A Language for Interactive Cooperative Agents

Hendrik Skubch, Michael Wagner, Roland Reichle
Distributed Systems Group
University of Kassel
{skubch, wagner, reichle}@vs.uni-kassel.de

February 26, 2009

Abstract

Cooperative behaviour of agents within highly dynamic and nondeterministic domains is an active field of research. In particular establishing highly responsive teamwork, where agents are able to react on dynamic changes in the environment while facing unreliable communication and sensory noise, is an open problem. Moreover, modelling such responsive, cooperative behaviour is difficult. In this work, we specify a novel model for cooperative behaviour geared towards highly dynamic domains. In our approach, agents estimate each other's decision and correct these estimations once they receive contradictory information. We aim at a comprehensive approach for agent teamwork featuring intuitive modelling capabilities for multi-agent activities, abstractions over activities and agents, and a clear operational semantic for the new model. This work encompasses a complete specification of the new language, ALICA.

Contents

1 Introduction

Mobile robots and software agents can substitute manpower, especially for operations in dangerous environments such as mine clearance or searching for human survivors after an earthquake. Even if the complete substitution is normally not the main driving force for building robots, rescue robotics are one example where robots might improve the safety of human beings by performing dangerous tasks for them.

One of the biggest challenges is the design of cooperative behaviours of teams of such agents[1]. However, many of the existing approaches like the BDI model originally by Bratman [3] focus only on a single agent. Similar to human beings, it is also for robots in many situations more effective to work in a team. For instance, a team of robots searches through a destroyed building much faster than a single robot with the same capabilities. Moreover, robots with different capabilities can complement one another to solve a common goal.

Unfortunately, uncertainties in complex and dynamic multi-agent domains obstruct coherent teamwork. In particular, team members often have different, incomplete and sometimes also inconsistent views of their environment. Acting in such environments often requires swift reactions, which do not allow for communication of particular decisions before acting on them. Instead, agents are required to make autonomous decisions taking the team and the common goal into account, while estimating decisions of their teammates.

Furthermore the support of heterogeneous (hardware and software) platforms of mobile robots and software agents is a challenge. In particular, supporting agents with different capabilities and sensors which still can cooperate and predict each other's decisions is essential for any comprehensive teamwork model.

As our work not only focuses on team coordination but also on teamwork, we first have to explain the difference between two topics. Tambe [30] explains:

> [...] the difference between simple coordination and teamwork [...] focuses on the distinction between ordinary traffic and driving in convoy. Ordinary traffic is simultaneous and coordinated by traffic signs, but it is not considered teamwork. Driving in a convoy, however, is an example of teamwork. The difference in the two situations is that while teamwork does involve coordination, in addition, it at least involves a common team goal and cooperation among team members.

According to this example, team coordination can be seen as a prerequisite of teamwork. Even if team coordination is enough under certain circumstances, there are many situations, in which teamwork is needed to achieve a certain goal. One very interesting domain to show this is robotic soccer.

RoboCup is an international project to foster AI and robotics research by providing a standard problem where a wide range of technologies can be integrated and examined. RoboCup chose to use soccer as a central example scenario, aiming at innovations to be applied for socially significant problems and industries.

[1]We use the term "agent" as a synonym for "software agent" and "robot".

In this work, we present *A Language for Interactive Cooperative Agents* (ALICA), a formal language describing strategies for cooperative teams of agents. We focus on its semantics, and explain in detail how agents execute ALICA programs. For most examples, we use the robotic soccer domain, as it is not only very dynamic and demanding, but also provides an intuitive understanding.

Agents and their behaviour are often described using logic. This is very useful to analyse and specify a system in terms of concepts like belief, desire, intention or plan. But as these systems are normally implemented by using an arbitrary programming language, it is often very difficult to verify whether this implementation satisfies its specification. This problem is already described by Wooldridge [33], pg. 298, as the problem of ungrounded semantics. We tackled this shortcoming by introducing a one-to-one correspondence of the concepts used in this specification and those used for the implementation (which is at least partly described in [32, 28]).

This document describes the specification of ALICA and is organised as follows: In the following section, we will state requirements for a language used to describe cooperative behaviour. Section 3 gives an overview of our approach. Afterwards, in Section 4, we discuss the most important related work, describe their influence on this work, and examine them with respect to the requirements stated beforehand.

Section 5 introduces ALICA's syntax formally, and is followed by the main part of this work, a detailed discussion of its semantics in Section 6. We conclude and point out future work in Section 7.

2 Requirements

Suppose a team of robots has to search an earthquake site for survivors. This team could be composed of several robots with quite different capabilities, such as an avionic robot for scouting, some ground based robots equipped with gear to clear debris and some robots with high precision sensors. Suppose furthermore, a human should model the possible behaviour of these robots in advance, without knowing the precise layout of the earthquake site. The task of modelling this behaviour can be very complex, given conditions such as unreliable communication, robots moving completely out of communication range of the rest of the team, incomplete information about the site, the possibility of robots breaking down, and hence the need for other robots to take over their task. In some cases, the situation can change very quickly and dramatically, e.g., due to a building collapsing. Hence, the robots need to react quickly and possibly independently of each other.

Such highly dynamic and nondeterministic domains impose a number of challenges for the realisation of responsive yet coherent teamwork of autonomous agents. Teams of agents operating in such domains have to be robust against sensory noise, breakdown of individual agents, and unexpectedly changing situations. Such changes in the environment require the agents to react and adapt under tight time-constraints, which entails that it is often not possible to explicitly communicate, or even agree upon, a decision before acting on it. Unreliable communication makes maintaining a highly responsive coherent teamwork even more difficult. On the other hand, some tasks might require agents to agree upon their decisions explicitly before acting, such as cooperatively lifting a heavy object.

This small example identifies several essential requirements posed on robotic teams acting in this or similar domains:

High Responsiveness: Individual agents as well as the team as a whole have to be able to react swiftly to unexpected changes in the environment and newly discovered information.

Coherent Teamwork: While doing so, the exhibited cooperation must not degrade completely. Minor temporal breakdowns can be tolerated as long as they are repaired dynamically.

Robustness The team has to be able to cooperate even confronted with unreliable communication and a high amount of sensory noise.

Adaptivity The team must be able to compensate for lost team members, and be able to integrate new team members in case a replacement robot is introduced during runtime.

A language that is designed to describe such robust cooperative behaviour should provide means to support all the requirements above. This again gives rise to requirements on the modelling language itself:

Formal Grounding Any language describing complex behaviour for autonomous agents needs a formal grounding to allow for automated analyses, such as model checking, of the modelled entities.

Intuitive Understanding Nevertheless, modelling cooperative behaviour should be intuitive and possible without in-depth knowledge about the language's implementation.

Support Heterogeneous Capabilities The language needs to provide facilities to treat agents with different capabilities, such that each agent can be employed to the best of its capabilities and without unnecessarily endangering an agent by giving it a task it is not suited for.

Global Perspective The scope of the language is teams of agents. Hence, the behaviour of the team should not emerge from individual single-agent programs, instead, the language provide means to model team behaviour explicitly.

Efficiency During runtime, the agents have to react quickly, hence decision making within the language should be efficient. If planning or other potentially costly reasoning techniques are employed, interleaving with more reactive decision making techniques must be possible.

Abstraction To foster reusability as well as provide a degree of robustness against changes in the team composition, the modelling language should allow for abstractions from agents and from activities. This abstraction needs to be instantiated during runtime, i.e., autonomously by the agents in question.

Explicit Degree of Commitment As stated earlier, some tasks require tight synchronisations between cooperating agents. Establishing such synchronisations can be costly. Hence, the modelling language should support multiple degrees of commitment to be explicitly modelled.

Powerful Failure Handling Finally, given that failures on different levels will eventually occur during runtime, the language should provide means to deal with them. That is, there should be default failure handling mechanisms for different kind of failures, e.g., inability to establish a commitment, unexpected action effects, conflicting beliefs of team members, etc. Moreover, it should be possible to use domain specific knowledge in order to define specific failure handling routines.

3 Design Overview

The main notion of ALICA are so-called *plans*, which capture a specific activity within a team meant to achieve a certain goal.

A particular plan abstracts from concrete agents by relating to a set of *tasks* that need to be accomplished in order for the plan to succeed. This abstraction allows particular agents to dynamically reassign themselves to tasks they deem themselves more suited to or tasks that are deemed more important. Hence, the break-down of a robot can be compensated on the fly. Choosing which task within a plan they take on is an autonomous decision done by the agents. The calculated mapping from agents to tasks is called an *allocation*.

This decision is guided by the *role* an agent takes on within a team, each role provides a set of *preferences* for specific tasks, and is mapped via capabilities onto agents. In the example in Section 2 for instance, the avionic robot would be assigned a surveyor role, which probably has a high preference for scouting tasks and a negative preference for any task that requires physical manipulation.

Due to the two-fold abstraction we introduce here using roles and tasks, roles can be used rather statically, i.e., the role of an agent changes only if the team structure changes after an agent leaves the team. In Section 6.1, we give an overview about our role assignment. However, role assignment is not the main focus of this paper, therefore we present only a general approach without details.

Plans can be grouped together to *plantypes*, providing the agents with sets of alternative ways of solving a particular problem. Choosing a specific plan from such a plantype is done autonomously by the agents in question. This way, a modeller can provide means to accomplish a certain goal under different circumstances.

The autonomous decisions made by the agents are guided by *utility functions* that evaluate a specific allocation for a specific plan based on the beliefs agents have about the current state of affairs.

Plans are modelled similar to petri-nets, containing *states* and *transitions* guarded by conditions between them. However, each state can contain an arbitrary number of plantypes, spanning a hierarchy of plans with which complex strategies can be composed out of simple ones. Furthermore, at each level, plantypes can used to abstract from specific plans.

Finally, at the most basic level, agents execute *behaviours*, small single-agent programs that implement specific actions, such as moving to a certain location or grabbing an object. These behaviours are atomic from the point of view of ALICA, however, we assume that they can signal success or failure of their action upwards.

In order to deal with the unreliability and potential cost of communication, ALICA provides different notions of commitment, by default, an agent assumes it can estimate decisions made by all other agents and revises its estimations if it is provided with additional evidence, e.g., through a periodic communication act. Stronger commitment can explicitly be modelled by *synchronised* transitions, which force the involved agents to establish mutual belief about the corresponding conditions and goals. This is done by explicit communication acts.

4 Related Work

There are plenty of languages describing agent behaviour. Roughly, they can be divided in two groups: BDI-based and action-theoretic approaches. Furthermore we can distinguish between single agent and multi-agent approaches.

BDI The BDI model by Bratman [3] is a model for practical reasoning agents, imbued with particular mental attitudes, viz: **Beliefs**, **Desires** and **Intentions**. **Beliefs** represent the informational state of the agent about the world (including herself and other agents). The **Desires** of an agent represent objectives or situations that the agent would like to accomplish or bring about. **Intentions** represent the deliberative state of the agent: what the agent has chosen to do.

Even if BDI is only a single agent model, we have introduced it here as it builds the base for several existing works. ALICA can be seen as a BDI language, although in its current form, it lacks an explicit representation of desires or goals, as it focuses on plans as intentions.

Joint Intentions Theory The Joint Intentions Framework [4] is a theoretical framework founded on BDI logics. The framework focuses on a team's joint mental state, called a **joint intention**. A team jointly intends a team action if team members jointly committed to perform an action while in a specified mental state.

In order to enter a joint commitment, the team members have to establish appropriate mutual beliefs and individual commitments. Although the Joint Intentions Theory does not mandate communication and several techniques are available to establish mutual beliefs about actions from observations (see for example [15]), currently communication seems the only feasible way to attain joint commitments. A very interesting key aspect of the Joint Intention Theory is the commitment to attain mutual belief about the termination of a team action. This helps to ensure that the team stays updated about the status of the team actions. Joint intentions and joint commitments provide a basic framework to reason about coordination required for teamwork as well as guidance for monitoring and maintaining team activities. However, a single joint intention for a high-level team goal seems not appropriate to model team behaviour in detail and to ensure coherent teamwork.

There exists several different implementation such as GRATE* [17] based on the Joint Intention Theory. Unfortunately, here also the *problem of ungrounded semantics appears* and it is hard to evaluate to which extend the implementations follow the theory.

Shared Plans Theory In contrast to Joint Intentions, the Shared Plans Theory [11, 10] employs a hierarchical structures over intentions, thus overcoming the shortcoming of single Joint Intention for complex team tasks. The Shared Plans Theory is not based on a joint mental attitude but on an intentional attitude called **intending that**, which is very similar to an agent's normal intention to perform an action. However, an individual agent's 'intention that' is directed towards its collaborator's action or towards a group's joint action. 'Intention that' is defined via a set of axioms that guide an individual to take actions (including the communication), that enable or facilitate its team-mates, sub-team or team to perform assigned tasks.

A SharedPlan for group action specifies beliefs about how to do an action and sub actions [11, 10]. The formal model captures intentions and commitments toward the performance of individual and group actions. A collaborative plan is composed of a **Mutual belief**, of a (partial) recipe, individual **intentions to** perform the actions, individual **intentions that** collaborators succeed in their sub actions and individual or collaborative plans for sub actions. With the concept of actions and sub-actions the Shared Plans Theory describes a hierarchy of plans to reach a common goal. This is also the main difference between the Joint Intentions Theory and the Shared Plans Theory; the Shared Plans Theory describes the way to achieve a common goal whereas the Joint Intentions Theory describes only this common goal. However, the lack of principles like joint intentions and joint commitments results in limited possibilities to reason about team coordination and team activities.

There exist several implementations based on the Shared Plans Theory, such as CAST [34, 35]. Unfortunately, here also the *problem of ungrounded semantics appears* and it is hard to evaluate to which extend the implementations follow the theory.

STEAM & Machinetta STEAM [30, 23] builds on both Joint Intention Theory and Shared Plan Theory and tries to overcome their shortcomings. Based on joint intentions, STEAM builds up hierarchical structures that parallel the Shared Plan Theory as described in the previous paragraph. So STEAM formalises commitments by building and maintaining joint intentions and uses Shared Plans to treat team's attitudes in complex tasks, as well as unreconciled tasks.

ALICA is very similar to and borrows a number of ideas from STEAM, and thus also from the Joint Intentions Theory and from the Shared Plans Theory. Just as STEAM, ALICA builds hierarchical structures of team plans that cover the collaborative behaviour of whole teams and sub-teams, provides mechanisms to assign agents to (sub-)teams and identifies the need for tracking of actions performed by teammates. ALICA also parallels the Joint Intentions Theory. In fact, ALICA can be considered to be a BDI language, although in its current form, it lacks an explicit representation of desires or goals, as it focuses on plans as intentions.

However, in contrast to STEAM, ALICA agents in general do not establish joint intentions before acting towards a cooperative goal. Instead, each agent estimates the decisions of its team mates and acts upon this estimation. Conflicting individual decisions are detected and corrected using the periodically communicated internal states of team-mates. Although STEAM provides approaches for selective communication and tracking of mental attitudes of team-mates, we argue that for highly dynamic domains and time-critical applications the strict requirement to establish or estimate a joint commitment before a joint activity is started has to be skipped. In our opinion, agents that decide and act until contradictory information is available seem to be much more suitable for such applications. Nevertheless, ALICA provides language elements to enforce an explicit agreement, resulting in a joint intention, for activities that require time critical synchronisations, such as cooperative lifting of an object. Also the assignment of agents to teams and teams to operators (which encapsulate the actual team-behaviour) done by STEAM seems to be too static for highly dynamic domains. For example in a soccer game, a robot that is assigned as defender, should also be able to take over the tasks of an attacker if it obtains the ball and the game situation seems to be promising to start an attack over the side-line. In order to facilitate such behaviour, we provide a slightly different definition of roles and incorporate the concept of tasks and task prefrences. With its coordination approach, ALICA also skips the concept of a 'team-leader' which STEAM assumes for different purposes in team coordination and in resolving conflicts.

The project “Machinetta” [27] is based on STEAM. In order to provide a lightweight and portable implementation of the teamwork framework, “Machinetta” uses the concept of proxies to build a reusable software package that encapsulates the teamwork model. Each proxy works closely with a single domain agent, representing that agent in the team.

All previously described teamwork models have in common, that they provide mechanisms to reason about or to establish teamwork, but they do not go into much detail for the description of the internals of plans or operators. STEAM for example, and its implementation TEAMCORE [24] just assume reactive or situated plans, and do not provide support to really 'program' plans with regard to sequential and/or parallel actions and do not really specify the internal control cycle of an agent. Here, agent programming languages have inspired the design of ALICA, in particular 3APL [14] and its successor 2APL [6].

3APL is an agent oriented programming language ([14, 5]), aiming at modelling cognitive agents and high level control of cognitive robots. It implements the BDI model originally by Bratman [3].

ALICA shares many concepts of 3APL, e.g., the definition of the belief base, substitution of variables and the interpretation of goals as 'goals-to-do', which are not described declarative but via plans that are directed towards achieving a goal. However, in contrast to ALICA, 3APL also facilitates explicit specification of goals. It introduces rule sets and beliefs to allow reasoning over both, goals and plans. Moreover, ALICA defines it operational semantics much in the same way as 3APL through a transition system. In fact, 3APL distinguishes between a transition system for the pure language elements and a transition system for the meta-language to specify the control structures of an agent. In ALICA, we do not provide this distinction, and thus, the two transition systems are merged to a single one. Although 3APL implementations support communication in a FIPA[2] [8, 9] compliant manner, explicit multi-agent plans as we support in ALICA cannot be expressed in 3APL.

2APL by Dastani et al. [6] is a successor of 3APL with a strong emphasis on multi-agent systems. However, currently 2APL does not feature any way to model multi-agent plans from a global perspective. Instead, single agent plans need to be devised which interact with one another.

AgentSpeak(L) allows BDI agents to be specified similar to logic programs [25]. Rao [25] identified the problem of ungrounded semantics independently of Wooldridge [33]. Rao [25] tries to overcome this shortcoming by introducing the AgentSpeak(L) which abstracts an implemented BDI system. AgentSpeak(L) is a programming language based on a restricted first-order language with events and actions. Unfortunately, the modelling of multi-agent plans is not possible in AgentSpeak(L). Interestingly enough, AgentSpeak(L) could be embedded into 3APL [12]. The main result of the work of Hindriks et al. [12] is a proof that 3APL can simulate AgentSpeak(L) and that as a consequence, 3APL has at least the same expressive power as AgentSpeak(L).

KARO is not a programming language, but an agent logic based on dynamic logic. However, Hindriks and Meyer [13] proposed a programming language that directly relates to the logic. We argue that the modalities of dynamic logic are only of limited use in robotic scenarios, where actions happen over possibly concurrent time intervals. As such, robotic scenarios are potentially easier to describe in theories working with time intervals, such as the event calculus [29], for which no full agent oriented languages exist yet.

Action Theoretic Languages such as GOLOG [20], and FLUX [31], feature clear and rich second order semantics. However, to our knowledge, there is no practical action theoretic language that describes multi-agent behaviour. In general, these languages focus more on knowledge representation and reasoning about actions than on behaviour specification. However, the strong emphasis on powerful reasoning techniques appeals in comparison to typical BDI languages.

Hierarchies of Abstract Machines (HAMs), originally by Parr and Russell [21], are used to describe agent programs in an abstract way in order to improve performance of reinforcement learning algorithms. This is done by providing a hierarchical structure in advance to the learning algorithm, such that several smaller policies have to be learnt instead of a single monolithic one. HAMs have been extended by Andre and Russell [1] to *Programmable* HAMs (PHAMs), providing additional modelling power such as parameters and interrupts. The general structure of PHAMs, albeit limited to the single agent case, is quite similar to the structure provided by an ALICA program.

[2]Foundation for Intelligent Physical Agents

Moise+ by Hübner et al. [16] introduces the notion of "Agent Organisations" which rely on the notion of openness and heterogeneity of MAS and poses new demands on traditional MAS models. These demands include the integration of organisational and individual perspectives and the dynamic adaptation of models to organisational and environmental changes. Agent Organisations have been advocated to deal with agent coordination and collaboration in open environments [16]. In this context, an organisation is the set of behavioural constraints adopted by, or enforced on, a group of agents to control individual autonomy and achieve global goals. Moise+ provides no means to specify actual behaviour, but instead is ment to be used as an organisational model in conjunction with an agent language.

Whereas other multi-agent coordination approaches concentrate on or are more suitable for the coordination of large-scale teams involving a high number of agents or swarms, ALICA is focused on providing a teamwork model for small-scale teams of autonomous agents, involving only up to about 20 teammates.

Considering all the aspects discussed above, we argue that ALICA enhances the state of the art in team-work models, as it provides another step towards a comprehensive approach that provides support for all aspects of team coordination and also for explicit programming of team behaviour from a global perspective at the same time. With its approach to allow agents to decide and act towards a certain team-goal without explicit establishment of a joint commitment, it is also very suitable for highly dynamic domains that require fast decisions and actions and do not allow explicit communication and negotiations beforehand.

5 Syntax

In this section, we introduce the language elements of ALICA and their syntactic relationships. In Section 6, we explain their semantics in detail.

5.1 Conventions

In this work, we assume the following notional conventions:

- Set union is denoted by $+$ and subtraction of finite sets by $-$:

$$A - B \overset{def}{=} \{a | a \in A \wedge a \notin A \cap B\}$$

- Free variables in formulae are universally quantified if not otherwise stated.

- The following abbreviations are used in first-order formulae:

$$(\forall x \in S)\phi \overset{def}{=} (\forall x) x \in S \rightarrow \phi$$

$$(\exists x \in S)\phi \overset{def}{=} (\exists x) x \in S \wedge \phi$$

- By $\mathrm{img}(f)$ we denote the image of function f in the usual sense.

5.2 Syntactic Elements of ALICA

An ALICA domain signature consists of

- a set of Agents $\mathcal{A}$, which form the cooperative team;
- a logic $\mathcal{L}$, with language $\mathcal{L}(Pred, Func)$ meant to describe the agents' belief bases with a set of predicates $Pred$ and a set of function symbols $Func$.

Given a signature, an ALICA program consists of:

- a set of Roles $\mathcal{R}$, which contains all available roles an agent can take on;
- a set of Plans $\mathcal{P}$, each describing a specific cooperative activity;
- a set of Tasks $\mathcal{T}$, each intuitively describing a function or duty within plans, meant to be fulfilled by one or more agents;
- a set of States $\mathcal{Z}$, a state occurs within a plan as a step during an activity;
- a set of Transitions $\mathcal{W} \subseteq \mathcal{Z} \times \mathcal{Z} \times \mathcal{L}$, connecting states within plans;
- a set of Synchronisations $\Lambda \subseteq 2^{\mathcal{W}}$, connecting transitions and denoting the need to synchronise certain actions;
- a set of Behaviours $\mathcal{B}$, encapsulating lower level actions, thus forming the atomic activities within ALICA;
- a top-level plan $p_0 \in \mathcal{P}$;
- a top-level state $z_0 \in \mathcal{Z}$;
- a top-level task $\tau_0 \in \mathcal{T}$;
- a set of functions, PlanType, each of type $2^{\mathcal{L}(Pred,Func)} \mapsto \mathcal{P}$, abstracting from a set of related plans.

With the exception of $\mathcal{L}(Pred, Func)$, all the above sets are considered to be finite.

The following functions are used to describe a specific ALICA program:

- States: $\mathcal{P} \mapsto 2^{\mathcal{Z}}$ States(p) denotes the states within a plan.
- Tasks: $\mathcal{P} \mapsto 2^{\mathcal{T}}$ Tasks(p) denotes the tasks of a plan.
- $\xi \colon \mathcal{P} \times \mathcal{T} \mapsto \mathbb{N}_0 \times (\mathbb{N}_0 \cup \{\infty\})$ 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 τ.
- Init: $\mathcal{P} \times \mathcal{T} \mapsto \mathcal{Z}$, Init($p, \tau$) denotes the initial state of of task τ in plan p
- Pre: $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}(Pred, Func)$, Pre($p$) denotes the precondition of plan or behaviour p.
- Run: $\mathcal{P} \cup \mathcal{B} \mapsto \mathcal{L}(Pred, Func)$, Run($p$) denotes the runtime condition of plan or behaviour p.

- PlanTypes: $\mathcal{Z} \mapsto 2^{2^{\mathcal{L}(Pred,Func)} \mapsto \mathcal{P}}$, PlanTypes$(z)$ denotes the set of plantypes to be executed on state z. We write PlanTypes$(z)(\mathcal{F})$ to denote the set of plans that are identified by the set of plantypes given a set of formulae $\mathcal{F}$.
- Behaviours: $\mathcal{Z} \mapsto 2^{\mathcal{B}}$, Behaviours$(z)$ denotes the set of behaviours to be executed in state z.
- Success: $\mathcal{P} \mapsto 2^{\mathcal{Z}}$, Success$(p)$ denotes the set of terminal states of plan p, which indicate successful execution of the plan.
- Fail: $\mathcal{P} \mapsto 2^{\mathcal{Z}}$, Fail$(p)$ denotes the set of terminal states of plan p, which indicate unsuccessful execution of the plan.
- Post: $\mathcal{Z} \mapsto \mathcal{L}(Pred, Func)$, 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, $\mathcal{U}_p \colon 2^{\mathcal{L}(Pred,Func)} \mapsto \mathbb{R}$, 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, plantypes 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 define the following derived functions:

- By PlanTypes$^*(z)$ we denote a transitive flavour of PlanTypes, inductively defined by:
  - PlanTypes$(z) \subseteq$ PlanTypes$^*(z)$
  - If $f \in$ PlanTypes$^*(z)$ then
    $(\forall z', p) p \in \text{img}(f) \wedge z' \in \text{States}(p) \rightarrow \text{PlanTypes}(z') \subseteq \text{PlanTypes}^*(z)$.
- Plans: $\mathcal{Z} \mapsto 2^{\mathcal{P}}$, Plans$(z)$ denotes the set of plans that can be executed within state z:

$$\text{Plans}(z) \overset{def}{=} \{p | (\exists f \in \text{PlanTypes}(z)) p \in \text{img}(f)\}$$

- Plans*: $\mathcal{Z} \mapsto 2^{\mathcal{P}}$ is the transitive flavour of Plans, defined by:

$$\text{Plans}(z)^* \overset{def}{=} \{p | (\exists f \in \text{PlanTypes}^*(z)) p \in \text{img}(f)\}$$

- Reachable: $\mathcal{P} \times \mathcal{T} \mapsto \mathcal{Z}$ denotes the set of states transitively connected to Init(p, τ), i.e., reachable in p by task τ. It is inductively defined by:
  - Init$(p, \tau) \in$ Reachable(p, τ)
  - If $z \in$ Reachable(p, τ) then
    $(\forall z', \phi)(z, z', \phi) \in \mathcal{W} \rightarrow z' \in \text{Reachable}(p, \tau)$

- Plans or Behaviours: $\mathcal{PB} \overset{def}{=} \mathcal{P} \cup \mathcal{B}$

Example 5.1 (Example Plan)**.** *Figure 1 shows a simple example plan. The plan has the initial state $Z0$ with the task $Task1$. Furthermore, the plan contains a second state $Z1$ which is reachable by the transition $Trans1$ from the initial state, and a final state $Z2$ which is reachable by the transition $Trans2$ from the second state $Z1$. In each of the first two states a plantype is to be executed: $Pt1$ in state $Z0$ and $Pt2$ in state $Z1$. $Z2$ is a successful terminal state, i.e.,* $\text{Success}(Plan1) = \{Z2\}$*, and as such it does not contain any plantypes. This restriction will be formally introduced in Section 5.3.*

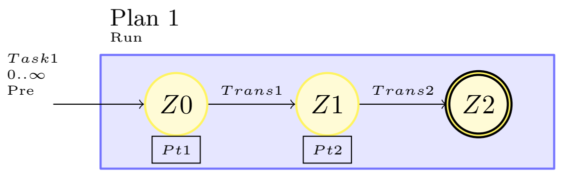

Figure 1: Example Plan

Next, let us illustrate the hierarchical structure of an ALICA program in Example 5.2.

Example 5.2 (A hierarchical Plan)**.** *This example shows a hierarchical ALICA program in the robotic soccer domain. In Figure 2 a simple high-level plan for a throw-in is depicted. The plan has the initial state* ***ThrowInPos*** *with the task* ***DefaultTask****. Furthermore, the plan contains a second state* ***DoThrowIn*** *which is reachable by a transition from the initial state. In each state a plantype has to be executed: the ThrowInPosPT in the state ThrowInPos and the DoThrowInPT in the state DoThrowIn. As aforementioned, a plantype abstracts from a set of related plans. In this small example, each plantype has only one realising plan: ThrowInPosPlan is a realisation of the plantype ThrowInPosPT and DoThrowInPlan is a realisation of the plantype DoThrowInPT.*

ALICA plans can have parameters, which are instantiated during runtime. This allows to model activities in a very expressive and compact way. The set of parameters, or variables a plan has is defined by its components. Intuitively, all free variables in any condition occurring in a plan are parameters. Definition 5.1 captures this relation.

Definition 5.1. We define sets of free variables in plans and behaviours based on the formulae relevant to a plan or behaviour, respectively.

- For a formula $\phi \in \mathcal{L}(Pred, Func)$, let $\text{vars}(\phi)$ denote the set of free variables in ϕ.
- For a behaviour $b \in \mathcal{B}$, let $\text{vars}(b)$ denote the union over the free variables in $\text{Pre}(b)$, $\text{Run}(b)$, and $\text{Post}(b)$.
- For a plan $p \in \mathcal{P}$, let $\text{vars}(p)$ denote the union over the free variables in all formulae occurring in p, i.e., in:
  - the precondition and the runtime condition,

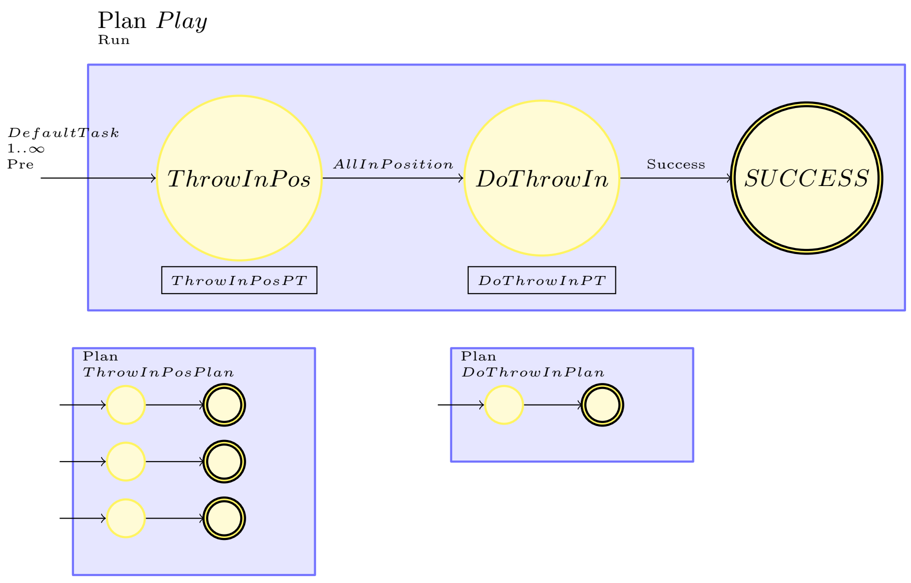

Figure 2: Example: Throw-In Plan

- all post conditions,
- all formulae in transitions within p.

Note that terms bound to vars(b) are passed onto the lower level behaviour. We therefore require that for all behaviours b, Pre(b) $\wedge$ Run(b) grounds all unbound variables in vars(b).

Definition 5.2. A *parametrisation* of a plan p is a substitution, denoted $\rho(p)$, connecting vars(p) with free variables in plans and behaviours occurring in p. For all plans p, the domain of $\rho(p)$ is a subset of

$$\{x|(\exists p', z)x \in \text{vars}(p') \wedge (p' \in \text{Plans}(z) \vee p' \in \text{Behaviours}(z)) \wedge z \in \text{States}(p)\}$$

and $\rho(p)$ maps onto terms with variables only among vars(p).

Intuitively, a parametrisation is used to hand over variables between plans. Each occurrence of a plan p in an ALICA program is unique in the sense that it is standardised apart from every other occurrence, much in the same way as clauses in logic programs are standardised apart.

Example 5.3 (Parametrisation)**.** *This example shows how parametrisation is used within ALICA. Therefore we introduce a scenario, where a robot should move a mountain, composed of several small stones, from one place to another. As the robot is not able to take more then one stone of the mountain, it has to move stone by stone. In an imperative language, a plan for this task could be modelled as depicteted in Figure 3. But since variables are referentially transparent in ALICA, this plan is unusable, as the robot would endlessly try to move the first stone bound to X. Figure 4 shows a referentially correct plan and its subplan to execute this task. Here the subplan $MoveStone$ to move a single stone is called as long as the original mountain exists. Each subsequent execution of*

MoveStone is standardised apart from the previous one, hence, every time a different stone is bound to X.

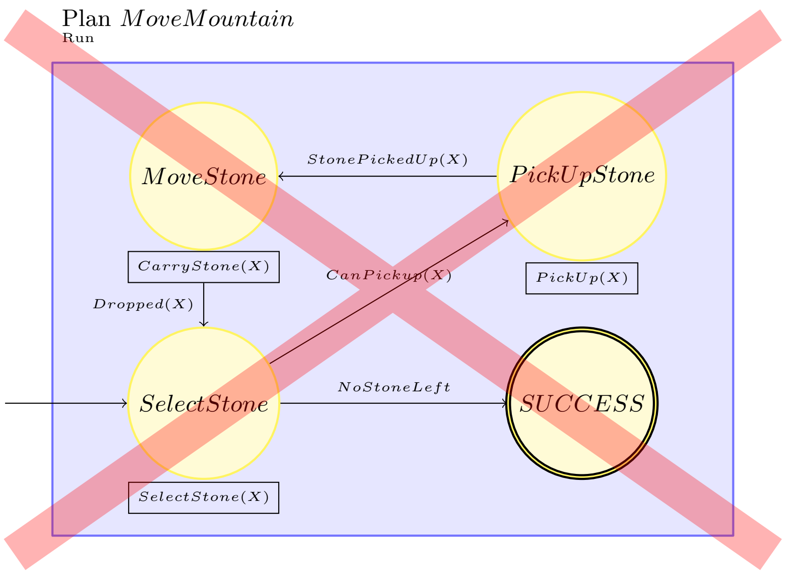

Figure 3: Example: Parametrisation

This declarative treatment of variables is similar to common logic programs. A rough, illustrative translation of the two plans to programs is given in Figure 5.

5.3 Syntactic Constraints

In the previous section, we introduced the language elements of ALICA and stated some relationships between them, such as the plantree. We now constraint the syntax of an ALICA program in a syntactical manner, in order to guarantee the intended structure of these relationships.

Let Σ_{syn} be the set containing the following axioms:

- The top-level plan contains precisely one state, z_0, and one task, τ_0:
  $\text{States}(p_0) = \{z_0\} \land \text{Tasks}(p_0) = \tau_0$ (A5.1)
- States belong to at most one plan:
  $(\forall p, p' \in \mathcal{P})\, \text{States}(p) \cap \text{States}(p') = \emptyset \lor p = p'$ (A5.2)
- All plan-task pairs have a valid cardinality interval associated:
  $(\forall p \in \mathcal{P}, \tau \in \mathcal{T})\tau \in \text{Tasks}(p) \rightarrow (\exists n_1, n_2)\, \xi(p, \tau) = (n_1, n_2) \land n_1 \leq n_2$(A5.3)
- No transition connects states in different plans:
  $(\forall (z_1, z_2, \phi) \in \mathcal{W})(\exists p \in \mathcal{P}) z_1 \in p \land z_2 \in p$ (A5.4)
- Failure and Success sets are disjoint subsets of the corresponding state set:
  $(\forall p \in \mathcal{P})\, \text{Success}(p) \subseteq \text{States}(p) \land \text{Fail}(p) \subseteq \text{States}(p)$ (A5.5)
  $\land\, \text{Success}(p) \cap \text{Fail}(p) = \emptyset$

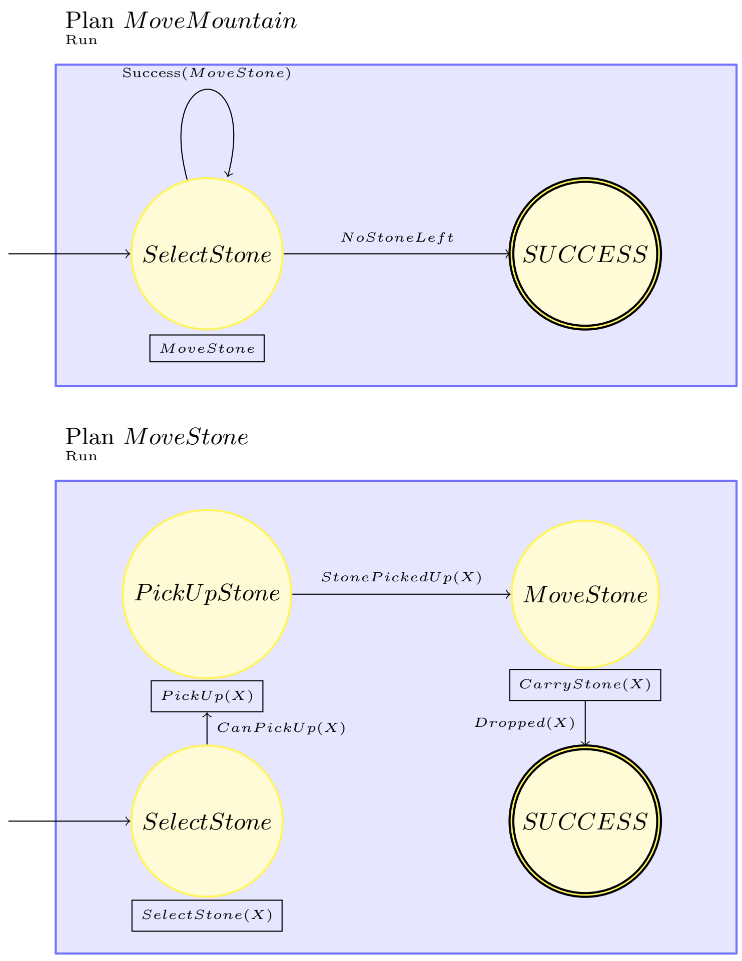

Figure 4: Example: Referentially Correct Parametrisation

- There is a post condition associated with each success and failure state:
$$(\forall p \in \mathcal{P})(\forall z \in \mathrm{Success}(p) \cup \mathrm{Fail}(p))(\exists \phi)\, \mathrm{Post}(z) = \phi \tag{A5.6}$$
- Synchronisations happen only within a plan:
$$(\forall s \in \Lambda)(\forall w, w' \in s) w = (z_1, z_2, \phi_1) \wedge w' = (z_3, z_4, \phi_2) \tag{A5.7}$$
$$\wedge(\exists p \in \mathcal{P}) z_1 \in \mathrm{States}(p) \wedge z_3 \in \mathrm{States}(p)$$
- The hierarchical plan structure does not contain cycles:
$$(\forall p, z) p \in \mathrm{Plans}^*(z) \rightarrow z \notin \mathrm{States}(p) \tag{A5.8}$$
- All plans and behaviours are standardised apart:
$$(\forall p, p' \in \mathcal{PB})\, \mathrm{vars}(p) \cap \mathrm{vars}(p') = \emptyset \vee p = p' \tag{A5.9}$$
- Terminal States do not have sub-plans or behaviours attached:
$$(\forall z)((\exists p) z \in \mathrm{Success}(p) \vee z \in \mathrm{Fail}(p)) \rightarrow \mathrm{PlanTypes}(z) = \emptyset \tag{A5.10}$$
$$\wedge\, \mathrm{Behaviours}(z) = \emptyset$$

```
%moveMountain1(X):- selectStone(X), canPickUp(X), pickUp(X),
                    pickedUp(X), moveStone(X), moveMountain1(X).

moveMountain1(X):- noStoneLeft.

moveMountain2   :- moveStone, moveMountain2.
moveMountain2   :- noStoneLeft.

moveStone       :- selectStone(X), canPickUp(X), pickUp(X),
                   pickedUp(X), carryStone(X), dropped(X).
```

Figure 5: Intuitive translation of two plans to programs.

An ALICA program is valid iff it satisfies Σ_{syn}.

5.4 Plantypes

Plantypes are used to abstract from concrete plans and allow agents to choose autonomously between different alternative ways of solving a problem or reacting to a certain situation. Intuitively, a plantype, mapping from set of formulae in $\mathcal{L}(Pred, Func)$ into $\mathcal{P}$, can be seen as a non-deterministic choice operator. An agent that enters a state z chooses a plan to execute from every plantype in PlanTypes(z). The possible choices each plantype offers is modelled through a set of plans. The choice depends on the agent's belief base and is encapsulated by the corresponding plan type. All plantypes f are defined by:

$$f(\mathcal{F}) \overset{def}{=} \underset{p \in S_f}{\operatorname{argmax}} \mathcal{U}_p(\mathcal{F})$$

S_f is called the *defining set* of f, and is modelled by the user as a subset of $\mathcal{P}$. A deterministic execution of a plan p can be modelled by a plantype with an associated set that contains only p.

By this definition, a plantype maps onto the plan with the highest utility with respect to the set of forumlae $\mathcal{F}$. This set should correspond to a believed or hypothesised situation the agent faces. Note that $\mathcal{F}$ is not constraint to believes about the environment, but can also encompass believes about the internal state of an agents teammates. If, for example, agent a believes that agent b executes plan p in plantype P, the utility functions involved can refer to this fact and evaluate p higher than other plans in P.

6 Semantics

We define an operational semantic for ALICA in terms of a transition system [22]. This transition system describes how and when an agent updates its internal state, and hence how it progresses within an ALICA program. Firstly, we formally introduce the notion of an agent and an *agent configuration*, which captures the internal state of an agent. Afterwards we describe the notion of roles and how roles can be mapped to agents. In Section 6.3, we describe the most important aspect of an agent, namely its belief base, in which an agent represents both the state of the environment and the state of the team it is part of.

Within ALICA, an agent is assumed to follow the execution loop depicted in Figure 6. The major steps within this loop are: (1) Update of its beliefs (e.g., by incorporating sensor information), (2) Update the plan base wrt. the transition system, and (3) Execution of all active behaviours according to its state computed in step (2).

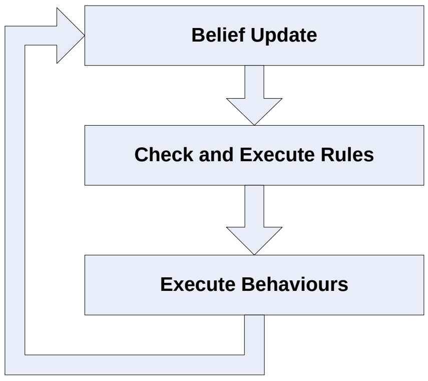

Figure 6: Agent Execution Loop

Definition 6.1 (Agent)**.** An agent $a \in \mathcal{A}$ is described by an associated set $\mathrm{Cap}_{\mathrm{prov}}(a)$. $\mathrm{Cap}_{\mathrm{prov}}(a)$ is the set of capabilities agent a possesses. Every capability $c \in \mathrm{Cap}_{\mathrm{prov}}$ is a tuple $(Key, Value)$, where Key refers to a fuzzy-set, e.g., speed and $Value$ is either a membership function of that fuzzy-set, representing notions such as "fast", or a concrete value, e.g., $4m/s$.

Definition 6.2 (Agent Configuration)**.** For any agent a, $\mathrm{Conf}(a)$ denotes its *configuration*. An agent configuration is a tuple $(B, \Upsilon, E, \theta, \mathrm{R})$, where

- B is the agent's belief base,
- Υ is the agent's plan base,
- $E \subseteq \mathcal{B} \times \mathcal{Z}$, the set of behaviours b the agent executes together with the state in which b occurs, called the context of b,
- θ a substitution,
- R is a set of roles, which are assigned to the agent according to his capabilities.

Intuitively, a configuration captures the dynamic part of an agent.The belief state refers to the current situation and is subject to dynamic changes. The plan base reflects an agent's intention in a procedural manner. The set of behaviours refers to actions that are currently ongoing. The substitution θ holds the current instantiations of plan and behaviour parameters. Whenever a condition is evaluated, θ is updated. The final part of an agent configuration is its role set. While roles are comparatively static, they are subject to change whenever the perceived team composition changes, e.g., if a team member is incapacitated. Note that an agent can be assigned multiple roles. However, the number of roles an agent can take, depends heavily on the domain, e.g., in the RoboCup domain one role per agent is usually sufficient, while in other domains multiple roles might be necessary, especially if an agent has multiple unrelated capabilities, such as infrared vision and a gripper. In the following sections, we will discuss each part of an agent's configuration in detail.

6.1 Roles

Within ALICA, tasks denote specific activities within plans, while roles are used to establish a more general team make-up. In realistic scenarios, it is always possible that an agent breaks down and can no longer participate within a team. In the same manner, an agent might be assigned to a team during runtime. In order to take these possibilities into account, we use roles to abstract from concrete agents and allow for roles to be reassigned dynamically if the team composition changes. Roles are then mapped to tasks through *preferences*.

Definition 6.3 (Role)**.** A Role $r \in \mathcal{R}$ is a tuple $(\mathrm{Pref}, \mathrm{Cap}_{\mathrm{req}}, \mathit{Priority})$, where

- Pref is a function $\mathrm{Pref}\colon \mathcal{T} \mapsto [-1, 1]$, that maps all available tasks to a real number between -1 and 1. $\mathrm{Pref}(\tau)$ expresses the preferences of agents assigned to role r towards task τ. A negative preference demonstrate the incapability of an agent to execute this certain task. If an agent has more then one role $r_1, \ldots, r_n$, its preference for a task τ is the maximum of all corresponding preferences.
- $\mathrm{Cap}_{\mathrm{req}}$ is a set of capabilities the role requires from the agent. Every capability $r \in \mathrm{Cap}_{\mathrm{req}}$ is a tuple $(Key, Value, Weight)$, where Key refers to a fuzzy-set, e.g., speed, $Value$ is a membership function of that fuzzy-set representing for instance "slow", and $Weight$ is a real value between 0 and 1 expressing the importance of this specific capability for describing the role.
- $\mathit{Priority}$ is the priority of the role within the complete set of roles.

Definition 6.4 (Role Utility)**.** Whenever an agent joins a team, its roles have to be assigned according to its capabilities. We calculate a utility $\mathcal{U}_r$ of a certain agent for a certain role using the similarity measure Δ [36] of the provided capability of the agent $\mathrm{Cap}_{\mathrm{prov}}$ and the required capability of the role $\mathrm{Cap}_{\mathrm{req}}$. The weighted sum of these values is the utility of one agent for a certain role: The utility $\mathcal{U}_r(a)$ of a role r has the form:

$$\mathcal{U}_r(a) = \begin{cases} \frac{\sum_{(V_{pro}, V_{req}, W_{req}) \in Y} W_{req} \cdot \Delta(V_{req}, V_{pro})}{|Y|} & \text{if } Y \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

$$\text{where}\quad Y = \{(V_{pro}, V_{req}, W_{req}) | \quad \begin{aligned}[t] &(\mathrm{Pref}, \mathrm{Cap}_{\mathrm{req}}, Prio) = r \\ &\wedge (K, V_{pro}) \in \mathrm{Cap}_{\mathrm{prov}}(a) \\ &\wedge (K, V_{req}, W_{req}) \in \mathrm{Cap}_{\mathrm{req}}\} \end{aligned}$$

Given role utilities, which measure the adequateness of an agent for a certain role, we can introduce role assignment procedures, which distribute roles among all agents of the team.

A role assignment procedure should calculate a role assignment maximising the sum of all role utilities involved under a certain set of constraints such as:

- an agent can take on more than one role, but multiple roles might lower its efficiency in each of them;
- a role can be assigned to multiple agents;
- all agents should have a role;
- roles can have constraints attached to them, excluding each other.
- ...

Since role assignment is not the main focus of this work, we simplify the problem by adopting the following assumptions:

- an agent can only be assigned to exactly one role;
- a role can only be assigned to exactly one agent;
- roles are assigned in order of their priority

Definition 6.5 (Role Assignment)**.** A role assignment is a mapping from the set of agents $\mathcal{A}$ to the set of roles $\mathcal{R}$. Let $\vec{R} = (r_1, \ldots, r_n)$ be a vector containing all roles in $\mathcal{R}$, such that

$$\begin{aligned} &\forall_{1 \leq i < j \leq n} (\mathrm{Pref}_{r_i}, \mathrm{Cap}_{\mathrm{req}_{r_i}}, Prio_{r_i}) = r_i \wedge (\mathrm{Pref}_{r_j}, \mathrm{Cap}_{\mathrm{req}_{r_j}}, Prio_{r_j}) = r_j \\ &\qquad \wedge Prio_{r_i} > Prio_{r_j} \end{aligned}$$

holds. Using this vector, role assignment can be recursively defined as a set $\mathcal{RA}$:

$$\mathcal{RA} = \left\{ (r_i, a) \mid (a \in \mathcal{A}' = \mathcal{A} - \{b | (r_j, b) \in \mathcal{RA}, j < i\}) \wedge a = \underset{c \in \mathcal{A}'}{\operatorname{argmax}} \mathcal{U}_{r_i}(c) \right\}$$

Note that this procedure requires $\mathcal{R}$ to be at least as large as $\mathcal{A}$ in order to assign a role to each agent.

Example 6.1 (Example for Role Assignment)**.** *Let a and b two agents such that Agent a provides the capabilites*

$$\mathrm{Cap}_{\mathrm{prov}}(a) = \{(Speed, low), (Kicker, true), (Width, large)\}$$

and b the capabilities

$$\mathrm{Cap}_{\mathrm{prov}}(b) = \{(Speed, high), (Kicker, true), (Width, small)\}$$

Furthermore, let Attacker, Defender and Goalie be three roles in $\mathcal{R}$:

$$\begin{aligned}
\mathit{Attacker} &= (\mathrm{Pref}_{\mathit{Attacker}}, \mathrm{Cap_{req}}_{\mathit{Attacker}}, \mathit{Prio}_{\mathit{Attacker}})\\
\mathit{Defender} &= (\mathrm{Pref}_{\mathit{Defender}}, \mathrm{Cap_{req}}_{\mathit{Defender}}, \mathit{Prio}_{\mathit{Defender}})\\
\mathit{Goalie} &= (\mathrm{Pref}_{\mathit{Goalie}}, \mathrm{Cap_{req}}_{\mathit{Goalie}}, \mathit{Prio}_{\mathit{Goalie}})
\end{aligned}$$

Suppose the corresponding priorities are $\mathit{Prio}_{\mathit{Attacker}} = 1$, $\mathit{Prio}_{\mathit{Defender}} = 0.5$ and $\mathit{Prio}_{\mathit{Goalie}} = 0.7$. Furthermore, the corresponding required capabilities are:

$$\begin{aligned}
\mathrm{Cap_{req}}_{\mathit{Attacker}} &= \{(\mathit{Speed}, \mathit{high}, 0.5), (\mathit{Kicker}, \mathit{true}, 1)\}\\
\mathrm{Cap_{req}}_{\mathit{Defender}} &= \{(\mathit{Kicker}, \mathit{true}, 0.5)\}\\
\mathrm{Cap_{req}}_{\mathit{Goalie}} &= \{(\mathit{Kicker}, \mathit{true}, 1), (\mathit{Width}, \mathit{large}, 0.5)\}
\end{aligned}$$

For simplicity, assume $\Delta(x, y) = \begin{cases} 1 & \textit{if } x = y \\ 0 & \textit{otherwise} \end{cases}$

Then, after Definition 6.4 the utility of the agents for the roles is as following:

- *The utility of agent a for role Attacker is* $\mathcal{U}_{\mathit{Attacker}}(a) = \frac{0.5\cdot 0+1\cdot 1}{2} = 0.5$
- *The utility of agent b for role Attacker is* $\mathcal{U}_{\mathit{Attacker}}(b) = \frac{0.5\cdot 1+1\cdot 1}{2} = 0.75$
- *The utility of agent a for role Defender is* $\mathcal{U}_{\mathit{Attacker}}(a) = \frac{0.5\cdot 1}{1} = 0.5$
- *The utility of agent b for role Defender is* $\mathcal{U}_{\mathit{Attacker}}(b) = \frac{0.5\cdot 1}{1} = 0.5$
- *The utility of agent a for role Goalie is* $\mathcal{U}_{\mathit{Goalie}}(a) = \frac{1\cdot 1+0.5\cdot 1}{2} = 0.75$
- *The utility of agent b for role Goalie is* $\mathcal{U}_{\mathit{Goalie}}(b) = \frac{1\cdot 1+0.5\cdot 0}{2} = 0.5$

After Definition 6.5, agent a is assigned Goalie and agent b is assigned Attacker. The role Defender is not assigned to any robot, as only two agents are available and as this role has the lowest priority.

6.2 Plan Base

The plan base of an agent captures its current intentional state. Specifically, it denotes which states the agent inhabits for each plan it participates in and which tasks it committed to. Hence, each element holds a procedural represented intention.

Definition 6.6 (Plan Base)**.** An agent's plan base is a set of triples (p, τ, z), consisting of a plan p, a task τ and a state z. The plan base of an agent a is denoted by $\mathrm{PBase}(a)$. A plan base contains at most one triple for each plan p.

If (p, τ, z) is an element of $\mathrm{PBase}(a)$ for an agent a, we say a participates in p (or executes p), is committed to task τ and inhabits state z. We define the following macro over plan bases, which captures the hierarchical structure embedded in it:

Definition 6.7. $\mathrm{Plans}^*(\Upsilon, z)$ denotes the set of plans that are executed by an agent with plan base Υ in the context of z. It is defined as:

$$p \in \mathrm{Plans}^*(\Upsilon, z) \text{ iff } (\exists \tau, z')(p, \tau, z') \in \Upsilon \wedge p \in \mathrm{Plans}^*(z)$$

6.3 Belief Base

Let $\mathcal{L}(Pred, Func)$ be the language of the belief base. $\mathcal{L}$ is a first-order logic[3] extended by the modal operators Bel_a and K_a, for each agent a in $\mathcal{A}$.

The operator Bel_a expresses individual belief of agent a. $\mathrm{Bel}_a\,\phi$ denotes that agent a believes ϕ. Formally and according to the knowledge axioms by Fagin et al. [7], Bel is defined (as KD45 system) by the following axioms:

- $(\mathrm{Bel}_a\,\phi \to \psi) \to (\mathrm{Bel}_a\,\phi \to \mathrm{Bel}_a\,\psi)$ (Distribution Axiom)
- $\mathrm{Bel}_a\,\phi \to \neg\,\mathrm{Bel}_a\,\neg\phi$ (Consistency Axiom)
- $\mathrm{Bel}_a\,\phi \to \mathrm{Bel}_a\,\mathrm{Bel}_a\,\phi$ (Positive Introspection Axiom)
- $\neg\,\mathrm{Bel}_a\,\phi \to \mathrm{Bel}_a\,\neg\,\mathrm{Bel}_a\,\phi$ (Negative Introspection Axiom)
- $(\forall x)\,\mathrm{Bel}_a\,\phi(x) \to \mathrm{Bel}_a((\forall x)\phi(x))$ (Knowledge Quantifier)

These axioms form the notion of strongly rational belief. The modality K extends Bel towards knowledge: $\mathrm{K}_a\,\phi \overset{def}{=} (\mathrm{Bel}_a\,\phi) \land \phi$.

On top of individual belief, we use the usual notions of "everyone beliefs", EBel, and mutual belief, MBel. Everyone beliefs is defined by (after Rao et al. [26]):

$$\mathrm{EBel}_A\,\phi \overset{def}{=} (\forall a \in A)\,\mathrm{Bel}_a\,\phi$$

The formula $\mathrm{EBel}_A\,\phi$ is satisfiable if and only if all agents a in group A believe ϕ. The mutual belief of ϕ is defined as that all agents a of a group A believe ϕ and all of them believe ϕ mutually. $\mathrm{MBel}_A\,\phi$ is defined as the fixpoint of:

$$\mathrm{MBel}_A\,\phi = \mathrm{EBel}_A\,\phi \land \mathrm{EBel}_A\,\mathrm{MBel}_A\,\phi$$

This yields an infinite conjunction of the form:

$$\mathrm{MBel}_A\,\phi \leftrightarrow \mathrm{EBel}_A\,\phi \land \mathrm{EBel}_A\,\mathrm{EBel}_A\,\phi \land \ldots \land \mathrm{EBel}_A \ldots \mathrm{EBel}_A\,\phi \land \ldots$$

Meaning, everyone believes ϕ, believes that everyone believes ϕ, believes that everyone believes that everyone believes ϕ, and so on.

The set of terms in $\mathcal{L}(Pred, Func)$ contains certain ALICA specific terms, such as a constant for each agent in $\mathcal{A}$, and domain-specific terms, such as coordinates or terms representing physical objects. Formally, the set of terms in $\mathcal{L}(Pred, Func)$ is given by:

- A countable infinite set of variables, $X = \{x_1, x_2, \ldots, x_n, \ldots\}$,
- A set of n-ary function symbols ($n \geq 0$), *Func*. *Func* contains:
  - $\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, $\mathcal{P}$, $\mathcal{T}$,
  - a domain-specific set of function symbols, F_{dom}.

The set of predicates in $\mathcal{L}(Pred, Func)$, *Pred* contains:

[3] In principle, ALICA can also work with a propositional modal logic, since the set of ground terms is finite. However, compiling an ALICA program into a propositional logic causes $\mathcal{P}$ to grow exponentially.

- $\mathrm{In}(a, p, \tau, z)$, defined to hold iff $(p, \tau, z) \in \mathrm{PBase}(a)$. This allows an agent to reason about its beliefs about the internal states of other agents. For instance, $\mathrm{Bel}_a \, \mathrm{In}(b, p, \tau, z)$ denotes that a believes b to be committed to τ in p and that b is currently in state z.
- $\mathit{HasRole}(a, r)$, holding the mapping of an agent a to its roles r,
- $\mathit{Handle}_f(b, z)$, which is meant to hold if an agent should handle the failure of behaviour b in state z,
- $\mathit{Handle}_f(p)$, which is meant to hold if an agent should handle the failure of plan p,
- $\mathit{Failed}(p, i)$, indicating that plan p failed i-times,
- $\mathit{Failed}(b, z, i)$, indicating that behaviour b failed i-times in state z,
- $\mathrm{Succeeded}(a, p, \tau)$, true iff agent a successfully completed task τ in plan p,
- $\mathit{Alloc}(z)$, true iff an allocation of tasks to agents for state z is deemed necessary,
- $\mathit{Success}(b, z)$ and $\mathit{Fail}(b, z)$, indicating a success or failure signal from behaviour b, which is executed in state z, respectively,
- a domain-specific set of predicates, P_{dom}.

P_{dom} contains predicates relating to the world representation of the agent, e.g., $\mathit{DistanceTo}(\mathit{object}, \mathit{dist})$ or $\mathit{Carries}(\mathit{agent}, \mathit{object})$. The language elements introduced here allow agents to reason about the environment (with symbols in F_{dom} and P_{dom}), and about the internal states of itself and its teammates. Thus, calculations such as role assignment can be done with respect to an agent's beliefs.

In order to capture the relationship between the different predicates reflecting believes about internal states of agents, we define a set of axioms, Σ_{b}. Let Σ_{b} contain for each agent a in $\mathcal{A}$ the following:

- Unique Name Axioms over agents, behaviours, plans, and states:

$$UNA(\mathcal{A}, \mathcal{B}, \mathcal{P}, \mathcal{Z}, \mathcal{T})$$

- If failure handling for a behaviour is needed, it is relevant:

$$(\mathrm{Bel}_a \, \mathit{Handle}_f(b, z)) \rightarrow (\exists p, \tau) \, \mathrm{In}(a, p, \tau, z)$$

- If failure handling for a plan is needed, it is relevant:

$$\begin{aligned}(\mathrm{Bel}_a \, \mathit{Handle}_f(p) \vee \mathit{Failed}(p, i)) \quad \rightarrow \quad & p = p_0 \vee (\exists p', z, \tau) \, \mathrm{In}(a, p', \tau, z) \\ & \wedge p \in \mathrm{Plans}(z)\end{aligned}$$

- In the same way, if failure handling for a behaviour is needed, it is relevant:

$$(\mathrm{Bel}_a \, \mathit{Handle}_f(b) \vee \mathit{Failed}(b, z, i)) \rightarrow (\exists p, \tau) \, \mathrm{In}(a, p, \tau, z)$$

- 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:

$$\mathrm{Succeeded}(a, p, \tau) \rightarrow (\exists z) p \in \mathrm{Plans}(z) \wedge (\exists a', \tau', p') In(a', p', \tau', z)$$

- Task allocation is only needed for a state inhabited by the agent:

$$(\mathrm{Bel}_a\, Alloc(z)) \rightarrow (\exists p, \tau)\, \mathrm{In}(a, p, \tau, z)$$

Definition 6.8 (Common Knowledge)**.** Let Σ_B be the set given by:

$$\Sigma_\mathrm{B} = \Sigma_\mathrm{syn} \cup \Sigma_\mathrm{dom} \cup \Sigma_b$$

where Σ_dom is a set of domain-specific axioms, describing the domain, Σ_b is the set of axioms regarding an agent belief and Σ_syn is the set of syntactic constraints (see Section 5.3). Σ_B is assumed to be common knowledge in $\mathcal{A}$, i.e., $\Sigma_\mathrm{B} \wedge \mathrm{MBel}_\mathcal{A}\, \Sigma_\mathrm{B}$ holds.

Definition 6.9 (Belief Base)**.** A set of formulae $B \subset \mathcal{L}(Pred, Func)$ is a *belief base* for agent a iff

$$\Sigma_\mathrm{B} \cup B \not\models \bot$$

and

$$\mathrm{In}(a, p, \tau, z) \in B \leftrightarrow (p, \tau, z) \in \mathrm{PBase}(a)$$

Thus, an agent's belief base reflects its belief about the world as well as its belief about all other agents' internal states, i.e., plan bases. The above definition results in a belief base that reflects the intuition that an agent always believes it does what it intentionally is doing. Moreover, the belief base is always consistent wrt. Σ_B. By BelBase(a) we denote the belief base of agent a.

Definition 6.10 (Agent Proof)**.** Let a be an agent, $\mathcal{F}$ be a set of formulae, η a substitution, and ϕ a formula. Then we denote that a proves ϕ wrt. $\mathcal{F}$ and Σ_B by

$$\mathcal{F} \vdash^a_\eta \phi$$

where η is the computed answer of the proof. If it is clear from the context, we omit a or η.

6.4 Belief Update

The belief base of an agent is updated frequently, either to accommodate for new sensory data, communication acts or internal updates. Here, we only treat the last case explicitly. We write $B + F$ to denote that the belief base B is updated by the (finite) set of formulae $F = \{f_1, f_2, \ldots, f_n\}$.

$$\begin{aligned} B + F &\overset{def}{=} B + \bigwedge_{f \in F} f \\ B - F &\overset{def}{=} B + \bigwedge_{f \in F} \neg f \end{aligned}$$

We require the belief update operator + to satisfy the KM-postulates [18] U1 - U4, and U8, adopted to accommodate for the static common knowledge, Σ_{B}, and thus + forms an inertial basic update operator (after Lang [19]):

$$\Sigma_{\mathrm{B}} \cup (B + f) \models f \quad \text{(U1)}$$

$$\Sigma_{\mathrm{B}} \cup B \models f \rightarrow (B + f) \leftrightarrow B \quad \text{(U2)}$$

$$\text{If } \Sigma_{\mathrm{B}} \cup B \text{ and } f \text{ are both satisfiable then } \Sigma_{\mathrm{B}} \cup (B + f) \text{ is also satisfiable} \quad \text{(U3)}$$

$$\text{If } B \leftrightarrow C \text{ and } f \leftrightarrow g \text{ then } B + f \leftrightarrow C + g \quad \text{(U4)}$$

$$(B \cup C) + f \leftrightarrow (B + f) \cup (C + f) \quad \text{(U8)}$$

Most importantly, the belief base must be consistent at all times. Note that consistency wrt. Σ_{b} can easily be established by removing literals of the form $Handle_f(p, z)$, $Handle_f(b)$, $Failure(p, i)$, $\mathrm{Succeeded}(a, p, \tau)$, $Alloc(z)$, $Success(b, z)$, and $Fail(b, z)$.

Assume for example an agent a executes a plan p, $In(a, p, \tau, z)$ holds in $\mathrm{BelBase}(a)$, and the agent aborts the execution of p, due to a reaction on a higher level in the plan hierarchy. Then $\mathrm{BelBase}(a)$ is update: $\mathrm{BelBase}(a)' = \mathrm{BelBase}(a) - In(a, p, \tau, z)$. If for example an allocation for state z was pending, $Alloc(z) \in \mathrm{BelBase}(a)$, it is no longer relevant, and hence $Alloc(z)$ is removed as well.

6.5 Team Configuration

In ALICA, each agent constantly monitors the actions of its team members wrt. the plans it participates in. Hence, an agent can not only react to another agent breaking down and consequently being removed from the team, but also each agent considers the progress of all other agents when making a decision, such as committing to a task within a plan.

For this notion of monitoring, we first define what it means for a team to be working on a plan.

Definition 6.11. Let a be an agent in A and let $\mathrm{TeamIn}(A, p)$ denote that team $A \subseteq \mathcal{A}$ executes a plan p, formally:

$$\begin{aligned} \mathrm{TeamIn}(A, p) \overset{def}{=} \; & (\forall \tau \in \mathrm{Tasks}(p))(\exists n_1, n_2)\, \xi(p, \tau) = (n_1, n_2) \wedge \\ & (\exists A')A' \subseteq A \wedge n_1 \leq |A'| \leq n_2 \wedge \\ & (\forall a' \in A)a' \in A' \leftrightarrow (\exists z)\, \mathrm{In}(a', p, \tau, z) \vee \mathrm{Succeeded}(a', p, \tau) \end{aligned}$$

If $A = \mathcal{A}$, we abbreviate $\mathrm{TeamIn}(\mathcal{A}, p)$ by $\mathrm{TeamIn}(p)$.

This definition is non-monotonic in the sense that there can be two sets A, A' with $A' \supseteq A$ such that $\mathrm{TeamIn}(A, p)$ and $\neg\,\mathrm{TeamIn}(A', p)$. Hence, an agent's assumptions regarding its team are vital for its evaluation of plans.

6.6 Utility Functions

We already introduced a certain kind of utility functions, namely role utilities in Section 6.1. Here, we are concerned with functions evaluating plans wrt. situations. These utility functions are pivotal to the way an ALICA program is executed.

Definition 6.12 (Utility Function)**.** The utility $\mathcal{U}_p(\mathcal{F})$ of a plan p wrt. belief base $\mathcal{F}$ has the form:

$$\mathcal{U}_p(\mathcal{F}) = \begin{cases} -1 & \text{if } pri(\mathcal{F}) < 0 \\ w_0\, pri(\mathcal{F}) + \sum_{1 \leq i \leq n} w_i\, f_i(\mathcal{F}) & \text{if } \mathcal{F} \models \mathrm{TeamIn}(p) \\ 0 & \text{otherwise} \end{cases}$$

A utility function is a weighted sum of several functions $pri, f_0 \dots f_n$ over belief states. The function pri evaluates the preferences of all agents involved in executing plan p towards their current tasks in p:

$$pri(\mathcal{F}) = \begin{cases} -1 & \text{if } (\exists a)\phi[\tau, \mathrm{Pref}] \wedge \mathrm{Pref}(\tau) < 0 \\ \sum_{\phi[\tau,\mathrm{Pref}]} \mathrm{Pref}(\tau) & \text{otherwise} \end{cases}$$

where $\phi[\tau, \mathrm{Pref}] = HasRole(a, (\mathrm{Pref}, \mathrm{Cap}_{\mathrm{req}}, Prio)) \in \mathcal{F} \wedge \mathrm{In}(a, p, \tau, z) \in \mathcal{F}$. The functions f_i are arbitrary functions over belief bases. However, we require that $(\forall f_i, w_i) w_i f_i(\mathcal{F}) \geq 0$ for all belief bases $\mathcal{F}$.

We refer to the belief which agent executes which task in a certain plan p as an *allocation* of plan p. A formal definition is given in Section 6.7. Intuitively, a utility function reflects the value of a plan together with an allocation in a certain situation. The value such a combination has for a team is always positive, unless the combination is deemed useless, in which case the utility should evaluate to zero. This is the case if the assumptions $\mathcal{F}$ do not satisfy all requirements the cardinalities pose on an allocation, i.e., if $\mathrm{TeamIn}(p)$ does not follow from $\mathcal{F}$.

Moreover, if a preference of a role r assigned to an agent a towards a task τ is negative and a is allocated to τ, the whole utility is evaluated to -1. This ensures that an agent will not under any circumstances take on a task with negative preference. Instead, it would do nothing, which yields a utility of at least 0. The strict rule employed here is very important in order to describe roles (and thus, agents) which cannot take on certain tasks. For instance, fragile avionic robots should never commit to tasks which require physical manipulation of objects in order to keep themselves from harm.

Definition 6.13. The utility of a set of plans, $\vec{p} = (p_1, \dots, p_n)$, $\mathcal{U}_{\vec{p}}(\mathcal{F})$ is a vector:

$$\mathcal{U}_{\vec{p}}(\mathcal{F}) \overset{def}{=} (\mathcal{U}_{p_1}(\mathcal{F}), \dots, \mathcal{U}_{p_n}(\mathcal{F}))$$

Example 6.2 (Simple Utility Function)**.** *The following formula shows a simple example of a utility function.*

$$\mathcal{U}_p(\mathcal{F}) = 0.15 \cdot pri(\mathcal{F}) + 0.8 \cdot DistBallAttack(\mathcal{F}) + 0.05 \cdot DistGoalDefend(\mathcal{F})$$

with $pri(\mathcal{F})$ as defined in Definition 6.12,

$$DistBallAttack = \max_{\mathrm{In}(a,p,Attack,z) \in \mathcal{F}} \frac{1 - Dist(Ball, a)}{FieldDiagonal}$$

and

$$DistGoalDefend = \max_{\mathrm{In}(a,p,Defend,z) \in \mathcal{F}} \frac{1 - Dist(OwnGoal, a)}{FieldDiagonal}$$

The utility function in the example consists of three summands:

- $Pri(\mathcal{F})$ evaluates the preferences of each participating agent given by their roles,
- *DistBallAttack* depends on the distance between the ball and the robot committed to task *Attack*,
- $DistBallDefend$ evaluates the distance between the own goal and the robot committed to task $Defend$.

As such, this utility function highly depends on the belief base of the evaluating agent $\mathcal{F}$. In the next section, we discuss how these utility functions are used to choose a plan from a plantype and to allocate agents to tasks within plans.

6.7 Task Allocation

Whenever an agent enters a state, it has to decide for all plans within that state, to which, if any, task it commits to. That is, it has to calculate a task allocation for all participating agents. Ideally, every agent involved computes the same task allocation. Within ALICA, the degree of communication involved in establishing these allocations can be explicitly modelled. Here, we model the most basic case, which involves no communication during task allocation.

A task allocation C for plan p is a set of formulae of the form $\mathrm{In}(a_i, p, \tau, z)$, one for each agent a_i allocated to participate actively in p. For each $\mathrm{In}(a_i, p, \tau, z)$ in C, τ is a task relevant for p, i.e., an element of $\mathrm{Tasks}(p)$, and z is the initial state of τ in p, $\mathrm{Init}(p, \tau)$.

In order for an agent to commit to a task, it has to compute such a task allocation. Under the weak commitment used here, it is not guaranteed that all involved agents compute the same allocation, since their belief bases may differ. These conflicting allocations however can be resolved during the execution of the plan, see Section 6.9.2.

An agent a with configuration (B, Υ, E, θ) computes a task allocation C for plan p under a valid belief base $\mathcal{F}$, which acts as an assumption, such that C satisfies the following constraints:

- $\Sigma_{\mathrm{B}} \cup \mathcal{F} \cup C \not\models \bot$
- $\Sigma_{\mathrm{B}} \cup \mathcal{F} \cup C \vdash_\eta (\mathrm{Pre}(p) \wedge \mathrm{Run}(p))\theta$
- $\Sigma_{\mathrm{B}} \cup \mathcal{F} \cup C \vdash \mathrm{TeamIn}(p)$
- $\mathcal{U}_p(\mathcal{F} \cup C) \geq 0$

such that $\mathcal{U}_p(\mathcal{F} \cup C)$ is maximised.

For a plan p and agent a we denote a task allocation satisfying the above conditions by $\mathrm{TAlloc}(a, p|\mathcal{F})$. $\mathcal{F}$ acts a set of assumptions describing the belief state for which the allocation should be valid. If an agent allocates wrt. the current state, $\mathcal{F}$ should equal its belief base. Hence, an agent believed to be already committed to a task in p will be considered. This allows for a dynamic repair in case some agents withdraw from a plan and need to be replaced.

Additionally, a task allocation defines a corresponding substitution, $\sigma_{(a,p,\mathcal{F})}$, called *allocation substitution* of task allocation $\mathrm{TAlloc}(a, p|\mathcal{F})$. $\sigma_{(a,p,\mathcal{F})}$ is the application of η, the computed answer of the proof of the precondition and runtime condition of p, to $\rho(p)$.

$$\sigma_{(a,p,\mathcal{F})} \overset{def}{=} \rho(p) \circ \eta$$

Based on this notion of task allocation, we define a recursive version, which allocates agents to a branch of plans in the plan tree. Intuitively, whenever an agent enters a state and computes a task allocation for a subplan, it has to enter the initial states of the tasks it allocated itself to. Hence, it also needs to compute a task allocation for those initial states and so on.

Definition 6.14 (Ordered Plan List)**.** We call a list of plans $\vec{P} = p_1, \ldots, p_n$ ordered wrt. the plan tree of a valid ALICA program iff $(\forall p_i, p_j \in \vec{P})(\exists z')z' \in \mathrm{States}(p_i) \wedge p_j \in \mathrm{Plans}^*(z) \rightarrow i < j$.

Proposition 1. *Given a valid ALICA program, for all subsets P of $\mathcal{P}$, there is an ordered list $\vec{P}$ containing precisely all elements of P.*

Proof. Let $p < p'$ denote that $(\exists z')z' \in \mathrm{States}(p) \wedge p' \in \mathrm{Plans}^*(z)$. Since the empty list is ordered wrt. any plan tree and since from Syntactic Axiom A5.7 it follows that for all plans p and p', $p < p' \rightarrow \neg p' > p$ the claim holds. □

Definition 6.15 (Recursive Task Allocation)**.** Given an agent a with configuration $(B, \Upsilon, E, \theta, \mathrm{R})$, a valid belief base $\mathcal{F}$, a recursive task allocation for state z, $\mathrm{TAlloc}^*(a, z|\mathcal{F})$, is a set of formulae of the form $\mathrm{In}(a', p_i, \tau, z')$. Let $\vec{P} = p_1, p_2, \ldots, p_n$ be the ordered plan list containing exactly all plans p_i such that $\mathrm{In}(a', p_i, \tau, z')$ is in $\mathrm{TAlloc}^*(a, z|\mathcal{F})$ for some agent a', some task τ, and some initial state z'. Furthermore, let $\vec{b}$ be the set of behaviours such that $b \in \vec{b} \leftrightarrow \mathrm{In}(a, p_i, \tau, z') \in \mathrm{TAlloc}^*(a, z|\mathcal{F}) \wedge b \in \mathrm{Behaviours}(z')$ and let $\mathcal{G}$ denote $\mathrm{TAlloc}^*(a, z|\mathcal{F}) \cup \mathcal{F}$.

Then $\mathrm{TAlloc}^*(a, z|\mathcal{F})$ is valid iff:

$$\Sigma_{\mathrm{B}} \cup \mathcal{G} \not\models \perp \tag{1}$$

$$\Sigma_{\mathrm{B}} \cup \mathcal{G} \vdash_\eta \left(\left(\bigwedge_{p \in \vec{p}} \mathrm{Pre}(p) \wedge \mathrm{Run}(p) \right) \wedge \right. \tag{2}$$

$$\left. \left(\bigwedge_{b \in \vec{b}} \mathrm{Pre}(b) \wedge \mathrm{Run}(b) \right) \right) \theta\, \rho(p_1) \ldots \rho(p_n)$$

$$(\forall p) p \in \vec{P} \rightarrow \mathcal{G} \vdash \mathrm{TeamIn}(p) \tag{3}$$

$$\begin{aligned}(\forall p) p \in \vec{P} \rightarrow & p \in \mathrm{PlanTypes}^*(z)(\mathcal{G}) \\ & \wedge (\exists p', \tau', z')\, \mathrm{In}(a, p', \tau', z') \in \mathcal{G} \\ & \wedge p \in \mathrm{PlanTypes}(z')(\mathcal{G})\end{aligned} \tag{4}$$

$$\begin{aligned}&(\forall a', p', \tau', z')\, \mathrm{In}(a', p', \tau', z') \\ &\in \mathrm{TAlloc}^*(a, z|\mathcal{F}) \rightarrow z' = \mathrm{Init}(p', \tau')\end{aligned} \tag{5}$$

$$\begin{aligned}&(\forall a', p, \tau', z')\, \mathrm{In}(a', p, \tau', z') \\ &\in \mathrm{TAlloc}^*(a, z|\mathcal{F}) \rightarrow (\exists p', \tau'', z'')\, \mathrm{In}(a', p', \tau'', z'') \in \mathcal{G} \wedge \\ &\qquad p' \in \mathrm{PlanTypes}(z'')(\mathcal{G})\end{aligned} \tag{6}$$

$$\begin{aligned}&(\forall p')\, \mathrm{In}(a', p', \tau', z') \\ &\in \mathrm{TAlloc}^*(a, z|\mathcal{F}) \rightarrow \mathcal{U}_{p'}(\mathcal{G}) \geq 0\end{aligned} \tag{7}$$

$$\tag{8}$$

By Condition 1, a valid task allocation has to be consistent with the assumptions and the common knowledge. Hence, by Definition 6.6 and 6.9, an agent cannot be assigned two conflicting tasks. In particular, an agent already believed to be participating in a plan p cannot be reassigned. Condition 2 ensures that all preconditions and runtime conditions of plans assigned are met and yields an answer substitution an agent uses later on. Furthermore, the yielded answer substitution are used in the *allocation substitution* (see Definition 6.16). Condition 3 rules out partially assigned plans, i.e., enforces that each plan is executed by a proper number of agents. Condition 4 limits the task allocation to plans mapped onto by plantypes occurring in states the allocation agent enters by adopting the allocation.

By Condition 5 each agent that is newly allocated to a task is believed to be in the corresponding initial state. Condition 6 requires a task allocation to be complete in the sense that an agent allocated to a plan is either allocated to or was already believed to be in the state that contains the corresponding plantype. Finally, Condition 7 ensures that all resulting utility values are not below 0 and hence the allocation is not considered harmful.

Condition 4 enforces the allocating agent to take on a "local view" during task allocation, as it does not take into account plans other agents may be forced to enter by adopting the allocation, but assumes an outcome that is consistent with its calculations. Obviously, this can lead to inconsistencies between the different local views of the participating agents. For instance, it might be the case that a subplan not within an agent's local view cannot be executed for some reason, i.e., due to an insufficient number of agents. Since such inconsistencies cannot be ruled out completely due to potentially different belief bases, we tolerate them here instead of enforcing agents to calculate a more computationally expensive global view. Section 6.9.2 explains how such inconsistencies are handled. Additionally, it is possible to alleviate this problem at compile time by propagating cardinalities and preconditions upwards the plan tree.

The precise choice of $\text{TAlloc}(a, z|\mathcal{F})^*$ is subject to a set of utility functions:

$$\{\mathcal{U}_p(\mathcal{F} \cup \text{TAlloc}^*(a, z|\mathcal{F}))|\, \text{In}(a', p, \tau, z') \in \text{TAlloc}^*(a, z|\mathcal{F})\}$$

We leave the way this optimisation problem is handled open. In our implementation, a greedy backtracking algorithm is used to search through the plan tree, starting from the top-most plans. In each step, a plan and an allocation for that plan is chosen from a single plantype, such that the corresponding utility is maximised globally.

Definition 6.16 (Recursive Task Substitution)**.** Let $\vec{P} = p_1, p_2, \ldots, p_n$ be the ordered list of all plans allocated in $\text{TAlloc}^*(a, z|\mathcal{F})$, and η be the answer substitution for the proof of the pre- and runtime conditions. Then, the *allocation substitution* of $\text{TAlloc}^*(a, z|\mathcal{F})$, $\sigma^*_{(a,z,\mathcal{F})}$ is defined by:

$$\sigma^*_{(a,z,\mathcal{F})} \overset{def}{=} \rho(p_1) \circ \ldots \circ \rho(p_n) \circ \eta$$

Example 6.3 (Recursive Task Allocation)**.** *Suppose three agents, $\mathcal{A} = \{a, b, c\}$, have to compute an allocation for plan p_1 depicted in Figure 7. For simplicity, assume all plantypes contain just a single plan. Furthermore, assume that for all tasks involved, at least one agent needs to commit to it in order to satisfy Condition 3 of Definition 6.15.*

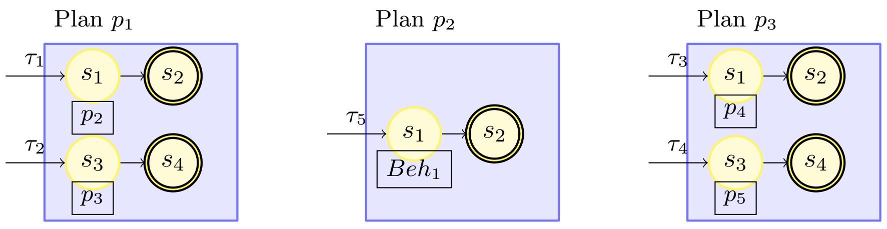

Figure 7: Example: Plans

Figure 8 shows the procedure of the recursive task allocation from the perspective of agent a. First the task allocation for the initial plan p_1 is computed. Suppose, the best allocation for this plan and its tasks is as following:

- *τ_1: c,*
- *τ_2 : a, b.*

Since agent a has allocated itself to task τ_2, $\text{In}(a, p_1, \tau_2, s_3)$ has to be a member of its task allocation set. Hence, a has to allocate itself, along with b to plan p_3, the single member of $\text{Plans}(s_3)$. However, a does not consider plan p_2, which c has to execute according to the calculation so far. For p3*, suppose, a calculates:*

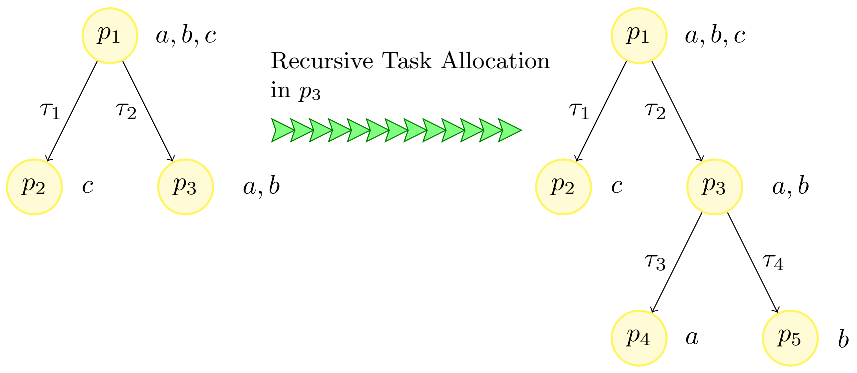

Figure 8: Example: Recursive Task Allocation

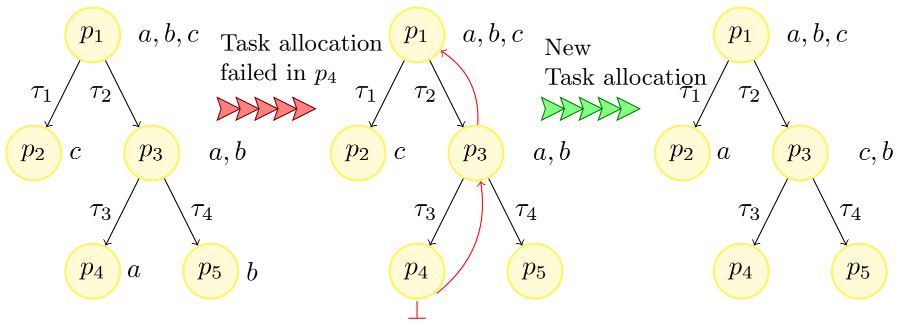

Figure 9: Task Allocation Failure

- $\tau_3 : a$,
- $\tau_4 : b$.

Now, a has to allocate itself within p_4. Assume, it cannot find an allocation satisfying the corresponding preconditions. In this case, backtracking occurs (see the tree in the middle of Figure 9). The tree on the right hand side shows the final task allocation, where agent a has swapped places with agent c within plan p_1. Note, that a has not allocated b and c within p_3. This reflects the local view of a on this plan tree.

6.8 Communication

In the following section, we will introduce a transition rule system, which describes how agent configurations are changed while an agent executes an ALICA program. For the transition system to work, agents must either be able to track each other through the plan tree, by modelling their configuration, or the agents have to communicate the configurations.

We think that in any practical setting one would adopt both ways to a certain degree. Communication would be employed as often as it is feasible and tracking wherever communication is unreliable, costly, or takes too much time. Here, we assume that the agents inform each other periodically of their plan base and, if applicable, their success status. This can introduce inconsistencies since communication is unreliable and delayed. Most importantly, agents have to take decisions in between communication periods and in many cases cannot wait for a communication act to be acknowledged. However, periodic communication guarantees that each agent is able to update its agent model eventually, assuming the probability of a sent message being received in finite time is different from 0. Tracking can be used to fill the gaps in between messages.

In this work, we only present a very limited way of tracking, which we deem sufficient if most communication attempts succeed and the communication period is not too large in comparison with the duration of the deliberation cycle of an agent. Future work will evaluate this assumption and discuss tracking abilities in more detail.

Each agent a communicates a set of literals of the form $\mathrm{In}(a, p, \tau, z)$ and $\mathrm{Succeeded}(a, p, \tau)$ periodically. Each literal is communicated iff it is an element of $\mathrm{BelBase}(a)$. Note also that each agent only communicates its own internal status, not the status of other agents. In domains, where communication range is limited and agents have to relay information to each other, these communication messages need to be extended. A received message of this kind is treated the same way as new sensor information, and update the receiver's belief base accordingly. Hence, each agent is assumed to be truthful.

6.9 Transition Rules

Transition rules define how an agent's configuration changes during a single computation step. Each rule is guarded by a condition and transforms a given configuration into a new one:

$$RuleName\colon \frac{Conditions}{CurrentAgentConfigurations \longrightarrow NewAgentConfigurations}$$

In the following a denotes the agent subject to the transition system. A rule is applied only if its condition is satisfied, agent configuration $\mathrm{Conf}(a)$ unifies with the left hand side of the transition and no rule with higher precedence is applicable. If rule r_1 has a higher precedence than r_2, we write $r_1 > r_2$. The precedence relation over rules, $>$, is transitive, irreflexive, and antisymmetric.

We distinguish two kinds of transition rules, *operational* rules, which describe an agent's normal operation and *repair* rules, which provide means to recover from a failure. Execution of an ALICA program begins with the $Init$ rule. From there on, the rules $Trans$ and $STrans$ describe how an agent reacts to transitions within plans. $Trans$ captures the case of normal transitions, $STrans$ the case of synchronised transitions, which require establishing mutual belief between all involved agents. The application of both kind of rules is followed by the application of rule *Alloc*, which handles recursive task allocation for agents believed to be in the newly entered state. Finally, the rules $BSuccess$ and $PSuccess$ modify an agent's configuration due to success signals from a lower level (in case of $BSuccess$) and due to reaching a success state within a plan ($PSuccess$).

In principal, these six rules are sufficient to describe the operative behaviour of the agents involved, unless a failure occurs. Failure handling is done using repair rules. Here we present a set of ten repair rules, capturing different kinds of reaction to failures. Some rules presented here are not applicable in all domains, hence repair rules are somewhat domain dependent.

The rule *BAbort* stops a behaviour that is executed if it signals a failure. *BRedo* and *BProp* handle this failure. *BRedo* tries to re-execute a failed behaviour if possible, while *BProp* propagates the failure upwards to the plan in whose context the failed behaviour was executed. *PAbort* acts similar towards plans as *BAbort* does towards behaviours, it stops the failed plan and all plans and behaviours executed in its context. However, it can be overridden by *PRedo* which resets the agent's state within a failed plan if possible. This avoids computional and possibly communication overhead, as the agent continues to work on its task, and does not calculate a new allocation.

In case such a "soft" restart through *PRedo* is impossible and *PAbort* stops a plan, *PReplace* triggers a new task allocation. A new task allocation can also choose an alternative plan from the corresponding plan type. If all other means of failure handling are exhausted, *PProp* propagates a failure upwards to the parent plan. Finally, *PTopFail* captures the case where the top-level plan has failed, and simply triggers a clear initialisation through *Init*.

There are two special rules, *NExpand* which triggers a failure if a due task allocation cannot be performed and *Replan* which is used to periodically check the utility of a task allocation, and triggers a new task allocation if the current utility is deemed to be unsatisfying. *Replan* thereby allows for highly dynamic changes in the allocation and thus accommodates for swift changes in the environment.

6.9.1 Operational Rules

Operational rules always take precedence over repair rules, following the idea that a failure might become irrelevant by a change in an agent's configuration. Repair rules are geared to preserve a certain status or provide alternatives for a currently pursued, but failed, intention.

The Initialisation Rule

$$Init : \frac{\Upsilon = \emptyset}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B + \{\mathrm{In}(a, p_0, \tau_0, z_0), alloc(z_0)\}, \{(p_0, \tau_0, z_0)\}, \emptyset, \emptyset, \mathrm{R})}$$

Intuitively, the Initialisation Rule obligates the agent to start the execution of the plan tree. This is due whenever an agent's planbase is empty, i.e., after start up and whenever an agent's plan base has been emptied completely due to plan failures.

The Trans Rule The first rule we discuss controls when and how an agent follows a transition from one state to another.

$$Trans : \frac{B \vdash_\eta \phi\theta \wedge (p, \tau, z) \in \Upsilon, (z, z', \phi) \in \mathcal{W} \wedge \neg(\exists s \in \Lambda)(z, z', \phi) \in s}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, (\Upsilon - \vartheta_p^-) + \vartheta_p^+, E', \theta \circ \eta, \mathrm{R})}$$

where

- $\vartheta_b^- = \{\mathrm{In}(a', p, \tau', z) | a' \in \mathcal{A}, \tau' \in \mathcal{T}\}$
  $\cup \{\mathrm{In}(a', p', \tau', z'') | a' \in \mathcal{A}, p' \in \mathrm{Plans}^*(z), z'' \in \mathrm{States}(p')\}$
- $\vartheta_b^+ = \{\mathrm{In}(a', p, \tau', z') | \, \mathrm{In}(a', p, \tau', z) \in \vartheta_b^-\} \cup \{alloc(z')\}$

- $\vartheta_p^+ = \{(p, \tau, z')\}$
- $\vartheta_p^- = \{(p, \tau, z)\} \cup \text{Plans}^*(\Upsilon, z)$
- $E' = E - \{(b, z) | (p, \tau, z) \in \vartheta_p^-\}$

That is, an agent will follow an outgoing transition from state z to z' if it currently resides in z and believes the condition ϕ annotating the transition to hold. Furthermore, this transition must not belong to a synchronisation set. Following a transition entails that the agent stops executing all plans and behaviour that are executed in the context of z, i.e., are in $\text{Plans}^*(P, z)$. The addition of $alloc(z')$ to the belief base encodes the need for a task allocation with respect to the newly entered state z'. Note that an agent applying this rule also assumes that every other agent currently in z applies it, i.e., believes its precondition. This realises a partial tracking of other agents through the plan tree. Furthermore, the computed answer of the proof of ϕ, η is appended to the runtime substitution θ.

The Synchronised Transition Rule handles a transition within a synchronisation set. Intuitively, a synchronisation models the start of a cooperative act that depends on the involved agents to act in a very small time-frame. The upper bound on the size of this time-frame depends on the latency and reliability of the communication and the precision with which agents can track their teammates' intentions. In the worst case, the condition guarding the Synchronised Transition Rule cannot be established.

$$STrans : \frac{(\exists A \subseteq \mathcal{A}) a \in A (\exists s \in \Lambda)(z, z', \phi) \in s \wedge \psi}{(B, \Upsilon, E, \theta, \text{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, (\Upsilon - \vartheta_p^-) + \vartheta_p^+, E', \theta \circ \eta, \text{R})}$$

where

- $\psi = (\forall (z'', z''', \phi_i) \in s)(\exists a' \in A) B \vdash_\eta \text{MBel}_A(\text{In}(a', p, \tau', z'') \wedge \phi^* \theta)$
- $(p, \tau, z) \in \Upsilon$
- $\phi^* = \bigwedge_{(z'', z''', \phi_i) \in s} \phi_i$
- $\vartheta_b^- = \{\text{In}(a', p, \tau', z) | a' \in A, \tau' \in \mathcal{T}\}$
  $\cup \{\text{In}(a', p', \tau', z') | a' \in A, p' \in \text{Plans}^*(z)\}$
- $\vartheta_b^+ = \{\text{In}(a', p, \tau', z') | a' \in A, \tau' \in \mathcal{T}\} \cup \{alloc(z')\}$
- $\vartheta_p^+ = \{(p, \tau, z')\}$
- $\vartheta_p^- = \{(p, \tau, z)\} \cup \text{Plans}^*(\Upsilon, z)$
- $E' = E - \{(b, z) | (p, \tau, z) \in \vartheta_p^-\}$

Here, an agent will follow a synchronised transition, if and only if it can identify a group A of agents it is part of, such that a believes that there is mutual belief in A that all relevant conditions ϕ_i hold. Moreover, a has to believe that there is mutual belief in A that all agents in a are in the correct states, that is, that every transition in the synchronisation set will be used by one agent.

Hence, there is mutual belief about the individual intentions to progress along the synchronised transitions. If the agent believes this condition to hold, it will act in the same manner as in the case of a normal Transition Rule.

Synchronised Transition Rules take precedence over normal Transition Rules, $STrans > Trans$. This is done following the intuition that a synchronisation guards a part of a plan that is of higher benefit to the team and more difficult to reach. It is easy to see that the condition of $Trans$ would subsume the condition of $STrans$ if it were not for the exclusion of synchronisations ($\neg(\exists s \in \Lambda)(z, z', \phi) \in s$).

The Allocation Rule takes over where the transition rules above left an agent. It causes a task allocation to be performed, usually in a state just entered.

$$Alloc : \frac{alloc(z) \in B \wedge \mathrm{In}(a, p', \tau', z) \in B \wedge \mathrm{TAlloc}^*(a, z|B) \neq \emptyset}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B - \{alloc(z)\}, \Upsilon + \vartheta_p^+, E + \vartheta_e^+, \theta', \mathrm{R})}$$

where

- $\theta' = \theta \circ \sigma_{(a,p,B)}$
- $\vartheta_b^+ = \mathrm{TAlloc}^*(a, z|B)$
- $\vartheta_p^+ = \{(p', \tau', z'')|\, \mathrm{In}(a, p', \tau', z'') \in \vartheta_b^+\}$
- $\vartheta_e^+ = \{(b(\vec{x})\theta', z)|(p, \tau, z) \in \vartheta_p^+ \wedge b \in \mathrm{Behaviours}(z)\}$

That is, if a believes a task allocation wrt. state z is needed and possible, it will update its believe base with the allocation $\mathrm{TAlloc}^*(a, z|B)$, insert all plans it is involved in into its plan base, and start to execute all behaviours relevant to the added plan-task-state-triples, if they are deemed possible. By appending $\sigma_{(a,p,B)}$ to the runtime substitution, the agent is bound to the consequences of the proved conditions during the recursive task allocation.

Since the result of an allocation is only relevant until the agent leaves the corresponding state, Transition Rules have a higher precedence than the Allocation Rule, $Trans > Alloc$.

Behaviour Success Rule We treat behaviours as atomic actions, which can fail or succeed at any point in time while running. Such a termination is reflected by the atoms $Success(b, z)$ and $Fail(b, z)$, where b is the behaviour in question and z the state in which it is called. These hold exactly if (b, z) is an element of the agent's behaviour base and b failed or succeeded in the last computational step, respectively.

$$BSuccess : \frac{Success(b, z) \in B}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', \Upsilon, E - \{(b, z)\}, \theta, \mathrm{R})}$$

where $B' = (B - \{Success(b, z)\}) + \{\mathrm{Post}(b)\theta\}$.

If a behaviour succeeds, it is stopped and its postcondition is assumed to hold. Since this rule processes a signal and updates the belief base accordingly, it takes precedence over the previous rules that modify the plan base, $BSuccess > STrans$.

Plan Success Rule A plan p succeeds iff a state $z \in \text{Success}(p)$ is reached.

$$PSuccess : \frac{(p, \tau, z) \in \Upsilon, z \in \text{Success}(p)}{(B, \Upsilon, E, \theta, \text{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, \Upsilon - \vartheta_p^-, E, \theta, \text{R})}$$

where

- $\vartheta_b^- = \{\text{In}(a', p, \tau', z) | a' \in \mathcal{A}, \tau' \in \mathcal{T}\}$
- $\vartheta_b^+ = \{\text{Succeeded}(a, p, \tau), \text{Post}(z)\theta\}$
- $\vartheta_p^- = \{(p, \tau, z)\}$

Similar to the Behaviour Success Rule, this rule updates the belief base with the post condition attached to the terminal state reached. Moreover, the successful completion of task τ in p is recorded as well, so the agent is not any longer committed to τ, although the success needs to be communicated to other agents working on p. Else, it might be the case that they abort the plan due to an insufficient number of agents working on it. (see Example 6.4). We give precedence to the success of lower level behaviours, but still treat this success rule with a higher priority than the Transition Rules, $BSuccess > PSuccess > STrans$.

Example 6.4 (Asynchronous Plan Success)**.** *This example shows the case that one task of a plan is finished before another task of that plan. As aforementioned the successful completion of a task is represented in the belief base and communicated to other agents. Figure 10 illustrates such a case. Initially the two agents a and b are allocated to the tasks τ_1 and τ_2 of plan p. Both tasks have an associated cardinality of* 1..1*, which means that exactly one agent must be allocated to each task for the execution of the plan. Both agents start with the initial state of their task within the plan. Agent a executes plan p*2 *and agent b plan p*3*. After some time, agent a progresses to the next state s*2*. Agent b leaves state s*3 *following the transition to the success state s*4*. Now agent b leaves the plan and therefore it is not longer committed to task τ_2. If the successful completion of task τ_2 were not recorded, the plan would be aborted as the cardinalities of the task are not satisfied. See also Definition 6.11.*

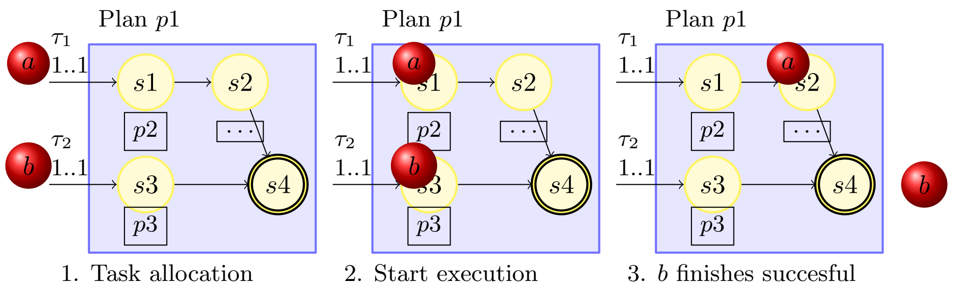

Figure 10: Example: Asynchronous Plan Success

6.9.2 Repair Rules

Typical BDI languages feature techniques handling failures that occur, e.g., due to unexpected changes in the dynamic environment agents act in. Classic BDI languages also distinguish between plan failures and goal failures. The first can be due to some side constraint being violated, the latter due to the goal itself becoming impossible to achieve. In ALICA, we have not yet introduced an explicit representation of goals, this is subject of future work.

Here, we limit ourselves to plan failures, which can happen frequently, depending on the scenario. For instance in robotic soccer, the domain is highly dynamic and each robot has to make assumptions about the status of its teammates, which can prove to be wrong. Repair Rules are special transition rules which are meant to recover from such failure. There are different ways to handle failures and a specific way can depend on the domain. A failed plan can be retried, replaced by an alternative, or the failure can be propagated up the plan tree. In some languages, such as AgentSpeak [25], a failure can even raise a specific goal, triggering custom plans meant to handle precisely the occurred failure.

ALICA features a similar way to deal with plan failures explicitly. A failed plan or behaviour is recognised and causes a corresponding believe to be inserted in the belief base ($Handle_f(p)$ for a plan, and $Handle_f(b, z)$ for a behaviour, respectively). Since the Transition Rule takes precedence over repair rules, an explicit mechanism can be modelled via a transition. Otherwise, default handling takes place.

The Behaviour Abortion Rule removes a behaviour from the behaviour base if its runtime condition is violated or a failure is signalled from a lower level process by $Fail(b, z)$.

$$BAbort : \frac{(b, z) \in E \wedge (B \nvdash \mathrm{Run}(b)\theta \vee Fail(b, z) \in B)}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', \Upsilon, (E - \{(b, z)\}), \theta, \mathrm{R})}$$

where

- $B' = (B - \{(\forall i)Failed(b, z, i), Fail(b, z)\})$ $+\{Failed(b, z, j), Handle_f(b, z)\}$,
- $j = \begin{cases} i+1 & \text{if } Failed(b, z, i) \in B, \\ 1 & \text{otherwise.} \end{cases}$

The belief $Failed(b, z, i)$ keeps track how many times a behaviour had to be aborted. This allows to consecutively apply different failure recovery rules. Note that if the corresponding state z is left, e.g., through a transition, this belief is dropped to keep the belief base consistent with Σ_{b}.

The Behaviour Repair Rules act as a default mechanism to handle behaviour failure.

$$BRedo : \frac{Handle_f(b, z) \in B \wedge B \vdash (\mathrm{Pre}(b) \wedge \mathrm{Run}(b))\theta\, \rho(b) \wedge Failed(b, z, 1) \in B}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', \Upsilon, E', \theta, \mathrm{R})}$$

where

- $E' = E + \{(b, z)\}$
- $B' = B - \{Handle_f(b, z)\}$

$$BProp : \frac{Handle_f(b, z) \in B \wedge Failed(b, z, i) \in B \wedge (i > 1 \vee B \nvdash (\mathrm{Pre}(b) \wedge \mathrm{Run}(b))\theta)}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', \Upsilon, E, \theta, \mathrm{R})}$$

where

- $B' = (B - \{Handle_f(b, z), Failed(b, z, i), (\forall k)Failed(p, k)\})$
  $+ \{Handle_f(p), Failed(p, j)\}$
- $(p, \tau, z) \in \Upsilon$
- $j = \begin{cases} j' + 1 & \text{if } Failed(p, j') \in B \\ 1 & \text{otherwise} \end{cases}$

BRedo restarts a failed behaviour if possible, *BProp* propagates the failure upwards if restarting has already been tried or is not possible. The universal quantification in $(\forall k)Failed(p, k)$ ensures that $Failed(p, k)$ is removed regardless of the current value of k. It is the case that at most one instance of $Failed(p, k)$ holds per plan p. Note that the applicability of these rules is subject to the concrete domain. For instance, retrying a failed behaviour might not make sense at all in certain scenarios. In others, a behaviour's success can be associated with a known probability distribution, in which case the utility of a retry can be estimated.

The Plan Abortion Rule is quiet similar to the Behaviour Abortion Rule:

$$PAbort : \frac{(p, \tau, z) \in \Upsilon \wedge (z \in \mathrm{Fail}(p) \vee B \nvdash \mathrm{Run}(p)\theta \vee B \vdash \neg\,\mathrm{TeamIn}(p))}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, \Upsilon - \vartheta_p^-, E - \vartheta_e^-, \theta, \mathrm{R})}$$

- $\vartheta_b^- = \{\mathrm{In}(a', p, \tau', z) | a' \in \mathcal{A}, \tau' \in \mathcal{T}\}$
  $\cup \{\mathrm{In}(a', p', \tau', z') | p' \in \mathrm{Plans}^*(z)\} \cup \{(\forall k)Failed(p, k)\}$
- $\vartheta_b^+ = \{Handle_f(p), Failed(p, j)\}$
- $\vartheta_p^- = \{(p, \tau, z)\} \cup \mathrm{Plans}^*(\Upsilon, z)$
- $\vartheta_e^- = \{(b, z) | (p, \tau, z) \in \vartheta_p^-\}$
- $j = \begin{cases} j' + 1 & \text{if } Failed(p, j') \in B \\ 1 & \text{otherwise} \end{cases}$

This rule aborts a plan if the corresponding runtime condition is violated, a failure state is reached or if the agent believes that the team no longer executes the plan. Additionally, the rule aborts all plans and behaviours executed in the context of the current state z, and assumes that all agents involved in executing p detect the plan failure. Hence, this potentially leaves the agent in a conflicting belief state, where it still believes that $\mathrm{Bel}_{a'}\,\mathrm{TeamIn}(p)$ for some other agent a'. Resolving this state requires the agent to inform its team-mates about the plan failure. Here, this is done only implicitly through the periodic communication of the plan base, see Section 6.8.

The Plan Repair Rules implement default ways to handle failed plans. Intuitively, a plan can be restarted, replaced by an alternative or the failure can be propagated up.

$$PTopFail : \frac{Handle_f(p_0) \in B}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B - \{Handle_f(p_0)\}, \emptyset, \emptyset, \emptyset, \mathrm{R})}$$

PTopFail handles failures of the top level plan by resetting the agent configuration, thus triggering *Init* again. The only way to handle a failure at this level is to retry the whole program.

$$PRedo : \frac{(p, \tau, z) \in P \wedge z \in \mathrm{Fail}(p) \wedge \neg((\exists x) Failed(p, x) \in B) \wedge B' \vdash \psi}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', (\Upsilon - \vartheta_p^-) + \{(p, \tau, z')\}, E', \theta, \mathrm{R})}$$

where

- $B' = (B - \vartheta_b^-) + \{In(a, p, \tau, z'), Alloc(z'), Failed(p, 1)\}$
- $\vartheta_b^- = \{\mathrm{In}(a, p, \tau, z)\} \cup$
  $\{\mathrm{In}(a, p', \tau', z') | \tau' \in \mathcal{T} \wedge p' \in \mathrm{Plans}^*(z)\}$
- $z' = \mathrm{Init}(p, \tau)$
- $\psi = \mathrm{TeamIn}(p) \wedge (\mathrm{Pre}(p) \wedge \mathrm{Run}(p))\theta$
- $\vartheta_p^- = \{(p, \tau, z)\} \cup \mathrm{Plans}^*(\Upsilon, z)$
- $E' = E - \{(b, z) | (p, \tau, z) \in \vartheta_p^-\}$

By applying *PRedo*, an agent retries to fulfil its task within a plan p, if it has reached a failure state and believes its team is still working on p and that the preconditions and runtime conditions are still met. Note that evaluating the condition of this rule requires the agent to hypothesis B' before applying *PRedo*. *PRedo* takes precedence over *PAbort*, $PRedo > PAbort$, so a less extensive failure handling is tried first. Subsequent failures of the same plan will not be handled by *PRedo* due to the insertion of $Failed(p, 1)$ into the belief base. Since there is only one way to handle failure of the top-level plan, $PTopFail > PRedo$.

$$PReplace : \frac{Handle_f(p) \in B, Failed(p, 1) \in B}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow (B', \Upsilon, E, \theta, \mathrm{R})}$$

where

- $B' = (B - \{Handle_f(p)\}) + \{Alloc(z)\}$
- $p \in \mathrm{Plans}(z)$

PReplace handles a failure by triggering a new task assignment for the state in which p is executed.

$$PProp : \frac{Handle_f(p) \in B, Failed(p, 2) \in B, (p', \tau, z) \in \Upsilon}{(B, \Upsilon, E, \theta, \mathrm{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, \Upsilon - \vartheta_p^-, E - \vartheta_e^-, \theta, \mathrm{R})}$$

where

- $p \in \text{Plans}(z)$
- $\vartheta_b^- = \{\text{In}(a', p', \tau', z) | a' \in \mathcal{A}, \tau' \in \mathcal{T}\}$
  $\cup \{\text{In}(a', p'', \tau', z') | p'' \in \text{Plans}^*(z)\} \cup \{(\forall k) \mathit{Failed}(p', k)\}$
- $\vartheta_b^+ = \{\mathit{Handle}_f(p'), \mathit{Failed}(p', j)\}$
- $\vartheta_p^- = \{(p', \tau, z)\} \cup \text{Plans}^*(\Upsilon, z)$
- $\vartheta_e^- = \{(b, z) | (p', \tau, z) \in \vartheta_p^-\}$
- $j = \begin{cases} j' + 1 & \text{if } \mathit{Failed}(p', j') \in B \\ 1 & \text{otherwise} \end{cases}$

The last option for an agent confronted with a plan failure is to propagate the failure upwards, which is done here by aborting the parent plan and triggering failure handling rules for it. Since a failure should be handled at the lowest level possible, *PReplace* > *PProp*.

The Allocation Failure Rule handles the case where a task allocation cannot assign any agent to a plan, for instance if a precondition cannot be met. If an allocation for state z fails, a failure for the corresponding plan p is raised.

$$NExpand : \frac{\mathit{Alloc}(z) \in B, \text{TAlloc}^*(a, z|B) = \emptyset, (p, \tau, z) \in \Upsilon}{(B, \Upsilon, E, \theta, \text{R}) \longrightarrow ((B - \vartheta_b^-) + \vartheta_b^+, \Upsilon - \vartheta_p^-, E - \vartheta_e^-, \theta, \text{R})}$$

where

- $\vartheta_b^- = \{\mathit{Alloc}(z)\} \cup \{\text{In}(a', p, \tau', z) | a' \in \mathcal{A}, \tau' \in \mathcal{T}\}$
  $\cup \{\text{In}(a', p', \tau', z') | p' \in \text{Plans}^*(z)\} \cup \{(\forall k) \mathit{Failed}(p, k)\}$
- $\vartheta_b^+ = \{\mathit{Handle}_f(p), \mathit{Failed}(p, j)\}$
- $\vartheta_p^- = \{(p, \tau, z)\} \cup \text{Plans}^*(\Upsilon, z)$
- $\vartheta_e^- = \{(b, z) | (p, \tau, z) \in \vartheta_p^-\}$
- $j = \begin{cases} j' + 1 & \text{if } \mathit{Failed}(p, j') \in B \\ 1 & \text{otherwise} \end{cases}$

The Replanning Rule treats the case where a plan has not failed but has a comparatively low utility evaluation. In this case, an agent can trigger a new task allocation if it believes there exists an allocation which is more suitable to the current situation.

$$Replan : \frac{(p, \tau, z) \in P \wedge \mathcal{U}_p(B'') - w_s \mathit{Sim}(p, B'', B) > \mathcal{U}_p(B) + t_p}{(B, \Upsilon, E, \theta, \text{R}) \longrightarrow ((B' + \vartheta_b^+), (\Upsilon - \vartheta_p^-) + \vartheta_p^+, (E - \vartheta_e^-) + \vartheta_e^+, \theta', \text{R})}$$

where

- $B' = B - \{\text{In}(a', p', \tau', z') | p' \in \text{Plans}^*(z)\}$
- $B'' = B' + \text{TAlloc}^*(a, z|B')$

- $\theta' = \theta \circ \sigma_{(a,z,B')}$
- $\vartheta_b^+ = \{\mathrm{In}(a', p', \tau', z') | \, \mathrm{In}(a', p', \tau', z'') \in \mathrm{TAlloc}(a, z, B') \wedge (z' = z'' \vee \mathrm{In}(a', p', \tau', z') \in B)\}$
- $\vartheta_p^- = \{(p', \tau', z') | p' \in \mathrm{Plans}^*(z)\}$
- $\vartheta_p^+ = \{(p', \tau', z') | \, \mathrm{In}(a, p', \tau', z') \in \vartheta_b^+\}$
- $\vartheta_e^- = \{(b, z') | (p', \tau', z') \in \vartheta_p^- \wedge b \in \mathrm{Behaviours}(z')\}$
- $\vartheta_e^+ = \{(b, z') | (p', \tau', z') \in \vartheta_p^+ \wedge b \in \mathrm{Behaviours}(z')\}$
- t_p is a threshold value,
- w_s is a weight,
- $Sim(p, \mathcal{F}, \mathcal{G})$ is a similarity measure:

$$Sim(p, \mathcal{F}, \mathcal{G}) = 1 - \frac{|\{a | \, \mathrm{In}(a, p, \tau, z) \in \mathcal{F} \wedge (\exists z') \, \mathrm{In}(a, p, \tau, z') \in \mathcal{G}\}|}{|\mathcal{G}|}$$

That is, if a new task allocation for state z has a utility higher than the current allocation wrt. the situation at hand, the agent will adopt the new allocation. The threshold value t_p limits the applicability of this rule. t_p as well as w_s is specific to each plan p. Similar to the transition rule, the current allocation is removed from belief, plan and behaviour base. The new allocation is adopted directly, similar to the Task Allocation Rule. This rule has the lowest precedence.

This rule enables a team to react swiftly to changing situations. Due to the threshold value and the similarity measure, oscillation can be avoided. However, it highly depends on the utility function and is therefore domain dependent. In general, as an agent progresses towards successful completion of its task, the overall utility should increase. One way to achieve this, given reward functions for successful completion, is using utilities which obey the Bellman equations [2].

In summary, the presented repair rules form a flexible and customisable system to react on failures. Although a complete evaluation as well as a formal analysis is still pending, preliminary tests have shown promising results, especially with regards to robustness and adaptiveness of the executing team.

7 Conclusions & Future Work

In this work, we presented a specification language for cooperative behaviour of autonomous teams. The language, ALICA, is based on BDI languages such as 3APL [14] and teamwork frameworks such as STEAM [30]. With the presented approach we extend the state of the art towards a comprehensive teamwork model, especially geared towards highly dynamic domains.

The syntax allows for intuitive description of complex cooperative strategies in the form of hierarchical plans, while annotations such as preconditions, postconditions and utility functions allow to model autonomous decisions in a very precise manner. Through different degrees of commitments, either by estimation or by explicit synchronisation, a team's performance can be optimised with regards to the precise task at hand.

Roles and tasks form a two-layered abstraction between agents and plans, thus allowing for both to be specified independently of each other. Roles are assigned to agents based on their capabilities, and agents are allocated to tasks based on the current situation as well as the preferences of the roles they are assigned to. By allocating roles to agents based on the offered and required capabilities, we directly address one of the most important aspects of heterogeneity in teams of heterogeneous agents. Heterogeneity in multi-agent systems can roughly be divided into the subclasses "communication heterogeneity" (e.g., by different communication protocols), "information representation heterogeneity" (e.g., different units), "capability heterogeneity" (different agents can have different capabilities), and "concept heterogeneity" (with different modelling approaches and reasoning paradigms). In this work we presented a general role allocation approach, which is to be evaluated in the future.

The semantics of ALICA is described by a transitional rule system, which defines how agents update their internal state while they execute an ALICA program. The internal state of an agent consists of its beliefs, its intentions in the form of procedural plans it is committed to, its roles, its behaviours in execution and a runtime substitution. This substitution allows for further abstraction through employment of parametrised plans and actions.

The rule system is both flexible and robust, as it already features rich failure handling procedures and can easily be extended, e.g., to accommodate planning and other sophisticated reasoning techniques.

Moreover, the operational nature of the given semantics allow for a one-on-one correspondence between specification and implementation, thus circumventing the problem of ungrounded semantics. However, the relationship between the semantics and BDI theories still needs be examined, in order to bridge this gap completely. More concrete, we want to evaluate the convertibility of ALICA to other approaches like 3APL. Convertibility would be also a big step to address heterogeneity in teams of heterogeneous agents, as this would directly address the issue of using different behaviour modelling approaches within one team.

Besides pending practical evaluations, the presented system offers plenty of room for future work. We plan on a complete system to track agents through an ALICA plan tree by estimating which rules an agent will apply. This should result in cooperation even more robust against unreliable communication and sensor noise.

The notion of utility functions to evaluate plans defined here gives rise to the possibility of employing reinforcement learning through decision theoretic functions. This can then be combined with symbolic learning techniques to learn and optimise complex team strategies out of simple ones.

References

[1] David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *In Advances in Neural Information Processing Systems: Proceedings of the 2000 Conference*, pages 1019–1025. MIT Press, 2000.

[2] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, N.J., 1957.

[3] Michael Bratman. *Intentions, Plans, and Practical Reason.* Harvard University Press, 1987.

[4] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42(2-3):213–261, 1990. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(90)90055-5. URL `http://portal.acm.org/citation.cfm?id=77757&dl=GUIDE,`.

[5] Mehdi Dastani, M. Birna, Riemsdijk Frank Dignum, and John jules Ch. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *In Proc. ProMAS'03. 2003*, pages 111–130. Springer, 2003.

[6] Mehdi Dastani, Dirk Hobo, and John-Jules Ch. Meyer. Practical extensions in agent programming languages. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–3, New York, NY, USA, 2007. ACM. ISBN 978-81-904262-7-5.

[7] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge.* MIT Press, 1995. ISBN 0262061627.

[8] FIPA. *FIPA Communicative Act Library Specification.* FIPA, December 2002.

[9] FIPA. *FIPA ACL Message Structure Specification.* FIPA, 2001. URL `http://www.fipa.org/specs/fipa00061/`.

[10] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86:269–357, 1996.

[11] Barbara J Grosz and Candace L Sidner. Plans for discourse. In *Intentions in Communication.* MIT Press, 1990.

[12] Koen Hindriks, Frank S. De Boer, Wiebe Van Der Hoek, and John jules Ch. Meyer. A Formal Embedding of AgentSpeak(L) in 3APL. Technical report, Advanced Topics in Artificial Intelligence, Springer Verlag LNAI 1502, 1998.

[13] Koen V. Hindriks and John-Jules Ch. Meyer. Agent Logics as Program Logics: Grounding KARO. In Christian Freksa, Michael Kohlhase, and Kerstin Schill, editors, *KI*, volume 4314 of *Lecture Notes in Computer Science*, pages 404–418. Springer, 2006.

[14] Koen V. Hindriks, Frank S. De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999. ISSN 1387-2532.

[15] Marcus J. Huber and Edmund H. Durfee. On acting together: Without communication. In *Working Notes of the AAAI Spring Symposium on Representing Mental States and Mechanisms*, pages 60–71, 1995.

[16] Jomi Fred Hübner, Jaime Simao Sichman, and Olivier Boissier. A model for the structural, functional, and deontic specification of organizations in multiagent systems. In *In Guilherme Bittencourt and Geber L. Ramalho, editors, Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA02), LNAI 2507*, pages 118–128. Springer, 2002.

[17] Nicholas R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995. URL `http://citeseer.ist.psu.edu/jennings95controlling.html`.

[18] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR91)*, pages 387–394. Morgan Kaufmann, 1991.

[19] Jérôme Lang. Belief update revisited. In *IJCAI*, pages 2517–2522, 2007.

[20] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[21] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 1043–1049, Cambridge, MA, USA, 1998. MIT Press. ISBN 0-262-10076-2.

[22] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981. URL `http://ttic.uchicago.edu/~blume/classes/aut2008/proglang/papers/plotkin81structural.pdf`.

[23] D. Pynadath and M. Tambe. Multiagent teamwork: Analyzing the optimality and complexity of key theories and models. In *In Proceedings of the 1st conference of autonomous agents and multiagent systems (AAMAS-2002)*, 2002. URL `http://citeseer.ist.psu.edu/article/pynadath02multiagent.html`.

[24] David V. Pynadath, Milind Tambe, and Nicolas Chauvat. Toward team-oriented programming. In *Intelligent Agents VI: Agent Theories, Architectures, and Languages*, pages 233–247, 1999.

[25] Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc. ISBN 3-540-60852-4.

[26] Anand S. Rao, Michael P. Georgeff, and E. A. Sonenberg. Social plans: A preliminary report. In E. Werner and Y. Demazeau, editors, *Decentralized AI 3 — Proceedings of the Third European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-91)*, pages 57–76, Kaiserslautern, Germany, 1992. Elsevier Science B.V.: Amsterdam, Netherland. URL `http://citeseer.ist.psu.edu/rao92social.html`.

[27] Paul Scerri, David V. Pynadath, Nathan Schurr, Alessandro Farinelli, Sudeep Gandhe, and Milind Tambe. Team oriented programming and proxy agents: The next generation. In Mehdi Dastani, Juergen Dix, and Amal El Fallah-Seghrouchni, editors, *PROMAS*, volume 3067 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 2003. ISBN 3-540-22180-8.

[28] Andreas Scharf. Grafische Verhaltensmodellierung kooperativer autonomer Robotersysteme. Bachelor thesis, University of Kassel, Germany, 2008.

[29] Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 409–430. Springer, 1999.

[30] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997. URL `http://citeseer.ist.psu.edu/tambe97towards.html`.

[31] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory Pract. Log. Program.*, 5(4-5):533–565, 2005. ISSN 1471-0684.

[32] Stefan Triller. A Cooperative Behaviour Model for Autonomous Robots in Dynamic Domains. Diplom(I), University of Kassel, Germany, 2008.

[33] Michael Wooldridge. *Introduction to MultiAgent Systems.* John Wiley & Sons, June 2002. ISBN 0 47149691X.

[34] John Yen, Jianwen Yin, Thomas R. Ioerger, Michael S. Miller, Dianxiang Xu, and Richard A. Volz. Cast: Collaborative agents for simulating teamwork. In *IJCAI*, pages 1135–1144, 2001. URL `http://citeseer.ist.psu.edu/yen01cast.html`.

[35] Jianwen Yin, Michael S. Miller, Thomas R. Ioerger, John Yen, and Richard A. Volz. A knowledge-based approach for designing intelligent team training systems. In Carles Sierra, Maria Gini, and Jeffrey S. Rosenschein, editors, *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 427–434, Barcelona, Catalonia, Spain, 2000. ACM Press. URL `http://citeseer.ist.psu.edu/yin00knowledgebased.html`.

[36] H.-J. Zimmermann. *Fuzzy set theory—and its applications (3rd ed.).* Kluwer Academic Publishers, Norwell, MA, USA, 1996. ISBN 0-7923-9624-3. URL `http://portal.acm.org/citation.cfm?id=236293`.