external triggers to compute a new deployment and then redeploy the agents to their new locations. The distributed algorithms can be built on top of any centralised survivability algorithm (or CSA). We assessed and summarised their performance based on a set of experimental results at the end of this chapter.

The CSA we used in this chapter is the one proposed by Kraus el.al. [KST03], where they are completely ignorant about the dependencies between node failures in the network. However, this assumption is not always valid. Moreover, because of the ignorance assumption, survival probabilities tend to be extraordinarily pessimistic (low). Since there are many cases where the ignorance assumption of [KST03] is not appropriate while the independence assumption is valid, in the next chapter, we will investigate the case where we know the node failures are independent.
Chapter 7

Centralised Survivability Algorithms

In the previous chapter, we introduced three distributed algorithms (DSA) which are build upon any arbitrary but fixed centralised survivability algorithm (CSA). In this chapter, we are going to study the centralised survivability algorithm CSA.

The centralised algorithm we used in our distributed models is COD proposed by Kraus et al. in [KST03]. It uses the assumption that we have no knowledge about the dependencies between the disconnections of nodes. This assumption is general enough so that it can be applied to any multiagent environments even when the relationship of the failures of the hosting nodes is unknown. In this chapter, however, we point out (in Section 7.1) that this assumption is not appropriate especially for real MAS applications. Instead, we now assume that we know that the failures of the nodes in the network are independent. We then study how to develop the improved CSA for computing the survivability of a given MAS deployment under this independence assumption.

First, Section 7.1 discusses in more details why the ignorance assumption is not appropriate. Section 7.2 then defines the axiomatic survivability function and Section 7.3 shows how to compute the survivability of the deployment under independence assumption. In addition, we consider the complexity of computing the survivability. After that, in Section 7.5 we develop several approximation algorithms for computing survivability, and report experimental results in Section 7.6.
7.1 Introduction

Kraus et al. [KST03] have previously given a probabilistic model of survivability in which they propose a centralised survivability algorithm COD to compute survivability. COD uses a linear programming model to define the survivability of a deployment under the assumption that we are completely ignorant about node failure dependencies. The advantage of this assumption is that the algorithm COD is general enough and is not constrained by whether failures of hosting nodes in the network are independent or not. Furthermore, COD can be used even if we do not have knowledge about the node failure dependencies. However, we point out that this ignorance assumption is not appropriate because of the following reasons.

- The ignorance assumption about the relationship of the disconnections of the nodes is not always valid—for example, if there is an attack on UK government computers, it is unlikely that the computers of UNESCO (the United Nations Educational, Scientific and Cultural Organisation) are under attack as well. Thus, the events “UMD computers go down” and “UNESCO computers go down” are probably independent.

- Likewise, because of the ignorance assumption, survival probabilities computed by COD tend to be very low. Kraus et al. showed in [KST03] that finding an optimal deployment is intractable under their ignorance assumptions. The algorithm COD that they provided is exponential for computing the survivability of a given deployment. Moreover, it is double exponential to find a deployment that maximises the probability of survival. High complexity makes COD inapplicable for the online computation when we intend to apply the distributed algorithms to ensure the adaptive deployment, as shown in the previous chapter (not to mention if the size of the involved network or the multiagent application is large).

As we can see above, there are some cases where the ignorance assumption is not appropriate but the independence assumption is valid. Therefore, in this chapter, we assume that we know the failures of the nodes in the network are independent of one another. Based on this assumption, we study how to develop algorithms to measure the survivability of a MAS deployment that can be applied for real applications.
7.2 Axiomatic Survivability Functions

As our aim is to develop algorithms to measure the survivability of a given MAS, we are interested to define the rules that such algorithms should satisfy. In this section, we introduce a notion of an *axiomatic survivability function* that satisfies various survivability axioms. As we will see the approach in this section generalises the approach of Kraus et al.

Before presenting the notion of an axiomatic survivability function, we first briefly review the notions and the assumptions that we will use in this chapter. We assume that a multiagent system \( \mathcal{M} \) is a finite set of agents. Beyond that, we make no assumptions about what language the agents are written in. We assume that for a multiagent system \( \mathcal{M} \) to function, at least one copy of each agent in the system must be accessible. We assume the existence of a network \( \mathcal{N} = (V, E) \), where \( V \) is the set of nodes in the network and \( E = V \times V \)—that means that \( \mathcal{N} \) is a fully connected overlay network. We further assume that a *deployment function* w.r.t. \( \mathcal{M}, \mathcal{N} \) is a mapping \( \mu \) from \( V \) to \( 2^\mathcal{M} \) such that:

1. For each node \( n \in V \), \( \Sigma_{a \in \mu(n)} \text{space}(a) \leq \text{space}(n) \).
2. For each \( a \in \mathcal{M} \), there exists an \( n \in V \) such that \( a \in \mu(n) \).

Throughout the rest of this chapter, we will assume that \( \mu \) is an arbitrary but fixed deployment, and that \( \mathcal{N} \) is an arbitrary, but fixed network. As a consequence, we will just say “deployment” instead of “deployment w.r.t. \( \mathcal{M}, \mathcal{N} \)”.

We now start by giving the definition of *ordering on deployments* as follows.

**Definition 7.2.1 (ordering on deployments).** Suppose \( \mu, \mu' \) are deployments. We say that \( \mu \leq \mu' \) if and only if \( \forall n \in V, \mu(n) \subseteq \mu'(n) \).

Intuitively, if \( \mu \leq \mu' \), then this means that the deployment \( \mu' \) starts from the deployment \( \mu \) and adds zero or more agents to each node.

A *disconnect probability function* (dp for short) is a mapping \( dp : V \rightarrow [0, 1] \) that assigns to each node \( n \in V \), a probability of “going down” or somehow being disconnected from the network.

**Definition 7.2.2 (ordering on disconnect probability functions).** Suppose \( dp, dp' \) are two disconnect probability functions. We say that \( dp \leq dp' \) if and only if \( \forall n \in V, dp(n) \leq dp'(n) \).
Intuitively, \( dp \leq dp' \) means that the disconnect probabilities assigned by \( dp' \) to any node is always at least the value assigned by \( dp \) (but could be more).

We are now ready to axiomatically define a **survivability function**.

**Definition 7.2.3.** (survivability function) A survivability function \( SF \) is any mapping that takes as input, a deployment function \( \mu \) and a disconnect probability function \( dp \), and returns as output, a real number in the unit interval \([0, 1]\) such that the following axioms are satisfied:

1. If \( \mu \leq \mu' \), then \( SF(\mu', dp) \geq SF(\mu, dp) \).
2. If \( dp' \leq dp \) then \( SF(\mu, dp') \geq SF(\mu, dp) \).

The first axiom says that when the disconnect probability is fixed, and we make a deployment “larger” (by possibly adding more agents to zero or more nodes), the survivability of the deployment cannot go down. This makes sense as increasing the number of nodes at which an agent is located should not reduce survivability. The second axiom says that if we hold the deployment fixed and replace the disconnect probability function by one that assigns potentially higher disconnect probabilities to zero or more nodes, then the survivability should decrease. This also makes sense as the probability of disconnection of a node should reduce the probability of survival of the whole deployment. The following example illustrates these concepts.

**Example 17.** Consider a fully connected overlay network with \( V = \{n_1, n_2, n_3\} \) and a multiagent application \( M = \{a_1, a_2, a_3\} \). Suppose the disconnect probability function \( dp \) is given by:

\[
\begin{align*}
dp(n_1) &= 0.7, \ dp(n_2) = 0.6, \ dp(n_3) = 0.4.
\end{align*}
\]

Suppose the current deployment \( \mu \) is given by:

\[
\begin{align*}
\mu(n_1) &= \{a_1, a_2, a_3\}, \ \mu(n_2) = \{a_1\}, \ \mu(n_3) = \{a_2, a_3\}.
\end{align*}
\]

Clearly, if \( dp' \) assigns \( 0.7, 0.7, 0.7 \) to \( n_1, n_2, n_3 \), then \( dp \leq dp' \).

Likewise, if \( \mu' \) is:

\[
\begin{align*}
\mu'(n_1) &= \{a_1, a_2, a_3\}, \ \mu'(n_2) = \{a_1, a_2, a_3\}, \ \mu'(n_3) = \{a_1, a_2, a_3\}.
\end{align*}
\]

then we have \( \mu \leq \mu' \).

It is obvious that COD of Kraus et al. in [KST03] (see Chapter 5) satisfies the above two axioms.
7.3 Computing Survivability

Kraus et al. define the probability of survival of a deployment under the assumption that they are *completely ignorant* about the dependencies between node failures. As we stated above, an alternative more reasonable assumption is that we know that all node failures are mutually independent. In this section, we therefore define algorithms for computing the survivability of a given deployment under this independence assumption. In particular, we introduce the three algorithms:

- *SF1* is based on the linear programming;
- *SF1\textsubscript{n}* is a node based approach;
- *SF1\textsubscript{a}* is an agent based algorithm.

**Theorem 7.3.1 (Assumption).** As we always make the investigation under the independence assumption in this chapter, for simplicity, throughout the rest of this chapter, we will say “survivability” instead of “survivability under independence assumption”.

We now extend a linear programming based algorithm COD to compute the survivability of a given deployment under independence assumption.

**SF1: a linear programming approach**

Suppose we are given a fully connected network $\mathcal{N} = (V, E)$, a multiagent application $\mathcal{M}$, and a deployment $\mu$. For any two nodes $n_i$ and $n_j$, assuming that the disconnection of $n_i$ and $n_j$ are independent leads to the following constraint:

$$\sum_{\mathcal{N}'' \in F N \wedge n_i \in \mathcal{N}'' \wedge n_j \in \mathcal{N}''} P(\mathcal{N}'') = (1 - dp(n_i))(1 - dp(n_j)) \quad (7.1)$$

This constraint says that for any two nodes $n_i, n_j \in V$, the sum of the probabilities of all possible future networks $\mathcal{N}''$ where both node $n_i$ and $n_j$ survive must be the product of the survival probability of node $n_i$ and probabilities of the survival probability of node $n_j$. Similar constraints must be added for sets of 3 nodes, 4 nodes, and so on. As a consequence, we expand the set $CONS(dp, \mathcal{N})$ of constraints given in COD described in Chapter 5 by including this set of new
7.3. COMPUTING SURVIVABILITY

Given constraints and get the following linear program.

\[
\text{minimize} \quad \sum_{N' \in FN \land (\mu \text{ is a deployment w.r.t. } N')} P_{N'}
\]

subject to \(CONS2(dp, V)\):
- \(\sum_{N' \in FN \land n \in N'} P_{N'} = 1 - dp(n)\)
- \(\sum_{N' \in FN} P_{N'} = 1\)
- \(P_{N'} \geq 0\) for any \(N' \in FN\)
- \(\sum_{(N'' \in FN \land n_i, n_j)} P_{N''} = (1 - dp(n_i))(1 - dp(n_j))\) for any \(n_i, n_j \in V\)
- \(\vdots\)
- \(\sum_{N'' \in FN \land n_i, \ldots, n_m} P_{N''} = \sum_{n_k \in V} (1 - dp(n_k))\) for all \(n_k \in V\)

We can get the survivability of the deployment \(\mu\) by solving the above linear program. We call this linear programming approach \(SF1\).

The following result is an immediate consequence of the fact that the set of constraints used in \(CONS2(dp, V)\) is a superset (or equal to) the set of constraints used in \(CONS1(dp, V)\).

**Proposition 7.3.2.** Given a deployment \(\mu\) and a network \(N\), \(SF1(\mu, dp) \geq COD(\mu, dp)\).

Proposition 7.3.2 says that assuming independence always causes survival probability to go up as compared to assuming ignorance of dependencies between node failures.

**Two Exact Algorithms**

As computing the survivability of a deployment is at least \(NP\)-hard (we will show it later in this section), we can present only exponential algorithms to compute the survivability of a given deployment. However, computing the survivability of a given deployment by solving the linear program is more complex because of the enormous size involved. We now provide two algorithms \(SF1_n\) and \(SF1_a\) to compute the survivability without using linear programming:

- \(SF1_n\): The complexity of the algorithm \(SF1_n\) is exponential in the number of nodes. Thus, it can be used in environments where the number of nodes is small and there are many agents,
• **SF1**: The complexity of the algorithm $SF1$ is exponential in the number of agents. It is good for computing the deployments where the number of nodes is large, but there are only few agents.

### SF1: a node-based algorithm

In order to compute the survivability of a given deployment we need to consider all possible future networks. A possible future network consists of a subset of nodes of the original network. We are interested in future networks in which at least one copy of each agent is deployed somewhere. Formally, given a deployment $\mu$, suppose $N = \{N_i | N_i \subseteq V \text{ and } \mu \text{ is a valid deployment w.r.t. } N_i\}$. We say that $\mu$ is valid w.r.t. $N_i$ if and only if for each agent $a \in M$, $\{n | a \in \mu(n)\} \cap N_i \neq \emptyset$.

The survivability of $\mu$ is the probability that one of the future networks $N_i$ of $N$ will survive. Suppose $N_i = \{n_{i1}, n_{i2}, \ldots n_{il}\}$, $n_{ij} \in V$. The survivability of $N_i$ is given by:

$$surv(N_i) = \prod_{n_p \in N_i} (1 - dp(n_p)) \cdot \prod_{n_q \in V \setminus N_i} dp(n_q) \tag{7.2}$$

For the deployment $\mu$ to survive, all the nodes in at least one of the $N_i$'s must be connected. Since the event of the survivability of $N_i$ is mutually exclusive from the survivability of $N_j$, $i \neq j$, the survivability of $\mu$ is the sum of the survivability of each of the $N_i$'s. The following result shows this.

**Proposition 7.3.3 (Correctness of $SF1$).** Suppose $\mu$ is a deployment w.r.t. an overlay network $\mathcal{N} = \{V, E\}$ and suppose the disconnect probabilities of all nodes are independent. Let $N$ be the set of all feasible networks. Then,

$$SF1(\mu) = surv(N_1 \lor N_2 \lor \ldots \lor N_{|N|}) = \sum_{N_i \in N} surv(N_i).$$

The above proposition tells us that in order to find the survivability of $\mu$ under the assumption of independence, we have to find all subsets of $V$ w.r.t. which $\mu$ is a valid deployment. For each such subset $N_i$, the probability that $N_i$ survives is a straightforward product computation. The probability that $\mu$ survives (under independence) is the sum of the probabilities that the $N_i$'s survive.

This is illustrated below.

**Example 18.** Suppose that a network and a deployment is given in Example 17. The possible future minimal networks are: $N_1 = \{n_1\}$, $N_2 = \{n_2, n_3\}$, $N_3 =$
The survivability of each network is:

\[ \text{surv}(N_1) = (0.3)(0.6)(0.4) = 0.072, \]
\[ \text{surv}(N_2) = (1 - 0.6)(1 - 0.4)(0.7) = 0.168. \]

Similarly we have \( \text{surv}(N_3) = 0.072, \) \( \text{surv}(N_4) = 0.048, \) \( \text{surv}(N_5) = 0.108. \)

The survivability of the deployment is the sum of the probability that \( N_i \) survives:

\[ SF_1(n) = 0.072 + 0.168 + 0.072 + 0.048 + 0.108 = 0.468. \]

**SF1\(_a\): an agent-based algorithm**

\( SF1_a \) is another algorithm to find the survivability of \( \mu \) which is exponential in the number of agents.

Given a deployment \( \mu \), let \( A_i \) be the event that all the nodes that agent \( i \) is deployed on are disconnected. Let \( A_d \) be the event that at least one of \( A_i \) occurs. The probability of the event \( A_i \) is that all the nodes on which \( a_i \) is deployed are disconnected, i.e., \( \text{surv}(A_i) = \prod_{i \in \mu(n_k)} (1 - dp(n_k)). \) In order for the \( \mu \) to survive, none of \( A_i \) should occur. Unfortunately, the \( A_i \)'s events are not mutually exclusive. Thus, in order to compute the survivability of \( \mu \) using \( A_i \) we need to apply the rule of the probability of the disjunction of not mutually exclusive events as presented below.

**Proposition 7.3.4.** Suppose \( \mu \) is a deployment w.r.t. an overlay network \( \mathcal{N} = \{V, E\} \) and suppose the disconnect probabilities of all nodes are independent. Then

\[
SF_1(a)(\mu) = 1 - P(A_d) \\
P(A_d) = P(A_1 \lor A_2 \lor \ldots \lor A_{|M|}) \\
= 1 - \sum_{i \in M} P(A_i) + \sum_{i,j \in M} P(A_i \land A_j) \\
+ \ldots + (-1)^{|M|+1} P(A_1 \land \ldots \land A_{|M|}) \tag{7.3}
\]

\( SF_1_a \) finds all the \( A_i \)'s, computes the probability of each \( A_i \) and then computes the above formula.

\( SF_1_a \) is illustrated as follows.

**Example 19.** Suppose that a network and a deployment is given in Example 17. As agent \( a_1 \) locates at nodes \( n_1 \) and \( n_2 \), the probability of the disconnection of both \( n_1 \) and \( n_2 \) is:

\[ P(A_1) = dp(n_1)dp(n_2) = 0.7 \times 0.6 = 0.42. \]
Similarly, we have:

\[ Pr(A_2) = Pr(A_3) = dp(n_1)dp(n_3) = 0.28; \]
\[ Pr(A_1 \land A_2) = Pr(A_1 \land A_3) = dp(n_1)dp(n_2)dp(n_3) = 0.168; \]
\[ Pr(A_2 \land A_3) = dp(n_1)dp(n_3) = 0.28; \]
\[ Pr(A_1 \land A_2 \land A_3) = dp(n_1)dp(n_2)dp(n_3) = 0.168 \]

Thus, the survivability of the deployment is given by:

\[ SF1_a(\mu) = 1 - (Pr(A_1) + Pr(A_2) + Pr(A_3)) + (Pr(A_1 \land A_2) + Pr(A_1 \land A_3) + Pr(A_2 \land A_3)) - Pr(A_1 \land A_2 \land A_3) = 0.468. \]

Note that the time required to compute both \(SF1_n\) and \(SF1_a\) is exponential, although \(SF1_n\) is exponential in the number of nodes while \(SF1_a\) is exponential in the number of agents. This comes not as a surprise as the problem of finding the survivability of an agent deployment is \(NP\)-hard even when the independence assumption is made.

We can improve efficiency if we use an idea of Kraus et al. [KST03] to reduce the number of agents and nodes without any loss of accuracy. Kraus et al. prove that the survivability of a deployment is unaffected if we eliminate irrelevant agents—an agent \(a\) is irrelevant if there exists any other agent \(a'\) which is deployed at a subset of nodes at which \(a\) is deployed, that is, \(\{n \mid a' \in \mu(n)\} \subseteq \{n' \mid a \in \mu(n')\}\). Throughout this chapter, when computing survivability with any algorithms, we always first apply this method to eliminate the irrelevant agents, and then carry out the computation on the simplified deployments.

The following example gives the reader a quick idea of how the elimination idea works:

**Example 20.** Consider the network \(N\) and a deployment \(\mu\) given in Example 17. We denote the nodes of an agent \(a_i\) locates by \(Loc(a_i)\). For each agent, we have \(Loc(a_1) = \{n_1, n_2\}, Loc(a_2) = \{n_1, n_3\}, and Loc(a_3) = \{n_1, n_3\}\). As \(Loc(a_3) \subseteq Loc(a_2)\), we remove \(a_3\) from the deployment \(\mu\) and update \(\mu\) as:

\[ \mu'(n_1) = \{a_1, a_2\}, \mu'(n_2) = \{a_1\}, and \mu'(n_3) = \{a_2\}. \]

We compute the survivability of the simplified deployment \(\mu'\) by \(SF1_a\):

\[ surv(\mu') = 1 - (Pr(A_1) + Pr(A_2)) + Pr(A_1 \land A_2) = 1 - (0.42 + 0.28) + 0.168 = 0.468. \]

Clearly, \(surv(\mu') = surv(\mu)\).

As we said before, the complexity of \(SF1_n\) and \(SF1_a\) is either exponential in the number of nodes or exponential in the number of agents. Therefore, they are only applicable for the deployments where either the number of nodes is small.
7.4 Complexity Results for Survivability

We have introduced two survivability algorithms $SF_1$ and $SF_2$. We now present the complexity results for computing the survivability following.

Given a network $N = \{V, E\}$, a multiagent application $M$ and a deployment $\mu$. Any node in $V$ can “go down” or somehow get “disconnected” from the network. Thus, any $(N, N \times N)$ where $N \subseteq V$ is a possible network that can arise in the future. The survivability of $N$ (under independence) is given by

$$
surv(N) = \prod_{n_p \in N} (1 - dp(n_p)) \cdot \prod_{n_q \notin V \setminus N} dp(n_q). \tag{7.4}\n$$

Suppose a valid possible future network is defined by $\text{Valid}N(\mu) = \{N_i | N_i \subseteq V\}$ and $\mu$ is valid w.r.t $N_i$. Then the probability of survival of $\mu$ is given by $\Sigma_{N_i \in \text{Valid}N(\mu)} \text{surv}(N_i)$.

Finding the probability of survival of a deployment is at least NP-hard even if we make the independence assumption. Similarly, finding an optimal deployment is at least NP-hard.

**Theorem 7.4.1.** The problem of computing the survival probability of a given deployment under the independence assumption and the problem of finding an optimal deployment are at least NP-hard.

**Proof.** The proof of NP-hardness uses a reduction from set-covering. Given a finite set $X$ and a family $F$ of subsets of $X$, such that every element of $X$ belongs to at least one subset in $F$: $X = \bigcup_{S \in F} S$. A subset $S \in F$ covers its elements. The decision version of the set-covering problem asks whether or not a covering $N \subseteq F$ whose members cover all of $X$ exists with size at most $k$. The set-covering problem is NP-complete [CLRS01].

Figure 7.1 illustrates how the set-covering is transformed into the problem of finding valid networks. The set $X$ corresponds to the multiagent system $M$. Each subset $S \in F$ corresponds to each node in the network $n \in V$. Edges between set $S \in F$ and $x \in X$ means that the element $x \in S$, which corresponds
to the deployment $\mu$ defining the nodes where the agents locate. Clearly, $N$ is an existing covering with size $k$ if and only if $N$ is a valid network with number $k$ of nodes w.r.t $\mu$.

One may be tempted to believe that we can find an approximation algorithm that is guaranteed to terminate in polynomial time and give a deployment whose survival probability is within $\epsilon$ of the survival probability of the optimal deployment ($\epsilon > 0$). Unfortunately, the best such $\epsilon$ is 1 (under the assumption that $P \neq \text{NP}$).

**Theorem 7.4.2.** If $P \neq \text{NP}$, then for each polynomial algorithm to compute a sub-optimal deployment, there are instances in which the optimal deployment’s survival probability is 1, but the algorithm returns a deployment with survival probability 0.

**Proof.** Suppose that the claim above is not correct. Then there exists a polynomial algorithm $AL$ that always returns a deployment with survival probability larger than 0, when the optimal deployment survival probability is 1. We will use $AL$ in order to solve the NP-complete problem “subset sum” [CLRS01]. Given a set $S = \{s_1, \ldots, s_n\}$ and a sum $S_1$, we will build the following network:

Each member of the set $s \in S$ will be represented by an agent $a_s$, whose memory requirement is $s$, $\text{mem}(a_s) = s$. There are 2 nodes, $n_1$ and $n_2$, with available memory $\text{mem}(n_1) = S_1$ and $\text{mem}(n_2) = \sum_{s \in S} s - S_1$, respectively. The disconnect probability for each node is 0, i.e. $\forall_i \text{dp}(n_i) = 0$. It is easy to see:

1. The survival probability of the optimal deployment is 1, if there exists a subset $S' \subseteq S$ such that $\sum_{s' \in S'} s' = S_1$.

2. If there is no subset $S' \subseteq S$ which its sum is $S_1$, the optimal deployment’s survival probability is 0.

Therefore we can define the following algorithm in order to solve subset sum:
7.5 APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

- By given a set \( S \), build the corresponding network \( N_s \) (as we described above).
- Run algorithm \( AL \) on \( N_s \).
- If \( AL \) returns a deployment that its survival probability is larger than 0, return Yes (There exists a subset \( S' \subseteq S \) which its sum is \( S_1 \)).
- Otherwise return No.

One could expect that the computation would be easier with the assumption of independence, however Theorem 7.4.1 shows this is not the case. Hence, it is tempting to find approximation algorithms. As we have already seen in Theorem 7.4.2, it is unlikely that there is a polynomial algorithm to compute the survivability (unless we can prove that \( P = NP \)). Moreover, computing survival probability of a fixed deployment is hard to approximate within some fixed bound of the optimal solution.

In the following sections, we propose several heuristics to compute the deployments which work well with some experiment environments defined in Section 7.6.

7.5 Approximate Algorithms for Computing Survivability

In this section, we first present an upper bound on \( SF_{1_a} \). We then introduce several approximations to compute the MAS deployments. We are interested in finding lower bounds for \( SF_{1_n} \) and \( SF_{1_a} \)—this is because we want to be sure that when we say deployment \( \mu \) has survival probability exceeding some threshold, this is in fact true.

An upper bound on \( SF_{1_a} \)

As \( SF_{1_a}, SF_{1_n} \) both take exponential time, we now develop a fast algorithm to compute an upper bound on \( SF_{1_a} \). This algorithm is Algorithm 8 below. In this algorithm, the upper bound computed can be used to evaluate heuristics proposed later in the chapter. Event \( A_d \) is the event that all nodes on which some agent is located get disconnected. We are therefore interested in the complement of
event $A_d$. In Equation 7.3, if we find a lower bound for $\text{Prob}(A_d)$ and subtract it from 1, we will get an upper bound on the survivability of $\mu$. It is easy to see that $\sum_{i \in M} \text{Pr}(A_i) - \sum_{i \neq j, i, j \in M} \text{Pr}(A_i \land A_j)$ is a lower bound for $\text{Prob}(A_d)$. Similarly, any even number of terms in the expression of equation 7.3 provides a lower bound. This lower bound can be calculated incrementally until we run out of time or the difference between what we add to the expression (an odd term) and what we subtract from the expression (an even term) is very small. We can then take the maximum among all the lower bounds that we computed. Subtracting this value from 1 will give us an upper bound on the survivability of $\mu$.

**Algorithm 8.** $UB(\mu, V, M, dp)$

(* Input: (1) a MAS deployment $\mu$ *)

(* (2) a set of nodes $V$ *)

(* (3) a set of agents $M$ *)

(* (4) a disconnect probability function $dp$ *)

(* (5) the predefined ratio $\text{RATIO}$ *)

(* Output: an upper bound on the survivability of $\mu$ *)

1. start timer $time$, $k = 1$; (* $k$ specifies the number of elements in the subsets *)

2. $value = 0$, $value_o = 1$;

3. $p = 0$, $minub = 1$, $maxlb = 0$;

4. while ($(|value - value_o| \leq \text{RATIO})$ and ($time < \text{MAXTIME}$))

   (a) $value_o = value$;

   (b) $value = k\text{subset}(\mu, k)$; (* returns the sum of the probability of $k$-subsets *)

   (c) if $k$ is odd, then $sign = 1$;

   else $sign = -1$;

   (d) $p = p + sign \times value$;

   (e) if ($sign = 1$)

       if $p < minub$, then $minub = p$;

   else

       if $p > maxlb$, then $maxlb = p$;

   (f) $k = k + 1$;

5. return $(1 - maxlb)$. 
The following theorem says that the upper bound computed by Algorithm 8 is correct.

**Theorem 7.5.1.** Given a deployment $\mu$ and a network $G$, the proposed upper bound algorithm gives an upper bound on $SF_{1a}$.

**Proof.** We first prove it is true when there are two terms in Equation 7.3. We need to show that

\[ P(A_1 \lor \ldots \lor A_n) \geq \sum_{i \in n} P(A_i) - \sum_{(i \neq j) \land (i,j \in n)} P(A_i \land A_j). \]

The statement holds for $n = 1$, where $P(A_1) = P(A_1)$;

For $n = 2$, $P(A_1 \lor A_2) = P(A_1) + P(A_2) - P(A_1 \land A_2)$.

Assume for $n = N$,

\[ P(A_1 \lor \ldots \lor A_N) \geq P(A_1) + \ldots + P(A_N) - P(A_1 \land A_2) - \ldots - P(A_{N-1} \land A_N) \]

holds.

We now show that for $n = N + 1$,

\[ P(A_1 \lor \ldots \lor A_{N+1}) \geq P(A_1) + \ldots + P(A_{N+1}) - P(A_1 \land A_2) - \ldots - P(A_{N} \land A_{N+1}). \]

We show it in the following steps:

\[
\begin{align*}
P(A_1 \lor \ldots \lor A_{N+1}) & = P(A_1 \lor \ldots \lor A_N) + P(A_{N+1}) - P((A_1 \lor \ldots \lor A_N) \land A_{N+1}) \\
& \geq (P(A_1) + \ldots + P(A_N) - P(A_1 \land A_2) - \ldots - P(A_{N-1} \land A_N)) + P(A_{N+1}) \\
& \quad - P((A_1 \land A_{N+1}) \lor \ldots \lor (A_N \land A_{N+1})) \\
& \geq (P(A_1) + \ldots + P(A_{N+1}) - P(A_1 \land A_2) - \ldots - P(A_{N-1} \land A_N)) - (P(A_1 \land A_{N+1}) \\
& \quad + \ldots + P(A_N \land A_{N+1})) \\
& = P(A_1) + \ldots + P(A_{N+1}) - P(A_1 \land A_2) - \ldots - P(A_N \land A_{N+1}).
\end{align*}
\]

The proof can be extended to more than two terms. \(\square\)

We now introduce several approximations which underestimate the survivability of the given deployment:

- $SF2$ is an anytime algorithm;
- $SF3$ is a tree-based approach;
- $SF4_1$ is a cover based algorithm;
- $SF4_2$ is a disjoint based algorithm;
- $SF4_g$ is a group algorithm; and
- $SF5$ is a split algorithm.
SF2: an anytime algorithm

$SF1_a$ is exponential in the number of agents, thus it is not feasible to compute deployments with many agents. We show in this section that it can be turned into an anytime algorithm.

Using the same idea that we used for computing the upper bound, we can also compute a lower bound on the survivability of $\mu$. Again, looking at the complementary event $A_d$, in Equation 7.3, if we compute an upper bound of $\text{Prob}(A_d)$ and subtract it from 1, we get a lower bound on the values $SF1_a, SF1_n$ return. Any odd number of terms of equation 7.3 provides an upper bound. An anytime algorithm can iteratively add terms until we run out of time or the ratio between the maximum among the lower bounds and the minimum among the upper bounds is smaller than a specified ratio. We describe the anytime algorithm as below.

Algorithm 9. anytime($\mu, V, M, dp, \text{RATIO}$)

(* Input: (1) a MAS deployment $\mu$ *)
(* (2) a set of nodes $V$ *)
(* (3) a set of agents $M$ *)
(* (4) a disconnect probability function $dp$ *)
(* (5) the predefined ratio $\text{RATIO}$ *)
(* Output: an underestimate of the survivability of $\mu$ *)

1. start timer $t$, $k = 1$; (* $k$ specifies the number of elements in the subsets *)
2. $p = 0$, $\text{minub} = 1$, $\text{maxlb} = 0$;
3. while ($\frac{\text{maxlb}}{\text{minub}} \geq \text{RATIO}$) and ($t < \text{MAXTIME}$)
   (a) value = $k$\text{subset}($\mu$, $k$); (* returns the sum of the probability of $k$-subsets *)
   (b) if $k$ is odd, then sign = 1;
       else sign = -1;
   (c) $p = p + \text{sign} \times \text{value}$;
   (d) if (sign = 1)
       if $p < \text{minub}$, then $\text{minub} = p$;
   (e) else
       if $p > \text{maxlb}$, then $\text{maxlb} = p$;
   (f) $k = k + 1$
Therefore, we need to show that:

\[ \text{The statement holds for } n \]

**Proof.** We first show that it is true when there are three terms in Equation 7.3. Therefore, we need to show that:

\[ P(A_1 \lor \ldots \lor A_n) \leq \sum_{i \in n} P(A_i) - \sum_{(i \neq j) \in (i, j \in n)} P(A_i \land A_j) \]

\[ + \sum_{(i \neq j \neq k) \in (i, j, k \in n)} P(A_i \land A_j \land A_k). \]

The statement holds for \( n = 2 \), where

\[ P(A_1 \lor A_2) = P(A_1) + P(A_2) - P(A_1 \land A_2). \]

When \( n = 3 \), it holds because:

\[ P(A_1 \lor A_2 \lor A_3) = P(A_1) + P(A_2) + P(A_3) - P(A_1 \land A_2) - P(A_1 \land A_3) - P(A_2 \land A_3) + P(A_1 \land A_2 \land A_3). \]

Assume that it holds for \( n = N \), thus we have:

\[ P(A_1 \lor \ldots \lor A_N) \leq P(A_1) + \ldots + P(A_N) - P(A_1 \land A_2) - \ldots - P(A_{N-1} \land A_N) \]

\[ + P(A_1 \land A_2 \land A_3) + \ldots + P(A_{N-2} \land A_{N-1} \land A_N). \]

We now show that for \( n = N + 1 \),

\[ P(A_1 \lor \ldots \lor A_{N+1}) \leq P(A_1) + \ldots + P(A_{N+1}) - P(A_1 \land A_2) - \ldots - P(A_N \land A_{N+1}) \]

\[ + P(A_1 \land A_2 \land A_3) + \ldots + P(A_{N-1} \land A_N \land A_{N+1}) \]

We show the above \( \text{LHS} \leq \text{RHS} \) in the following steps:

\[ \text{LHS} = P(A_1 \lor \ldots \lor A_{N+1}) \]

\[ = P(A_1 \lor \ldots \lor A_N) + P(A_{N+1}) - P((A_1 \lor \ldots \lor A_N) \land A_{N+1}) \]

\[ \leq (P(A_1) + \ldots + P(A_N) - P(A_1 \land A_2) - \ldots - P(A_{N-1} \land A_N) + P(A_1 \land A_2 \land A_3) + \]

\[ \ldots + P(A_{N-2} \land A_{N-1} \land A_N)) + P(A_{N+1}) - P((A_1 \lor \ldots \lor A_N) \land A_{N+1}) \]

We remove the same terms in LHS and RHS. Therefore, to show LHS ≤ RHS, we need to show:

\[ -(P(A_1 \land A_2) + \ldots + P(A_{N-1} \land A_N) + P((A_1 \land A_{N+1}) \lor (A_2 \land A_{N+1}) \ldots \]

\[ \lor (A_N \land A_{N+1}))) + P(A_1 \land A_2 \land A_3) + \ldots + P(A_{N-2} \land A_{N-1} \land A_N)) \]

\[ \leq -(P(A_1 \land A_2) + \ldots + P(A_N \land A_{N+1})) + P(A_1 \land A_2 \land A_3) + \ldots \]

\[ + P(A_{N-1} \land A_N \land A_{N+1}). \]

Removing the same terms, we need to show:

\[ P(A_1 \land A_2) + \ldots + P(A_{N-1} \land A_N) + P((A_1 \land A_{N+1}) \lor (A_2 \land A_{N+1}) \ldots \lor (A_N \land A_{N+1})) \]

\[ \geq P(A_1 \land A_2) + \ldots + P(A_N \land A_{N+1}) - (P(A_1 \land A_2 \land A_{N+1}) + \]

\[ + P(A_{N-1} \land A_N \land A_{N+1})). \]

After we remove the same terms from both of sides, we need to show that:
7.5. APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

\[ P((A_1 \land A_{N+1}) \lor (A_2 \land A_{N+1}) \lor \ldots \lor (A_N \land A_{N+1})) \geq P(A_1 \land A_{N+1}) + P(A_2 \land A_{N+1}) + \ldots + P(A_N \land A_{N+1}) - (P(A_1 \land A_2 \land A_{N+1}) + \ldots + P(A_{N-1} \land A_N \land A_{N+1})). \]

Now, we let \( B_i = A_i \land A_{N+1} \), the above inequality becomes:

\[ P(B_1 \lor B_2 \lor \ldots \lor B_N) \geq P(B_1) + P(B_2) + \ldots + P(B_N) - (P(B_1 \land B_2) + \ldots + P(B_{N-1} \land B_N)). \]

We already proved it holds in Proof 7.5.1. Thus, we have proved \( \text{LHS} \leq \text{RHS} \) for the case \( n = N + 1 \).

Similarly, the proof can be extended to any odd terms (5, 7, \ldots) in Equation 7.3.

\[ \text{SF3: a tree-based approximation} \]

Algorithm SF3 is an approximation which provides a lower bound on \( SF_{1n} \).

The \( SF_{1n} \) algorithm uses the property shown in Theorem 7.3.3 that

\[ SF_{1n}(\mu) = \text{surv}(N_1 \lor N_2 \lor \ldots \lor N_m) = \sum_{N_i \in N} \text{surv}(N_i). \]

It enumerates all subsets \( N_i \subseteq V \) w.r.t. which the deployment \( \mu \) is a valid deployment, finds the survivability of \( \mu \) w.r.t. of each of subsets \( N_i \), and adds up the results. The complexity of \( SF_{1n} \) is caused by the fact that there can be exponentially many subsets.

Instead of enumerating all subsets for which \( \mu \) is a valid deployment of \( M \), SF3 attempts to find \( \text{surv}(N_i) \) only for a bounded number of \( N_i \)'s. By choosing only a subset of all possible \( N_i \)'s, we know immediately that \( SF3(\mu) \leq SF_{1n}(\mu) \). That is, SF3 gives us an approximation from below of \( SF_{1n} \). Obviously, in order to make the value returned by \( SF3(\mu) \) as close as possible to \( SF_{1n}(\mu) \), when we choose which subsets to be included, we would like to pick the subsets \( N_i \) such that \( \text{surv}(N_i) \) is as large as possible.

\( SF3 \) does this via a tree search in which the root of every node is labeled with a subset of \( V \). The root is labeled with \( V \). The probability of \( V \) is computed. For every node \( n \in V \) there is a vertex labeled \( V - \{n\} \) in the second level of the tree. For each label of such a vertex the algorithm checks if the resulting \( \mu \) is valid. For all such vertices, the probability of their labeled possible future networks is computed. Only the \( \alpha \) vertices with the highest probability are further expanded in the same way. If a vertex labeled \( N_i \) is expanded, its children will be labeled
by $N_i \setminus \{n\}$ for each node $n \in N_i$. Again only $\alpha$ vertices will be expanded and so on. We stop when there are no more nodes to expand. $SF3$ sums the probability of all the future networks in the search tree that $\mu$ is valid with respect to them.

If $\alpha$ is polynomial in $V$, $SF3$ considers only a polynomial number of future networks. Therefore it may return very poor results if there is a large number of nodes. $SF3$ is bounded by the number of developed subsets multiplied by the largest subset probability. The largest subset probability is bounded by $\prod_{i=1}^{N}(1 - dp(n_i))$. Assume the disconnect probability of nodes is distributed normally in $[0, 0.5]$. The survivability given by $SF3$ is no greater than $\alpha 0.9^N$, which is smaller than $10^{-N/2}$. Since the performance of $SF3$ could be very poor, we propose two heuristics to improve its value.

1. The first heuristics is that prior to running $SF3$, we first remove a set of agents $A'$ whose locations (i.e. the nodes they are located) are disjoint with the locations of any other agents. We can compute the survivability of $A'$, denoted by $surv(A')$, directly. We then apply $SF3$ on the remaining agents $A \setminus A'$. At the end of the algorithm, we multiply the returned value by $surv(A')$, i.e. $surv(\mu) = surv(A') \cdot SF3(A \setminus A')$.

2. The second heuristics is based on the idea that if the number of nodes involved in $SF3$ is large (e.g 20), we want to reduce the number of nodes by removing some nodes which contribute less to the survival of the $\mu$. We sort the nodes in the ascending order according to the values of $\sqrt{dp(n)}$, where $\alpha'$ denotes the number of agents at node $n$. The first $K$ number of nodes can be deleted from the deployment. In this way, we discard nodes whose $dp$ is very low or who has very few agents on it. Note that after removal action, it may be possible to get rid of more irrelevant agents by the idea in [KST03] in order to further simplify computation.

The algorithm 10 uses the 20-number as a bound.

**Algorithm 10.** $SF3(\mu, V, M, dp, \alpha)$

(* Input: (1) a MAS deployment $\mu$ *)

(* (2) a set of nodes $V$ *)

(* (3) a set of agents $M$ *)

(* (4) a disconnect probability function $dp$ *)

(* (5) the predefined number of selected vertices $\alpha$ *)

(* Output: an underestimate of the survivability of $\mu$ *)
7.5. APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

1. disjoint\textsubscript{surv} = rmvdisjoint(\mu, G, \mathcal{M}, dp);
   (* remove the agents with disjoint locations, return the survivability of the removed agent set *)

2. if |V| > 20, then
   rmvnodes(\mu, V, \mathcal{M}, dp); (* remove some nodes according to the criteria *)

3. best\textsubscript{val} = calc\textsubscript{surv}(V); (* compute the survivability of the future network V *)

4. temp = \{V\}, done = 0, flag = 0;

5. while (¬done), do
   (a) \(X' = \emptyset\);
   (b) while (temp \neq \emptyset )
      i. \(X = headof(temp), temp = temp \setminus X\);
      ii. \(X' = X' \cup \{X \setminus \{x_i\} \mid x_i \in X\} \);
   (c) \(N = \emptyset\);
   (d) while (\(X' \neq \emptyset\)), do (* remove invalid sets and repetitive sets in \(X'\) *)
      i. \(X'_\text{sub} = headof(X')\);
      ii. if \(X'_\text{sub} \notin N \) and \(\bigcup_{n \in X'_\text{sub}} \mu(n) = S\), then
          \(N = N \cup X'_\text{sub}, flag = 1\);
   (e) if (¬flag), then done = 1;
   (f) else, do
      i. for (\(i = 0, i < |N|, i + +\))
         \(best\textsubscript{val} = best\textsubscript{val} + surv(N_i)\)
      ii. \(N = sort(N, surv(N))\);
          (* sort sets in \(N\) in descending order according to survivabilities *)
      iii. for (\(j = 0, j < \alpha, j + +\)) (* keep the first \(\alpha\) of sets *)
          \(\bullet temp = temp \cup headof(N)\);
          \(\bullet N = N \setminus headof(N)\);
      iv. \(flag = 0\);

6. return (best\textsubscript{val} \times disjoint\textsubscript{surv}).

The following proposition expresses that \(SF3\) is a correct polynomial time approximation of \(SF1_n\).
Proposition 7.5.3. SF3(\(\alpha, \mu, G, \mathcal{M}, dp\)) is an underestimation of SF1\(_n\). Suppose \(\alpha\) is fixed. The time complexity to compute SF3 is \(O(\alpha |V|^2 \log(\alpha |V|) + \alpha |V|^2 \mathcal{M})\), that is, the computation is polynomial if \(\alpha\) is a fixed constant.

Proof. Suppose the search starts from the set of nodes \(V\). The next level in the tree contains the number \(V\) of sets which are generated by removing exact one node from their parent set in the previous level. Only a fixed number \(\alpha\) of valid sets with highest survivabilities is used to generate the subtrees, thus the total number of generated subsets in the next level would be \(\alpha|V - 1|\). For each generated set of nodes, computing the probability of a set can be done by \(O(1)\), and checking if the set is valid (if it contains all agents) can be done by \(O(\mathcal{M})\), where \(\mathcal{M}\) is the number of agents. In addition, for each level, sorting the sets of the nodes takes \(O(\alpha|V - 1|\log(\alpha|V - 1|))\). As the maximum depth generated in the tree is \(|V|\), the time SF3 takes is: \(|V|(|\alpha|V - 1|\mathcal{M}| + \alpha|V - 1|\log(\alpha|V - 1|))\). Therefore, the complexity of the algorithm SF3 is:

\[
O(\alpha |V|^2 \log(\alpha |V|) + \alpha |V|^2 \mathcal{M}).
\]

The following example illustrates the working of SF3.

Example 21. Consider the network and the updated deployment of Example 20, where \(\mu(n_1) = \{a_1, a_2\}, \mu(n_2) = \{a_1\}, \text{ and } \mu(n_3) = \{a_2\}\). Suppose \(\alpha = 1\). The root is \(V = \{n_1, n_2, n_3\}\). Thus \(X_{s0} = \{n_1, n_2, n_3\}\), and

\[
\text{surv}(X_{s0}) = (1 - dp(n_1))(1 - dp(n_2))(1 - dp(n_3))
\]

\[
= (1 - 0.7)(1 - 0.6)(1 - 0.4) = 0.072.
\]

In the next level of the graph, 3 subsets are generated by removing one node from \(V\):

\[
X_{s11} = \{n_2, n_3\}, \text{ and } \text{surv}(X_{s11}) = (1 - 0.6)(1 - 0.4)(0.7) = 0.168;
\]

\[
X_{s12} = \{n_1, n_3\}, \text{ and } \text{surv}(X_{s12}) = (1 - 0.7)(1 - 0.4)(0.6) = 0.108;
\]

\[
X_{s13} = \{n_1, n_2\}, \text{ and } \text{surv}(X_{s13}) = (1 - 0.7)(1 - 0.6)(0.4) = 0.048.
\]

As \(\alpha = 1\), we use the set \(X_{s11}\) to generate subsets in the next level.

\[
X_{s21} = \{n_2\} \text{ is removed because it is not valid;}
\]

\[
X_{s22} = \{n_3\} \text{ is invalid thus removed.}
\]

The search terminates because no more valid subsets can be created. The survivability of the deployment is the sum of the survivabilities of all valid sets found so far:

\[
\text{SF3}(\mu) = \text{surv}(X_{s0}) + \text{surv}(X_{s11}) + \text{surv}(X_{s12}) + \text{surv}(X_{s13}) = 0.396.
\]
7.5. APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

**SF4₁: a set cover based algorithm**

The SF4₁ algorithm is a *set cover approach*. A cover is a set of nodes where each agent \( a \in \mathcal{M} \) is deployed at least once. The survivability of each set cover gives a lower bound on the actual survivability of the \( \mu \). We use a greedy set cover algorithm to find a cover. We iteratively pick the node with highest priority and remove the covered agents, until all agents are covered. Algorithm 11 shows how the greedy algorithm works.

**Algorithm 11.** \( FSC(\mu, V', \mathcal{M}, dp, PC, VC) \)

\begin{algorithmic}
  \State * Input: (1) a MAS deployment \( \mu \)
  \State (2) a set of nodes \( V' \)
  \State (3) a set of agents \( A \)
  \State (4) a disconnect probability function \( dp \)
  \State (5) the survivability of the cover set \( PC \)
  \State (6) a cover set \( VC \)

  \begin{enumerate}
  \item \( agents = A, nodes = V', VC = \emptyset \);
  \item while \((agents \neq \emptyset) \land (nodes \neq \emptyset)\) do
    \begin{enumerate}
    \item for each \( n \in nodes \), do
      \begin{enumerate}
      \item \( priority(n) = (1 - dp(n))^\frac{1}{k} \), where \( k \) is the number of agents on \( n \);
      \item compute the priority of each node *
      \end{enumerate}
    \item \( n' = \arg\max\{priority(n) \mid n \in nodes\}; \)
    \item \( agents = agents \setminus \mu(n'), nodes = nodes \setminus \{n'\}; \)
    \item \( VC = VC \cup \{n'\}; \)
    \end{enumerate}
  \item \( PC = \prod_{n \in VC}(1 - dp(n)); \)
  \item return \( PC \) and \( VC \).
  \end{enumerate}

**Algorithm 11** returns a set cover \( VC \) and its survival probability \( PC \).

The \( SF4₁ \) algorithm works by repeatedly finding the disjoint set cover based on the algorithm 11. Thus the survivability of the deployment \( \mu \) is the probability that at least one disjoint set cover will survive.
Algorithm 12. $SF_4^1(\mu, V, M, dp)$
(* Input: (1) a MAS deployment $\mu$ *)
(* (2) a set of nodes $V$ *)
(* (3) a set of agents $M$ *)
(* (4) a disconnect probability function $dp$ *)
(* Output: an underestimate of the survivability of $\mu$ *)

1. $V' = V$, $\text{flag} = 1$;
2. $\text{DPC} = 1$; (* the probability that all disjoint set cover will be disconnected. *)
3. while($\text{flag}$)
   (a) $A' = M$, $PC = 0$, $VC = \emptyset$;
   (b) $\text{FSC}(\mu, V', A', dp, PC, VC)$; (* find and compute a set cover. *)
   (c) if ($VC \neq \emptyset$), then
      i. $V' = V' \setminus VC$;
      ii. $\text{DPC} = \text{DPC}(1 - PC)$;
   else $\text{flag} = 0$;
4. $P = 1 - \text{DPC}$;
5. return $P$.

Proposition 7.5.4. The survivability function $SF_4^1$ is a polynomial-time approximation algorithm for $SF_1$. The computational complexity of $SF_4^1$ is $O(|M| \cdot |V|^2)$.

Proof. The number of iterations of the loop on line 3 in Algorithm 12 is bounded by $|V|$, where $|V|$ is the number of nodes. The algorithm $\text{FSC} \ 11$ runs in time $O(|V| \cdot |M|)$, where $|M|$ is the number of agents. Thus, the algorithm $SF_4^1$ runs in time $O(|M| \cdot |V|^2)$.

The following example illustrates the working of $SF_4^1$.

**Example 22.** Consider the deployment $\mu$ of Example 17. The deployment is updated as shown in Example 20 after removing all irrelevant agents. We have $M = \{a_1, a_2\}$ and $V = \{n_1, n_2, n_3\}$. We start to find the first set cover. $VC_1 = \emptyset$ and agents $= M$, for each node in $V$, we have:

- $\text{priority}(n_1) = (1 - 0.7)^{1/2} = 0.5477$,
- $\text{priority}(n_2) = (1 - 0.4)^{1/1} = 0.6$, 

...
7.5. APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

\[ \text{priority}(n_3) = (1 - 0.6)^{1/1} = 0.4; \]

As \( n_2 \) has the highest priority, we add \( n_2 \) to the set cover \( VC_1 = \{n_2\} \). As \( \mu(n_2) = \{a_1\} \), agent \( a_1 \) is removed from the agent set: agents = \( \{a_2\} \). The new priority of nodes are computed as:

\[ \text{priority}(n_1) = (1 - 0.7)^{1/1} = 0.3, \text{priority}(n_3) = (1 - 0.6)^{1/1} = 0.4. \]

We add \( n_3 \) to the cover set: \( VC_1 = \{n_2, n_3\} \), and agents = \( \emptyset \). The set of nodes is updated by removing the nodes in \( VC_1 \):

\[ V' = V' \setminus VC_1 = \{n_1\}. \]

We compute the survivability of the first cover set \( VC_1 \) as follows:

\[ PC_1 = (1 - dp(n_2))(1 - dp(n_3)) = (1 - 0.4)(1 - 0.6) = 0.24. \]

We now start to find the second set cover. Since \( n_1 \) includes all the agents, obviously it is another cover set, that is, \( VC_2 = \{n_1\} \). The survivability of the second cover set is:

\[ PC_2 = 1 - dp(n_1) = 0.3. \]

\( V' \) is updated by removing the nodes in \( VC_2 \): \( V' = \emptyset \). Thus, the survivability of the deployment \( \mu \) is computed by:

\[ SF_{41} = 1 - (1 - PC_1)(1 - PC_2) = 1 - (0.76)(0.7) = 0.468. \]

**SF42: a disjoint based algorithm**

For each agent in the multiagent application \( a_i \in \mathcal{M} \), let \( N^i = \{n^i_1, \ldots, n^i_k\} \) be the set of nodes where \( a_i \) is located. Let \( E^i_j \) be the event that the node \( n^i_j \) will survive, then the event that at least one copy of \( a_i \) will keep functioning is denoted by \( E^i = E^i_1 \lor \ldots \lor E^i_k \). The probability of the event \( E^i \) can be computed by

\[ P(E^i) = 1 - dp(n^i_1)dp(n^i_2)\ldots dp(n^i_k) \quad (7.5) \]

We can now define the event that a MAS deployment \( \mu \) will survive by:

\[ E(\mu) = (E^1_1 \lor \ldots \lor E^1_k) \land \ldots \land (E_{|\mathcal{M}|}^1 \lor \ldots \lor E_{|\mathcal{M}|}^j) \]

The probability of the event \( E(\mu) \) represents the survivability of the deployment \( \mu \). Unfortunately, the \( E_i \)s are not mutually exclusive. However, **SF42**
assumes that the events $E^1, E^2, \ldots, E^{\lvert M \rvert}$ are pairwise disjoint. We have
\[
SF_4(\mu) = P(E^1)P(E^2) \cdots P(E^{\lvert M \rvert}) \\
= (1 - dp(n_1) \cdots dp(n_k)) \times \cdots \\
\times (1 - dp(n_1^{\lvert M \rvert}) \cdots dp(n_l^{\lvert M \rvert})).
\]

We show in the following theorem that $SF_4(\mu)$ gives an underestimation of the survivability of $\mu$.

**Theorem 7.5.5.** $SF_4(\mu, G, \mathcal{M}, dp)$ underestimates the actual survivability of $\mu$. 

\[
P(E(\mu)) \geq SF_4(\mu, G, \mathcal{M}, dp) \\
= P(E^1)P(E^2) \cdots P(E^m)
\]

**Proof.** If there is only one node $n_1$, the statement clearly holds:
\[
P(E^1 \wedge \ldots \wedge E^m) = 1 - dp(n_1) \geq (1 - dp(n_1))^m = P(E^1) \cdots P(E^m).
\]
Let $n = 2$, for the left side of Equation 7.7, we have
\[
(1 - dp(n_1))(1 - dp(n_2)) \leq LHS \leq (1 - dp(n_1)dp(n_2)).
\]
For the right side of Equation 7.7,
\[
(1 - dp(n_1))^m(1 - dp(n_2))^m \leq RHS \leq (1 - dp(n_1)dp(n_2))^m.
\]
where $m' > 0$, $m'' > 0$ and $m' + m'' = m$.

As the best settings and the worst settings for values in LHS is same as those in RHS, we have:
\[
P(E^1 \wedge \ldots \wedge E^m) \geq P(E^1) \cdots P(E^m).
\]
Now we want to prove that the inequality holds for the deployment with $N$ nodes if it holds for any configuration with less than $N$ nodes.

We take a configuration with $N$ nodes and $m$ agents, and choose one node $n_c$. Without loss of generality we assume that the agents locating on the node $n_c$ are the first $r$ agents.

Let $p_c$ be the probability that $n_c$ will survive, then for $i \leq r$,
\[
P(E^i) = P(E^i \wedge \{n_c \text{ is alive}\}) + P(E^i \wedge \{n_c \text{ is dead}\}) \\
= p_c + P(\{a_i \text{ survives without } n_c\}) \times (1 - p_c)
\]
and for $i > r$, we have
\[
P(E^i) = P(\{a_i \text{ survives without } n_c\}).
\]
Let denote the event of $a_i$ surviving without $n_c$ by $B^i$, then 
\[
LHS = P(E^1 \wedge \ldots \wedge E^m)
\]
\[ P(\mathcal{E}^1 \land \ldots \land \mathcal{E}^m \land \{n_i \text{ is alive}\}) + P(\mathcal{E}^1 \land \ldots \land \mathcal{E}^m \land \{n_i \text{ is dead}\}) \]
\[ = P(\mathcal{E}^{r+1} \land \ldots \land \mathcal{E}^m \land \{n_i \text{ is alive}\}) + P(\mathcal{E}^1 \land \ldots \land \mathcal{E}^m \land \{n_i \text{ is dead}\}) \]
\[ = P(\mathcal{B}^{r+1} \land \ldots \land \mathcal{B}^m)p_c + P(\mathcal{B}^1 \land \ldots \land \mathcal{E}^m)(1 - p_c) \]
\[ \geq P(\mathcal{B}^{r+1}) \cdots P(\mathcal{B}^m)(p_c + P(\mathcal{B}^1) \cdots P(\mathcal{B}^r)(1 - p_c)) \]

The right side of \( \geq \) above can be achieved by induction hypothesis, since \( \mathcal{B}^i \) are using only the first \( N-1 \) node.

For the RHS, we have
\[ \text{RHS} = (p_c + P(\mathcal{B}^1)(1 - p_c)) \times \ldots \times (p_c + P(\mathcal{B}^r)(1 - p_c)) \times P(\mathcal{B}^{r+1}) \times \ldots \times P(\mathcal{B}^m). \]

In order to show \( \text{LHS} \geq \text{RHS} \), we need to check that if \( P(\mathcal{E}^i) \) are numbers in 0 and 1, then
\[ p_c + P(\mathcal{B}^1) \times \ldots \times P(\mathcal{B}^r) \times (1 - p_c) \]
\[ \geq (p_c + P(\mathcal{B}^1)(1 - p_c)) \times \ldots \times (p_c + P(\mathcal{B}^r)(1 - p_c)). \]

Let us call it the \( \Delta \) inequality. We consider the case \( r = 2 \) first:
\[ (p_c + P(\mathcal{B}^1)P(\mathcal{B}^2)(1 - p_c)) \geq (p_c + P(\mathcal{B}^1)(1 - p_c)) \times (p_c + P(\mathcal{B}^2)(1 - p_c)) \]

Since \( \text{LHS} \) and \( \text{RHS} \) are linear in \( P(\mathcal{B}^i) \), it suffices to check the inequality in the endpoints of \([0; 1]\). The inequality holds when we substitute \( P(\mathcal{B}^1) = 1 \) or \( P(\mathcal{B}^1) = 0 \), therefore it also holds in between.

Now we have:
\[ (p_c + P(\mathcal{B}^1)(1 - p_c))(p_c + P(\mathcal{B}^2)(1 - p_c)) \cdots (p_c + P(\mathcal{B}^r)(1 - p_c)) \]
\[ \leq (p_c + P(\mathcal{B}^1)P(\mathcal{B}^2)(1 - p_c)) \cdots (p_c + P(\mathcal{B}^r)(1 - p_c)) \]
\[ \leq \ldots \leq p_c + P(\mathcal{B}^1)P(\mathcal{B}^2) \cdots P(\mathcal{B}^r)(1 - p_c) \]

Thus the \( \Delta \) inequality holds, and we get what we set out to prove. \( \square \)

An example of the working of \( SF4_2 \) is shown below.

**Example 23.** Consider the updated deployment \( \mu' \) of example 20. Agent \( a_1 \) is located on nodes \( N = \{n_1, n_2\} \). Agent \( a_2 \) is located on nodes \( N' = \{n_1, n_3\} \). Thus, we have:
\[ P(\mathcal{E}^1) = 1 - dp(n_1)dp(n_2) = 1 - 0.42 = 0.58; \]
\[ P(\mathcal{E}^2) = 1 - dp(n_1)dp(n_3) = 1 - 0.28 = 0.72. \]

So the survivability of \( \mu \) is computed by:
\[ SF4_2(\mu) = P(\mathcal{E}^1)P(\mathcal{E}^2) = 0.4176. \]

Algorithms \( SF4_1 \) and \( SF4_2 \) can be combined together to compute the survivability of a given deployment \( \mu \). The combination algorithm \( SF4 \) works by
computing $SF_{41}$ and $SF_{42}$, and then returning the maximal value between them, 
$SF_{4} = \max(SF_{41}, SF_{42})$.

**SF$_{4g}$: a group algorithm**

$SF_{4}$ computes each agent’s survival probability and then returns the product of these survival probabilities. If no node contains more than one agent, then $SF_{4}$ returns the exact answer. However, in general, when the number of agents is large and there is a large number of nodes in which many agents are located, $SF_{4}$ can return a very low probability. To improve this, if there are agents in a deployment that coexist in various nodes, we would consider these agents as a group and compute the group’s survivability. We divide all agents into several such groups, and then take the product of the survival probabilities of all groups as the survivability of the deployment. An intuitive way to group agents is to consider the agent $a$ who has the lowest survivability. We group $a$ with other agents who have the most common nodes with it. When we compute the survivability of each agent group, we use the algorithm $SF_{1a}$. As $SF_{1a}$ takes exponential time in the number of agents, we limit the size of each group. The following algorithm explains this grouping idea.

**Algorithm 13. $SF_{4g}(\mu, V, \mathcal{M}, dp)$**

(* Input: (1) a MAS deployment $\mu$ *)
(* (2) a set of nodes $V$ *)
(* (3) a set of agents $\mathcal{M}$ *)
(* (4) a disconnect probability function $dp$ *)

(* Output: an underestimate of the survivability of $\mu$ *)

1. $s = 4$, $agents = \mathcal{M}$, $surv = 1$; (* we set the number of agents in one group as 4 *)

2. for each agent $a \in agents$, do

   $surv(a) = 1 - \prod_{n \in Loc(a)} dp(n)$;

3. while (agents! = NULL), do

   • choose an agent $a = \arg \min_{surv(a)agents}$;
     (* choose the agent with the lowest survivability *)

   • $A' = \text{group}(a, s, \mu, V, agents)$;
     (* group at most $s - 1$ agents who has the most common locations with $a$ into one group $A'$ *)
7.5. APPROXIMATE ALGORITHMS FOR COMPUTING SURVIVABILITY

- value = $SF1_a(\mu, A', dp)$; (* use $SF1_a$ to compute the survivability of $A'$ *)
- $surv = surv \times value$;
- $agents = agents \setminus A'$.

4. return $surv$.

We use the following example to demonstrate the working of algorithm $SF4_g$.

**Example 24.** Consider a fully connected overlay network with six nodes $V = \{n_1, n_2, n_3, n_4, n_5, n_6\}$ and a multiagent application $\mathcal{M} = \{a_1, a_2, a_3, a_4\}$. Suppose the disconnect probability function $dp$ is given by:

$$dp(n_1) = 0.7, \ dp(n_2) = 0.6, \ dp(n_3) = 0.4, \ dp(n_4) = 0.3,$$
$$dp(n_5) = 0.2, \ dp(n_6) = 0.1.$$  

Consider the current deployment $\mu$:

$$\mu(n_1) = \{a_1, a_2\}, \ \mu(n_2) = \{a_1\}, \ \mu(n_3) = \{a_2\}, \ \mu(n_4) = \{a_3\},$$
$$\mu(n_5) = \{a_3, a_4\}, \ \mu(n_6) = \{a_4\}.$$  

We set the number of agents in one group by 2. According to the algorithm $SF4_g$, we first compute the survivability of each agent based on the nodes on which the agent locates:

$$surv(a_1) = 1 - dp(n_1)dp(n_2) = 0.58, \ surv(a_2) = 1 - dp(n_1)dp(n_3) = 0.72,$$
$$surv(a_3) = 1 - dp(n_4)dp(n_5) = 0.94, \ surv(a_1) = 1 - dp(n_5)dp(n_6) = 0.98.$$  

Since agent $a_1$ is most likely to fail, we select $a_1$ to form the first group $g_1$. We then group agent $a_2$ and $a_1$ together since $a_2$ has the most number of common nodes to agent $a_1$. So we have $g_1 = \{a_1, a_2\}$. $SF1_a$ is applied to group $g_1$ to compute its survivability:

$$surv(g_1) = SF1_a(g_1) = 1 - (Pr(A_1) + Pr(A_2)) + Pr(A_1 \land A_2)$$
$$= 1 - (dp(n_1)dp(n_2) + dp(n_1)dp(n_3)) + dp(n_1)dp(n_2)dp(n_3) = 0.468.$$  

Similarly, the remaining agents, $a_3$ and $a_4$, form the second group $g_2 = \{a_3, a_4\}$.

$$surv(g_2) = SF1_a(g_2) = 1 - (Pr(A_3) + Pr(A_4)) + Pr(A_3 \land A_4)$$
$$= 1 - (dp(n_4)dp(n_5) + dp(n_5)dp(n_6)) + dp(n_4)dp(n_5)dp(n_6) = 0.914.$$  

Thus, the survivability of the deployment $\mu$ is:

$$SF4_g(\mu) = surv(g_1) \times surv(g_2) = (0.468)(0.914) = 0.42775.$$  

**SF5: a split algorithm**

Given a specific node, $n \in V$, we can consider two possible disjoint events. In the first event, $E_1$, the node will stay connected (and $E_1$’s probability is $1 - dp(n)$).
Alternatively, in event $E_2$, the node will be disconnected (with probability $dp(n)$). If $n$ stays connected, all the agents that are deployed on it will survive. Thus the survivability of the network, in this case, will depend on the survivability of the rest of the agents that are located on $V \setminus \{n\}$. If $n$ is disconnected, the survivability of the network depends on the rest of the nodes, i.e., $V \setminus \{n\}$. The survivability of the original network is thus $(1 - dp(n)) \text{Prob}(E_1) + dp(n) \text{Prob}(E_2)$. In both events the problem of computing the probability is smaller than the original problem and could be solved recursively. The sub problems usually become even smaller when getting rid of irrelevant agents using the idea in [KST03]. There are several stopping rules that are specified in the three first lines of the pseudo code. The first two rules refer to situations in which it is possible to compute the exact survival probability of the future network. The third one has to do with future networks that has very small probability (computing through the recursion using $p; p = 1$ in the first call to $SF5$). For these networks of very low probability, $SF4$ is applied to underestimate the survivability.

Algorithm 14. $SF5(\mu, \mathcal{N}, \mathcal{M}, dp, p, \epsilon)$

(* Input: (1) a MAS deployment $\mu$ *)

(* (2) a network $\mathcal{N} = \{V, \mathcal{N}\}$ *)

(* (3) a set of agents $\mathcal{M}$ *)

(* (4) a disconnect probability function $dp$ *)

(* (5) the survivability of the known nodes during split $p$ *)

(* (6) a predefined threshold $\epsilon$ *)

(* Output: an underestimate of the survivability of $\mu$ *)

1. if $\mathcal{M} = \emptyset$, return $1$;

   else, if the agents of $\mathcal{M}$ are located on disjoint sets of nodes,
   then return $(\prod_{a \in \mathcal{M}} (1 - \prod_{a \in \mu(n)} dp(n)))$;

   else, if $p < \epsilon$,
   then return $SF4(\mu, \mathcal{N}, \mathcal{M}, dp)$.

   else, choose a node $n \in V$,

   (a) $V' = V \setminus \{n\}$;

   (b) $\mathcal{N}' = \mathcal{N}' = (V', V' \times V')$;

   (c) $\mu' = \mu$; $\mathcal{M}' = \mathcal{M} \setminus \{a | a \in \mu(n)\}$;

   (d) get rid of irrelevant agents in $\mathcal{M}$ and $\mathcal{M}'$.
7.6 Experiments and Results

We recognise the importance of evaluating algorithms with actual implementations. We put a lot of efforts into the experiments including environmental design, implementation, and performance analysis. We give the details of the experiments in this section.
We implemented all the above algorithms and tested them on a Linux PC. The survivability algorithms we are going to compare in the experiments are approximations anytime algorithm $SF2$, $SF3$, $SF4$, group algorithm $SF4_g$ and split algorithm $SF5$.

We assessed the quality of a solution as follows:

- If the number of nodes or the number of agents is small (say, less than 16), we compare the values returned by different approximations with the exponential exact algorithm $SF1_n$—in the case where there are more agents than nodes or the algorithm $SF1_a$—if there are more nodes than agents.

- If it is not feasible to compute either $SF1_n$ or $SF1_a$ because both the number of agents and the number of nodes are larger than 16, we use the upper bound algorithm on $SF1_a$ for comparison.

We need to carry out the experiments with various environment settings. In this chapter, we consider instances taken from a (fictitious) company that is using many local servers, personal computers, and some web servers to locate and run multiagent applications. As we know, web servers and personal computers have high probabilities to go down, while local servers usually have low disconnect probabilities. In the next section, we describe the variations of settings we use in our experiments. We use the term *space ratio* to refer to the ratio of the total amount of space available on nodes to the total space requirements of agents.

**Environmental settings**

We use various environmental settings in the experiments. Suppose a multiagent application $\mathcal{M}$ includes a lot of agents but only a relatively small number of servers (or nodes) is available. We set the ratio of agents and nodes to $5/3$. We consider the following two environments for such problem size setting:

**s1**: The network consists of a small number of web servers $N_w$ which is 30% of the involved servers, and many local servers $N_l$, which is 70% of the involved servers. The disconnect probabilities of these servers are either very high—above 0.9 ($dp(N_w) \geq 0.9$), or very low—below 0.1 ($dp(N_l) \leq 0.1$). The space ratio of nodes and agents is between 2 and 3.

**s3**: The network includes local servers only. Suppose that some of these servers are new, while the others are old, and therefore disconnect probability of
these servers is distributed normally between 0 and 0.4. The space ratio of nodes and agents is around 4.

Consider another multiagent application $\mathcal{M}'$ which consists of a small number of agents. The company intends to deploy $\mathcal{M}'$ on many personal computers and local servers because the available resources on each server or PC are limited. We assume that the ratio of nodes and agents is $5/3$. The following environments are specified:

**s2:** Personal computers (30%) are employed, whose disconnect probabilities are over 0.9; they also use several local servers which have low disconnect probabilities (less than 0.1). The space ratio of nodes and agents is around 2–3.

**s4:** Only local servers of different ages are used to host $\mathcal{M}'$. The disconnect probability of the servers is distributed normally between 0 and 0.4. The space ratio of nodes and agents is around 4.

A sample of 31 existing agents is used to determine a distribution of agent sizes (in the range of 0 to 250 KB). As to the ratio of total available resource on nodes and total resource requirements of agents, we assume this ratio is 3 in all experiments. We are going to use the environments s1–s4 described above to test the survivability algorithms.

**Deployment of agents**

The method to generate deployments of the agents is important to the evaluations since different survivability algorithms may work well with different types of deployments. We generate deployments by the heuristics proposed in [KST03], namely node-based heuristics and agent-based heuristics. In addition, we use a random-based method to represent any other possible deployments. The deployment heuristics work as follows.

1. **Node-based:** This is based on the knapsack problem. We first sort nodes in ascending order according to their disconnect probabilities. We then place agents on the sorted nodes starting from the node with the lowest disconnect probability. We put as many agents as possible on this node, then go to nodes with the second lowest disconnect probability and so on.
2. **Agent-based:** This is based on the idea that we should first deal with agents with high resource requirements. Thus we sort agents in ascending order according to resource requirements, deploy them, then choose agents with the second highest resource requirements, as so on until there is no more space left for placing agents.

3. **Random-based:** We first randomly choose a node, and then randomly select and place agents on it following the resource constraints. We make sure the deployment use up all available resources on nodes.

We tested survivability algorithms on different types of deployments with various environment settings.

**Experimental results**

We are now ready to present the experimental results following. In the experiments, \( \alpha \) in the tree based algorithm (\( SF3 \)) is set as the number of nodes in the deployment. The threshold of approximation ratio in the anytime algorithm (\( SF2 \)) is defined as 0.9 and the time limit on the main part of the algorithm was set to 5 seconds. The time units are of microseconds. Every recorded observation was averaged over 50 runs.

We compared approximations in terms of:

- the computation time (in microseconds), and
- the approximation ratio, where
  \[
  \text{approximation ratio} = \frac{SFs}{\text{actual value}}. 
  \]

The algorithms were compared on various deployments (node-based, agent-based, and random-based) with different environment settings \( s1 \)–\( s4 \).

In all experiments, we varied the problem size, that is, the total number of agents and nodes. We present the results of the following experiments:

- **Experiment 1** compared different approximations in setting \( s1 \);
- **Experiment 2** compared different approximations in setting \( s2 \);
- **Experiment 3** compared different approximations in setting \( s3 \);
- **Experiment 4** compared different approximations in setting \( s4 \);
• *Experiment 5* compared different approximations in setting $s_1$ but increased the space ratio of nodes to agents from 2–3 to 3–4;  

• *Experiment 6* was carried out with setting $s_1$. It investigates how the value of $\alpha$ in $SF_3$ would effect the performance of $SF_3$.

We present the experimental results next.

Figure 7.2: Experiment 1: computation time using different algorithms with setting $S1$

![Figure 7.2](image)

Table 7.1: Experiment 1: Approximation ratio using different algorithms with setting $S1$

<table>
<thead>
<tr>
<th>Problem size</th>
<th>deployment</th>
<th>Anytime algo.</th>
<th>$SF_3$</th>
<th>$SF_4$</th>
<th>Split algo.</th>
<th>Group algo.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_{18}, a_{30}$</td>
<td>node-based</td>
<td>0.951475</td>
<td>0.998919</td>
<td>0.879377</td>
<td>0.999515</td>
<td>0.948644</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.930565</td>
<td>0.972725</td>
<td>0.879446</td>
<td>0.972050</td>
<td>0.947005</td>
</tr>
<tr>
<td>$n_{24}, a_{40}$</td>
<td>node-based</td>
<td>0.951971</td>
<td>0.964037</td>
<td>0.836856</td>
<td>0.965813</td>
<td>0.923682</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.861461</td>
<td>0.881713</td>
<td>0.796948</td>
<td>0.889531</td>
<td>0.859259</td>
</tr>
<tr>
<td>$n_{30}, a_{50}$</td>
<td>node-based</td>
<td>0.899324</td>
<td>0.937145</td>
<td>0.775934</td>
<td>0.938865</td>
<td>0.893412</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.811792</td>
<td>0.851059</td>
<td>0.731787</td>
<td>0.852492</td>
<td>0.827303</td>
</tr>
</tbody>
</table>

**Experiment 1.** In Experiment 1, we ran and compared five approximations in setting $s_1$ where the space ratio of nodes to agents is between 2 and 3 and the disconnect probabilities are distributed either in 0–0.1 or in 0.9–1. There are two kinds of deployments: one generated by agent-based heuristic and the other by node-based heuristic.

Table 7.1 illustrates the results of approximation ratio by different algorithms. The problem size varied from 48, 64 to 80—$n_{18}, a_{30}$ in the table refers to a multiagent system of 30 agents deployed over 18 nodes. $SF_3$ and split algorithms
return very good approximations. In particular, $SF3$ always gives higher accuracy than $SF4$, anytime and group algorithm. The split algorithm has the best approximation, though the difference between it and $SF3$ is very small.

Figure 7.2 shows the computation time (in logarithm scale) taken by different approximations with varying problem size on agent-based deployments. It is obvious that $SF3$ needs much less computation time than split algorithm and anytime algorithm (in the order of $10^3$, $10^4$, $10^5$–$10^7$ respectively), while $SF4$ is the fastest algorithm among all. It only takes several microseconds to compute the survivability of a given deployment. The time needed by group algorithm is close to that of $SF3$.

Overall, in experimental setting $s1$, if the space ratio of nodes to agents is below 3, $SF3$ outperforms other algorithms both w.r.t. approximation ratio and computation time into account.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>deployment</th>
<th>Anytime algo.</th>
<th>$SF3$</th>
<th>$SF4$</th>
<th>Split algo.</th>
<th>Group algo.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_{30}, a_{18}$</td>
<td>node-based</td>
<td>0.928353</td>
<td>0.990334</td>
<td>0.974706</td>
<td>0.99961</td>
<td>0.989189</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.939275</td>
<td>0.996662</td>
<td>0.973966</td>
<td>0.998569</td>
<td>0.998401</td>
</tr>
<tr>
<td>$n_{40}, a_{24}$</td>
<td>node-based</td>
<td>0.880772</td>
<td>0.980832</td>
<td>0.952544</td>
<td>0.98856</td>
<td>0.979202</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.907224</td>
<td>0.968986</td>
<td>0.941974</td>
<td>0.991491</td>
<td>0.988068</td>
</tr>
<tr>
<td>$n_{50}, a_{30}$</td>
<td>node-based</td>
<td>0.923707</td>
<td>0.981228</td>
<td>0.958711</td>
<td>0.973521</td>
<td>0.96628</td>
</tr>
<tr>
<td></td>
<td>agent-based</td>
<td>0.910653</td>
<td>0.972655</td>
<td>0.960914</td>
<td>0.976648</td>
<td>0.967943</td>
</tr>
</tbody>
</table>

Table 7.2: Experiment 2: Approximation ratio of different algorithms with setting S2

**Experiment 2.** Experiment 2 was carried out with setting $s2$, where the size ratio of nodes and agents is 5/3 and the space ratio of nodes to agents is 2–3. The disconnect probabilities are distributed dramatically either in 0–0.1 or in 0.9–1.

Table 7.2 reports the results of various approximation. In terms of accuracy, the split algorithm is still the best, followed by $SF3$, which outperforms the group algorithm both on the node-based deployments with problem size 48, 64, 80, and on the group-based deployments with size 80. In addition, the performance of $SF4$ and group algorithm is better than that in Experiment 1. This is not surprising as in $s2$, there are more agents who have disjoint locations with others compared to those deployments in setting $s1$. Thus $SF4$ and group algorithm should work better. The computation time taken by different approximations is similar to that shown in Figure 7.2. The results imply that $SF3$ is best among all approximations as far as both time and accuracy are concerned.
Figure 7.3: Experiment 3: approximation ratio using different algorithms with setting S3

**Experiment 3.** In Experiment 3, we ran and compared different approximations in setting $s3$ where the space ratio of nodes to agents is around 4. The disconnect probabilities are distributed normally in 0–0.4.

Figure 7.3 and 7.4 illustrate the approximation ratios and the computation times for different algorithms. In both figures, the x-axis represents the problem size, varying from 32 to 98 in step of 16. The figures show computation times on various deployments (i.e, node-based, agent-based, and random-based) respectively. We did not include the results of $SF3$ since its approximation ratio is much lower (below 0.8) than others in this setting.
Figure 7.4: Experiment 3: computation time using different algorithms with setting S3

Figure 7.3 shows the advantage of the split algorithm, which gives the best approximation ratio no matter what kind of deployments it works on. The figure illustrates that all algorithms have high approximate ratio (over 0.96) on node-based deployments. As for agent-based deployments, all algorithms return over 0.90 approximate ratio with problem size no larger than 80. Only the ratio of SF4 drops to 0.86 when the problem size goes up to 96. All algorithms but the anytime algorithm get above 95% accuracy on random-based deployments. The approximate performance of all algorithms decrease as the problem size increases. The anytime algorithm converges much faster on node-based deployments than agent-based deployments with setting s3—the computation time taken on the
latter is 100 times or so when the problem size is over 48 according to Figure 7.4. The time taken by split algorithm is about 10 times that taken by the group algorithm. Again, SF4 is the fast algorithm among all.

**Experiment 4.** Experiment 4 was carried out with environment setting s4, where the ratio of nodes to agents is 5 : 3, and disconnect probabilities are distributed normally in [0, 0.4].

Table 7.3 and Figure 7.5 show the effect of problem size on the approximation ratio and computation time by various approximations. We did not include the
results of $SF_3$ because of its low approximation ratio (below 0.8). As far as accuracy concerned, the performance of all approximations is very good—the accuracy is always over 0.97. In particular, $SF_4$, split and group algorithms could always achieve over 0.998 approximation ratio in Experiment 4 according to Table 7.3. Since there are more nodes than agents, the size of each node decreases compared with that in Experiment 3, thus most agents are disjoint with others w.r.t their locations, which makes algorithms, especially $SF_4$ and group algorithm, perform well. Unlike shown in Experiment 3, the anytime algorithm in Experiment 4 converged pretty fast no matter what kind of deployments. The computation time taken by $SF_4$, split and group algorithm is very close to those in Experiment 3.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{size} & \text{deployment} & \text{Anytime algo.} & \text{SF3} & \text{SF4} \\
\hline
n20,a12 & \text{node-based} & 0.99988 & 0.999971 & 0.999996 \\
& \text{agent-based} & 0.999845 & 0.99981 & 0.99994 \\
& \text{random} & 0.991724 & 0.999837 & 0.999999 \\
\hline
n30,a18 & \text{node-based} & 0.999922 & 0.999965 & 0.999983 \\
& \text{agent-based} & 0.99979 & 0.999635 & 0.999823 \\
& \text{random} & 0.991622 & 0.999918 & 0.999983 \\
\hline
n40,a24 & \text{node-based} & 0.999723 & 0.999969 & 0.999983 \\
& \text{agent-based} & 0.999588 & 0.999269 & 0.99996 \\
& \text{random} & 0.991255 & 0.999939 & 0.999889 \\
\hline
n50,a30 & \text{node-based} & 0.999661 & 0.999968 & 0.99989 \\
& \text{agent-based} & 0.999398 & 0.999136 & 0.999638 \\
& \text{random} & 0.990589 & 0.999655 & 0.999787 \\
\hline
n60,a36 & \text{node-based} & 0.999681 & 0.999923 & 0.999971 \\
& \text{agent-based} & 0.999217 & 0.998105 & 0.999097 \\
& \text{random} & 0.976279 & 0.998948 & 0.999713 \\
\hline
\end{array}
\]

Table 7.3: Experiment 4: Approximation ratio using different algorithms with setting s4

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Problem size} & \text{deployment} & \text{Anytime algo.} & \text{SF3} & \text{SF4} & \text{Split algo.} & \text{Group algo.} \\
\hline
n18, a30 & \text{node-based} & 0.981061 & 0.98647 & 0.96855 & 0.998956 & 0.98707 \\
& \text{agent-based} & 0.973769 & 0.986316 & 0.964955 & 0.998059 & 0.988083 \\
\hline
n24, a40 & \text{node-based} & 0.979052 & 0.991792 & 0.978524 & 0.99866 & 0.994861 \\
& \text{agent-based} & 0.980291 & 0.992817 & 0.978784 & 0.998752 & 0.99174 \\
\hline
n30, a50 & \text{node-based} & 0.98326 & 0.975502 & 0.982723 & 0.998131 & 0.994358 \\
& \text{agent-based} & 0.981894 & 0.975746 & 0.979898 & 0.997898 & 0.994035 \\
\hline
\end{array}
\]

Table 7.4: Experiment 5: Approximation ratio with setting S1 but using larger space ratio (3-4)

**Experiment 5.** In Experiment 5, we repeated Experiment 1 with setting s1 but increased the space ratio of nodes to agents from 2–3 to 3–4. Table 7.4 illustrates
that $SF3$ returns lower survivabilities than split and group algorithms, moreover, its accuracy is even lower than $SF4$ when the number of nodes and agents is 30 and 50. In addition, Figure 7.6 shows that $SF3$ is the most time consuming algorithm with the problem size 80 where $SF3$ needs 100 times of computation time than split and group algorithms. The experiment results imply that with such setting used in Experiment 5, $SF3$ is no longer the preferred approximation especially with the large problem size. Instead, the split algorithm demonstrated its capability of fast and accurate computation.

**Experiment 6.** In this set of experiments, we investigate how the value of $\alpha$ in $SF3$ would effect the performance of $SF3$. We only use $SF3$ as the survivability function. We run the program with varying $\alpha$. The measurements are computation time and survivability. The number of nodes in the network is 50 and the number of agents is set as 100. We randomly generated the deployments. Figure 7.7 shows the results of Experiment 6. With the increase of the value of $\alpha$, the value returned by $SF3$ increases from 0.246 to 0.273, while with the price of computation time that is enlarged by 16 times. When the problem size is large enough (100 in this experiment), even if we choose a small value of $\alpha$, the survivability given by $SF3$ is slightly lower than the value computed with a large $\alpha$. 

Figure 7.6: Experiment 5: computation time with setting S1 but using larger space ratio (3-4)
Figure 7.7: Experiment 6: Computation time and survivability returned by SF3 with varying $\alpha$.

Evaluation of results

We reported above the experiments which were carried out in various environment settings $s_1$—$s_4$ with different kinds of deployments. We now evaluate the experiment results that we have achieved.

- $SF3$ works well with some settings. As shown in Experiment 1 and 2, $SF3$ is preferable to other approximations as far as time and accuracy are concerned in settings $s_1$ and $s_2$. The settings $s_1$ and $s_2$ where (1) the disconnect probabilities of the nodes are distributed either in $[0, 0.1]$, or in $[0.9, 1]$; and (2) the space ratio of nodes to agents is less than 3. However,
Experiment 5 shows that when the space ratio is increased to 4, $SF_3$ no longer give the best performance. The split algorithm is the best instead.

- The Split algorithm outperforms others in setting $s_3$, where (1) the disconnect probabilities of the nodes follow normal distribution in $[0, 0.4]$; and (2) the ratio of nodes to agents is $3:5$.

- In setting $s_4$, where the disconnect probabilities are normally distributed in $[0, 0.4]$ and the ratio of nodes to agents is $5:3$, although every approximation does well in terms of accuracy, $SF_4$ is preferable to others when computation time is taken into account.

- All experiment results demonstrate that $SF_4$ has an unbeatable fast computation time.

We investigated the performance of different algorithms in various environment scenarios taken from a fictitious company. For applications where there are other possible environments for deploying agents and other possible deployments of agents, we can consult the experimental results about choosing appropriate algorithms.

### 7.7 Conclusion

The contributions in this chapter are as follows.

1. We gave an axiomatic definition of a survivability function. Our axiomatic definition captures the probability of survival of a deployment.

2. We provided two exact survivability functions called $SF_{1_n}$ and $SF_{1_a}$. $SF_{1_n}$ is exponential in the number of nodes, while $SF_{1_a}$ is exponential in the number of agents.

3. We showed that even if we assume independence, finding an optimal deployment and computing the survivability of a given deployment are both NP-hard.

4. We showed that if we want to use a polynomial approximation to find a sub-optimal deployment, there will be instances where the polynomial approximation says that the survivability of a deployment is 0, when in fact
the true survivability is 1. Thus, polynomial approximations are guaranteed to find at least one terrible solution.

5. We developed five approximations that underestimate the survivability of a deployment under the independence assumption.

6. We reported the results of an exhaustive set of experiments that we conducted to assess both the computation time and the quality of solutions found by the above approximations. We identified the situations under which different approximations work well.

The survivability algorithms proposed in this chapter can be considered as the candidates of the centralised algorithm CSA in our distributed algorithms ASA1, ASA2, and ASA3 in Chapter 6. Chosen a proper survivability function according to the environment where the multiagent application is deployed, the distributed survivability model can ensure the better survivability of the deployed MAS by adapting to the changes in the environment.
Chapter 8

Discussion and Future Work

We have introduced and investigated various survivability algorithms which provide a means of measuring the survival of a multiagent system. We have also proposed distributed models which can build on any survivability algorithms in order to ensure the maximal survivability of the system. Our survivability algorithms and models are developed based upon the idea of replicating agents.

In Section 8.1 of this chapter, we will review replication based approaches to improving survivability of multiagent systems. As extensions of our approach, some interesting future works will be suggested in Section 8.2.

8.1 Replication in Multiagent Systems

The applications of replication in agent systems can be classified into two kinds: one on regular agents, which are actually deployed to provide services; and the other on middle agents, which are specially added to the multiagent systems for maintaining fault tolerance. We discuss these two types of replication application in this section. Before discussing these approaches, we briefly describe two basic replication protocols.

Agent Replication Protocols

Agent replication creates one or more replicas for one or more agents in a multiagent system. Each of the replicas is able to perform the same task as the original agent. We refer to a group of replicas of an agent as a replicate group, and each replica within a replicate group as a replica. The idea of agent replication is
8.1. REPLICA TION IN MULTIAGENT SYSTEMS

simple—if one replica fails, there is another one in the replicate group ready to take over, thus the whole system will not compromised.

Among the approaches there are two basic types of replication protocols for generating replicas: passive replication and active replication.

- **Passive replication**: It assumes that there is one active replica, a leader, in each replicate group, who plays a special role. It receives and processes all input messages, periodically transmits its updated state to the other replicas in the same group in order to maintain consistency. If the active replica crashes, a new one must be elected as a leader among the remaining replicas based on some particular criteria, such as speed and reliability. Agents outside a replicate group can communicate with the group by interacting only with its leader.

- **Active replication**: This technique gives all replicas the same role without the centralised control of the passive technique. All of the replicas receive and process the input messages. An agent outside the group can either sends requests to each replica in turn until it receives an appropriate reply, or interact with all replicas then synthesise all replies of the replicas.

Active replication protocols provide a fast recovery delay but lead to the overhead of CPU. The passive replication protocol needs more processing time for recovery delays but requires less CPU resources as it activates redundant replicas only in case of failures. Active replication has all of the replicas active at the same time. Thus it must deal with results synthesis and ensure that reads and writes from the replicas do not result in any inconsistencies. On the contrary, passive replication has the advantage that results synthesis is not required and read and write consistency problems are avoided.

The choice of the most suitable protocol is dependent of the applications and the environment context, such as the failure rate, or the current available resources.

**Replication of Regular Agents**

Replication of agents provides an intuitive yet efficient way to achieve fault tolerance in multiagent systems.
Marin et al. [MSBG01, MBS03] introduce a framework, called *Dynamic Agent Replication eXtension* (DARX for short), to design reliable multiagent applications. A replicate group is an entity underlying every agent. Each group has one leader who is responsible to communicate with the other groups and detect failures within its group. The leader dynamically adds or removes replicas, carries out the change of the current replication strategy, and handles failure recovery within the replication group. In case of failure of a leader, the naming service elects a new one among the set of remaining replicas. Once an active replica fails, it selects other replicas within the group as the new active replica.

This work intends to bring some amount of flexibility to the system. DARX provides both passive and active replication strategies that are not fixed. Each agent can change, at runtime, its replication protocol and tune internal parameters such as the number of replicas. In addition, it allows to decide, though by the designer, which agents are more critical than the others, and hence should be made to bypass failures through replication.

Fedoruk and Deters [FD02] introduce proxies for groups of replicas representing agents. In order to minimise additional complexity and system loads that are introduced by the use of replication, they propose a *transparent replication technique*, which makes the group proxy act as an interface between the replicas and the rest of the multiagent system. In this way, the proxies make the group appear to be a single entity and they control execution and state management of a replicate group. To do so, they introduce several proxies: *communication proxy* handles all communications between replicas and other agents in the multiagent system; *data proxy* ensures all replicates receive the same percepts from the environment and handles results synthesis to ensure write consistency.

The approaches described above mainly investigate the problems introduced by the use of replication techniques within a multiagent system. Such problems include agent communication methods and results synthesis, read and write consistency among replicas, and state synchronisation of agents. On the other hand, they did not address some important issues how to apply replication technique, which are, for example:

- which agent should be replicated?
- where should the replica be deployed?
- how many copies of replicas should be made for each of the agent?
The problems above could be crucial to the performance of the multiagent system. The solutions of [MSBG01, MBS03, FD02] that apply replication schemes to each agent may be infeasible in practice since replication is often very costly in processing as well as in communications. Therefore, we should use replication optimally when and where and how many they are most needed. The DARX approach realises the importance of addressing these problems, however, in their approach, it is the designer’s responsibility to decide in advance, for every agent, which one, how many, where, and when to modify replication scheme.

Among the advantages of our approach, in contrast to theirs, are: firstly, a measurement of the survival level of the multiagent system is given, thus we have a way to decide on the number and the locations of the agent replicas according to the current environment; secondly, other than regular agents, the fault tolerant entity in our model, deployment agent da, is also distributed over the network, which not only takes the survivability burden off the regular agents, but also avoids the single point of failure.

Replication of Middle Agents

Rather than replicating regular agents to prevent themselves from failures, other specially designed agents can improve the survivability in a multiagent system. This kind of special agents, which is distinguished from the regular agents who provide services in the system, is called a middle agent. Middle agents maintain a continually updated information about regular agents in the system. In the context of fault tolerant multiagent systems, they are also responsible for detecting failures and recovering from failures. As we described in Chapter 4, Hagg [Hag96] introduced sentinels to monitor agent behaviours in order to improve the survivability of the whole agent system. In our approach, we have a special deployment agent da to in charge of the survival issue of the whole multiagent system.

The middle agent based approaches increase the number of agents in the system, however, they offload fault tolerance from regular problem solving agents. The middle agent can be designed and adapt to particular faults. Moreover, it can be incorporated into legacy systems and it reduces the system load. However the middle agent based approach is usually centralised, thus it becomes a bottleneck.

Some researchers have realised the single point of failure problem of the middle agent, and therefore applied replication on the middle agent in the system.
In [KCL00], Kumar et al. propose an adaptive multi-brokered agent system, named AAA, which apply the replication technique to middle agents (or broker) rather than regular problem solving agents. Thus, AAA is robust to the broker unavailability. In this system, several brokers could substitute for each other when one becomes unavailable. As the reduced number of brokers may degrade the system’s performance, they hypothesise that the team commits to maintain a specified minimum number of brokers. However, the number of brokers in their system must be predefined by the persistent team and will not change during the execution of the system. Their approach to survivability is completed by introducing the redundancy into the critical broker agents in a multiagent system. They only considered the potential failures of brokers but ignored the possible faults of the regular agents.

In the context of mobile agent systems, survivability focuses on how to avoid the loss of agents during execution. Mishra and Huang [MH00] introduce a Dependable Mobile Agent System (DaAgent for short) to recover node and communication failures. The middle agents, sentinels, are distributed on every node in the network. They monitor the movement of agents and ensure that agents can arrive at their destinations reliably. The DaAgent system consists of five modules: a DaAgent server, an agent watchdog, an agent consultant, and a communication module. A DaAgent server runs on every node in the network. Each mobile agent in the system is associated with an agent watchdog. Agent consultants, which locate on every node, determine the migration of agents from one node to another node. Survivability in DaAgent mainly relies on the middle agents in the system.

Similar with our method, the approaches of [KCL00] and [MH00] to ensure survivability are distributed. [KCL00] decides the number of replicas of the middle agent by the developer in advance—it does not consider the issue of maximal survivability of the system. [MH00] deploys the middle agent to each node in the network—they do not take into account the resource consumption of the middle agents. Moreover, as far as the adaptability is concerned, their approaches are both inferior to ours since our model is capable of re-evaluating the multiagent survivability with the changes of the external environment.

**Agent Cloning**

The designs and applications of above replication techniques are mostly static, in the sense that the quantity and the locations of agent replicas will not change once
the applications start. However, often re-evaluating the quality of the replication is necessary due to some possible changes of environment or agent workloads.

Recently, a number of agent cloning techniques have been proposed in order to achieve more adaptive systems. Agent cloning can be considered as an extension or a variation of agent replication. Cloning is a superset of task transfer and agent migration. A new agent is created on the local or a remote machine.

There are various ways of cloning: the simplest one is to copy all components of the template agent and give a new identifier to the resulting agent; in some cases, we would prefer clones of an agent smaller, smarter and more efficient than the original agent. So splitting clones into several smaller pieces may result in better performance, especially when we want to transfer an agent from one host to another host to carry out a task while only part of the knowledge and resources of the original agent is sufficient to perform the desired computation.

Decker et al. [DSW97b] use agent-cloning and agent-merging techniques to mitigate agent over-loading and promote system load balancing. Shehory et al. [SSCJ98] use agent-cloning to mitigate agent overloads in an open environment. In their paradigm, agents may clone, pass tasks to others, die or merge. Agents use standard dynamic programming to compute the optimal decision regarding the optimal time for cloning, with the knowledge of current load, current task queue, and future expected task flow. The cloning procedure consists of the following processes:

- **Reasoning before cloning:** This includes the reasoning about the task list with respect to time restrictions and capability and resource requirements.

- **Dividing the list of tasks:** This includes reasoning that considers the time intervals in which overloads are expected and accordingly selects tasks to be transferred.

- **Cloning:** This includes the creation and activation of the clone, the transfer of tasks, and the resulting inevitable updates of connections between agents.

- **Reasoning after cloning:** This collects information regarding the benefits of the cloning and environmental properties for future statistical analysis.

Fan [Fan01] furnishes each local agent with the capability of load-balancing and investigates how to split an agent into smaller and smarter ones. The approach is to build a BDI agent model, which allows to define different kinds of
cloning mechanisms—logical splitting, behaviour splitting, retrogressive spitting, and prototype cloning—and analyse properties and relationships between them. They define the semantics of constructing new agents by inheritance and self-identifying behaviour of existing agents in the BDI agent model.

Frederic et al. [FJ95] propose a model that reorganises its architecture by allowing not only agent-splitting but also agent-merging. Two primitives, composition and decomposition are defined in order to adapt the model to the workloads and the changes of environment. Composition allows the transformation of an entity into another entity, and decomposition allows one or several agents to delegate part of its skills. Their approach focuses on modification of the content and the number of agents in the system.

Agent-cloning approaches in [SSCJ98, Fan01, FJ95] provide more adaptive and dynamic multi-agent systems. Cloning is performed only when some overloads are detected in the system. This “replicating when necessary” idea makes cloning more applicably and significantly reduce the complexity introduced by replication. The capability of reorganisation provides a better tolerance to breakdowns.

These cloning approaches mainly target at the agent overload problem. The re-evaluation takes place when there is a need of maintaining the load balance—either in agents or in the system. Our approach takes into account more factors other than workload that could trigger the re-evaluations. We have not yet considered agent-merging or agent-splitting in our framework, however, it has been identified as future work (see Section 8.2 in this chapter).

8.2 Future Work

It is not surprising that we have not addressed all issues which are related to our approach. For example, we do not concern the technical details of replication, such as which replication protocol to be used and so on. We only consider, for the moment, the possible changes or failures of the disconnection of nodes, however extending our approach by taking account of other types of failures or environment changes is possible and not difficult to realised.

There are still many other interesting issues that remain unaddressed in our work. We only mention some of them in the following.
8.2. FUTURE WORK

Agent Mutation

In our current framework, we assume that the agent provides one service (or task) only. We can extend it by assuming that an agent may offer several services. The survivability of the system is then defined as: at least one node in the system must offer each service in the multiagent system at all time. Thus, our aim moves away from the old one: we now ensure all the services provided by the system must survive.

Therefore, when we re-deploy the replicas of agents in order to increase the survival of the whole system, sometimes we do not need to clone the whole agent but only a part of it—we then consider how to factor the techniques of agent-splitting, agent-merging into the current framework.

Computational Bounded Survivability

One issue studies the trade-off between the survivability of the multiagent system and its performance. When regular agents keep ensuring survivability, they must consume some resources such as the process cycles and the network bandwidth in order to communicate with the deployment agent \( da \). However, this comes at the cost of actually providing the services that the multiagent system is supposed to provide. This problem is significant for multiagent applications where resources are limited.

We investigate how to balance these two concerns. We can factor the resource consumption into the frameworks introduced in Chapter 6 so that instead of maximising the survivability, we find the deployment that maximises the expected performance subject to the additional requirement that survivability of the deployment exceeds the given threshold.

Coordinative Survivability

Another interesting problem arises if we assume the deployment agent \( da \) is not always having complete knowledge about the whole network and environment.

In our current approach, we make some assumptions on the deployment agent \( da \). We assume that \( da \) has full knowledge about the deployment of agents and the disconnect probability function of all of the nodes in the network. Furthermore, we assume that the knowledge of all replicas of \( da \) is consistent at all time. As one of our future work, we can investigate various \( da \) models where the replica
of da is in charge of only a subnetwork of the entire network. All da replicas collaborate together to fulfil the task.

We assume that the overall structure of the system consists of a set of subnetworks. Now suppose that there are \(k\) number of subnetworks \(SN\), and the probabilities of survival of these subnetworks are \(surv_1, \ldots, surv_k\). Thus the overall survivability of the whole network is the worst case among all of the subnetworks, that is, \(surv = \min\{surv_i | 1 \leq i \leq k\}\). We could predefine a threshold \(\theta_N\) that a network must fulfil. The aim of the system is to ensure that the survivability must not be less than the threshold; that is, \(surv \geq \theta_N\).

Each subnetwork \(SN_i\) has the following components: (1) a set of nodes, which are assigned to the subnetwork based on some algorithms. The nodes may be lent by other subnetworks in order to optimise the whole network; (2) a deployment agent \(da\), or a replica group of deployment agents \(da\) in each subnetwork has the responsibility of both maintaining the survivability of its subnetwork, and of communicating with other subnetworks to ensure the survivability of the network as a whole; (3) a set of regular agents that provide services.

Such assumption brings up many interesting problems.

**Design of the survival protocol:** The survival protocol specifies the rules of communications between the da in different subnetworks as well as the da and the regular agents. It defines the circumstances under which the da’s should talk to each other, what messages are allowed to be exchanged and what offers can be made between them.

Resources, such as nodes and regular agents, are distributed in the subnetworks. When some changes of the network occur, subnetworks may need to be reorganised and resources may be relocated. The protocol should maintain the survival of both the deployment agents and the regular agents in each subnetwork. In addition, it optimises the survivability of the whole network by relocating resources, such as borrowing or lending nodes from or to other subnetworks. This results in reorganisation of the network. Contract Nets can be applied in order to find proper subnetworks for the task of reorganisation.

**Split the network:** Splitting the whole network involves the problems of how to group nodes and how many subnetworks should be generated, and of how the da replicas behave in each subnetwork.
8.2. FUTURE WORK

We could split the network by putting the nodes which are close together into a subnetwork. In such a situation, we could assume that the da is deployed on each node, however, only one replica of da is active within one subnetwork.

Another possible way is that we can assume a network is partly connected and da is deployed over the network with regular agents by the algorithms developed in Chapter 6. If $da_i$ is the replica who locates on the node $n_i$. We then find a subnetwork $SN_i$ such that for all nodes $n'$ in $SN_i$, the length of the shortest path from the node $n_i$ to node $n'$ is less than $k$. That is, each da is responsible for its neighbour nodes within $k$-path length. It is obvious that with smaller $k$, the network is more reliable, however the communication cost is increased in the network.

The third possible model is Similar to the second one—each replica of da is responsible for one subnetwork. The difference comes at the way to split the network into several disjoint subnetworks by the idea of minimum spanning tree. More specifically,

- we first find a minimum spanning tree $ND = \{VD, ED\}$, where $ED \subseteq E$, and $VD = \{n_i | da \in \mu(n_i)\}$ is a set of nodes where the replicas of da are deployed;

- then based on $ND$, find a minimum spanning tree $T = \{V, E'\}$ such that $(ND \subseteq T) \lor (T \in N)$;

- we then split the MST $T$ into subtrees $ST$ by disconnecting edges $ED$, where $ED = \{(n_u, n_v) : n_u, n_v \in VD\}$;

- finally, we build subnetwork by expanding each subtree $ST$. 
Chapter 9

Conclusion

We conclude our work in this chapter. Recall our objectives of this thesis:

**Objective 1:** To develop the debugging technique to verify whether a multiagent system is correctly designed and implemented, and to monitor online behaviours of collaborative agents to reach some certain goals.

**Objective 2:** To develop distributed approaches which ensure the survivability of a multiagent system and are capable of adapting to the changing environment.

**Objective 3:** To develop efficient survivability algorithms which are able to measure and compute the survivability of a given multiagent system, and thus to guide replications in the multiagent system.

9.1 Part I: Monitoring Multiagent Systems

Part 1 of this thesis has achieved **Objective 1**.

**Objective 1:** Chapter 3 introduced a monitoring approach, which can be used for offline debugging and online monitoring for a multiagent system. The approach is based on monitoring the message exchange between agents using planning methods. We implemented running examples within a multiagent platform **IMPACT** and a planning system **DLV**. We also set up a project homepage at http://www.in.tu-clausthal.de/~yzhang/monitoring.html, where two animations demonstrate how our method works.
The contributions we have made in Part 1 are the following:

- Our approach can be seen as a very *useful debugging tool* specifically designed for multiagent systems. It detects potential coding and design errors in MASs. Moreover, it works for arbitrary agent systems.

- Our approach can be also used to monitor the online behaviours of agents. Since the approach adapts to any planning formalism, it is flexible and even applicable to those multiagent systems where agents may have different attitudes and collaborative objectives.

- In addition, since the action theory forms a formal system specification, it is possible to reason about it and to be used in other contexts other than debugging and monitoring.

### 9.2 Part II: Survivability of Multiagent Systems

Part 2 of this thesis has achieved *Objective 2* and *Objective 3*.

**Objective 2:** Chapter 6 investigated the *multiagent survivability* problem in changing environments. Unlike static approaches, we introduced three distributed algorithms, which extend *centralised survivability algorithms* but are completely distributed and can redeploy agents when there is a need to re-evaluate the survivability of agents. We also reported a set of experimental results to assess how good these algorithms are in terms of the computation and network time and the solution quality.

**Objective 3:** Chapter 7 provided two survivability algorithms to compute the survival probability of the given multiagent system under the independent assumption of failures on network nodes. We showed that computing survivability in this environment is intractable. As a result, five fast sub-optimal approximations have been introduced. We described results of an exhaustive set of experiments that we conducted to assess both the computation time and the approximation ratio of different algorithms. We reported on the advantages and disadvantages of different heuristics in different environmental settings, and identified the situations under which different approximations work well.
We make the following contributions in Part 2 of this thesis:

- We provide a *formal probabilistic survivability model* for ensuring the survivability of multiagent applications. It is especially useful for the agent applications in environments which are *hostile, changing, and resource bounded*, such as those for military tasks.

- The distributed architectures and algorithms (Chapter 6) are distributed and adaptive to the changing environments. They ensure maximal, online, and distributed survivability of multiagent applications. Moreover, they can be built upon any arbitrary but fixed centralised survivability algorithm.

- We introduce both exact and approximate algorithms (Chapter 7) to measure the survivability of multiagent systems. Different algorithms can be chosen to use for different environmental settings.

- In addition, the proposed fast algorithms in Chapter 7 can be used in other context. For instance, they can adapt to solve *file replication problem* to determine where to locate different file replica on a large-scale network such as the grid.

However, as we pointed out in Chapter 4 and Chapter 8, there are still some open problems left in our work. We also leave some interesting problems for future work.
Bibliography


Appendix A

Specification of the Complete Gofish MAS

The specification of the extended Gofish Multiagent system as a planning problem using DLV notation:

actions:

DropOff(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

add_package(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority, DropTime) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

DistCenter(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

Truck(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

send_ziptozip(Pid, RecipEmail, OrigZip, DestZip, DropTime) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

send_centertoohouse(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

send_trucktohouse(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
APPENDIX A. SPECIFICATION OF THE COMPLETE GOFISH MAS 176

get_recipient_info(Pid,L) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime),
location(L).

recipient_info(Pid,L,RecipTel,RecipEmail,DestZip,DestStreet,DropTime) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime),
location(L).

Delivery(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

set_delivery_time(Pid, DelivTime) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

statistic_info(Pid,OrigZip,DestZip,DestStreet) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

CustomerPickup(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

fluents:
% Fluents describing the internal states of believe of the monitoring agent:

undelivered.

packageAt(Pid, L) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime),
location(L).

added(Pid) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

tobesentfrom(Pid,L) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime),
location(L).

informed(Pid,L) requires
package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail,
OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime),
location(L).

tobeinformed(Pid,L) requires
appendixA.specificationOfTheCompleteGofishMAS

package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime), location(L).

delivered(Pid) requires package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

stats_sent(Pid) requires package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

recipAtHome(Pid) requires package(Pid, PType, Wt, Vol, LSender, FSender, LRecip, FRecip, RecipTel, RecipEmail, OrigZip, DestZip, DestStreet, DestNum, Priority, Cost, DropTime, DelivTime).

initially: recipAtHome(Pid).

always: noConcurrency.

described undelivered if not stats_sent(Pid).

described inertial packageAt(Pid,L).

described inertial added(Pid).

described inertial delivered(Pid).

described inertial informed(Pid,L).

described inertial tobeinformed(Pid,L).

described inertial tobesentfrom(Pid,L).

described inertial stats_sent(Pid).

described inertial recipAtHome(Pid).

executable DropOff(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority).

described packageAt(Pid, "DropOff") after DropOff(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority) if packageAt(Pid,L).

% DropOff cannot occur if the package is SOMEWHERE or if has been added already:

described nonexecutable DropOff(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority) if packageAt(Pid,L).

described nonexecutable DropOff(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority) if added(Pid).

executable add_package(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority, DropTime) if packageAt(Pid, "DropOff").

% Package cannot be added again if it has already been added:

described nonexecutable add_package(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority, DropTime) if added(Pid).

described caused added(Pid) after add_package(Pid, PType, Wt, Vol, FSender, LSender, OrigZip, FRecip, LRecip, RecipTel, RecipEmail, DestNum, DestStreet, DestZip, Priority, DropTime).

executable send_ziptozip(Pid, RecipEmail, OrigZip, DestZip, DropTime)

if packageAt(Pid, "DropOff"), added(Pid).

described packageAt(Pid,"ontheway") after send_ziptozip(Pid, RecipEmail, OrigZip, DestZip, DropTime).

described -packageAt(Pid,"DropOff") after send_ziptozip(Pid, RecipEmail, OrigZip, DestZip, DropTime).
APPENDIX A. SPECIFICATION OF THE COMPLETE GOFISH MAS

executable DistCenter(Pid) if packageAt(Pid,"ontheway").
caused packageAt(Pid, "centertohouse") after DistCenter(Pid).
caused -packageAt(Pid,"ontheway") after DistCenter(Pid).

executable send_centertohouse(Pid) if packageAt(Pid,"centertohouse").
caused tobesentfrom(Pid, "centertohouse") after send_centertohouse(Pid).
nonexecutable send_centertohouse(Pid) if tobesentfrom(Pid, "centertohouse").

executable get_recipient_info(Pid,L) if tobesentfrom(Pid,L), packageAt(Pid,L).
nonexecutable get_recipient_info(Pid,L) if tobeinformed(Pid,L),packageAt(Pid, L).
caused tobeinformed(Pid,L) after get_recipient_info(Pid,L).

executable recipient_info(Pid,L,RecipTel,RecipEmail,DestZip,DestStreet,DropTime)
if tobeinformed(Pid,L), packageAt(Pid,L).
nonexecutable recipient_info(Pid,L,RecipTel,RecipEmail,DestZip,DestStreet,DropTime)
if packageAt(Pid, L), informed(Pid,L).
caused informed(Pid,L) after packageAt(Pid, L),
recipient_info(Pid,L,RecipTel,RecipEmail,DestZip,DestStreet,DropTime).

executable Truck(Pid) if packageAt(Pid, "centertohouse"), informed(Pid,"centertohouse").
caused packageAt(Pid, "trucktohouse") after Truck(Pid).
caused -packageAt(Pid,"centertohouse") after Truck(Pid).

executable send_trucktohouse(Pid) if packageAt(Pid,"trucktohouse").
caused tobesentfrom(Pid, "trucktohouse") after send_trucktohouse(Pid).
nonexecutable send_trucktohouse(Pid) if tobesentfrom(Pid, "trucktohouse").

executable Delivery(Pid) if packageAt(Pid, "trucktohouse"), informed(Pid,"trucktohouse"),
recipAtHome(Pid).
caused packageAt(Pid,"Dest") after Delivery(Pid).
caused -packageAt(Pid,"trucktohouse") after Delivery(Pid).

customer PickUp(Pid) if informed(Pid, "centertohouse"), not informed(Pid, "trucktohouse").
nonexecutable CustomerPickUp(Pid) if packageAt(Pid,"Dest").
caused packageAt(Pid,"Dest") after packageAt(Pid, "centertohouse"), CustomerPickUp(Pid).
caused -packageAt(Pid,"centertohouse") after packageAt(Pid, "centertohouse"), CustomerPickUp(Pid).

total recipAtHome(Pid) after CustomerPickUp(Pid).

goal: not undelivered?

The specification of background knowledge:

ev(dropoff).
ev(distcenter).
ev(delivery).
ev(truck).

location("DropOff").
location("ontheway").
location("centertohouse").
location("trucktohouse").
location("Dest").

package(pidVar, pTypeiVar, wtVar, volVar, lSenderVar, fSenderVar, lRecipVar, fRecipVar, recipTelVar, recipEmailVar, origZipVar, destZipVar, destStreetVar, destNumVar, priorityVar, costVar, dropTimeVar, delivTimeVar).
Index

≃, 46, 50
\text{M}_{\text{log}}, 28, 37, 38, 40, 41, 43, 47–53, 73
\mathcal{O}, 55
A, 27, 28, 30–32
Sem, 55

A_{PF}, 44

action

of an IMPACT agent, 56
action theory, 26, 31, 63
ADD, 89–91, 94
agent program, 55
AgentDE, 56–59
anytime algorithm, 124, 139, 141, 143, 145
approximation algorithm, 110, 120, 121, 123, 126, 139, 143, 146, 149, 150
ASA1, 86, 87, 89, 91, 92, 94–96, 98–109, 151
ASA2, 86, 87, 96–98, 100, 102, 103, 105, 106, 108, 109, 151
ASA3, 86, 87, 98–100, 102–109, 151
atom

status, 56
AUML, 65, 66

background knowledge, 33, 36
broker, 21, 156

C, 28

centralised survivability algorithm, 80, 83, 86, 87, 96, 99, 109, 110, 151, 152
CHKSWITCH, 90, 91
COD, 78, 80, 84, 101, 110, 111, 113–115
code call, 57
coding error, 37, 40, 51, 61, 62
collaboration strategy, 45–52
compatible plan, 40, 41, 52, 53
compile-time check, 57
conc, 55
conditional plan, 48, 49
conformant plan, 33, 51, 52
Contract Nets, 21, 160
CP, 48–52
C-Plans, 40, 41, 52, 53
CSA, 83, 86, 87, 96, 99, 109, 110, 151
Culprit, 41, 47, 52, 53
da, 86–89, 96–102, 104–108, 155, 159–161
DEL, 89–91, 94
deployment, 78, 84, 85, 86–88, 91, 96, 97, 100, 107, 109, 111–114, 118, 119, 126, 127, 135, 140–144, 147, 148, 150
ordering on, 112
survivability of, 110, 111, 113–118, 122, 126, 130, 132, 134, 135, 150
design error, 37, 40, 51, 52, 60, 62

180
dif, 87, 91–94, 96

difference table, 87, 88–92, 99, 108
disconnect probability, 78, 100–103, 112, 113, 127, 139, 150
ordering on, 112
disp, 29, 30, 39
distributed survivability algorithm, 83, 84, 86, 109

DLV\(^K\), 35, 36, 53, 54, 59, 62, 72, 162
DSA, 83, 86, 109, 110

dm, 29, 30, 39, 41
event dispatcher agent, 29, 30
event management agent, 29, 30

Gofish, 16, 28–31, 35, 39, 41, 56, 58–62, 72, 175
group algorithm, 135, 139, 143, 146–148

HTN, 72
HTN Planning, 31, 32, 72
method, 32
operator, 32

IMPACT, 8, 54–60, 62, 100, 162, 180

IMPACT
agent development environment, 56
agent log, 59
agent program, 55
agent roost, 58
atom
status, 56
connection, 57
message box, 58
notion of concurrency, 56
server, 57, 58

insdif, 89–91

intended collaborative behaviour, 45, 47–53
intended plan, 26, 27
I-Plans, 26, 27
irrelevant agent, 118, 127, 137
\(\mathcal{I}S\), 45, 47–53

JADE, 65

\(\mathcal{K}\), 8, 25, 30–36, 39–41, 44, 45, 48, 49, 51–54, 57, 59, 62, 72, 162, 175, 181

lower bound, 122, 124, 126, 130

message box, 58
message log file, 28, 37, 38, 40, 41, 43, 47–53, 73
message performatives, 28, 32
message repertoire, 28, 31
monitor, 26, 27, 37–41, 43, 48–50, 56–62, 73
multiagent application, 85, 108

notification agent, 29, 41
notify, 29, 30, 41

notion of concurrency, 56

\(O\), 55

\(CP\), 49, 51, 53
optimistic plan, 33, 34, 41, 45, 52, 53, 59
optimistic plan covering, 49, 51, 53

\(P\), 31, 32, 34, 38–41, 43, 46–53, 59, 73, 181

\(P\)-Plans, 32, 39, 40, 46

\(P^M\), 48
pa, 29, 30, 35, 39, 41, 42
package agent, 29, 30, 41
Petri nets, 65–67
\( \mathcal{PF} \), 44–51, 180, 182
plan covering, 48–52
plan recognition, 23, 69, 70
planning framework, 44–51, 180, 182
planning problem, 25, 26, 30–32, 34, 38, 40
    actions, 26, 31–35, 39, 42, 44
    fluents, 31–33, 35, 44, 46
    goal, 26, 28, 32, 34, 39, 44, 45
    theory, 32, 63
remdif, 89–91
replication, 14, 77, 79, 152, 153, 155
    protocol, 152, 154
Runs, 45–50
S, 45–52
\( S_{\mathcal{PF}} \), 44
safely deleted, 89, 90, 91, 95, 99
secure plan, 33, 35, 40, 41, 45, 48, 49,
    51, 52, 59
Sem, 55
sentinel, 21, 69, 155, 156
server, 54, 57, 58
SF1, 114, 115
SF1_a, 115, 117, 118, 121, 124, 135, 139,
    150
SF1_n, 116, 118, 121, 126, 128, 139, 150
SF2, 124, 139, 141
SF3, 126, 127, 139, 141–144, 147–149
SF4, 134, 137, 139, 143, 145–148, 150
SF4_g, 135, 136, 139
SF5, 136, 137–139
similarity, 46
split algorithm, 136, 139, 142, 143, 145,
    147, 148, 150
status set, 57
survivability function, 110, 112, 150
survivability of a MAS, 14, 75, 77, 79,
    80, 83–86, 100, 104, 109, 152,
    155, 159
SWITCH, 89–91, 94, 95
\( tr \), 44, 45, 51
trajectory, 35, 44, 46, 49
transition, 44, 45, 51
underestimation of survivability, 125, 133,
    137, 138, 151
upper bound, 121, 124, 139
zip, 30
zip agent, 30