
A coordination language for agents interacting in distributed plan-execute cycles

W. de Vries and J-J.Ch. Meyer*

Intelligent Systems Group,
Department of Information and Computing Sciences,
Utrecht University,
P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
E-mail: vlinderwiekje@hotmail.com
E-mail: jj@cs.uu.nl
*Corresponding author

F.S. de Boer

Intelligent Systems Group,
Department of Information and Computing Sciences,
Utrecht University,
P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
and
Department of Software Engineering,
Centrum voor Wiskunde en Informatica (CWI),
Kruislaan 413, P.O. Box 94079,
1090 GB Amsterdam, The Netherlands
and
Leiden Institute of Advanced Computer Science (LIACS),
Universiteit Leiden Niels Bohrweg 1,
2333 CA Leiden, The Netherlands
E-mail: F.S.de.Boer@cwi.nl

W. van der Hoek

Department of Computer Science,
University of Liverpool,
Liverpool L69 3BX, UK
E-mail: Wiebe.Van-Der-Hoek@liv.ac.uk

Abstract: An architecture is proposed for agents to coordinate the formation and execution of plans. Along with the architecture, a new coordination language is introduced that agents may employ to form temporary alliances for planning and plan execution. This language is based on ideas from constraint programming and a formal operational semantics is given for it.

Keywords: multi-agent systems; coordination; planning; constraint programming; operational semantics.

Reference to this paper should be made as follows: de Vries, W., Meyer, J-J.Ch., de Boer, F.S. and van der Hoek, W. (2009) 'A coordination language for agents interacting in distributed plan-execute cycles', *Int. J. Reasoning-based Intelligent Systems*, Vol. 1, Nos. 1/2, pp.4–17.

Biographical notes: Wieke de Vries obtained her PhD in 2002 at the Utrecht University, in the field of formal programming languages for agent systems and agent verification. Since then, she had been working as a Documentation Writer for Parallax, an IT company in planning software and has set up her own catering business.

John-Jules Ch. Meyer obtained his PhD from the Vrije Universiteit in Amsterdam in 1985. Since 1993, he has been a Professor of Computer Science at Utrecht University. At the moment, he is heading the Intelligent Systems Group. He is a member of the IFAAMAS board, steering the international AAMAS conferences and of the editorial boards of various international journals. His current research interests include artificial intelligence and intelligent agents in particular. In 2005, he was appointed as a fellow of the European Coordinating Committee for Artificial Intelligence.

Frank de Boer is a Leader of the research group ‘Coordination Languages’ at the Centrum Wiskunde and Informatica (Amsterdam, NL) and Professor of the Foundation of Software Technology Department at the Leiden University (NL). He obtained his PhD at the Free University of Amsterdam in 1991. He works on formal methods for concurrent programming languages including object- and agent-oriented languages.

Wiebe van der Hoek is a Professor in the Agent ART Group of the Computer Science Department of the University of Liverpool (UK) since 2002. He obtained his PhD at the Free University of Amsterdam in 1992. He works on formal approaches to multi-agent systems, including logics for knowledge and belief, coordination and cooperation.

1 Introduction

An important motivation for the study and deployment of multi-agent systems is the idea of using these systems to perform non-trivial tasks in a distributed manner, where the individual agents in the system cooperate and coordinate their individual activities in order to achieve the overall goal (task) in a rational (‘intelligent’) way. To realise this idea, agents have to be ‘social’ in the sense of (Wooldridge and Jennings, 1995; Wooldridge, 2002): they have to communicate with each other, in order to exchange information and request and provide services. They must be able to perform actions for or together with other agents, depending on their own motivations and available resources. When realised, these social agent capabilities then contribute to agents being interacting pieces of software. Agents must pay attention to the actions of their fellow agents in the environment they inhabit and sometimes will have to try to persuade other agents to help them. In this paper, we focus on agents that form temporary alliances to jointly achieve common goals.

Collaborative agents can cooperate to achieve a mutually beneficial goal by developing a plan together for this goal. This is clearly a form of ‘distributed planning’. We ‘abstract’ from the specific planning algorithms the agents use (see desJardins and Wolverson, 1999; Kautz and Selman, 1998; Penberthy and Weld, 1992; Rao et al., 1992; Weld, 1999) and instead focus on coordination between the agents. Our main interest in this paper is to provide programming language constructs to perform coordination of the kind described, together with a precise formal semantics of these constructs. To this end, we provide a ‘coordination language’ and an ‘agent architecture’ that ‘facilitate’ distributed planning and synchronised execution of the resulting plan. A coordination language provides statements for coordination of agents, ‘abstracting’ from the internal make-up and processes of the agents. Aspects of agents like reasoning or message passing should be programmed in an ‘ordinary’ programming language, like AGENT-0 (Shoham, 1993), 3APL (Hindriks et al., 1998) or JAVA. The statements of the coordination language should also be implemented in these lower level languages. The statements of our language enable agents to ‘form groups’ dedicated to constructing a distributed plan together, to ‘communicate’ about details of the plan (as produced by the planning algorithms used by the agents), to jointly decide that the plan is sufficiently elaborated by ‘committing’ to it

and finally to ‘execute’ it. The benefit of using a separate coordination language is that it focuses on one aspect of agents, namely coordination and defines primitives for this in a clean and abstract manner, without distractions caused by details of internal agent statements. A coordination language can be used as a ‘wrapper’, to go around agents written in different programming languages. In the program of an agent, statements from the coordination language appear among statements from the programming language used for the agent. The primitives of the coordination language enable the agents to ‘interact’ with each other in order to form and perform plans. This way, heterogeneous agents can coordinate their activities. The coordination language provides a neat separation of concerns and enables reuse.

Our approach is based on ‘constraint programming’ (Saraswat, 1993). In constraint programming, indeterminacy about the value of a variable is gradually resolved by agents adding constraints on the value of this variable. In our approach, a ‘plan’ is a named tuple consisting of several elements, like the set of actions, the order of these actions and the set of agents involved in execution of the plan. By establishing constraints on these components, the plan is formed. A ‘constraint solver’ (Tsang., 1993) is used to check whether the constraints of the agents forming the plan are consistent. After the negotiating agents have committed to the plan, the plan might still be partial and the constraint solver uses its built-in planning knowledge to arrive at a sufficiently specified plan.

The starting points of this new coordination language are the same as those of the programming language GrAPL (de Vries et al., 2002): we use constraints and the synchronous primitives of our coordination language are similar (to a large extent) to the statements of GrAPL. But as we widen the scope of constraints from single actions to entire plans, there are many new aspects to be dealt with.

The set of actions of a plan can contain individual actions and group actions as well as see-to-it actions. These last actions simply specify that a certain goal is to be achieved, without details on how to go about this. When a plan is executed and a see-to-it action is encountered, a plan for the goal has to be formed and subsequently executed. So, a plan does not need to be fully elaborated before execution. Plans like this, which are gradually refined by introducing subgoals, are called ‘hierarchical plans’; hierarchical planning is also used in Rao et al. (1992) and Tambe (1997). Hierarchical plans have the advantage that

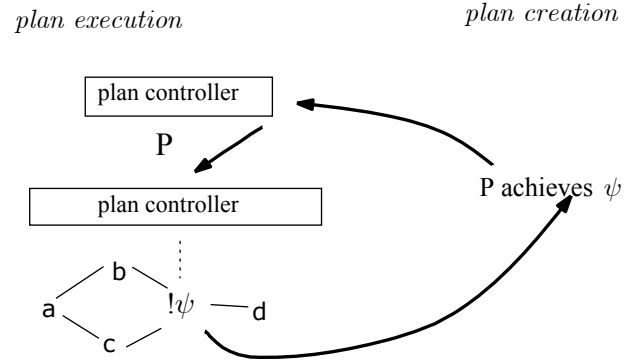
part of the plan formation can be postponed and thus can take into account more up-to-date environment circumstances.

For each goal, the agents want to achieve, there are two phases: the ‘plan creation phase’ and the ‘plan execution phase’. In the plan creation phase, there is a group of ‘negotiator’ agents that communicate with each other, using a constraint store, to find a plan for the given goal. This means that the agents pick a plan name and then start inserting constraints on the actions, the agents performing the actions and the order of the actions of that plan. For each activated plan name, there is a (semi-global) constraint store, which is shared by the group of negotiator agents, but not accessible to agents outside this group. A plan can be compared to a treaty between autonomous parties, which will govern the future behaviour of the parties. Like treaties, distributed plans are independent concepts that surpass the local agents that form and execute the plan. Therefore, it is intuitively obvious that plans in creation or execution are represented using a constraint store which is semi-global, instead of using local constraint stores. When the negotiators have inserted all of their constraints into the store of the plan, they commit to the plan and the ‘execution phase’ can start. Because the plan can contain synchronised group actions, it is useful to have a ‘plan controller’ that decides on the details of plan execution. Each plan in execution has its own associated plan controller. This controller has authority over the actors of the plan; when it tells certain agents to synchronously perform an action, they have to do this. When the next action to be executed is a see-to-it action, the plan controller suspends and a new plan-execute cycle is started for the new goal. After a group of agents has formed a plan to obtain the goal and has executed it, the plan controller resumes with the rest of the outer plan. This way, the agents execute ‘distributed plan-execute cycles’.

It might seem that a global plan with a plan controller that dictates agent behaviours conflicts with agent autonomy, but as the plan being formed is agreed upon by the negotiating agents, these agents later on are obliged to honour the agreement.

In Figure 1, we depict a plan-execute cycle. There is an outer plan (the partially ordered plan with actions a , b , c , d and $!\psi$) which is in execution; it has an associated plan controller, which already has taken care of the execution of a , b and c and is currently suspended at action $!\psi$. The goal ψ is broadcast to the agents and some of them react by starting a new plan-execute cycle. They develop a plan P that will lead to ψ and execute it. After the plan controller of this plan finishes, the controller of the outer plan takes over again. If plan execution finishes successfully, the plan controller terminates and a new planning phase can start. If actions fail due to synchronisation problems or environment conditions, the plan controller first informs the negotiators of the plan about the failure, such that they can decide to replan. After this, the plan controller terminates just as in the case of successful plan execution.

Figure 1 The plan-execute cycle



Thus, we propose a ‘generic architecture’ for plan formation and execution. This architecture contains the agents, which form and execute plans, the constraint solver with the constraint stores of the plans and the plan controllers. The statements of the coordination language enable the agents to interact with each other, update the constraint stores that represent plans being created and start plan controllers. The constraint solver contains additional planning knowledge and can thus detect errors the agents make in their planning process and complete the plan when the agents have inserted their constraints. The set of plan controllers is dynamic; whenever a new plan starts execution, a new plan controller springs up, which can suspend when a subplan has to be formed and which disappears when the execution of the plan is finished.

In the next section, we go into the nature of plans and the constraint stores that contain constraints on plans. In Section 3, we formally define the syntax and semantics of our coordination language. In Section 4, we describe the coordination architecture. Finally, Section 5 concludes this paper.

2 Plan features and constraint stores

Underlying the definition of both plans and statements of the coordination language are these sets:

- \mathcal{A} the set of atomic physical actions. For simplicity, we focus on unparameterised actions here.
- \mathcal{I} the set of agent identities.
- \mathcal{PN} the set of plan names.
- \mathcal{PV} the set of plan variables; each agent has at its disposal its own plan variables, a subset of \mathcal{PV} .
- \mathcal{LV} the set of local variables.
- \mathcal{L} a predicate logical language used to formalise constraints on plans. We assume \mathcal{L} is equipped with a consequence relation denoted by \models_{cs} (CS abbreviates constraint solver). Furthermore, $free(\varphi)$ is the set of free variables in the formula φ

As stated in Definition 1 below, actions from the set \mathcal{A} form the basis of plans. To keep things manageable, we look at unparameterised actions in this paper. In the concluding section, we will briefly discuss a combination of the coordination language with GrAPL (de Vries et al., 2002), in which actions are parameterised.

The allocation of actions to particular agents or groups of agents is part of the planning process. The task allocation determines whether a certain action will be an individual action or a group action. Of course, some actions from \mathcal{A} can only be allocated to one agent, while other actions always must be performed by a group. For example, in the domain of cleaning office buildings, the action of vacuuming a room is an individual action, the action of moving a heavy cupboard is a group action and the action of cleaning a window can be both individual or collective, depending on the amount of glass present in the office.

The sets \mathcal{PV} and \mathcal{PN} are closely related. Both plan variables and plan names are used to refer to different plans. At each moment during execution of a multi-agent system, there can be various plans under construction and in execution, by disjoint or even overlapping groups of agents (see also Section 4). Plan variables from \mathcal{PV} are assigned values from the set of plan names \mathcal{PN} . When including a coordination statement in an agent program, the programmer should use a plan variable to refer to a plan and its associated constraint store. When the coordination statement is executed and there is not yet a binding to the plan variable, the system chooses a new, unused plan name and binds this to this variable. The programmer is not allowed to choose a plan name himself, because he could choose a plan name already used by another agent for another plan and thus create confusion. The set of plan variables is divided into disjoint subsets, such that each agent has its own private plan variables. The set \mathcal{LV} contains local variables that agents can use to test the constraint store of a plan (named ‘plan store’ from now on). These may be employed to refer to whatever is adequate for the plan: objects, actions, agents,...

Our notion of plans is very close to the notion of ‘operators’, used in e.g., Tambe (1997). Though the definition of operators differs, an operator usually contains a precondition, a postcondition and a plan body. The precondition specifies environment circumstances in which the plan body can be executed, with the goal of achieving the postcondition. The plan body specifies the actions and the order in which they need to be done. In our architecture, the plan body is represented by a partially ordered set of actions. These actions can be atomic actions from \mathcal{A} or actions of the form $!\psi$. This should be read as ‘see to it that ψ becomes true’. Actions like these allow hierarchical planning. If a plan contains $!\psi$, then a subplan to achieve ψ is made when $!\psi$ is executed and not earlier. This is useful, because it enables agents to adjust the plan to the circumstances in the environment close to the time of execution.

Definition 1 (plan components): The set \mathcal{PC} of plan components is defined:

$$\mathcal{PC} = \mathcal{A} \cup \{!\psi \mid \psi \text{ is a closed formula of } \mathcal{L}\}$$

So, a plan can have two different kinds of actions: basic actions, which are simply executed and see-to-it-actions, where a whole new planning and execution phase are started in a recursive manner. As the actions in the set \mathcal{A} are physical in nature, we do not have mental actions in the plans. In our view, plan execution is an ‘interaction’ with the environment. The coordination language entirely abstracts from internal agent reasoning. The coordination language enables the agents to communicate with each other and execute plans, which both are viewed as interaction.

As described in the introduction, a plan is formed by inserting constraints into the constraint store associated with the plan name. The constraints pertain to certain features of the plan, such as the action set or the task allocation. Each plan feature is represented by a special variable.

Definition 2 (plan features): We define six plan features (*Purpose*, *Circumstances*, *Actions*, $<$, *Negotiators* and *Actors*), where *Purpose* and *Circumstances* are variables of type \mathcal{L} , *Actions* is a variable of type $\wp(\mathcal{PC})$, $<$ is a variable of type $\wp(\mathcal{PC} \times \mathcal{PC})$, *Negotiators* is a variable of type $\wp(\mathcal{I})$ and *Actors* is a variable of type $\mathcal{A} \rightarrow \wp(\mathcal{I})$ [the set of partial functions from \mathcal{A} to $\wp(\mathcal{I})$].

The six features of a plan all are ‘variables’, which gradually are constrained during the plan creation phase. To be precise, the values of the plan features *Purpose*, *Circumstances* and *Negotiators* are determined at the start of the planning phase, when the group of negotiators is formed. The other three variables are worked out during the planning phase. The name of the plan actually does not refer to a plan, but to the constraint store containing the demands on the plan. We will often be sloppy with this and talk about ‘the plan called P ’. During the plan creation phase, the constraints in the plan store allow a set of possible plans. Example 1 below illustrates this. At the end of the planning phase, when the agents have committed to the plan, the plan is ready and ‘definite’. All six plan components then have been completely determined, that is, the constraints imply that *Actions* is a specific set, *Purpose* is equivalent to a ground formula from \mathcal{L} , etc. As the six variables each have a definite value, the space of possible plans has been narrowed down to one.

A ‘plan’ thus is an association of values with a six-tuple of plan features. After the planning phase has ended, the formula associated with *Purpose* formalises the goal of the plan and the value of *Circumstances* is a formula describing suitable environment conditions for the plan to be performed. These two formulas correspond to the post and precondition of operators. The planning agents use these two formulas as an input to their planning routines, from which we abstract and which yield contributions to the distributed plan under construction. *Purpose* and

Circumstances can also play a role at the start and end of plan execution, to check whether proper *Circumstances* indeed are present in the world before execution and to check whether the *Purpose* of the plan indeed has been achieved after execution. These checks are performed at the lower agent level and so we do not go into them. The value of *Actions* in a finished plan is the set of plan components constituting the plan. Note that a committed plan is still partial, in the sense that it can contain see-to-it-statements! The value of $<$ is a partial order on the action set. To ensure this, the consequence relation \models_{cs} used by the constraint solver has the properties of partial orders as axioms. The partial order fixes the execution order of some actions. Actions that are not ordered by it are independent and can be executed in parallel (according to the planning agents). The value of *Negotiators* is the set of agents that have agreed to form the plan together and the value of *Actors* is a function, associating a group of agents to each atomic action in the plan. Actions which have been allocated to a group containing more than one agent must be done by these agents in a synchronised manner. So, the value of *Actors* describes the ‘task allocation’ of the plan.

During the plan creation phase, the plan is partial, because the six variables are only determined by the constraints given by the planning agents (the *Negotiators*) and generally these will not completely fix the values of the variables during the planning process. The language \mathcal{L} for phrasing constraints at least contains set-theoretic operations (e.g., \in , \subseteq) among its predicates. Additionally, there can be predicates expressing characteristics of agents. For example, the constraint $reliable(\mathbf{James}) \leftrightarrow \mathbf{James} \in Actors(\mathbf{transport_diamond})$ demands that **James** should be involved in transporting a certain diamond if and only if he is a reliable agent. Typically, during the plan creation phase we will have constraints like $A \subseteq Actions$, where A is a set of plan components. So, other plan components could also become part of the plan later on. This is an example of a plan store during the planning phase, specifying a set of possible plans:

Example 1 (A plan under construction)

$$\begin{aligned} Purpose &= room_clean \wedge \\ Circumstances &= room_messy_and_dirty \wedge \\ Negotiators &= \{\mathbf{James}, \mathbf{Martha}\} \wedge \\ \{!floor_very_clean, empty_ashtray, \\ &\quad open_windows\} \subseteq Actions \wedge \\ open_windows &< !floor_very_clean \wedge \\ Actors(empty_ashtray) &= \{\mathbf{John}\} \end{aligned}$$

Note that in the example above, the set of actions is not fixed yet and the allocation and the ordering of the actions are only partial.

After the negotiators have committed to the plan and the plan is finished by the constraint solver, the constraints allow only one plan. This is an example of a finished plan:

Example 2 (A definite plan)

$$\begin{aligned} Purpose &= room_clean \wedge \\ Circumstances &= room_messy_and_dirty \wedge \\ Negotiators &= \{\mathbf{James}, \mathbf{Martha}\} \wedge \\ Actions &= \{!floor_very_clean, empty_ashtray, \\ &\quad open_windows, !\neg room_messy\} \wedge \\ open_windows &< !\neg room_messy \wedge \\ !\neg room_messy &< !floor_very_clean \wedge \\ Actors(empty_ashtray) &= \{\mathbf{John}\} \wedge \\ Actors(open_windows) &= \{\mathbf{Mary}, \mathbf{Martha}\} \end{aligned}$$

Though this plan is definite, the order of the actions is still partial and not total.

To store the constraints on plans, the state of the system has to contain a plan constraint store association:

Definition 3 (plan constraint store association): A plan constraint store association is a partial function $\mu : \mathcal{PN} \rightarrow \mathcal{L}$.

This function maps plan names to the present constraint on that plan. The function is partial, as only part of the plan names are in use for plans being formed or in execution.

We also provide a definition that formalises that a plan store is ‘definite’, which is the case when the constraints on the plan allow only one value for each of the six plan features.

Definition 4 (definiteness of a plan store): A plan store $P \in \mathcal{PN}$ is *definite* in the context of a constraint store association μ when the following holds:

- There is a closed formula $\varphi \in \mathcal{L}$, such that $\mu(P) \models_{cs} Purpose = \varphi$.
- There is a closed formula $\psi \in \mathcal{L}$, such that $\mu(P) \models_{cs} Circumstances = \psi$.
- There is a set of plan components $A \subseteq \mathcal{PC}$, such that $\mu(P) \models_{cs} Actions = A$.
- There is a relation $R \subseteq \mathcal{PC} \times \mathcal{PC}$, such that $\mu(P) \models_{cs} < = R$.
- There is a set of agents $N \subseteq \mathcal{I}$, such that $\mu(P) \models_{cs} Negotiators = N$.
- There is a partial function $f : A \rightarrow \wp(\mathcal{I})$ with domain $A \cap \mathcal{A}$ (all atomic actions in the plan), such that $\mu(P) \models_{cs} Actors = f$.

When a plan store is definite, we often say that ‘the plan is definite’ and identify the constraints on the plan with an association of specific values with the plan features.

3 Syntax and semantics

3.1 Syntax

After establishing this background, we define the statements of the coordination language:

Definition 5 (coordination statements): The coordination language contains the following statements:

- **FormGroup**($\alpha, \omega, p, \varphi$), where $\alpha, \omega, \varphi \in \mathcal{L}$ are closed formulas and $p \in \mathcal{PV}$.
- **CommGroup**(φ, p), where $p \in \mathcal{PV}$ and $\varphi \in \mathcal{L}$ is a closed formula.
- **?**(φ, p), where $\varphi \in \mathcal{L}$ and $p \in \mathcal{PV}$.
- **Commit**(p), where $p \in \mathcal{PV}$.
- **Execute**(p), where $p \in \mathcal{PV}$.

3.2 Two-layered semantics

To define agent programs, the statements above are combined with a lower level (agent) programming language. This programming language provides statements for other agent capabilities, such as reasoning and ordinary communication. Also, this programming language provides control constructs, such as conditional branching, recursion, sequential composition and non-deterministic choice. Below, we explain the coordination statements and give their formal operational semantics.

The semantics we use is two-layered. As we only look at agent coordination and abstract from the local agent reasoning, we only need to give transition rules for the ‘global’ level semantics of the coordination statements. We assume the underlying programming languages (or language) used to program the agents have well-defined ‘local’ semantics.

In order to define global transition rules, we first define global system configurations. The global system configuration at each moment contains at least the local agent configurations, the set of plan controllers that are active at that time, the set of plans that are committed and ready but not in execution yet and a plan constraint store association:

Definition 6 (global system configuration): A ‘global system configuration’ is a tuple $\langle \dots, \{ \langle S_\iota, \pi_\iota \rangle \mid \iota \in \mathcal{I} \}, \chi, \rho, \mu \rangle$.

Here,

- $\langle S_\iota, \pi_\iota \rangle$ is a local agent configuration of agent $\iota \in \mathcal{I}$, where π_ι is the local program left to be executed and S_ι is the local agent state.
- $\chi \subseteq \mathcal{PN}$ is the set of plans that are currently being executed.
- $\rho \subseteq \mathcal{PN}$ is the set of plans that are ready for execution, but are not in execution yet.

- $\mu : \mathcal{PN} \rightarrow \mathcal{L}$ is a plan constraint store association
- ... are other global configuration elements which we abstract from, because these are not relevant for plan coordination (for example, the state of the world).

We demand that $\chi \cap \rho = \emptyset$ and that for each $P \in \rho \cup \chi$ it holds that P is definite.

The demand that χ and ρ have no elements in common is made because a plan cannot be in execution and waiting to be executed at the same time. The plans in χ each have an associated plan controller, to which we also refer with the name of the plan. The plans in the sets ρ and χ must be definite, because otherwise they are still too undetermined to be executed. The plan constraint store association μ binds a constraint to each plan name which is in use by the system of agents. We largely abstract from the contents of the local agent configurations $\langle S_\iota, \pi_\iota \rangle$, as the details of the internal make-up of the agents belong to the lower agent level. We do explicitly represent the local agent program, as execution of coordination statements changes the program left to be executed. S_ι represents the local agent state, which can for example consist of a belief base, a set of current intentions and a set of joint intentions. We sometimes use A_ι as a shorthand for $\langle S_\iota, \pi_\iota \rangle$. The global configuration can contain additional elements from which we abstract, such as the state of the world. In the above definition, these elements are suggested by including ... as an element of the global configuration. For readability and brevity, we leave out these dots in the sequel of this paper.

In general, a global transition rule of a coordination statement C will have the format as the rule (General Format) in Figure 2.

In (General Format), A_ι abbreviates $\langle S_\iota, \pi_\iota \rangle$, and A'_ι abbreviates $\langle S'_\iota, \pi'_\iota \rangle$. In this rule, a group of agents J synchronously executes the coordination statement C . Each agent has its own coordination parameters: agent ι tries to execute $C(\text{args}_\iota)$. This coordination statement is part of the program of the agent, which is largely written in some other programming language. In order to be able to phrase global transition rules for the coordination statements, we have to make assumptions about their local semantics, even if we are not looking at the local semantics of the programming language, the rest of the agent program is written in. As can be seen above, we assume that the local semantics of a coordination statement yields a local transition labelled with the coordination statement and its arguments, $C(\text{args}_\iota)$. By putting the coordination statements in the labels of local transitions, they are available for the global semantics to process. The meaning of the coordination statements is largely determined at the global level, though there can also be local effects. For example, when agents form a group to make a plan, they can establish a joint intention to achieve the *Purpose* of the plan to be formed. Whether this happens depends on the mental make-up of the agents. If a mental

update is a result of executing **FormGroup**, this is encoded in the local semantics of this coordination statement, by transforming the local state S_i into S'_i . The local transitions of coordination statements always model successful executions of the coordination statements. Of course, the coordination statement can fail; the global semantics is defined in such a way that the local updates to the agent states are not performed in case of failure.

On the basis of the information in the labels of the local transitions above the line of the transition rule, a global transition is constructed. This global transition transforms the old global configuration into the new one and it can yield additional information, such as the information stating whether the coordination statement has succeeded. This information can be used by the local agent programs in the sequel of the execution. Depending on the design of the agents, this information will become part of the local agent states in some way or another. However, since we abstract from the structure of the local agent states, we cannot indicate concretely how this additional information will affect the configuration. This is the reason why we use the operator Γ_{add_inf} in the template transition rule above: it models the incorporation of the additional information into the (local and hence also the) global configuration.

Remark (failure of coordination statements): There are two manners in which coordination statements can fail. In the

first place, synchronisation can fail. This happens for the coordination statements that require the whole group of *Negotiators* to be present. When not all negotiators are synchronously performing their coordination statement, then no global transition is generated. This means that the global computation ‘blocks’ and the local agents have to ‘wait’ until all *Negotiators* are present and the coordination statement can be performed. The global computation also blocks when an agent unsuccessfully tests a plan store. This does not mean that the computation of the system is over; agents can take other branches of their local programs that do not lead to deadlock.

The second way of failing happens when agents try to form the plan by executing **CommGroups**. When the *Negotiators* propose constraints on the plan which are not consistent with each other or earlier constraints on the plan, then this type of failure occurs. The constraints are not added to the plan store, as this would result in the constraint \perp (inconsistency) and all earlier planning information would be lost. Instead, a global transition is generated that keeps the plan store unchanged and returns a boolean value *false* to the negotiating agents. The agents thus can find out that the group communication has failed because of disagreement and subsequently try other proposals in group communication.

Figure 2 Transition rules

$$\begin{array}{c}
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{C(\text{args}_i)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \Gamma_{add_inf}(\langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi', \rho', \mu') \rangle)} \quad \text{(General Format)} \\
\\
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{FormGroup}(\alpha, \omega, p_i, \varphi_i)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \Gamma_P(\langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu') \rangle)} \quad \text{(FormGroup)} \\
\\
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{CommGroup}(\varphi_i, P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \Gamma_{true}(\langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu') \rangle)} \quad \text{(SuccCommGroup)} \\
\\
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{CommGroup}(\varphi_i, P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \Gamma_{false}(\langle \{S_\iota, \pi'_\iota\} | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho, \mu') \rangle)} \quad \text{(FailCommGroup)} \\
\\
\frac{A_\iota \xrightarrow{?(\varphi, P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \Gamma_\emptyset(\langle \{A'_\iota\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus \{\iota\}\}, \chi, \rho, \mu') \rangle)} \quad \text{(Test)} \\
\\
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{Commit}(P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi, \rho \cup \{P\}, \mu' \rangle)} \quad \text{(Commit)} \\
\\
\frac{\text{for all } \iota \in J : A_\iota \xrightarrow{\text{Execute}(P)} A'_\iota}{\langle \{A_\iota | \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle \longrightarrow \langle \{A'_\iota | \iota \in J\} \cup \{A_\iota | \iota \in \mathcal{I} \setminus J\}, \chi \cup \{P\}, \rho \setminus \{P\}, \mu' \rangle)} \quad \text{(Execute)}
\end{array}$$

Notes: (General Format) gives the general format, the others formalise the meaning of our coordination statements. Explanations of and conditions on the old and new variables (with accents) are given in the running text.

3.3 Semantics of FormGroup

Now, we come to the discussion of the statements of our coordination language. The statement $\text{FormGroup}(\alpha, \omega, p, \varphi)$ informally translates into: ‘Form a group which will negotiate about a plan referred to by p aimed at realising ω starting from situations where α holds and the group composition has to obey the constraint φ .’ A FormGroup is synchronously performed by a group of agents. This is the semi-formal semantics of FormGroup :

Whenever a number of agents $J \subseteq \mathcal{I}$ try to form a group by each ($\iota \in J$) executing $\text{FormGroup}(\alpha, \omega, p_\iota, \varphi_\iota)$, then this will succeed if

$$\bigwedge_{\iota \in J} \varphi_\iota \wedge \text{Negotiators} = J \mid \neq_{\text{CS}} \perp$$

that is, if the composition of the group is consistent with the conjunction of the demands on group composition. Note that the agents have to agree upon the purpose ω and circumstances α of the plan. If successful, the system chooses a new, unused plan name $P (P \notin \text{domain}(\mu))$ and passes this name to the agents, who associate the name with the plan variables they use ($p_\iota, \iota \in J$). Also, the FormGroup -operation initialises the constraint bound to P , as the purpose, circumstances and negotiator group are fixed.

Formally: $\mu(P) := \text{Purpose} = \omega \wedge \text{Circumstances} = \alpha \wedge \text{Negotiators} = J$. If the group of agents J does not satisfy the constraints of the agents, then execution blocks; no global transition is generated.

When a group is formed, only these agents are allowed to add constraints to the plan. A successful FormGroup -statement initiates a plan store for the new plan with *Purpose* ω and *Circumstances* α and the fixed group of *Negotiators* J . All (and thus, in the light of the above, exactly all) members of the group J have to be present when the plan is updated. This way, we can be sure that intruders from outside cannot influence the plan and that the negotiators agree on the shared plan being formed. If group formation fails, because the present group does not satisfy the demands of the group members, then it is possible that there is another group (at the same time or later on) that is suitable. If there is no such group available, all agents wait at their FormGroup statements for a suitable group to arise, of which they may or may not be part.

Another result of FormGroup is that the new plan is ‘named’. This is important, because this name will be bound to the constraint store shared by the negotiators, that will gradually define the plan and the name of the plan is needed to be able to execute it and to phrase constraints on the plan. Though the name formally only refers to a constraint on the plan (through the plan constraint store association), we will also use the name to refer to the definite plan resulting at the end of the plan creation phase and to refer to the plan

controller of the plan in the execution phase. The agents use different plan variables to refer to the plan and a result of a FormGroup is that these variables can be bound to the chosen plan name by the local agents.

Now, we explain the formal transition rule for the statement FormGroup . Let $J \subseteq \mathcal{I}$ be a non-empty set of agent names, let $P \in \mathcal{PN}$ be a plan name, let $A_\iota = \langle S_\iota, \pi_\iota \rangle$ and let $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$. Then (FormGroup) in Figure 2 gives the transition rule for FormGroup , where the following conditions hold:

- $P \notin \text{domain}(\mu)$
- $\bigwedge_{\iota \in J} \varphi_\iota \wedge \text{Negotiators} = J \mid \neq_{\text{CS}} \perp$
- $\mu' = \mu[\text{Purpose} = \omega \wedge \text{Circumstances} = \alpha \wedge \text{Negotiators} = J / P]$

This rule is just a formalisation of the informal semantics we gave earlier. The result of the global transition is a new global configuration, incorporating as additional information (by means of the operator Γ_P) the name P of the new plan in creation which is now available for the local agents to bind to their plan variables p_ι . The local updates to the states of the agents as a result of forming a group (for example, a joint intention towards the *Purpose* of the plan) indeed take place; the local changes to the agent configurations ($A_\iota \rightarrow A'_\iota$) are part of the change to the system configuration. The configurations of agents that do not participate in the group formation stay the same. Forming a group fails if the group composition is inconsistent with the demands of the group members or these demands are inconsistent among themselves. No global transition results for this group J trying to initiate a plan creation phase. The local agent programs of the agents in J are stuck at the FormGroup statement. This situation remains like this until a suitable group of agents synchronously attempts to form a group. It seems likely that some agents from J will not be part of this new group.

3.4 Semantics of ComGroup

To actually plan, we have $\text{CommGroup}(\varphi, p)$, which informally translates to ‘add the constraint φ to the present constraint on the plan referred to by p ’. Here, φ is a formula stating demands of the agent executing the CommGroup on the properties of the plan (that is, on the actions in the plan, their order and the agents performing them). As CommGroup -statements must always be preceded by a FormGroup -statement, the plan variable p is already bound to a plan name $P \in \mathcal{PN}$. CommGroup is a synchronous communication statement that can only be successful if all of the *Negotiators* of plan P execute it simultaneously. This is the semi-formal semantics of CommGroup :

Whenever a number of agents $J \subseteq \mathcal{I}$ try to modify a plan by each executing **CommGroup** (φ_ι, p_ι) , then this will succeed if all $p_\iota, \iota \in J$ are bound to the same plan name P , $\mu(P)|_{=cs} \text{Negotiators} = J$ (that is, the proper group of agents present) and $\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota \not|_{=cs} \perp$ (that is, the constraints of the agents are consistent with each other and the stored plan constraint). If successful, the operation updates the constraint bound to P .

Formally: $\mu(P) := \mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota$. Finally, the execution yields a Boolean value, indicating whether the attempt to update the plan succeeded.

If the constraints of these agents are consistent, then the conjunction is added to the plan store. If not, then the plan store remains unchanged. The Boolean is available for the agents to include in their local state; it can be tested by the agents involved to determine their next step.

The coordination statement **CommGroup** has no effect on the plan store in case of inconsistency of proposals of the group members involved. We have chosen for this option, because otherwise one would lose all information about the plan in creation stored in the plan store, as it is overwritten with \perp .

We give the two global transition rules for **CommGroup**, for successful and failing group communication, respectively. With failing group communication, we refer to the situation that the proper group of *Negotiators* takes part in the group communication, but with proposals which are not consistent with each other and/or the previous constraint in the plan store. For the situation that the proper *Negotiators* are not present, no global transition results. Then, the agents which are present have to wait for the rest of the negotiating agents. We again assume that $J \subseteq \mathcal{I}$ is a set of agent names, $P \in \mathcal{PN}$ is a plan name, $A_\iota = \langle S_\iota, \pi_\iota \rangle$ and $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$. We start with the rule (**SuccCommGroup**) (Figure 2) for a successful **CommGroup**, where the following conditions hold:

- $\mu(P)|_{=cs} \text{Negotiators} = J$
- $\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota \not|_{=cs} \perp$
- $\mu' = \mu \left[\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota / P \right]$

Again, we simply formalise the semi-formal semantics given earlier. The result of the global transition is a new global configuration, in which the local configurations of the communicators are modified and the plan store of P is updated and in which the Boolean *true* to signal that the group communication has succeeded, has been incorporated (by Γ_{true}).

Equation (**FailCommGroup**) in Figure 2 describes the meaning of an unsuccessful **CommGroup**. Here, the following conditions hold:

- $\mu(P)|_{=cs} \text{Negotiators} = J$
- $\mu(P) \wedge \bigwedge_{\iota \in J} \varphi_\iota \not|_{=cs} \perp$

Because of the first condition above, the group of communicators must still be the group of *Negotiators* of the plan, as was the case for successful group communication. In case the second condition also holds, the demands of the negotiating agents are inconsistent with each other or with the previous demands on the plan. As a result, the plan store is not changed and the Boolean *false* is returned. The **CommGroup** statement is removed from the programs left to be executed of the participants ($\pi_\iota \rightarrow \pi'_\iota$) but the update to the local agent states ($S_\iota \rightarrow S'_\iota$) which was locally computed is not performed. For example, if in a particular implementation of **CommGroup** an agent forms an intention towards each action it adds to the plan, this results in a transformation of the local agent state (S_ι becomes S'_ι) when a proposed constraint of this agent implies that actions are added to the plan. If the **CommGroup** does not succeed, this intention should not be established, so the local agent state stays S_ι .

3.5 Semantics of $?(\varphi, p)$

The **CommGroup**-statement is close to the **tell**-statement from constraint programming (Saraswat, 1993), in that it adds information to the constraint store. We also need an equivalent of **ask**, to test the constraint store. For this, we have $?(\varphi, p)$, with informal reading: ‘Test whether φ follows from the constraint on the plan referred to by p ’. This is the semi-formal meaning of this statement:

Whenever an agent $\iota \in \mathcal{I}$ tries to test the store of a plan by executing $?(\varphi, p)$ and p is associated with the plan name P , then this will succeed if

$\mu(P)|_{=cs} \iota \in \text{Negotiators}$ (that is, ι is one of the *Negotiators* of the plan) and there is a ground substitution θ , with $dom(\theta)$ equal to the set of free local variables in φ , such that $\mu(P)|_{=cs} \varphi\theta$ (that is, the store of P entails this instantiation of φ). If successful, the operation yields the substitution θ , which represents information about the values of features of P . If not, the execution blocks.

Typical tests of a plan are $?(\mathbf{a} \in \text{Actions}, p)$, which tests whether \mathbf{a} is an action of the plan p , $?(\mathbf{a} < \mathbf{b}, p)$, which tests whether action \mathbf{a} precedes action \mathbf{b} and $?(\text{Actions} = x, p)$, where x is a free local variable and which tests whether the action set of p is fixed to one specific set and (in case there was not yet a substitution for x available) results in a substitution of x with this specific set of actions. We will explain the semantics of this last statement in more detail. Suppose P is the plan name bound to p . The test succeeds if the testing agent is allowed access to the information about the plan P and if there is a

substitution which binds a ground value to x such that the constraints on P entail that the set of actions of P equals this value. This implies that the ground value bound to x must be a set of plan components. If the test succeeds, that is, if indeed the action set of P has been constrained to one specific set of plan components, then this set is bound to x , such that the agent which performed this test can use this information.

We only have a global transition rule for successful tests. In case the test fails, no global transition results, which means that the local execution of the test blocks. Let $\iota \in \mathcal{I}$ be an agent name, let $P \in \mathcal{PN}$ be a plan name, let θ be a ground substitution with $dom(\theta) = free(\varphi) \cap \mathcal{LV}$, let $A_\iota = \langle S_\iota, \pi_\iota \rangle$ and let $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$. Then (Test) in Figure 2 gives the meaning of a successful test. It is due to the following conditions:

- $\mu(P) \models_{cs} \iota \in Negotiators$
- $\mu(P) \models_{cs} \varphi\theta$

These conditions state that only *Negotiators* of a plan have access to its plan store and that an instantiation of the formula tested ($\varphi\theta$) follows from the plan store. Furthermore, the substitution θ is incorporated into the resulting configuration as additional information (via Γ_θ) so that it can be used by the agent having performed the test.

3.6 Semantics of *Commit*

Next is *Commit*(p), which informally reads ‘complete the plan referred to by p by filling in missing details’. This statement is also executed synchronously by the group of negotiators of the plan referred to by p . By jointly executing *Commit*-statements, the negotiators end the plan creation phase of this plan. This does not have to mean that all plan features have been totally determined by the negotiations (*CommGroups*) of these agents, but just that the negotiators have established all their demands on the plan and that they do not care about the further details. Now it is up to the constraint solver to fill in the rest of the plan. We assume the constraint solver incorporates planning knowledge in its associated *planning function* $\bar{\omega} : \mathcal{L} \rightarrow \mathcal{L}$ to allow it to finish plans. This function takes a constraint on a plan, which describes a set of possible plans and returns a strengthened, definite constraint which only allows one plan, which means that the new constraint prescribes one specific value for each of the six plan features. We demand that $\bar{\omega}(\varphi) \models_{cs} \varphi$, that is, the resulting constraint on the plan must imply the original constraint. This way, the demands of the *Negotiators* always are respected. The planning knowledge can be very powerful, enabling the constraint solver to even construct plans when the agents did not add any constraints to the plan store. The constraints of the *Negotiators* reduce the size of the search space for the constraint solver, which speeds up the planning process. A *Commit*-statement fails in case of absence of one or more

negotiators in the synchronous execution. This is the semi-formal semantics:

Whenever a number of agents $J \subseteq \mathcal{I}$ try to commit to a plan by each ($\iota \in J$) executing *Commit*(p_ι), then this will succeed if all p_ι , $\iota \in J$ are bound to the same plan name P and $\mu(P) \models_{cs} Negotiators = J$ (that is, the proper group of agents present). If successful, the operation updates the plan store of P , such that the resulting store is definite. Formally: $\mu(P) := \bar{\omega}(\mu(P))$. Finally, the *Commit* statement adds the plan name P to the set ρ of plans which are ready to be executed.

Note that this rule does not yield a Boolean indicating whether the *Commit* operation succeeded. This is so because when all *Negotiators* take part in the *Commit* operation by synchronously executing the *Commit* statement, the operation always succeeds. If not all *Negotiators* are present, then the global transition cannot be taken, so the agents wait until all *Negotiators* *Commit* to the plan. As the planning function $\bar{\omega}$ yields a definite plan, it is safe to add the plan to the set ρ of definite plans which are ready to be executed.

Assuming that $J \subseteq \mathcal{I}$ is a set of agent names, $P \in \mathcal{PN}$ is a plan name, $A_\iota = \langle S_\iota, \pi_\iota \rangle$ and $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$ (*Commit*) of Figure 2 gives the global transition rule for the *Commit* statement, where the following conditions hold:

- $\mu(P) \models_{cs} Negotiators = J$
- $\mu' = \mu[\bar{\omega}(\mu(P)) / P]$

3.7 Semantics of *Execute*

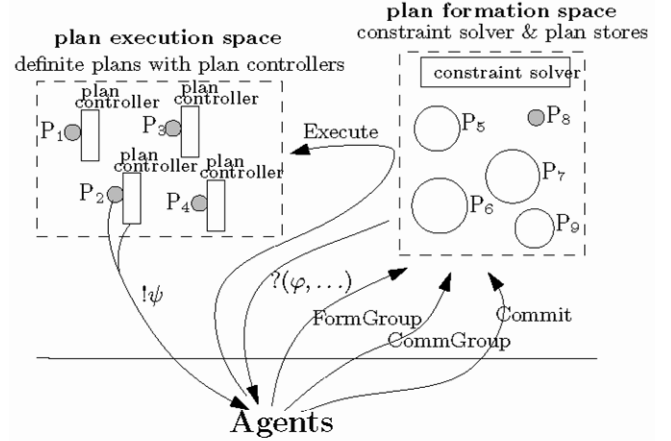
After committing to a plan, the plan creation phase for this plan has ended and the plan execution can start. To start the execution phase, we have *Execute*(p), which simply means ‘execute the plan referred to by p ’. This statement again has to be synchronously performed by the negotiators of the plan and it results in starting a plan controller for the plan referred to by p . A plan controller is a special process that coordinates and synchronises plan execution and takes care of proper execution of the plan. The global semantics of an *Execute* statement transfers the plan name from the set ρ (plans ready to be executed) to the set χ (plans in execution). As the global transition rule is relatively simple, we skip the semi-formal semantics and present the global transition rule. Again, assume that $J \subseteq \mathcal{I}$ is a set of agent names, $P \in \mathcal{PN}$ is a plan name, $A_\iota = \langle S_\iota, \pi_\iota \rangle$ and $A'_\iota = \langle S'_\iota, \pi'_\iota \rangle$. Then (execute) gives the rule of *Execute*, where the following condition holds:

$$\mu(P) \models_{cs} Negotiators = J$$

This rule only implements the *start* of the plan execution, in which the plan is transferred to the set of plans in execution. The plan controller which is associated with the plan then directs the agents that are *Actors* of the plan to execute the actions of the plan. We will restrict ourselves to an informal account of plan execution, because plan execution takes place at the lower agent level. An execution step of a plan controller amounts to finding the minimal elements of the partial order of the plan and executing these, either simultaneously or one after the other. Execution of an action \mathbf{a} of the plan means synchronising this action with the agent(s) involved, as laid down in $Actors(\mathbf{a})$; these agents take care of the actual execution. When the action has been done, it is removed from the set of actions of the plan. In case the action to be executed is a $!\psi$, then the goal ψ is broadcast in the hope there will be a group of agents willing to form and execute a plan resulting in ψ . If so, a new plan-execute cycle takes place and the controller of the outer plan suspends until such a plan has been made and executed. The plan made in this inner plan-execute cycle can contain other see-to-it statements, recursively resulting in new cycles. In case the plan fails during execution, for example because there are no agents willing to make a plan for a goal which has been broadcast through a see-to-it action, then the plan controller informs the negotiators of the plan, such that they can replan. The advantage of using plan controllers for plan execution over letting the agents execute the plan on their own is that it minimises communication between the actors. An example of a plan part that can be easily executed using a coordinator is $\mathbf{a} < \mathbf{b}$ (no actions in between), with $Actors(\mathbf{a}) = \{\iota_1, \iota_2\}$ and $Actors(\mathbf{b}) = \{\iota_3, \iota_4\}$. So, two consecutive group actions have to be performed by disjoint groups of agents. Without a plan controller, this would involve communication between these two groups.

As explained in the previous paragraph, the start of the execution phase is decided by the *Negotiators* of the plan, not by the *Actors*. We now have to direct our attention towards the group of actors. Until this point, we simply assumed that this group of actors will accept the plan and the role delegation made by the planners. As agents are (supposed to be) autonomous, we can only assume this if all actors were part of the group of negotiators or if all actors stand in an authority relation below at least one planner. Nothing in our coordination architecture enforces this, so it is possible that an actor is suddenly called upon by the plan controller to perform an action, while this actor agent is otherwise engaged and not willing to assist in the plan at all. It can be argued that this aspect of the architecture conflicts with agent autonomy. This problem is easily solved; we could add a coordination statement to the language to inform actors which are not negotiators about the contents of the plan. Then, we change the semantics of the *Execute*-statement by demanding that both the negotiators and the actors synchronously execute it. This way, the actors can refuse their cooperation. If they agree to take part in the plan execution, a (joint) commitment towards the plan can be established among the acting agents.

Figure 3 The distributed plan coordination architecture



The reader might wonder why we opted for separate groups of negotiators and actors, which can overlap, coincide or even be disjoint. The reason for this is that we gain expressivity. Our coordination language allows ‘manager agents’ to construct plans through negotiation, which ‘labourer agents’ have to execute. The language also allows the negotiators to be part of the group of actors or vice versa. These possibilities yield a coordination language which is suitable for writing coordination protocols for different organisational profiles.

4 The coordination architecture

In Figure 3, we depict the coordination architecture described earlier.

There are two levels in the architecture. In the lower level are the agents, programmed in some (agent) programming language. The ‘ordinary’ statements of the agent programs are executed in the lower level. The higher level is the plan coordination level, on which we focus in this paper. This level contains two spaces: the plan formation space and the plan execution space. The plan formation space contains the plan stores of plans still in the plan creation phase. In the figure, each store is indicated by a circle. The size of the circle indicates the size of the set of plans, which are consistent with the constraints in the store; that is, the smaller the circle, the further the creation of the plan has advanced. The little grey circles indicate plan stores that are definite and thus only allow one plan. These plan stores can be equated with plans. The constraint solver interacts with the plan stores, checks the constraints the negotiating agents want to add to plan stores and uses its planning knowledge to aid in creating and finishing the plan. The plan execution space contains finished plans, which are being executed. Each plan has an associated plan controller to coordinate plan execution with the agents involved.

Looking at the global system configuration $\langle \{A_\iota \mid \iota \in \mathcal{I}\}, \chi, \rho, \mu \rangle$, we can pinpoint the spaces of the architecture: $\{A_\iota \mid \iota \in \mathcal{I}\}$ are the agents in the lower level, χ are the plans in the plan execution space, ρ matches the

plans that are definite (the little grey circles from Figure 3) and μ are the other plans in the plan formation space.

As the figure shows, there can be many plans under construction and in execution simultaneously. This feature of the architecture is useful, as independent plan coordination activities can be done in parallel. One group of agents does not have to wait until another group has finished executing its plan before starting on a new plan. Of course, multiple plans of possibly overlapping groups of agents require more deliberation and communication of the agents, to decide whether they are available to contribute to a plan. This reasoning takes place at the lower agent level.

The programs of the agents contain statements from the plan coordination language, which are executed at the higher level. The arrows in the figure mainly indicate data flow. For example, when agents do a `CommGroup`, information is added to a certain plan store. When agents test a plan store, agents obtain information about the plan being formed. Execution of coordination statements in the plan formation store involves the constraint solver. For example, when a group of negotiators commits to a plan, then the constraint solver checks whether this plan is definite and if not, uses its planning knowledge to arrive at a definite plan. When a finished plan is executed, it moves from the plan formation space to the plan execution space and gets a plan controller associated. The plan controller coordinates the agents to perform the actions of the plan, according to the order dictated by the partial order. So, if the next action to be executed is a group action for agents $\{t_2, t_5, t_6\}$, the plan controller synchronises with these agents, such that they perform the action together. When the next action to be executed is a see-to-it action, the plan controller broadcasts this to the agents, such that a new distributed plan-execute cycle can start, inside the current one. After execution of the plan is finished, the plan is disposed of. In case a plan fails during execution, for example because not all actors are present to execute a group action or because no agent reacts to a see-to-it message, the plan controller reports the failure to the negotiators of the plan, after which it terminates.

5 Conclusions

We defined an architecture for agents that coordinate with each other in order to create and execute plans. Also, we introduced a novel coordination language which can be used to enable agents to form temporary alliances for planning and performing the group plan. Coordination languages are very useful for agent applications, as agents often are heterogeneous and thus need a common language of a high abstraction level to coordinate their behaviour. [In fact, coordination and coordination languages are increasingly recognised as being of crucial importance for the design of multi-agent systems (Ciancarini et al., 2000; Dastani et al., 2005; Omicini et al., 2001; Arbab, 2004) and one may view our work as contributing to this area.]

The coordination language presented here uses ideas from constraint programming (Saraswat, 1993) and is similar to GrAPL (de Vries et al., 2002), a language to coordinate group actions in a multi-agent setting. When we compare the coordination language to GrAPL, GrAPL is more expressive with respect to details of the actions: actions are parameterised and the agents negotiate about the parameters. A combination of GrAPL and the coordination language is well conceivable. In this new language, there are constraints on two levels, namely the constraints on the structure of the plan and the task allocation and the constraints on parameters of actions which are part of the plan.

Our work is also related to classical Linda-like coordination languages, which use a blackboard to exchange information. Instead of the blackboard, our architecture features constraint stores, which have the added benefit of logical structure and inference mechanisms.

In a way, the coordination language can be compared with agent communication languages like KQML (Finin et al., 1994). The coordination statements (`FormGroup`, `CommGroup`, `Commit` and `Execute`) correspond to speech act types and the constraints and other parameters of these statements form the content of the messages. The statements of our language are of a higher abstraction level; one can imagine implementing them in KQML. The benefit of our language is that it is specifically designed for the coordination of agents creating and executing plans and that its semantics precisely describes the manipulation of the plan stores. In order for communication and coordination to be successful, the agents must have ontological agreement on the language they use. It remains future work to show how our coordination language can be useful in negotiating about ontological issues (cf. van Diggelen et al., 2006).

The coordination architecture is generic and is to be combined with planning methods for the agents and the constraint solver. The agents use these planning algorithms to decide which constraints they want to add to a plan store and the constraint solver uses a planning algorithm to check whether the combined proposed constraints of agents indeed contribute to a plan that reaches its purpose, as well as to complete the plan when the agents have committed to it. By abstracting from specific planning methods, the coordination language can be suitably used by agents using different planning approaches.

Plans in our approach are partially ordered sets of actions. This matches with the plan format in partial order planning (e.g., Penberthy and Weld, 1992), where plans are formed by gradually constraining the action set and the order of the actions, as is necessary to obtain the goals of the plan. Planning algorithms like these can be very well combined with constraint satisfaction techniques, to be implemented in the constraint solver of our coordination architecture. Other, related planning methods which combine well with our coordination architecture are distributed hierarchical task network planners (desJardins and Wolverson, 1999), which work by gradually refining a goal into subgoals and/or actions. Recently, a new

generation of fast planners like Graphplan (Blum and Furst, 1997) and satisfiability-based planning (Kautz and Selman, 1998) has arisen. These planners are not of the usual deductive kind, but use satisfiability instead. As this view matches with constraint satisfaction, these new planners are very usable in the coordination architecture. There are also planners that formalise the planning problem as a constraint satisfaction problem and then use constraint solving to achieve a solution (van Beek and Chen, 1999). This results in a very fast planner, according to van Beek and Chen (1999). In this planning method, the variables which are constrained describe the *states* which are achieved by performing actions, while in our formalism the variables (plan features) describe the *actions* in the plan and their properties. The multi-agent constraint-based planner MACBeth (Goldman et al., 2000) combines constraint programming with hierarchical task network planning. The (human) user and the planner cooperate to tailor a sketchy plan to a specific situation. The user posts constraints on the relations between sub-tasks in the plan under construction and on the value of task parameters. Thus, the constraints both describe demands on the structure of the plan and on the resource allocation. MACBeth uses constraint management techniques to arrive at a definite plan. This seems similar to the set-up in our coordination architecture, where the agents which negotiate about the plan first post constraints on the plan, thus narrowing down the search space for the constraint solver. But our coordination architecture is of a different nature; it is not entered around a planner (while MACBeth is), but provides a generic system structure for agents doing distributed planning and a language through which the agents can jointly post their demands on the plan.

In this paper, we only briefly touched upon mentalistic aspects of group activity (Levesque et al., 1990; Dunin-Kępicz and Verbrugge, 1996; Rao et al., 1992; Tambe, 1997). We see work in this area as valuable and establishing connections between the construction of plans and the proper mental attitudes as very relevant. But presently, we chose to focus on plan construction and execution. Another relevant field of research concerns the structuring of coordination processes, for example using coordination protocols Omicini et al. (2001). Here, we do not mean the lower-level coordination protocols used to implement the coordination statements, but the higher-level coordination protocols constructed using statements from the coordination language. Structuring coordination is essential, to guide the agents in using the proper coordination statements at the right times. This prevents them from arbitrarily trying to achieve agreement on matters, which will take a lot of communication effort. Our coordination language can be used to implement coordination protocols.

Our coordination architecture and language enable the agents to negotiate about demands on the plan being created by posing constraints on the plan. These constraints shrink the space of possible plans, after which the planner implemented in the constraint solver can complete the plan.

This way, the preferences of the agents and the knowledge of the planner can be combined. The coordination language offers abstract and formally well-defined primitives, enabling agents to interact through the construction and execution of distributed plans.

References

- Arbab, F. (2004) 'REO: a channel-based coordination model for component', *Composition Mathematical Structures in Computer Science*, Vol. 14, No. 3, pp.329–366.
- Blum, A. and Furst, M. (1997) 'Fast planning through planning graph analysis', *Artificial Intelligence*, Vol. 90, Nos. 1–2, pp.281–300.
- Ciancarini, P., Omicini, A. and Zambonelli, F. (2000) 'Multi-agent system engineering: the coordination viewpoint', in Jennings, N.R. and Lesperance, Y. (Eds.): *Intelligent Agents VI. Agent Theories, Architectures and Languages*, LNAI 1757, pp.250–259, Springer-Verlag, Berlin.
- Dastani, M., Arbab, F. and de Boer, F. (2005) 'Coordination and composition in multi-agent systems' in Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M.P. and Wooldridge, M. (Eds.): *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS'05)*, pp.39–446, ACM Press, New York.
- de Vries, W., de Boer, F.S., Hindriks, K.V., van der Hoek, W. and Meyer, J-J.Ch. (2002) 'A programming language for coordinating group actions', in Dunin-Kępicz, B. and Nawarecki, E. (Eds.): *From Theory to Practice in Multi-Agent Systems, Proceedings of the 2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS'01)*, pp.313–321, LNAI 2296, Springer, Berlin.
- desJardins, M. and Wolverson, M. (1999) 'Coordinating a distributed planning system', *AI Magazine*, Vol. 20, No. 4, pp.45–53.
- Dunin-Kępicz, B. and Verbrugge, R. (1996) 'Collective commitments', in Tokoro, M. (Ed.): *Proceedings of the 2nd International Conference on Multi-agent Systems (ICMAS'96)*, pp.56–63, AAAI Press, Menlo Park, CA.
- Finin, T., McKay, D., Fritzson, R. and McEntire, R. (1994) 'KQML: an information and knowledge exchange protocol', in Fuchi, K. and Yokoi, T. (Eds.): *Knowledge Building and Knowledge Sharing*, Ohmsa and IOS Press.
- Goldman, R.P., Haigh, K.Z., Musliner, D.J. and Pelican, M. (2000) 'MACBeth: a multi-agent constraint-based planner', in *Working Notes of the AAAI Workshop on Constraints and AI Planning*, AAAI.
- Hindriks, K.V., de Boer, F.S., van der Hoek, W. and Meyer, J-J.Ch. (1998) 'Formal semantics for an abstract agent programming language', in Singh, M., Rao, A. and Wooldridge, M (Eds.): *Intelligent Agents IV, Proceedings of the 4th International Workshop on Agent Theories, Architectures and Languages (ATAL 1997)*, pp.215–229, LNAI 1365, Springer, Berlin.
- Kautz, H. and Selman, B. (1998) 'Blackbox: a new approach to the application of theorem proving to problem solving', in *AIPS'98 Workshop on Artificial Intelligence Planning Systems*, pp.58–60.

- Levesque, H.J., Cohen, P.R. and Nunes, J.T. (1990) 'On acting together', in *Proceedings of the National Conference on Artificial Intelligence*, pp.94–99, AAAI Press, Menlo Park, CA.
- Omicini, A., Zambonelli, F., Klusch, M. and Tolksdorf, R., (Eds.) (2001) *Coordination of Internet Agents – Models, Technologies and Applications*, Springer, Berlin.
- Penberthy, J.S and Weld, D.S. (1992) 'UCPOP: a sound, complete, partial order planner for ADL', in Nebel, B., Rich, C. and Swartout, W. (Eds.): *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, pp.103–114, Morgan Kaufmann Publishers, Cambridge, MA.
- Rao, A.S., Georgeff, M.P. and Sonenberg, E.A. (1992) 'Social plans: a preliminary report', in Werner, E. and Demazeau, Y. (Eds.): *Decentralized AI 3, Proceedings of the 3rd European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAA-MAW'91)*, pp.57–76, Elsevier, Amsterdam.
- Saraswat, V.A. (1993) *Concurrent Constraint Programming*, MIT Press, Cambridge, Massachusetts.
- Shoham, Y. (1993) 'Agent-oriented programming', *Artificial Intelligence*, Vol. 60, pp.51–92.
- Tambe, M. (1997) 'Towards flexible teamwork', *Journal of Artificial Intelligence Research*, Vol. 7, pp.83–124.
- Tsang, E.P.K (1993) *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego.
- van Beek, P. and Chen, X. (1999), 'CPlan: a constraint programming approach to planning', in *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pp.585–590, AAAI Press/The MIT Press, Stanford, CA.
- van Diggelen, J., Beun, R.J., Dignum, F., van Eijk, R.M. and Meyer, J-J.Ch. (2006) 'ANEMONE: an effective minimal ontology negotiation environment', in Stone, P. and Weiss, G. (Eds.): *Proceedings of the Fifth International Joint Conference On Autonomous Agents and Multi-agent Systems (AAMAS06)*, pp.899–906, ACM Press, New York.
- Weld, W.S. (1999) 'Recent advances in AI planning', *AI Magazine*, Vol. 20, No. 2, pp.93–123.
- Wooldridge, M. (2002) *An Introduction to Multi-Agents Systems*, Wiley, Chichester, England.
- Wooldridge, M. and Jennings, N.R. (1995) 'Intelligent agents: theory and practice', *The Knowledge Engineering Review*, Vol. 10, No. 2, pp.115–152.