Cooperative Behaviour of Autonomous Vehicles

Research Study

Authors:
M.Sc. Stephan Opfer
M.Sc. Andreas Witsch

January 2014
Contents

1 Objectives 1

2 Robots and Vehicles 2
  2.1 RoboCup Middle-Size-League 2
  2.2 Differences to Autonomous Driving 3
  2.3 Shared Characteristics 3

3 ALICA 5
  3.1 Fundamental Design Principles 5
    3.1.1 Domain Independence 5
    3.1.2 Autonomy 6
    3.1.3 Locality 6
  3.2 Intuitive Overview 6
  3.3 ALICA Language Definition 9
    3.3.1 Syntax 9
    3.3.2 Semantics 10
  3.4 General Alica 12
    3.4.1 Behaviour Parameters and Plan Variables 13
    3.4.2 Agent Variables 14

4 Evaluating ALICA in Autonomous Driving Scenarios 16
  4.1 The Simulation Environment 16
  4.2 Scenario I: Merging from Two to One Lane 19
  4.3 Scenario II: Give Way to an Emergency Vehicle 22
  4.4 General Scenario Results 25
  4.5 Quality of Coordination under Unreliable Communication 25
  4.6 Backwards Compatibility 27
  4.7 Conclusions 28

5 Project Proposal 29
  5.1 Open Research Questions for Autonomous Driving with ALICA 29
  5.2 Work Packages 29
    5.2.1 WP1 Porting the ALICA Implementation 29
    5.2.2 WP2 Development of an Autonomous Driving ALICA Program 30
    5.2.3 WP3 Development of a World Model 31
    5.2.4 WP4 A Smart Communication Middleware and Team Observation Algorithm 31
    5.2.5 WP5 Simulator Evaluation 32
  5.3 Cost Estimation 32
1 Objectives

The objective of this study is to elaborate the suitability and necessary adaptations of the formal language ALICA for the autonomous car driving domain. Based on these results we will derive a project recommendation to resolve these adaptations.

In particular, the study will:

1. describe the formal language ALICA and give an example for an autonomous car driving program.

2. elaborate similarities and differences between autonomous car driving and multi-robot systems to analyse the applicability of ALICA from a theoretical point of view.

3. evaluate ALICA’s coordination mechanisms for the two scenarios Merge Traffic and Emergency Vehicle with respect to CPU runtime, memory consumption, communication bandwidth, and required informations. These evaluations are based on the ALICA implementation used in the RoboCup Middle-Size League.

4. provide a conceptual elaboration how to deal with agents that are not equipped with ALICA.

5. propose a concrete research project that leads towards ALICA controlled autonomous cars.

The study is structured as follows. Chapter 2 shows differences between autonomous car driving and robotic football. Afterwards, Chapter 3 gives first an intuitive and then a formal description of ALICA. In Chapter 4 we will apply the ALICA approach to two scenarios in a simulated environment and analyse its properties. Finally, Chapter 5 describes a possible future project to enhance ALICA for the use in real autonomous cars.
2 Robots and Vehicles

In the first chapter of the study, we describe the RoboCup Middle-Size-League (see Section 2.1), as it is one of the typical application domains for ALICA. Afterwards, in Section 2.2 we point out what are the main differences between the Middle-Size-League and the domain of autonomous driving. Furthermore, Section 2.3 focusses on the shared characteristics of both domains and points out, why ALICA is suitable for the modeling and control of autonomous driving in general.

2.1 RoboCup Middle-Size-League

The application domain, which we had in mind when we designed ALICA was the Middle Size League (MSL) of the RoboCup research initiative\(^1\). In the MSL two robot teams, each consisting of five football robots, play against each other. A match is divided into two halves of 15 minutes each. The football field has a size of \(12 \times 18\) m and an original Fifa football is used. The football robots are not allowed to be larger than \(52 \times 52\) cm and taller than \(80\) cm. Only the goal keeper may extend its size by \(10\) cm for one second, in order to catch the ball. Furthermore, the robots of a team are allowed to communicate with each other via WiFi. Despite of these limitations, the MSL rules \(^1\) are only a slightly changed version of the original football rules \(^2\). Figure 2.1 shows a robot of the team Carpe Noctem Cassel developed in our institute.

\[\text{www.robocup.org}\]

Robot football is a very dynamic game. The average MSL robots are able to drive and simultaneously dribble the ball at a velocity of up to \(5\) m/s. The kick distance reaches up to \(16\) m, whereby the ball is accelerated to velocities above \(12\) m/s. During the last years, the robots dramatically improved their general cooperation and pass playing, making strategies like man-to-man marking mandatory for every successful team. From an agent researchers’ point of view \(^1\)\(^7\), the football robots match several well known agent properties:

\[\text{Figure 2.1: A Carpe Noctem Football Robot}\]
**Rational** agents always choose the optimal actions, according to their knowledge and beliefs, to achieve a certain goal.

**Autonomous** agents are always acting independently, sovereign, and uncontrolled.

**Proactive** agents act self-initiated to achieve a certain goal.

**Reactive** agents handle changes in their environment by adapting their behaviour accordingly.

**Communicative** agents exchange their knowledge and beliefs with each other in order to cooperate.

### 2.2 Differences to Autonomous Driving

Considering the necessity of cooperation between agents and the properties of the environment, we encountered some differences between autonomous driving and robotic football during the research study. The autonomous driving domain has *strict traffic rules*. If an agent complies with the traffic rules, the intentions and actions of this agent are predictable by just observing the current situation. The current situation can be described by the agent’s driving lane, its direction indicators, its breaking lights, its position, its velocity, and the traffic lights or signs at the side of the road. This means that if another agent has perfect sensors for reliably and accurately perceiving these values, communicating about intentions or plans is not required in the current situation. There are some exceptions to this rule, like a deadlock situation at crossings without traffic lights and give way signs or lane narrowings from three to two lanes. However, coordinated autonomous driving is beneficial because of a higher car throughput in high traffic situations, while also increasing the level of safety by avoiding user errors. In robotic football the individual agents often have several rational options for acting without breaking the rules. This makes active communication about the agent’s current intentions more necessary in robotic football than in autonomous driving.

Another difference is that in robotic football the available *background knowledge* about the environment is more precise than in the autonomous driving domain. The exact position and size of every field marking, the maximum size of each team, and the colour and the approximate form of each object on the field is known in advance. This makes sensor data processing easier in robotic football, compared to autonomous driving, and makes it necessary to qualify the statement about the predictability of the autonomous driving domain. Furthermore, considering the background knowledge about the environment in autonomous driving, the structure of the environment is known only approximately in advance.

Especially the *formation of teams* is different in autonomous driving. While in robotic football all possible team members are known in advance and can communicate with each other, in autonomous driving neither of these two properties hold. Moreover, an agent can be located in between two agents, which are not aware of each other, because of range limitations of the communication devices. As a result, this agent in between has to decide how it cooperates with these two agents, e.g. decide whether it should make room for the agent in front of it or behind it, while both are expecting cooperation, because of their unawareness of the third agent.

### 2.3 Shared Characteristics

A popular model that characterises the processing cycle of autonomous agents and many control loops is the *MAPE cycle* [7] as depicted in Figure 2.2. Each letter of *MAPE* stands for one
activity in the control loop of an autonomous agent: monitor, analyse, plan, and execute. The agents of both domains, described in the last two sections, interact with their environments in the sense of the MAPE cycle. They perceive their environment through sensors and process the acquired data in order to gain more abstract information like positions, velocities, or general states of other objects in the environment. Based on the results of this analysis they plan and decide the next steps and realise them through their actuators. As their actuators execute the given actions, the agents change their environments according to their intentions.

![Agent Environment Relation (MAPE Cycle)](image)

The general model of an agent, induced by the MAPE cycle, matches the assumptions about agents made by our multi-agent coordination framework “ALICA – A Language for Interactive Cooperative Agents”. Therefore, both kind of agents can be handled by ALICA. The details about ALICA are given in Chapter 3.

Another property, which both domains have in common, are their potentially imprecise and noisy sensor values. This property has the effect, that ALICA is designed for robustness with noisy and imprecise sensor values in mind. Even if two agents have completely different sensor values about their environment, ALICA is still able to make these two agents cooperate with each other to a certain degree. Furthermore, ALICA handles the unreliable communication, which also both domains have in common, due to its fully decentralized approach. In ALICA no central coordinating or managing agent exists. All agents are completely independent of each other. The only exception for this rule occurs, if agents have conflicting plans. In this case, the agent with the lower unique ALICA identifier commands which strategy will be executed.

Finally, there is another commonality of both domains, which stems from different reasons in the domains. It is the necessity for a dynamic and efficient task-allocation algorithm. In robotic football the opponent team makes it necessary to adapt to unpredictable situations. An opponent can shoot the ball with unpredictable power, which creates also unpredictable rebounces. The velocity of the ball makes it necessary to immediately reallocate tasks and thereby adapt to ever changing situations. There is a similar situation in autonomous driving, due to the ever changing and unpredictable environment. Construction sites make it necessary to merge from two to one lanes on a high way, or as another example, suddenly changing driving conditions forces to cancel an overtaking manoeuvre. In order to adapt to these unforeseen events, an efficient reallocation of the tasks in the team of agents is necessary.
ALICA is a comprehensive solution for describing the behaviour of multi-agent teams [11]. Its unique combination of paradigms constitutes a novel way to model the cooperative behaviour of multi-robot systems. The approach encompasses description of capabilities, role allocation, task descriptions similar to hierarchical state machines, task allocation based on utility functions, explicit coordination through synchronised transitions, implicit coordination in a broadcast-and-compute fashion, and conflict resolution by switching the decision making protocol on the fly. Finally, non-linear constraint satisfaction problems are integrated in order to allow reasoning over domain-specific positions or configurations of joints. We see this combination of paradigms as the biggest strength of ALICA.

The implementation has been successfully employed by the RoboCup MSL team Carpe Noctem Cassel since 2009 and is currently being used in the project IMPERA\(^1\), which researches the coordination of extraterrestrial exploration missions. The C\# source code is available under a BSD-based open source license\(^2\).

On the theoretical level, we provide new insights into hierarchical task allocation leading to two different allocation schemes with different requirements on the task structure. Furthermore, an anytime constraint optimisation solver is included, which extends state-of-the-art approaches for non-linear continuous satisfaction problems. Most notable are the solver’s abilities to track solutions over time, coordinate solutions within the team, and distributively solve problems while retaining reactivity of the individual robot.

3.1 Fundamental Design Principles

The design of the ALICA semantics is guided by the following principles: domain independence, locality, and autonomy. They are respectively meant to guarantee portability, scalability, and robustness against adverse domain features such as unreliable communication.

3.1.1 Domain Independence

Although robotic scenarios are in the original focus of ALICA, it is designed domain independent, such that the results can be used in other domains like autonomous car driving. Therefore, ALICA makes as little assumptions as possible about the way of representing an agent’s environment. It is possible to use a classical first-order logic, a modal logic, or even a hybrid approach such as a probabilistic logic with ALICA. A specific ALICA program references the world representation, which must be defined as part of the process. In other words, an ALICA process includes a domain specific world model that can be queried by the ALICA program for information about the agent’s environment.

\(^1\)http://www.uni-kassel.de/eecs/fachgebiete/vs/research/impera.html
\(^2\)http://ros.org/wiki/cn-alica-ros-pkg
3.1.2 Autonomy

The ability to cope with unreliable communication is a crucial requirement in robotic domains. ALICA tackles this problem by performing calculations redundantly, that is, decisions regarding the team are made by all agents individually and autonomously. Inconsistent decisions can be subsequently detected and corrected once corresponding information are available, e.g. through communication or action recognition. This principle allows ALICA to operate under highly degraded network conditions, as shown in [14]. Due to decentralized decision making, message delivery times can be almost unbounded. There is only one exception to this rule, which is related to the recognition of an incapacitated agent. Our implementation assumes that if no message was received from an agent for a certain period of time, this agent is no longer able to function properly. The time period used depends on the expected network quality, the degree of dynamics in the domain, and the likelihood of an agent breaking down. This exception will need some adaptation for domains like autonomous car driving, where no broadcast infrastructure is available that connects all agents with each other.

3.1.3 Locality

The global state of a team of agents executing an ALICA program is represented by the combined states of all agents involved. In order to cope with the potential complexity of the problem tackled by the team, ALICA exploits the hierarchical structure of the program. Solutions for encountered problems are described by plans, which split the team using tasks, and employ sub-plans to solve potential sub-problems.

Maintaining this tree structure and performing all necessary calculations to make all decisions involved can very well overwhelm the computational power of a central coordinator. Hence, ALICA adopts a locality principle. Each agent keeps track of the plans it is involved in and only participates in decisions regarding these plans. Parts of the global state of the team in which an agent does not participate are ignored by this agent.

If the problem can be decomposed into sub-problems, this principle significantly reduces the computation costs in large teams. Thereby the locality principle fosters scalability with respect to the number of agents participating. This is also a principle that comes very naturally to human beings. For instance, a driver is not interested in the precise position of all other cars on the street. He is just interested in the cars near to him. This is equivalent to the ALICA notion of knowing which agents are allocated to which tasks of the plan hierarchy in an ALICA program. The same principle is also applied by humans in less dynamic scenarios. If your department participates in a large collaborative project, you will not concern yourself with every detail of your partners’ doings. Instead, it suffices to know an abstract progress status.

3.2 Intuitive Overview

ALICA allows robots to interact in plans that are made up of different states that are connected with transitions (called state machine). Each transition reflects a condition, which has to be true in order to apply a state-switch of the agents. Each state may contain further plans, behaviors, or plantypes. In the context of ALICA a behavior refers to an atomic behavior or manipulation task of an agent e.g. “Emergency Break”. Thus, plans are designed to model more complex tasks like “DriveHome”. As states of plans can again include plans, an ALICA program is usually a hierarchical structure. Plantypes are an abstraction of a task e.g. if two different plans are
available to achieve the same goal in a different way. During runtime the active plan of a plantype is selected based on a utility function that is part of each plan.

A plan can be annotated with pre-, runtime-, and postconditions. The precondition needs to be true before a plan can be executed but not necessarily while it is active. In contrast, runtime conditions need to be true continuously while a plan is active. Postconditions become true after a plan was successfully executed and are mainly designed to allow the implementation of planning algorithms.

In order to model interaction between different agent tasks, a plan can be composed by more than one state machine. In this case, each state machine represents an agent task. Note that a task can be executed by more than one agent and is therefore annotated by a minimum and maximum cardinality. The optimal mapping from agents to tasks is determined by the plan’s utility function.

ALICA plans, states, and behaviors can be parameterized with variables. These variables are currently determined by a constrained satisfaction problem.

ALICA Plans can be modeled using a tool called Carpe Noctem Plandesigner, as shown in Figure 3.1. The Plandesigner can store ALICA programs on the file system in XML format and generates C# code for all conditions and constraint definitions. During runtime our ALICA implementation can load all source code parts together with the XML definition of an ALICA program, interpret it, and provide runtime support e.g. for synchronization, conflict resolution, or distributed constraint solving. Note that during runtime each agent interprets the ALICA program by a tree structure including the currently active states.

An overview over the ALICA implementation is given in Figure 3.2. It is composed of the following components:
**Castor** is a shared library that supports easy access to configuration files, which are used to store configurations persistently.

**Plan Repository** includes a model parser, that loads XML descriptions of ALICA programs and connects all conditions, utilities, and constraints to their C# implementation.

**Constraint Solver** is an (incomplete) distributed constraint solver that determines plan variable values with gradient ascent and a automatic differentiation library [12, 16].

**Plan Base** is the main ALICA interpreter that applies all ALICA rules e.g. task allocation.

**Role Allocation** determines the agent’s role based on its capabilities.

**Behaviours** is the set of all behaviours, which are all located in separate threads.

**World Model** includes all necessary information about the agent’s environment and a model of shared data for implicit synchronisation. The world model is accessible everywhere in the framework by the Singleton pattern.

**Authority Handler** manages conflict resolution for task allocation [13].

**Team Observer** tracks the current team of agents that cooperate with each other.

![Figure 3.2: Component Model of an ALICA Agent](image)

Figure 3.2: Component Model of an ALICA Agent
3.3 ALICA Language Definition

The following sections will give a more formal language definition of ALICA. Therefore we will first state the language syntax in Section 3.3.1. Afterwards, in Section 3.3.2 we will define the most important rules of ALICA’s semantics. Finally, in Section 3.4 we present the generalised form of ALICA, which lifts ALICA to a first-order logic approach.

3.3.1 Syntax

In this section, we formalise the elements of the ALICA language. The central notion within ALICA are plans. A plan describes specific activities, a team of agents has to execute in order to achieve a certain goal. Plans are modelled similar to Petri-nets, containing states and transitions between the states. Each transition is guarded by a condition, which must be believed to hold by an agent in order to progress along the transition.

The set of all plans in an ALICA program is denoted by $\mathcal{P}$. $\mathcal{Z}$ denotes all states. The belief base of each agent is described in a logic with language $\mathcal{L}$, hence all conditions are elements of $\mathcal{L}$. We do not enforce a specific logic, here however, we assume a first-order language e.g. a high-level programming language like C++, C#, or Java. Each transition $\tau$ is an element of $\mathcal{Z} \times \mathcal{Z} \times \mathcal{L}$. A transition may belong to a synchronisation $\lambda$, meaning it can only be progressed along by an agent if it believes that for all transitions belonging to $\lambda$, an agent will do so, and that there is mutual belief among all involved agents about this fact.

ALICA abstracts from agents in two ways, by tasks and roles. A role is assigned to an agent based on its capabilities. This assignment is treated to be relatively static, i.e., it holds until the team composition changes. This is the case if an agent joins the team, leaves the team, or its capabilities change, for instance, due to being incapacitated.

Tasks on the other hand abstract from specific paths within plans. As such, they denote similar activities in different plans. If a group of agents has to execute a plan, the corresponding tasks are allocated to the available agents based on the situation at hand and the roles the agents are assigned. This two-layered abstraction allows for programs to be specified independently of the team composition that will exist at execution time. A team composition can be described solely by the roles each agent is assigned, while plans can be described without referring to roles. Each role has a numeric preference towards tasks, which allows roles to be mapped onto tasks dynamically during runtime.

Since plans describe activities for multiple agents, they have multiple initial states, each of which is annotated by a task. Hence, a task abstracts from specific activities within plans, and multiple plans can be annotated with the same tasks. For instance, a plan to build a tower and a plan to build a bridge might both contain the task of moving building blocks into position. Certain tasks might require multiple agents, each task-plan pair $(p, \tau)$ is annotated by a cardinality, i.e., by an interval over $\mathbb{N}_0 \cup \{\infty\}$, denoting how many agents must at least and may at most be allocated to $\tau$ in $p$, in order to execute $p$.

The applicability of a plan in a certain situation is measured in two ways. Firstly, each plan $p$ is annotated by a precondition $\text{Pre}(p)$, which has to hold when the agents start to execute it, and a runtime condition $\text{Run}(p)$, which has to hold throughout its execution. Secondly, each plan $p$ is annotated by a utility function $\mathcal{U}_p$, which is used to evaluate the plan together with a potential allocation of agents to tasks within $p$ wrt. a situation. A utility function maps believed or potential situations onto the real numbers. Both, conditions and utility functions, solely refer to the belief base of an agent, which contains believed allocations.

Plans can be grouped together in plan types, providing the agents with sets of alternative
ways of solving a particular problem. Choosing a specific plan from such a plan type is done autonomously by the agents in question. This selection is guided by the utility functions and conditions of the plans involved.

Each state contains an arbitrary number of plan types. For each plan type, the agents involved have to choose a plan and execute it, i.e., multiple plans, one from each plan type, are executed in parallel. Additionally, each state contains an arbitrary number of behaviours. Behaviours are atomic single-agent action programs, e.g. Park or DriveRoute. The set of all behaviours in an ALICA program is denoted by $B$. Each behaviour within a state is meant to be executed by all agents inhabiting the corresponding state.

The relationship between states, plans, and plan types spans a hierarchical structure, called the plan tree of an ALICA program.

### 3.3.2 Semantics

The semantics of ALICA is given by a transitional rule system, which describes how the internal states of agents change over time. These internal states are referred to as agent configurations. Additionally, a set of axioms, $\Sigma_B$ constrains these configurations.

**Definition 3.3.1 (Agent Configuration).** An agent configuration is a tuple $(B, \Upsilon, E, \theta, R)$, where $B$ is the agent’s belief base, $\Upsilon$ the agent’s plan base, $E \subseteq B \times Z$ the set of behaviours $b$ the agent executes together with the state in which $b$ occurs, $\theta$ a substitution, and $R$ is a set of roles assigned to the agent according to its capabilities.

The plan base contains triples of the form $(p, \tau, z)$, indicating that the agent is committed to task $\tau$ in plan $p$ and currently inhabits state $z$ within $p$. For each plan $p$ there is at most one triple $(p, \tau, z)$ in any plan base. Each of these triples is reflected by a literal in the belief base, $\text{In}(a, p, \tau, z)$, representing the belief that $(p, \tau, z)$ is an element of agent $a$’s plan base. In the same way, $\text{HasRole}(a, r)$ captures the belief that role $r$ is assigned to $a$.

An ALICA domain signature is a tuple, $(\mathcal{R}, \mathcal{P}, \mathcal{T}, \mathcal{Z}, \mathcal{W}, \Lambda, \text{PlanType}, p_0, z_0, \tau_0)$ where $\mathcal{R}$ is a set of roles, $\mathcal{P}$ a set of plans, $\mathcal{T}$ a set of tasks, $\mathcal{Z}$ a set of states, $\mathcal{W}$ a set of transitions, $\Lambda$ a set of synchronisations, and $\text{PlanType}$ a set of plan types. $p_0$ is the top-level plan, with the solitary state $z_0$ and the single task $\tau_0$. The structure of an ALICA program is formed by functions and relations between these elements.

- **States:** $\mathcal{P} \mapsto 2^Z$, $\text{States}(p)$ denotes the states within a plan.
- **Tasks:** $\mathcal{P} \mapsto 2^T$, $\text{Tasks}(p)$ denotes the tasks of a plan.
- **$\xi$:** $\mathcal{P} \times \mathcal{T} \mapsto \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\})$ is a partial function, associating cardinalities with tasks in plans. Intuitively, $\xi(p, \tau) = (n_1, n_2)$ denotes that in order to execute $p$, at least $n_1$ and at most $n_2$ agents have to commit to task $\tau$.
- **Init:** $\mathcal{P} \times \mathcal{T} \mapsto \mathcal{Z}$, $\text{Init}(p, \tau)$ denotes the initial state of task $\tau$ in plan $p$.
- **Pre:** $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}(\text{Pred, Func})$, $\text{Pre}(p)$ denotes the precondition of plan or behaviour $p$.
- **Run:** $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}(\text{Pred, Func})$, $\text{Run}(p)$ denotes the runtime condition of plan or behaviour $p$.
- **PlanTypes:** $\mathcal{Z} \mapsto 2^{\mathcal{L}(\text{Pred, Func}) \mapsto \mathcal{P}}$, $\text{PlanTypes}(z)$ denotes the set of plan types to be executed on state $z$. We write $\text{PlanTypes}(z)(\mathcal{F})$ to denote the set of plans that are identified by the set of plan types given a set of formulae $\mathcal{F}$. 

10
• Behaviours: $\mathcal{Z} \rightarrow 2^\mathcal{B}$, Behaviours($z$) denotes the set of behaviours to be executed in state $z$.

• Success: $\mathcal{P} \rightarrow 2^\mathcal{Z}$, Success($p$) denotes the set of terminal states of plan $p$, which indicate successful execution of the plan or one of its tasks.

• Fail: $\mathcal{P} \rightarrow 2^\mathcal{Z}$, Fail($p$) denotes the set of terminal states of plan $p$, which indicate unsuccessful execution of the plan or one of its tasks.

• Post: $\mathcal{Z} \rightarrow \mathcal{L}(\text{Pred, Func})$, Post($z$) is a partial function, that maps terminal states of a plan to postconditions.

• Each plan $p \in \mathcal{P}$ has a utility function, $U_p: 2^{\mathcal{L}(\text{Pred, Func})} \rightarrow \mathbb{R}$, associated with it. Intuitively, this utility evaluates the applicability of a plan together with an agent allocation with respect to a given situation. As aforementioned, a plan describes a specific activity of one or more agents. An allocation for a plan is a subset of agents executing the tasks of the plan.

The relationship between transitions and states form a directed graph for each plan with transitions as edges and states as nodes. Plans, plan types and states form a tree-like structure, called plantree. The root node of this tree is $p_0$, the top-level plan. Over these structures we denote transitive closures over the functions above with an asterisk, for instance, Plans*($z$) denotes all plans which can occur below state $z$ in the plan tree.

Plans and behaviours can have parameters, which are precisely the free variables in the pre-, runtime- and postconditions. The parameters of a plan or behaviour $p$ are denoted by $\text{vars}(p)$.

During the execution of an ALICA program the state of each agent is given by its configuration. Information an agent has about the state of the team is reflected in its belief base by the following predicates:

• $\text{In}(a, p, \tau, z)$, defined to hold iff $(p, \tau, z) \in \text{PBase}(a)$. This allows an agent to reason about its beliefs about the internal states of other agents. For instance, Bel$_a \text{In}(b, p, \tau, z)$ denotes that $a$ believes $b$ to be committed to $\tau$ in $p$ and that $b$ is currently in state $z$.

• $\text{HasRole}(a, r)$, stating that agent $a$ has taken on role $r$,

• $\text{Handle}_f(b, z)$, which is meant to hold if an agent should handle a failure of behaviour $b$ in state $z$,

• $\text{Handle}_f(p)$, which is meant to hold if an agent should handle the failure of plan $p$,

• $\text{Failed}(p, i)$, indicating that plan $p$ failed $i$-times,

• $\text{Failed}(b, z, i)$, indicating that behaviour $b$ failed $i$-times in state $z$,

• $\text{Succeeded}(a, p, \tau)$, true iff agent $a$ successfully completed task $\tau$ in plan $p$,

• $\text{Alloc}(z)$, true iff an allocation of tasks to agents for state $z$ is deemed necessary,

• $\text{Success}(b, z)$ and $\text{Fail}(b, z)$, indicating a success or failure signal from behaviour $b$, which is executed in state $z$, respectively.
Note that since the signature is finite, these predicates can still be compiled into propositional logic.

The beliefs of an agent are again subject to a set of axioms, $\Sigma_b$, which, for each agent $a$ in $\mathcal{A}$, contains the following:

- If failure handling for a behaviour is needed, it is relevant:
  \[(Bel_a Handle_f(b,z)) \rightarrow (\exists p, \tau) \text{In}(a,p,\tau,z)\]

- If failure handling for a plan is needed, it is relevant:
  \[
  (Bel_a Handle_f(p) \vee Failed(p,i)) \rightarrow 
  p = p_0 \vee (\exists p',\tau',z) \text{In}(a,p',\tau,z) 
  \land p \in \text{Plans}(z)
  \]

- In the same way, if failure handling for a behaviour is needed, it is relevant:
  \[(Bel_a Handle_f(b) \vee Failed(b,z,i)) \rightarrow (\exists p, \tau) \text{In}(a,p,\tau,z)\]

- An agent’s success in a task is only relevant as long as there is another agent still within the state that contains the corresponding plan:
  \[\text{Succeeded}(a,p,\tau) \rightarrow (\exists z)p \in \text{Plans}(z) \land (\exists a',\tau',p'')\text{In}(a',p'',\tau',z)\]

- Task allocation is only needed for a state inhabited by the agent:
  \[(Bel_a Alloc(z)) \rightarrow (\exists p, \tau) \text{In}(a,p,\tau,z)\]

Additionally, $\Sigma_b$ contains unique name axioms over all ALICA domain elements. These axioms are used to keep the belief base of an agent consistent during its runtime. For this we assume an inertial basic belief update operator (after [6]). $\Sigma_{syn}$ and $\Sigma_b$ together with a set of domain specific axioms form the complete set of ALICA axioms for a given domain, $\Sigma_B$. $\Sigma_B$ is assumed to be common knowledge in $\mathcal{A}$.

### 3.4 General Alica

The ALICA described so far offers a variety of modelling options to tackle multi-agent scenarios. However, the fact that it is limited to propositional semantics (therefore coined pALICA [15]) can cause an explosion of specific language elements, most strikingly, behaviours. In pALICA, every behaviour is static. Thus, each behaviour essentially performs the same action, regardless in what situation it is called. While this limitation is somewhat lifted by the fact that behaviours are black boxes, and thus can be Turing complete programs interacting with the belief base, this is not represented within ALICA and therefore cannot be coordinated or reasoned about by means internal to ALICA. Therefore, we developed a generalised version of ALICA, where plans and behaviours are parametrisable, and where the reasoning in parameter spaces is only bounded by memory.
3.4.1 Behaviour Parameters and Plan Variables

Each behaviour $b$ has a potentially empty list of variables $\vec{x}$, its parameters. We write $b(\vec{x})$ to indicate behaviour $b$ with its parameters $\vec{x}$. By allowing behaviours to feature parameters, the problem of common execution patterns can be dealt with. Each occurrence of a behaviour in the program can thereby have a different set of parameters. This allows the blocks world plan to be described using only two behaviours, $\textit{Pickup}(x)$ and $\textit{Put}(x, y)$. However, even though infinitely many different behaviour instances\(^3\) can now be expressed using a finite amount of behaviours, and thereby a finite amount of code, this is not sufficient to deal with parameters whose values are unknown or only partially known before runtime.

Consider the following simple example: A robot is tasked with finding a book in an office, and identifying it by moving towards it. Before runtime, the specific book the robot will find is unknown, as there are various different books in the office. The concrete book the robot should move towards only becomes apparent after the robot has searched for some time and the corresponding beliefs are inserted into its belief base. Figure 3.3 shows how a corresponding plan could look like. The robot executes the behaviour $\textit{LookAround}$ until a book is represented in the belief base, and that every object is passed as parameter to the behaviour $\textit{Goto}(x)$.

However, the plan $\textit{FindBook}$ is not expressible in pALICA, even when allowing behaviour to have parameters. What is needed additionally is a link between free variables in conditions and behaviour parameters. This allows the reasoning task, in this case the identification of a book to be defined within the plan.

In the following, we extend plans, states and plantypes, i.e., the elements which structure an ALICA program, with the appropriate notions to allow variables to occur in plans and relate them to each other. Thus, each plan $p$ has a potentially empty list of unique variables $\vec{x}$, written $p(\vec{x})$. Equally, each plantype $P$ has a potentially empty list of variables $\vec{x}$, written $P(\vec{x})$.

Thus, each plan and plantype has a set of unique variables available, which can be referred to by conditions. However, different plans and plantypes are associated with pairwise disjoint sets of variables. In order to relate variables of different plans to each other, we introduce the notion of bindings.

Each state $s$ within each plan $p(\vec{x})$ defines a possibly empty substitution $\theta(s)$, called a binding, such that $\theta(s) = \{ y_1 \mapsto x_1, \ldots, y_n \mapsto x_n \}$, all $x_i$ are terms with variables only among the variables of the corresponding plan, $(\forall x_i) \text{vars}(x_i) \subseteq \vec{x}$, and all $y_i$ are variables of behaviours or plantypes in $s$, $(\forall y_i)y_i \in \bigcup_{b(\vec{y}) \in \text{Behaviours}(s)} \bigcup_{P(\vec{y}) \in \text{PlanTypes}(s)} \vec{y}$.

Equally, each plantype $P(\vec{x})$ defines a possibly empty substitution $\theta(P)$, called a binding, such that $\theta(P) = \{ y_1 \mapsto x_1, \ldots, y_n \mapsto x_n \}$, all $x_i$ are terms with variables only among the variables of $P(\vec{x})$, $(\forall x_i) \text{vars}(x_i) \subseteq \vec{x}$, and all $y_i$ are variables of plans in $P$, $(\forall y_i)y_i \in \bigcup_{p(\vec{y}) \in P(\vec{x})} \vec{y}$.

Thus, each state and each plantype declares a local parametrisation of the plans, plantypes,

\(^3\)or, more precisely, up to a memory-bounded number of instances.
and behaviours it contains, allowing to relate variables with each other, but also allowing for static substitutions. Note that bindings do not necessarily bind all variables of sub-plans or behaviours, it is entirely possible leave sub-variables unbound. Similarly, variables of the parent plan can occur arbitrarily often in a binding.

Variables of plans can now serve as means to express dynamic bindings through conditions within plans. For instance, a condition attached to a transition can refer to a book just found and bind it to a plan variable, which in turn is bound to a behaviour variable by a state. This can easily be achieved by allowing conditions to have free variables among the variables of their plan. However, in some scenarios, one might not want to identify a unique ground value for each variable. For instance in the standard situation scenario in above, calculating precise positions is cumbersome, instead one might only want to list a set of properties such positions should fulfil. Furthermore, as bindings can be used to pass variables down the plan hierarchy, such that they are eventually used as behaviour parameters, one might only want to constrain the values for each variable to values suitable to the problem tackled at each level, and leave the details to sub-plans. In this particular scenario, a plan at a high level can describe the general problem constraints, such as the set of game rules the robots must observe. It can then make use of strategy specific sub-plans, bundled in a plantype. Each of these sub-plans can further constrain the variables according to a specific strategy, such as an aggressive, yet risky strategy for free-kicks in the opponent’s half, or a defensive strategy in case the team is in the lead.

This idea leads us to a constraint programming approach, where each individual plan does not ground parameters passed to behaviours, but states requirements, or constraints, ground solutions must fulfil. This way, the resulting behaviour of the multi-agent team can be expressed in a declarative manner, decoupled from implementation details of the underlying solver. Furthermore, a generative algorithm does not need to concern itself with the question whether or not generated conditions ground all variables passed to behaviours in all circumstances.

Languages such as FLUX or GOLOG [8] enforce that each action is fully grounded when posted for execution. This is vital, as there is no further reasoning happening before a corresponding action is executed, and the result of executing a non-ground action such as $\text{Goto}(x)$ is not defined. In contrast, we allow for non-ground constrained parameters, which can be grounded to appropriate values by issuing a query to a constraint solver on-demand, thus solving the problem of non-ground actions by an intermediate reasoning step. In ALICA we use constraint formulas to perform this reasoning step.

### 3.4.2 Agent Variables

While the notion of plan variables potentially allows for expressive descriptions of cooperation using first-order terms, it turns out that in some instances, this is not sufficient to capture the intended team behaviour in a concise manner.

Consider the popular Foraging Scenario [5], used for instance by Campbell and Wu[3]. Here a team, or a swarm of agents is tasked with searching and retrieving certain items, such as food or resources, while at the same time protecting their base or nest. The scenario has mainly been used to investigate role or task allocation techniques, which dynamically decide for each individual agent whether it should forage or protect. In this case, the precise number of agents assigned to a task is unknown until runtime, only an upper bound is given, even the total number of agents may be unknown until runtime. Further, the number of agents foraging or protecting can change dynamically.

Given a suitable role- and task allocation approach, the individual behaviours need to be specified. So far, expressing these behaviours using constraints in ALICA requires plan variables.
Each plan variable would represent an agent’s target position given, that it is tasked with foraging or protecting, respectively, thus twice as many variables are needed as agents can possibly participate. The resulting constraint problem would be overly complicated in all situation, as each agent only takes on one task at a time, so at most half the variables would be actually used. Furthermore, in a realistic scenario, the number of agents actually active would almost always be lower than the number of possibly active agents.

This way of formulating the constraints makes use of a domain-specific predicate within $\mathcal{L}(\text{Pred, Func}, \text{Pos})$, and implicitly assumes it is functional in the second parameter. Such predicates are similar to functional fluents in the Situation Calculus [10], however, they do not express the state of the world or knowledge about it, but represent intended values, or rather allow constraints to assert intentions or obligations. Thereby, such constraints can be seen as a declarative extension of plans, which otherwise formulate intentions in a procedural way.

In order to support the specification of constraints which abstract from the actual agents that execute a plan or a task at a specific point in time, a set of corresponding axioms is needed to fill in the gap between constraints and beliefs:

**Definition 3.4.1.** Let $F$ be a finite set of binary predicate symbols, and for every $P \in F$, let $\Sigma_A$ contain

$$((\forall a \in A)((\exists x)P(a,x)) \land ((\forall x,y)P(a,x) \land P(a,y) \rightarrow x = y))$$

Thereby, all elements of $F$ are functional in their second argument, given the first is an agent. We extend the belief axioms of pALICA, $\Sigma_b$ to include $\Sigma_A$. Predicates constrained by beliefs in this form are called functional agent fluents.

Thus, each formula in $\Sigma_A$ enforces the existence of a single value $x$ per agent $a$ such that $P(a,x)$ holds. In other words, $P(a,x)$ attaches a variable to agent $a$ under the name $P$. A domain-specific extension to $\Sigma_A$ in $\Sigma_{\text{dom}}$ can enable similar constructs for other domain-specific elements agents are able to modify and reason about. In the next section, we will clarify how constraints of this form can be incorporated into plans.
4 Evaluating ALICA in Autonomous Driving Scenarios

Depending on the domain, ALICA has different resource requirements and behaviour. In order to determine these parameters in case of the autonomous driving domain, ALICA was evaluated in two simulated scenarios. In this chapter the general simulation setup, the two scenarios, their implementation, and the test results are described in the following sections. The evaluated parameters are the bandwidth necessary for communication between ALICA agents, the memory and CPU utilisation of ALICA agents and some more general properties of ALICA, especially the quality of cooperation between ALICA agents under unreliable communication and in presence of NON-ALICA agents.

4.1 The Simulation Environment

In order to make results comparable to former results of experiments of VW, it is required to use the simulator SUMO – Simulation of Urban MObility [4]. SUMO is an open source, highly portable, microscopic and multi-modal traffic simulator for simulating continuous road traffic in large road networks. Microscopic means, that it allows to specify individual routes and behaviours for each vehicle. The simulator is mainly developed by the Institute of Transportation Systems at the German Aerospace Center. Its main purpose is the assistance in optimising urban traffic and therefore it is self-contained by design. Thus, a typical experiment would require to specify all parameters offline and run the experiment without any interaction. Nevertheless, a plug-in called Traffic Control Interface (TraCI) provides a limited network interface in order to interact with a running SUMO simulation. TraCI gives access to values of simulated objects and makes it possible to influence their behaviour.

Unfortunately, TraCI suffers from certain limitations concerning its interface and its interaction with SUMO. TraCI accepts only one TCP connection to one client. As ALICA is a completely decentralized approach, all agents control themselves independently and interact with their environment, in this case the simulation, asynchronously. In order to let all ALICA agents, controlling a simulated car, asynchronously interact with the simulation through a single TCP connection a proxy process is necessary. The developed proxy process, called SUMO-Proxy, manages the TCP connection to TraCI and distributes simulation data to all ALICA agents and forwards their control messages to TraCI.

During our experiments it turned out that the TraCI interface needs several milliseconds to process simple remote commands. This is slow compared to a common control cycle of an ALICA agent, which is usually triggered with a frequency of 30 Hz, i.e. at most 33 ms per cycle. Therefore, it is not possible to let all ALICA agents communicate asynchronously with the simulation through the proxy process. The simulation process would suffer a Denial-of-Service-like attack and produce unrealistic large delays for the simulated sensor values for all ALICA agents. In order to solve this issue, we are forced to run all ALICA agents as well as the simulation process in a synchronised and event-driven way. Thus, due to the deficiencies of the simulator, we have a complex simulation setup, as shown in Figure 4.1 and further described in the next paragraphs.
The communication channels between the ALICA agents themselves and between the proxy process and the ALICA agents are realised by the middleware provided by the Robot Operating System (ROS) [9]. With ROS it is possible to announce topics and pass messages through these topics to all registered listeners. This communication paradigm relies on common publish and subscribe mechanisms. In case of the WM-Update topic, updates about the environment are passed to each ALICA agent in order to update its world model. Triggered by these updates, each ALICA agent calculates the next command for controlling its simulated car. The calculated control commands are send through the Car-Ctrl topic, received by the proxy process, translated according to the TraCI-Protocol, and sent to the TraCI interface. The third topic includes the only ALICA specific messages, used in both test scenarios, namely the Plantree-Info messages. These messages encode the current state or progress of an ALICA agent in the executed ALICA program and enable all ALICA agents to cooperate.

The algorithm, which synchronises all ALICA agents with SUMO is described as pseudo code in Algorithm 1. The central data-structure of this algorithm is a queue with tasks for the proxy process. Initially the queue is filled with a world model update for each ALICA agent, which basically informs all agents about the start situation of the simulation. Afterwards, the algorithm processes one task of the queue per iteration of the loop (line 4–16). As the queue is sorted according to the timestamp of the tasks, the earliest task is processed first. In line 7 the proxy process triggers the simulation to run until the simulation time is the same as the time of the current dequeued task. This ensures, that the simulation is always in the state, it would have been when the simulation and the ALICA agents would have run asynchronously and the event of the current task would have taken place uncontrolled. In line 8-15 the proxy algorithm distinguishes between two kind of tasks, i.e. the WM-Update and the Car-Ctrl task. The WM-Update tasks, initially created in line 2, make the proxy receive the current state of the simulation (line 9) and send this update to the corresponding ALICA agent (line 10). Now the time is measured, which the ALICA agent needs to produce a corresponding answer, namely
the Car-Ctrl task. This control task is inserted in the queue with a timestamp which considers
the measured calculation time (line 11). Furthermore, another world model update is enqueued
(line 12), considering a realistic sensor frequency of 30 Hz. The Car-Ctrl task is much simpler
to handle, because it just has to be translated according to the TraCI-Protocol and sended
the TraCI connection afterwards (line 14). As a result of this algorithm, the simulation runs much
slower, than in a real scenario, but therefore produces realistic results. It seems to be necessary
to mention, that the minimum simulation step is 1 ms, which means, that the step size of the
simulation has the same magnitude as one iteration of the control loop of an ALICA agent.

There are two further deficiencies in relation to the simulator, which we weren’t able to compen-
sate totally. If requested by an ALICA agent, the TraCI interface sends lane change
commands to SUMO, in order to make the corresponding car switching lanes on its current road. These lane
change commands are, according to the developers we contacted through the SUMO developer
mailing list, just rough hints for SUMO. Thus, the ALICA agents can not reliably control the
lane, the car is driving on. Sometimes SUMO forbids to change the lane, although there would
be enough space for the car to change and sometimes SUMO switches the lanes of the cars by
itself. There are several parameters, which we could identify as influencing parameters for the
handling of lane change commands by SUMO: maximum possible acceleration and deceleration
of cars, current distances and velocities, and impatience. Nevertheless, tuning these parameters
made SUMO accepting lane change commands just a little bit more often and the number of
SUMO-initiated lane changes could not be reduced.

The other problem with the simulator is that it is not possible for a car to drive between two
lanes. Actually the cars are “beamed” from one lane to the other, which makes it possible change
the lane while standing still. Therefore, the requested scenario, where cars should make some
room for an emergency car, while waiting at a red traffic light, suffers tremendously under this
restriction of SUMO. In order to make the scenario work at all, we introduced an extra lane
for each lane to simulate the space a car would use to make some room for an emergency car.
Further details on this scenario are given in Section 4.3.

In order to combine both evaluation scenarios, we modelled a possible ALICA top-level plan
for autonomous car driving, as shown in Figure 4.2. The general purpose of this top-level plan
is to distinguish between inherent different situations, like normal driving, parking, emergency

```
foreach car in Cars do
  queue.add(WM-Update(car,t0));
end

while queue not empty do
  queue.sort();
  cmd ← queue.remove(0);
  sumo.step(cmd.time);
  if cmd is WM-Update then
    update ← sumo.do(cmd);
    cmd.car.send(update);
    queue.add(cmd.car.getCarCtrl(tnow+calc));
    queue.add(WM-Update(car,tnow+it));
  else cmd is Car-Ctrl
    sumo.do(cmd);
  end
end
```

Algorithm 1: Triggering Algorithm of the Proxy Process

18
situations, interactive check up routines and further possibilities, we did not model, yet. The usual state of the ALICA agents when driving towards a goal is the state DriveRoute, which has to handle all traffic rules. This state can be reached after the car check up, which is responsible for self-tests and other optional interaction with the human driver, has been executed successfully. Note, that the state Stop is the initial state were the car is doing nothing but expecting user commands like a route endpoint.

In the top-level plan example we modelled two emergency situations. First, the emergency stop, which is performed in case of safe driving is not possible anymore for some reason, e.g., if the car has a defect. Second, the aforementioned emergency car situation (see Section 4.3).

Figure 4.2: A Possible Master Plan for Autonomous Car Driving

In the next sections we will analyse two scenarios considering the following parameters. First, we will determine the required computational power and memory requirements of the ALICA processes. Therefore, we will execute the scenarios on an Intel Core i7-3770 CPU with 3.4 GHz using Ubuntu 12.04.04 LTS (64-bit) and 16 Gb of RAM. Additionally, we will monitor communication requirements between the ALICA processes that are used for coordination by counting the number of transmitted messages and bytes. Finally, we will analyse which information are necessary for each agent and how they could be acquired. Note, that the simulation is completely deterministic. Therefore, we can gather the experiment data from a single trial.

4.2 Scenario I: Merging from Two to One Lane

The goal of the Merge Traffic scenario is to show cooperation during a zip merge procedure. As shown in Figure 4.3, eight cars are approaching a narrow point. The narrow point is created by
the red car on the left lane which is unable to move e.g. due to an defect. The goal of the cars is to merge to the right lane without causing an accident. Note, that here we do not aim for an optimal vehicle throughput; this is beyond the scope of this scenario.

An ALICA plan that solves this task is depicted in Figure 4.4. It is a child plan of the DriveRoute state in Figure 4.2. By default all agents are in the Drive state. For all states that might lead to a conflict with another participant of the road traffic, a successor is foreseen. In our case we assumed possible conflicts at crossings, narrowings, and if another car wants to switch lanes. The latter two are required for the Merge Traffic scenario.

If a car in the Drive state detects an end of its lane that is approached in less than a configurable security interval, 4.5 seconds in our case, the car switches to the state Narrowing. In that state the car computes all gaps between the cars of neighbour lanes. Further, it selects the easiest reachable gap and aligns to its center. When the car is successfully aligned to a gap it changes the lane and switches back to the Drive state. During this process it also respects the security distance of the gap follower by rejecting gaps that deceed a minimum distance with respect to the current velocity.

Cars switch to the CreateGap state in case they detect a car in the state Narrowing that is willing to switch to their current lane. Although our implementation does this detection by checking whether a car is in the Narrowing state that is aligning to the gap in front of the current car, in practice this could also be determined by detecting direction indicators. Cars in the CreateGap state still follow their road, but slow down in order to create a gap. This facilitates the zip merge process and increases the security of the lane switch.

As shown in Figure 4.5, the communication rate per agent during the execution of this scenario is almost constant between 304 kb/s and 310 kb/s. The deviation is only indirectly coupled to the number of autonomous agents, because ALICA sends with a fixed communication rate, as long as no agent switches between states. However, state changes in the ALICA program are in our case determined by the current traffic situation and therefore limited to a maximum frequency of 15Hz. Note, that a single ALICA message for this scenario requires 60 bytes and is sent with 5 Hz, as long as no state changes occur. Thus, 300 b/s is the minimum required bandwidth per ALICA agent in this scenario.

Figure 4.6 shows the required memory with respect to the percentage of autonomy. Due to our analysis the fluctuations are only depending on the time the garbage collector of the Mono runtime environment\textsuperscript{1} performs clean ups. Nevertheless, the operations performed by ALICA during this scenario are assumed to require almost constant memory. The only situation dependent memory

\textsuperscript{1}Mono is the open source alternative to the Microsoft .Net C# runtime environment under Ubuntu.
Figure 4.4: Plan for Driving a Route

Figure 4.5: Required Bandwidth for Plantree Exchange
allocation is the computed number of gaps. However, the number of gaps is constant as is the number of cars.

Figure 4.6: Memory Consumption of ALICA

The required CPU time for this experiment is between 0.2 ms and 0.5 ms per agent and ALICA iteration, as shown in Figure 4.7. As the state machine computations are independent of the number of agents the slight increase in computation time with respect to the number of ALICA agents might be caused by the task allocation algorithm. Nevertheless, the order of magnitude is insignificantly low and therefore almost negligible.

Figure 4.7 also shows the cycle time per agent in an ALICA iteration. This time fully includes the communication time and all computations of the ALICA behaviours. The presented relationship is growing slightly with the number of autonomous agents. This is due to the fact that more ALICA agents make it necessary to consider more gaps for the computation of the optimal gap. However, we assume that the major offset between cycle time and iteration is caused by the ROS C# communication.

Table 4.1 gives a summary of the most important measurements averaged over all scenario trials. Note, that we experienced peaks of about 100 ms cycle time which are caused by the garbage collector of the Mono runtime environment. A detailed investigation of this effect is left for future work.

4.3 Scenario II: Give Way to an Emergency Vehicle

In the Emergency Vehicle scenario four cars are waiting at a traffic light, while an emergency vehicle is approaching from behind. The task of the cars is to make room for the emergency vehicle. The emergency vehicle can pass cars and traffic light without being slowed down.

As shown in Figure 4.8, we set up SUMO with a road consisting of four lanes. At its end is a traffic light and a crossing, to simulate a typical traffic situation. Due to the fact, that SUMO does not support continuous placement of cars between lanes, a simulation model with just two lanes does not allow a car to pass between two others. To simulate this behaviour we added two artificial lanes to give the cars on the route room to move when the emergency car is approaching.
Figure 4.7: Required CPU Time of ALICA Cycle

<table>
<thead>
<tr>
<th>Observed Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time</td>
<td>0.3324 ms/Iteration</td>
</tr>
<tr>
<td>CPU Peak</td>
<td>12.66 ms</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>0.99 ms</td>
</tr>
<tr>
<td>Cycle Peak</td>
<td>101.19 ms</td>
</tr>
<tr>
<td>Memory Requirement</td>
<td>48900.9 kilo bytes</td>
</tr>
<tr>
<td>Communication Rate</td>
<td>306.9 bytes/sec</td>
</tr>
<tr>
<td>Message Size</td>
<td>60 bytes</td>
</tr>
<tr>
<td>Required Informations</td>
<td>Distance to Leader resp. End of Lane, Own Velocity, relative Position of Gaps to Align to, Nearby Agents in “Narrow”-State</td>
</tr>
</tbody>
</table>

Table 4.1: Properties of ALICA for the *Merge Traffic* scenario
The ALICA plan to solve this scenario foresees two tasks. First, the *EmergencyCar* task is taken by the emergency car. This task only reaches the state *EmergencyDrive*, which is designed to let the emergency car drive its route. In contrast to the *DriveRoute* plan, this plan allows a much more aggressive driving, in order to minimize the travel time of the car. Second, the *DefaultTask* is taken by all other cars. This task continuously checks whether room should be made for the emergency car. Note, that our implementation detects the emergency car by its unique ALICA identifier. Two alternative solutions are surveilling the ALICA state of following cars or a sensor based detection and matching of the noise of the emergency horn.

As shown in Table 4.2 the measured properties of ALICA have the same order of magnitude than in the previous experiment (compare to Table 4.1). One reason for that is the fact, that the plan depth is the same. However, we observed a slightly increased cpu time for that task. This can be explained by the second emergency task, which needs to be allocated. Note, that the allocation task for the emergency car is very simple and therefore plays only a negligible role. In
RoboCup plans with more tasks and a more dynamic task allocation we observed cpu times just for the task allocation task of a few milliseconds.

<table>
<thead>
<tr>
<th>Observed Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time</td>
<td>0.4291 ms/Iteration</td>
</tr>
<tr>
<td>CPU Peak</td>
<td>8.45 ms</td>
</tr>
<tr>
<td>Cycle Time</td>
<td>1.04 ms</td>
</tr>
<tr>
<td>Cycle Peak</td>
<td>9.11 ms</td>
</tr>
<tr>
<td>Memory Requirement</td>
<td>47652.8 kilo bytes</td>
</tr>
<tr>
<td>Communication Rate</td>
<td>307.78 bytes/sec</td>
</tr>
<tr>
<td>Message Size</td>
<td>60 bytes</td>
</tr>
<tr>
<td>Required Informations</td>
<td>Own Position, Relative Leader Position, Relative Emergency Car Position</td>
</tr>
</tbody>
</table>

Table 4.2: Properties of ALICA for the Emergency Vehicle scenario

### 4.4 General Scenario Results

As shown by Table 4.1 and 4.2, our ALICA implementation needs less than 0.5 ms CPU time for a general autonomous driving task. The executed tasks only require weak synchronisation as the observation of the traffic situation is usually sufficient to avoid collisions. Therefore, little explicit communication is required to achieve consensus. Consequently, the communication rate is almost constant at 300-310 bytes/sec. These packets are only required to estimate the state of the other agents and the current team.

The memory consumption of ALICA seems with over 45 MB rather high, but is mainly influenced by the Mono runtime environment. In our opinion the memory requirements can be drastically reduced by porting ALICA to another programming language like C or C++.

Another problem of the Mono runtime environment is the garbage collector. The biggest issue with the garbage collector is the fact that it induces CPU peaks of over 100 ms. This fact excludes real-time capabilities of our implementation. These peaks are the main reason for the magnitude of the error bars in Figure 4.7.

As a final remark we want to mention that the SUMO simulator heavily disturbs the results of this experiment, as car placement is not continuous and does not necessarily guarantee the execution of a sent command. For example, lane change commands are only taken into account if they are not in conflict with SUMO’s built-in control logic. On the other hand SUMO controlled cars drive very aggressively and risky and are therefore no valid reference for car throughput on a certain road map.

### 4.5 Quality of Coordination under Unreliable Communication

The influence of packet delay and packet loss on the coordination of ALICA agents is domain-independent, because agents coordinate themselves by communicating their current state in the executed ALICA program. Therefore, we are able to present results on the quality of coordination under unreliable communication, which were created in the domain of robotic football. The test scenario includes four RoboCup Middle-Size League robots, which play football cooperatively.
Figure 4.10 shows how long the robots needed to coordinate with packet loss, after changes in the state of the execute ALICA program.

![Average Times To Coordinate on Packet Loss](image1)

**Figure 4.10: Real Robots: Average Time to Coordinate with Packet Loss**

Up to 40 percent packet loss, the average time to coordinate is around 200 ms. The average time to agree starts to increase significantly at 60 percent packet loss. The diagram in Figure 4.11 illustrates the average count of belief states in the team while experiencing packet loss. The count of belief states are the number of different beliefs about the current state of the ALICA program. The maximum number of belief states is four, as there are four robots.

![Average Belief Count on Packet Loss](image2)

**Figure 4.11: Real Robots: Average Count of Belief States with Packet Loss**

Figure 4.12 and 4.13 show the same situation with packet delay instead of packet loss. In general it is deduced that packet delay has an even worse effect on the coordination than packet loss has. Times to achieve the same belief keep increasing because every time a robot tries to adapt itself to the new state, it receives old information from its team members. This old information leads to updates on the state of the ALICA program. For instance, if agent A supposes that agent B also joins plan P, agent B is integrated into plan P. This assumption may be right, but now the delayed packets arrive at agent A and it removes agent B from plan P until the new packets,
containing information that agent B actually executes plan P, arrive. A loss of packets can be compensated until a certain level, because an agent does not remove another agent from plan if it does not receive anything from it for a certain time span (300ms). During this time span, the agent sticks to the state of the ALICA program. After the time span, it removes the robot, of whom it does not receive any data, from the team and thus from the state of the ALICA program.

![Average Times To Coordinate on Packet Delay](image1)

**Figure 4.12:** Real Robots: Average Time to Coordinate with Packet Delay

![Average Belief Count on Packet Delay](image2)

**Figure 4.13:** Real Robots: Average Count of Belief States with Packet Delay

### 4.6 Backwards Compatibility

Like all software we also expect ALICA programs to evolve over time. In order to maintain compatibility we propose the following approach for a graceful quality degradation of the cooperation between ALICA agents with different versions of the ALICA program. Each ALICA program is interpreted as a plan tree at runtime. The top-level plans are more abstract and general as
the low level plans. Therefore, we expect the higher levels of the ALICA program, which are responsible for distinguishing different traffic situations, to change less often than the lower levels, which define the precise driving behaviour of each car. ALICA could be extended in a way, that it does not expect completely consistent plan trees, but manages the depth of consistency an ALICA agent has with another one. In the merging scenario (see Section 4.2), two agents could still cooperate if they have the same plan tree down to the level of the DriveRoute plan, although they have different behaviours for making room and aligning to a gap at the lower levels.

4.7 Conclusions

The evaluations of the Emergency Vehicle and Merge Traffic scenario showed the ability of ALICA to control autonomous driving tasks with a relatively low amount of resources. A CPU time of less than 0.5 ms was sufficient for both scenarios, with a memory requirement of less than 55 MB. The communication infrastructure needs a broadcast channel that provides at least 300 bytes/sec per agent. However, the overall problem setting of the scenarios requires little explicit coordination, as the way, how cars coordinate, is often given by common traffic rules and local sensor information about other cars. Nevertheless, we think that explicit communication and cooperation can help to overcome limits induced by noisy and imprecise sensor values and furthermore, that the usage of ALICA with its coordination abilities can enhance autonomous car driving by increasing the traffic flow, while ensuring an increased amount of security.

A further advantage of ALICA in autonomous car driving is the increased maintainability. This was shown in our scenarios, where we demonstrated the handling of different autonomous car situations with three ALICA plans. However, in order to investigate whether ALICA can handle all possible situations of autonomous car driving more research is needed to develop a comprehensive autonomous driving ALICA program. Such a study should also be performed with a simulator environment that is closer to the real world than SUMO. Beside the fact that SUMO does not allow to freely control cars, its discrete lane model together with the prohibition of collisions are to stringent simplifications for an objective evaluation of an ALICA program.

Finally, we could not quantify the ability of ALICA to deal with a non-broadcasting communication infrastructures. Ignoring the lack of a broadcasting infrastructure could lead to inconsistent team estimates, plan bases and sensor data of the environment, which in turn might result in conflicting agent behaviour. Beside these issues we could not find any other problems in the ALICA approach that are not caused by the Mono implementation.
5 Project Proposal

Based on our experiences and results of the two evaluation scenarios of Chapter 4, we derive a project proposal in this chapter. Therefore, we will first summarize the open questions we experienced in this study and then propose concrete work packages. Finally, we provide a cost estimation for the proposed research.

5.1 Open Research Questions for Autonomous Driving with ALICA

The open research questions can be divided in four categories: First, the current ALICA implementation has drawbacks for the practical use in autonomous driving. This is due the use of the Mono C# which does not satisfy real-time requirements, as we could measure garbage collector peaks of about 100 ms. Furthermore, C# does not provide a good compatibility to typical embedded system programming languages like C. Finally, C# is not as memory efficient as more low-level programming languages like C++.

Second, the modelling of the domain specific parts of the ALICA process need to be researched in more detail. In particular, we did not model a complete autonomous car driving ALICA program. Moreover, there is no guarantee that the provided ALICA program is optimal in terms of data efficiency, maintainability, or extensibility. Finally, a world model needs to be investigated that fulfils the requirements of road traffic.

Third, necessary ALICA extensions caused by the communication infrastructure in autonomous car driving have to be addressed. As ALICA assumes a broadcast communication infrastructure, a smart middleware component is required that dynamically relays messages to cars that request them. In order to determine which car needs which information this middleware needs to use the knowledge encoded within the ALICA plans. In particular, the estimation of the team observer needs to be adapted. Finally, the scalability properties of ALICA under these conditions needs to be researched.

Fourth, the demand of the product market needs to be identified. This has to tackle properties like backward-compatibility, standardisation, testing, and quality control of ALICA programs. Although ALICA provides a good basis to address these problems by its formal approach, practical autonomous car driving problems so far have not been engaged in practice.

5.2 Work Packages

From Section 5.1 we can derive five work packages to develop a comprehensive autonomous car driving task.

5.2.1 WP1 Porting the ALICA Implementation

Goal Beside the fact that current computing platforms of autonomous cars do not provide a Mono virtual machine, our implementation can benefit from a better performance and less memory consumption. Therefore, we suggest to develop a C++ ALICA implementation.
A special issue for this work package is to ensure real-time capability of the implementation. This is important, as many control theoretic approaches can only ensure convergence if real-time capabilities of the controller is guaranteed. Therefore, all executions of ALICA rules have to be real-time capable and in particular the triggering of behaviour iterations.

Requirements  (None)

Results  A Real-Time C++ ALICA Library

Excluded Work  Evaluation of different programming paradigms for ALICA

Restrictive Interfaces  (none)

Steps

- Writing software tests
- Porting propositional ALICA C++
- Generating C++ code for conditions in Plandesigner
- Developing a real-time scheduler for ALICA threads
- Developing a C++ automatic differentiation library
- Porting the distributed CSP-solver
- Integrating C++ constraint descriptors code generation in Plandesigner

5.2.2 WP2 Development of an Autonomous Driving ALICA Program

Goal  As ALICA has never been used for autonomous driving tasks before, there is no knowledge base available how a good ALICA program for that domain look like. In particular, a methodical approach for the structure is required in order to provide maintainability and testability. Moreover, the structure needs to take care of the local traffic laws and needs to be adaptable to laws of different countries. Additionally, the ALICA program needs to consider cars that are not equipped with ALICA. Therefore, ALICA needs to be enhanced with a plan recognition algorithms, that anticipates the behaviour over other cars based on observations.

Ensuring that traffic laws are correctly modeled in the program is one of the most crucial points for the ALICA program. One part of this work package is therefore the investigation of a method to test and / or prove ALICA programs for their properties. In particular, the tests or proofs have to ensure a certain behaviour like avoiding crashes. Nevertheless, the program will focus on the common road traffic situations.

Requirements  Real-time ALICA C++ implementation

Results  Domain specific parts of the ALICA process and testing methodology

Excluded Work  Sensor data processing, complete traffic law implementation

Restrictive Interfaces  (none)

Steps

- Researching a testing methodology for ALICA programs

30
• Developing behaviour tests based on german traffic law
• Enhancing ALICA with an plan recognition algorithm
• Developing an autonomous car driving ALICA program considering unequipped cars
• Evaluating the program with german traffic law in an simulated environment with the testing methodology

5.2.3 WP3 Development of a World Model

**Goal** A world model is required that satisfies the demands of autonomous car driving. Thus, it has to deal with uncertainty and incomplete knowledge and to provide mechanisms to ensure conflict free knowledge bases. More precisely an ontology should be provided to achieve mutual understanding of data from different manufacturers and generations of autonomous cars.

**Requirements** Real-time ALICA C++ implementation, ALICA program for autonomous car driving

**Results** A world model for autonomous car driving

**Excluded Work** Research on sensor data processing and fusion

**Restrictive Interfaces** Car2Car and Car2X communication protocols

**Steps**
- Extracting the required knowledge for the autonomous car driving ALICA program
- Determining the uncertainty for commonly used sensors of autonomous cars
- Development of an ontology for an autonomous car driving world model
- Implementation of the world model
- Integrating mechanisms for achieving consensus

5.2.4 WP4 A Smart Communication Middleware and Team Observation Algorithm

**Goal** The communication range of autonomous cars is limited. Therefore, the current Team Observer component of different cars will estimate different teams of cooperating agents depending on their connection to other cars. This problem can affect task allocation in a sense that no common assignment can be found.

Even though our evaluation did show a required bandwidth of about 300 b/s per car, the bandwidth can be further reduced. The current implementation sends regularly new data to other agents to update the common knowledge base and therefore to achieve consensus. A smarter communication middleware could try to reduce the amount of transmitted data by checking, which data is required depending on the current situation and ALICA state. This could drastically decrease the required bandwidth of ALICA.

**Requirements** Real-time ALICA C++ implementation

**Results** Adaptation of ALICA to environments without a broadcast communication infrastructure

**Excluded Work** Generality of developed modules
Restrictive Interfaces  Car2Car and Car2X communication protocols

Steps  
• Determine when and which knowledge is required in the autonomous car driving ALICA program
• Deriving a information exchange and packet relay strategy
• Implementing communication middleware with that strategy
• Adapting team observer to the information exchange strategy
• Adapting task / role allocation to the new team observer
• Reevaluation of the autonomous car driving ALICA program with the testing methodology and the middleware

5.2.5 WP5 Simulator Evaluation

Goal  In order to summarise all results of the project and determine the project success an evaluation is required. As the evaluation on real cars is very time consuming we propose to evaluate in a simulated environment. The evaluation has to determine whether the developed ALICA program can handle standard road traffic situations also considering unequipped cars.

Requirements  All other Work Packages

Results  Full system test on a test track

Excluded Work  Hardware development, sensor Processing

Restrictive Interfaces  Car2Car and Car2X communication protocols

Steps  
• Elaborating simulator test setups covering the most notable autonomous driving tasks
• Linking the simulator to all required ALICA processes
• Performing tests
• Summerising the test Results and recording videos for documentation proposes

5.3 Cost Estimation

We assume that no hardware is required except the workstation PCs to develop the software. Furthermore, Volkswagen needs to provide a simulation environment that is suitable for all evaluation tasks. In particular, the communication interfaces Car2Car and Car2X have to be considered by the simulator.
<table>
<thead>
<tr>
<th>WP</th>
<th>Work Package</th>
<th>Duration</th>
<th>Workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP1</td>
<td>Porting the ALICA Implementation</td>
<td>9 M</td>
<td>9 PM</td>
</tr>
<tr>
<td>WP2</td>
<td>Development of an Autonomous Driving ALICA Program</td>
<td>9 M</td>
<td>15 PM</td>
</tr>
<tr>
<td>WP3</td>
<td>Development of a World Model</td>
<td>9 M</td>
<td>15 PM</td>
</tr>
<tr>
<td>WP4</td>
<td>A Smart Communication Middleware and Team Observation Algorithm</td>
<td>9 M</td>
<td>15 PM</td>
</tr>
<tr>
<td>WP5</td>
<td>Simulator Evaluation</td>
<td>15 M</td>
<td>6 PM</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>60 PM</td>
</tr>
</tbody>
</table>

Table 5.1: Estimation of Resources (M = months, PM = person months)
Bibliography

Middle Size Robot League Rules and Regulations for 2012.
16th ed.
url: https://docs.google.com/open?id=0B6cdZrEa8IsWZWU2M2M3YzctYjAxMyO0ZGRjL
Tk0MjctMTjGiyZDA0ZmVh (visited on 12/06/2013).

Laws of the Game.
2012.
URL: http://www.fifa.com/mm/document/affederation/generic/81/42/36/lawsofth
egame_2012_e.pdf (visited on 12/07/2013).

“Multi-agent role allocation: issues, approaches, and multiple perspectives”.
issn: 1387-2532.

“Recent Development and Applications of SUMO - Simulation of Urban MOibility”.
URL: http://elib.dlr.de/80483/.

“Division of labor in a group of robots inspired by ants’ foraging behavior”.
In: ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1 (1 Sept. 2006),
pp. 4–25.
issn: 1556-4665.

“Belief Update Revisited”.
In: IJCAI.
2007,
In: *Software Engineering for Self-Adaptive Systems*.  
Ed. by Rogerio de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw.  
Dagstuhl Seminar Proceedings 10431.  

“GOLOG: A Logic Programming Language for Dynamic Domains”.  

“ROS: an open-source Robot Operating System”.  
In: *ICRA Workshop on Open Source Software*.  
2009.

*Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*.  
isbn: 0262182181.

*Modelling and Controlling of Behaviour for Autonomous Mobile Robots*.  
isbn: 9783658008109.

“Solving Non-Linear Arithmetic Constraints in Soft Realtime Environments”.  
In: *27th Symposium On Applied Computing*.  
Vol. 1.  
ACM SAC.  

“Resolving Conflicts in Highly Reactive Teams”.  
In: *Kommunikation in Verteilten Systemen 2011*.  

“A modelling language for cooperative plans in highly dynamic domains”.  
“Towards a Comprehensive Teamwork Model for Highly Dynamic Domains”.
Ed. by Joaquim Filipe, Ana Fred, and Bernadette Sharp.
Vol. 2.
INSTICC Press. INSTICC Press, Jan. 2010,
121–127.

“Using incomplete satisfiability modulo theories to determine robotic tasks”.

Reasoning about Rational Agents.