Modeling Mental States in the Analysis of Multiagent Systems Requirements

Alexei Lapouchnian[1] and Yves Lespérance[2]

[1] Department of Computer Science, University of Toronto, Toronto, ON, M5S 3G4, Canada, alexei@cs.toronto.edu

[2] Department of Computer Science and Engineering, York University, Toronto, ON, M3J 1P3, Canada, lesperan@cs.yorku.ca

Abstract. This paper describes an agent-oriented requirements engineering approach that combines informal i^* models with formal specifications in the multiagent system specification formalism CASL. This allows the requirements engineer to exploit the complementary features of the frameworks. i^* can be used to model social dependencies between agents and how process design choices affect the agents' goals. CASL can be used to model complex processes formally. We introduce an intermediate notation to support the mapping between i^* models and CASL specifications. In the combined i^*-CASL framework, agents' goals and knowledge are represented as their mental states, which allows for the formal analysis and verification of, among other things, complex agent interactions where agents may have different goals and different (incomplete) knowledge. Our models can also serve as high-level specifications for multiagent systems.

1 Introduction

Modern software systems are becoming increasingly complex, with lots of intricate interactions. The recent popularity of electronic commerce, web services, etc. confirms the need for software engineering methods for constructing applications that are open, distributed, and adaptable to change. This is why many researchers and practitioners are looking at agent technology as a basis for distributed applications.

Agents are active, social, and adaptable software system entities situated in some environment and capable of autonomous execution of actions in order to achieve their set objectives [25]. Furthermore, most problems are too complex to be solved by just one agent; one must create a multiagent system (MAS) with several agents having to work together to achieve their objectives and ultimately deliver the desired application. Therefore, adopting the agent-oriented approach to software engineering means that the problem is decomposed into multiple, autonomous, interacting agents, each with their own objectives (goals). Agents in MAS frequently represent individuals, companies, etc. This means that there is an "underlying organizational context" [7] in MAS. Like humans, agents need to coordinate their activities, cooperate, request help from others, etc., often through negotiation. Unlike in object-oriented or component-based systems, interactions in multiagent systems occur through high-level agent communication languages, so these interactions are mostly viewed not at the syntactic level, but "at the knowledge level, in terms of goal delegation, etc." [7].

In requirements engineering (RE), *goal-oriented approaches* (e.g, KAOS [3]) have become prominent. In Goal-Oriented Requirements Engineering (GORE), high-level stakeholder objectives are identified as goals and later refined into fine-grained requirements assignable to agents/components in the system-to-be or in its environment. Reliance on goals makes goal-oriented requirements engineering methods and agent-oriented software engineering a great match. Moreover, agent-oriented analysis is central to requirements engineering since the assignment of responsibilities for goals and constraints among the components in the system-to-be and the agents in the environment is the main result of the RE process. Therefore, it is natural to use a goal-oriented requirements engineering approach when developing MAS. With GORE, it is easy to make the transition from the requirements to the high-level MAS specifications. For example, strategic relationships among agents will become high-level patterns of inter-agent communication. Thus, it would be desirable to devise an *agent-oriented RE approach with a formal component that supports rigorous formal analysis, including reasoning about agents' goals and knowledge*.

In the above context, while it is possible to informally analyze small systems, formal analysis is needed for any realistically-sized system to determine whether such distributed requirements imposed on each agent in a MAS are correctly decomposed from the stakeholder goals, consistent and, if properly met, achieve the system's overall objectives. Thus, the aim of this work is to devise an agent-oriented requirements engineering approach with a formal component that supports reasoning about agents' goals (and knowledge), thereby allowing for rigorous formal analysis of the requirements expressed as the objectives of the agents in a MAS.

In our approach, we integrate the i^* modeling framework [27] with CASL [19, 18], a formal agent-oriented specification language that supports the modeling of agent mental states. This gives the modeler the flexibility and intuitiveness of the i^* notation as well as the powerful formal analysis capability of CASL. To bridge the gap between informal i^* diagrams and formal CASL specifications we propose an intermediate notation that can be easily obtained from i^* models and then mapped into CASL. With our i^*-CASL-based approach, a CASL model can be used both as a requirements analysis tool and as a formal high-level specification for a multiagent system that satisfies the requirements. This model can be formally analyzed using the CASLve [20, 18] tool or other tools and the results can be fed back into the requirements model.

One of the main features of this approach is that goals (and knowledge) are assigned to particular agents thus becoming their subjective attributes as opposed to being objective system properties as in many other approaches (e.g., Tropos [1] and KAOS [3]). This allows for the modeling of conflicting goals, agent negotiation, information exchange, complex agent interaction protocols, etc.

The rest of the paper is organized as follows: Section 2 briefly describes the concepts of i^* and CASL, Section 3 discusses our approach in detail, and Section 4 concludes the paper.

2 Background

2.1 The i^* Framework

i^* [27] is an agent-oriented modeling framework that is mostly used for requirements engineering. i^* centers on the notion of *intentional actor* and *intentional dependency*. Actors are described in their organizational setting and have attributes such as goals, abilities, beliefs, etc. In i^*, an actor can use opportunities to depend on other actors in achieving its objectives, at the same time becoming vulnerable if those actors do not deliver. Dependencies in i^* are *intentional* since they appear as a result of actors pursuing their goals.

To illustrate the approach, we use a variant of the meeting scheduling problem, which has become a popular exemplar in Requirements Engineering. In the context of the i^* modeling framework a meeting scheduling process was first analyzed in [27]. We modified the meeting scheduling process to make our models easier to understand. For instance, we take the length of meetings to be the whole day. We also introduced a legacy software system called the Meeting Room Booking System (MRBS) that handles the booking of meeting rooms. The complete case study is presented in [8].

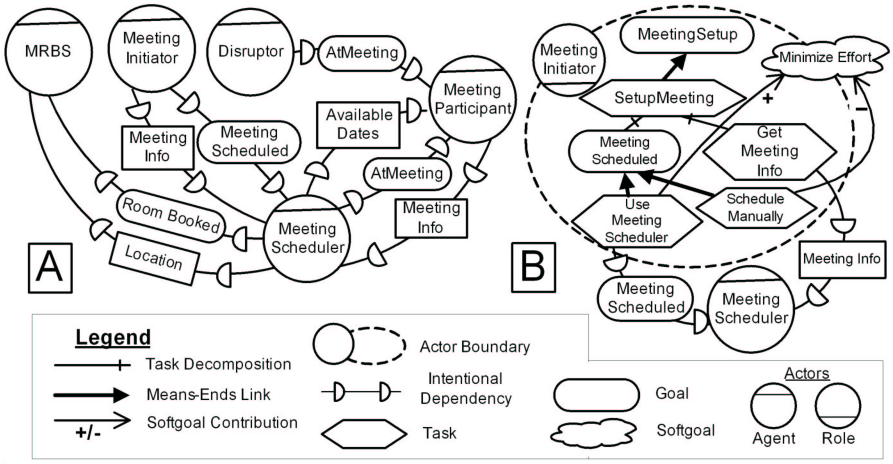

Fig. 1. SD (A) and SR (B) models for the meeting scheduler process

The i^* framework has two main components: the *Strategic Dependency (SD) model* and the *Strategic Rationale (SR) model*. The former describes the external relationships among actors, while the latter focuses on exploring the rationale behind the processes in organizations from the point of view of participating actors. SD models are networks of actors (which can be agents, positions, and roles) and dependencies. Depending actors are called *depender*s and depended-upon actors are called *dependee*s. There can be four types of dependencies based on what is being delegated — a goal, a task, a resource,

and a softgoal. Softgoals are related to the notion of non-functional requirements [2] and model quality constraints on the system.

Fig. 1A is the SD diagram showing the computerized Meeting Scheduler (MS) agent in its environment. Here, the role Meeting Initiator (MI) depends on the MS for scheduling meetings and for being informed about the meeting details. The MS, in turn, depends on the MP (Meeting Participant) role for attending meetings and for providing his/her available dates to it. The MS uses the booking system to book rooms for meetings. The Disruptor actor represents outside actors that cause changes in participants' schedules, thus modeling the environment dynamics.

SR models enable the analyst to assess alternatives in the definition of actor processes to better address their concerns. Four types of nodes are used in SR models — goals, tasks, softgoals, and resources — and three types of links — means-ends links, task decompositions links, and softgoal contribution links. Means-ends links specify alternative ways to achieve goals; task decomposition links connect tasks with components needed for their execution. For example, Fig. 1B is a simple SR model showing some details of the MI process. To schedule meetings, the MI can either do it manually, or delegate it to the scheduler. Quality requirements (softgoals), such as `Minimize-Effort` in Fig. 1B, are used to evaluate these alternatives. Contribution links specify how alternatives contribute to quality constraints.

2.2 The Formal Foundations: CASL

The Cognitive Agents Specification Language (CASL) [19, 18] is a formal specification language that combines theories of action [15, 16] and mental states [17] expressed in the situation calculus [10] with ConGolog [4], a concurrent, non-deterministic agent-oriented programming language with a formal semantics. In CASL, agents' goals and knowledge are modeled formally; communication actions are provided to update these mental states and ConGolog is then used to specify the behaviour of agents. This combination produces a very expressive language that supports high-level reasoning about the agents' mental states. The logical foundations of CASL allow it to be used to specify and analyze a wide variety of MAS including non-deterministic systems and systems with an incompletely specified initial state.

CASL specifications consist of two parts: the model of the domain and its dynamics (the declarative part) and the specification of the agents' behaviour (the procedural part). The domain is modeled in terms of the following entities: 1) *primitive actions* — all changes in the domain are due to primitive actions being executed by agents; 2) *situations*, which are states of the domain that result from the execution of sequences of actions (there is a set of initial situations, with no predecessor, corresponding to the ways agents think the world might be like initially); 3) *fluents*, which are predicates and functions that may change value from situation to situation. The fluent `Room(meetingID,date,room,s)`, where `s` is a situation parameter, models the fact that a room has been booked on some day for some meeting in a situation `s`.

To specify the dynamics of an application domain, we use the following types of axioms: 1) *action precondition axioms* that describe under what conditions actions can be performed; 2) *successor state axioms* (SSA), which were introduced in [15] as a solution to the frame problem and specify how primitive actions affect fluents; 3) *initial*

state axioms, which describe the initial state of the domain and the initial mental states of the agents; 4) *other axioms* that include unique name axioms for actions and domain independent foundational axioms.

Agents' behaviour is specified using a rich high-level programming language with recursive procedures, loops, conditionals, non-determinism, concurrency, and interrupts [4]. The following table lists most of the available constructs:

α,	primitive action
$\phi?$,	wait for a condition (test)
δ_1 `;` δ_2,	sequence
$\delta_1 \mid \delta_2$,	nondeterministic branch
$\pi\, v.\, \delta$,	nondeterministic choice of argument
δ^*,	nondeterministic iteration
`if` ϕ **`then`** δ_1 **`else`** δ_2 **`endIf`**,	conditional
`while` ϕ **`do`** δ **`endWhile`**,	while loop
`for` $v : \phi$ **`do`** δ **`endFor`**,	for loop
$\delta_1 \parallel \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\!\rangle \delta_2$,	concurrency with δ_1 at a higher priority
`guard` ϕ **`do`** δ **`endGuard`**,	guard
$\langle\, \boldsymbol{v} : \phi \rightarrow \delta$ **`until`** $\alpha \,\rangle$,	interrupt
$p(\boldsymbol{\theta})$,	procedure call

The guard operator (defined in [8]) blocks execution of the program δ until its condition ϕ holds. For an interrupt operator, when its condition ϕ holds for some value of $\boldsymbol{v}$, the interrupt triggers and the body δ is executed. Afterwards, the interrupt may trigger again provided that the cancellation condition α does not hold. The "for loop" construct is defined in [18].

Also, CASL supports formal modeling of agents' goals and knowledge. The representation for both goals and knowledge is based on a *possible worlds semantics* incorporated into the situation calculus, where situations are viewed as possible worlds [11, 17]. CASL uses accessibility relations `K` and `W` to model what an agent knows and what it wants respectively. `K(agt,s',s)` holds if the situation `s'` is compatible with what the agent `agt` knows in situation `s`, i.e., in situation `s`, the agent thinks that it might be in the situation `s'`. In this case, the situation `s'` is called `K`-*accessible*. When an agent does not know the value of some formula ϕ, it considers possible (formally, `K`-accessible) some situations where ϕ is true and some where it is false. An agent knows some formula ϕ if ϕ is true in all its `K`-accessible situations: **`Know`**`(agt,`ϕ`,s)` $=$ $\forall$ `s'(K(agt,s',s)` $\supset$ ϕ`[s'])`. Constraints on the `K` relation ensure that agents have positive and negative introspection (i.e., agents know whether they know/don't know something) and guarantee that what is known is true. Communication actions such as `inform` are used for exchanging information among agents. The preconditions for the `inform` action ensure that no false information is transmitted. The changes to agents' knowledge due to communication and other actions are specified by the SSA for the `K` relation. The axiom ensures that agents are aware of the execution of all actions. This formal framework is quite simple and idealized. More complex versions of the SSA can be specified, for example, to handle encrypted messages [19] or to provide

belief revision [21]. For convenience, abbreviations **KWhether**(agt,ϕ,s), which means that the agent knows either ϕ or its negation, and **KRef**(agt,θ,s), which indicates that the agent knows the value of θ, are used.

The accessibility relation W(agt,s′,s) holds if in situation s an agent considers that everything that it wants to be true actually holds in s′, which is called W-accessible. **Goal**(agt,ψ,s) indicates that in situation s the agent agt has the goal that ψ holds. The definition of **Goal** says that ψ must be true in all W-accessible situations that have a K-accessible situation in their past. This way, while agents may want something they know is impossible to obtain, the goals of agents must be consistent with what they currently know. In our approach, we mostly use achievement goals that specify the desired states of the world. We use the formula **Goal**(agt,**Eventually**(ψ),s) to state that agt has the goal that ψ is eventually true. The request and cancelRequest actions are used to request services and cancel these requests. Requests are used to establish intentional dependencies among agents. The dynamics of the W relation, which is affected by request, etc., are specified by a SSA. There are constraints on the W and K relations, which ensure that agents' goals are consistent and that agents introspect their goals. See [18, 19] for more details about CASL, as well as [8] for details about how we have adapted it for use with i^* for requirements engineering.

3 The i^*-CASL Process

3.1 Increasing Precision with iASR Models

Our aim in this approach is to tightly associate SR models with formal specifications in CASL. The standard SR diagrams are geared to informal analysis and can be very ambiguous. For instance, they lack details on whether the subtasks in task decompositions are supposed to be executed sequentially, concurrently, under certain conditions, etc. CASL, on the other hand, is a precise language. To handle this precision mismatch we use Intentional Annotated SR (iASR) models that help in bridging the gap between SR models and CASL specifications. Our goal is to make iASR models precise graphical representation for the procedural component of CASL specifications. The starting point for developing an iASR diagram for an actor is the regular SR diagram for that actor (e.g., see Fig. 1B). It can then be appropriately transformed into an iASR model through the steps described below.

Annotations. The main tool that we use for disambiguating SR models is *annotations*. Annotations allow analysts to model the domain more precisely and to capture data/control dependencies among goals and other details. Annotations, proposed in [24] for use with SR models and ConGolog, are textual constraints on iASR models and can be of several types: composition and link annotations, and applicability conditions. Composition annotations (specified by σ in Fig. 2A) are applied to task and means-ends decompositions and specify how the subtasks/subgoals are to be combined to execute/achieve the supertask/supergoal. Four types of compositions are allowed: sequence (";"), which is default for task decompositions, concurrency ("||"), prioritized concurrency ("⟫"), and alternative ("|"), which is the default for means-ends decompositions. These annotations are applied to subtasks/subgoals from left to right. The choice of composition annotations is based on the ways actions and procedures can be

composed together in CASL. In some approaches, for example, the Trust-Confidence-Distrust method of [6] that also uses i^*, sequencing among subtasks/subgoals is captured using precedence links. We believe that the arrangement of nodes from left to right based on their sequence/priority/etc. simplifies the understanding of models.

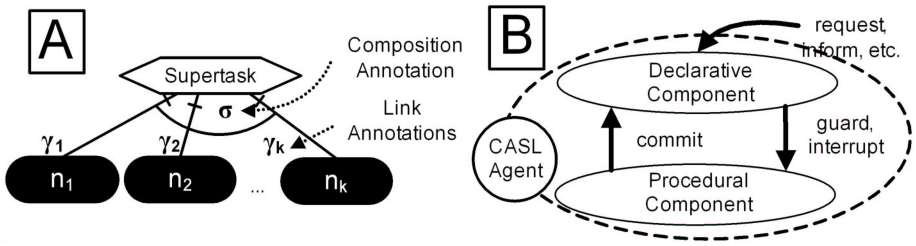

Fig. 2. Specifying annotations (A) and using them to synchronize procedural and declarative components (B) in CASL specifications

Link annotations (γ_i in Fig. 2A) are applied to subtasks/subgoals (n_i) and specify how/under what conditions they are supposed to be achieved/executed. There are six types of link annotations (corresponding to CASL operators): *while* and *for* loops, the *if* condition, the pick, the interrupt, and the guard. The pick (`pick(variableList, condition)`) non-deterministically picks values for variables in the subtask that satisfy the condition. The interrupt (`whenever(variableList,condition, cancelCondition)`) fires whenever there is a binding for the variables that satisfies the condition unless the cancellation condition becomes true. Guards (`guard(condition)`) block the subtask's execution until the condition becomes true. The absence of a link annotation on a particular decomposition link indicates the absence of any conditions on the subgoal/subtask.

The third annotation type is the *applicability condition* (`ac(condition)`). It applies to means-ends links used with goal achievement alternatives and specifies that the alternative is only applicable when the condition holds.

Agent Goals in iASR Models. A CASL agent has procedural and declarative components. iASR diagrams only model agent processes and therefore can only be used to represent the procedural component of CASL agents. The presence of a goal node in an iASR diagram indicates that the agent knows that the goal is in its mental state and is prepared to deliberate about whether and how to achieve it. For the agent to modify its behaviour in response to the changes to its mental state, it must detect that change and synchronize its procedural and declarative components (see Fig. 2B). To do this we use interrupts or guards with their conditions being the presence of certain goals or knowledge in the mental state of the agent. Procedurally a goal node is interpreted as invoking the means to achieve it.

In CASL, as described in [19], only communication actions have effects on the mental state of the agents. We, on the other hand, would like to let agents change their mental state on their own by executing the action `commit(agent,`ϕ`)`, where ϕ is a

formula that the agent/modeler wants to hold. Thus, in iASR diagrams all agent goals must be acquired either from intentional dependencies or by using the `commit` action. By introducing goals into the models of agent processes, the modeler captures the fact that multiple alternatives exist in these processes. Moreover, the presence of goal nodes suggests that the designer envisions new possibilities for achieving these goals. By making the agent acquire the goals, the modeler makes sure that the agent's mental state reflects the above intention. By using the `commit` action, the modeler is able to "load" the goal decomposition hierarchy into the agents' mental states. This way the agents would be able to reason about various alternatives available to them or come up with new ways to achieve their goals at runtime. This is unlike the approach in [24] where agent goals had to be operationalized before being formally analyzed.

Softgoals. Softgoals (quality requirements) are imprecise and thus are difficult to handle in a formal specifications language. Therefore, we use softgoals to help in choosing the best process alternatives (e.g., by selecting the ones with the best overall contribution to all softgoals in the model) and then remove them before iASR models are produced. Alternatively, softgoals can be operationalized or metricized, thus becoming hard goals. Since in this approach softgoals are removed from iASR models, applicability conditions can be used to capture in a formal way the fitness of the alternatives with respect to softgoals (this fitness is normally encoded by the softgoal contribution links in SR diagrams). For example, one can specify that phoning participants to notify them of the meeting details is applicable only in cases with few participants (see Fig. 5), while the email option is applicable for any number of participants. This may be due to the softgoal "Minimize Effort" that has been removed from the model before the iASR model was produced.

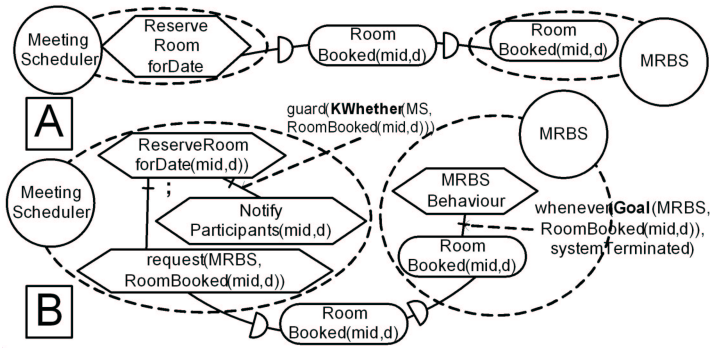

Fig. 3. Adding iASR-level agent interaction details

Providing agent interaction details. i^* usually abstracts from modeling any details of agent interactions. CASL, on the other hand, models high-level aspects of inter-agent communication. Because of the importance of agent interactions in MAS, in order to formally verify MAS specifications in CASL, all high-level aspects of agent interaction must be provided in the iASR models. This includes the tasks that request services or

information from agents in the system, the tasks that supply the information or inform about success or failure in providing the service, etc. We assume that the communication links are reliable.

For example, the SR model with the goal dependency `RoomBooked` (see Fig. 1A) in Fig. 3A is refined into the iASR model in Fig. 3B showing the details of the requests, the interrupts with their trigger conditions referring to mental states of the agents, etc. Here, the parameter `mid` ("meeting ID") is a unique meeting identifier. Since achieved goals remain in the mental state of an agent, all interrupts dealing with the acquisition of goals through dependencies must be triggered only for new instances of these goals. We usually leave these details out in iASR models. For instance, we have left out the part of the interrupt condition that makes sure that only unachieved instances of the goal `RoomBooked` trigger the interrupt in Fig. 3B. We present an example of the fully specified interrupt in the next section.

3.2 Mapping iASR Diagrams into CASL

Once all the necessary details have been introduced into an iASR diagram, it can be mapped into the corresponding formal CASL model, thus making the iASR model amenable to formal analysis.

The modeler defines a mapping **m** that maps every element (except for intentional dependencies) of an iASR model into CASL. Specifically, agents are mapped into constants that serve as their names as well as into CASL procedures that specify their behaviour; roles and positions are mapped into similar procedures with an agent parameter so that they can be instantiated by individual agents. For concrete agents playing a number of roles, the procedures corresponding to these roles will be combined to specify the overall behaviour of the agent. These procedures are normally executed in parallel. However, one may also use prioritized concurrency, which is available in CASL, to combine agent's roles. Leaf-level task nodes are mapped into CASL procedures or primitive actions; composition and link annotations are mapped into the corresponding CASL operators, while the conditions present in the annotations map into CASL formulas.

Mapping Task Nodes. A task decomposition is automatically mapped into a CASL procedure that reflects the structure of the decomposition and all the annotations. Fig.

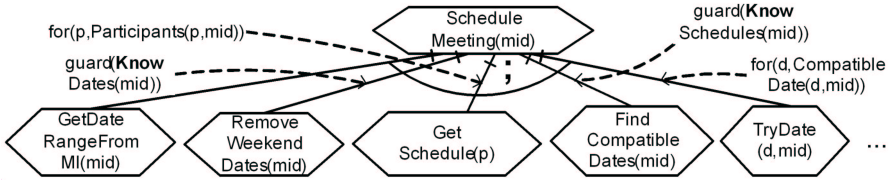

Fig. 4. Example iASR task decomposition

4 shows how a portion of the Meeting Scheduler's task for scheduling meetings can be

decomposed. This task will be mapped into a CASL procedure with the following body (it contains portions still to be recursively mapped into CASL; they are the parameters of the mapping **m**):

```
proc ScheduleMeeting(mid)
  m(GetDateRangeFromMI(mid));
  guard m(KnowDates(mid)) do
    m(RemoveWeekendDates(mid))
  endGuard;
  for p: m(Ptcp(mid)) do m(GetSchedule(p)) endFor;
  guard m(KnowSchedules(mid)) do
    m(FindCompatibleDates(mid))
  endGuard;
  for d: m(CompatibleDate(d,mid)) do
    m(TryDate(d,mid))
  endFor;
  ...
endProc
```

Note how the body of the procedure associated with the `ScheduleMeeting` task is composed of the results of the mapping of its subtasks with the annotations providing the composition details. This procedure can be mechanically generated given the mapping for leaf-level tasks and conditions.

Leaf-level tasks in our approach can be mapped either into primitive actions or CASL procedures. While mapping leaf-level tasks into CASL procedures may reduce model size and increase the level of abstraction (since in this way further details of agent processes will be hidden inside these procedures), restricting the mapping of the leaf-level tasks to primitive actions with the same name allows the CASL procedures to be automatically constructed from these actions based on iASR annotations.

Mapping Goal Nodes. In our approach, an iASR goal node is mapped into a CASL formula, which is the formal definition for the goal, and an achievement procedure, which encodes how the goal can be achieved and is based on the means-ends decomposition for the goal in the iASR diagram. For example, a formal definition for `MeetingScheduled(mid,s)` could be: $\exists$`d [AgreeableDate(mid,date,s)` $\wedge$ `AllAccepted(mid,date,s)` $\wedge$ `RoomBooked(mid,date,s)]`. This says that there must be a date agreeable for everybody on which a room is booked and all participants agree to meet. This definition is too ideal since it is not always possible to schedule a meeting. One can *deidealize* [22] `MeetingScheduled` to allow for the possibility of no common available dates or no available meeting rooms. To weaken the goal appropriately, one needs to know when the goal cannot be achieved. Modeling an achievement process for a goal using an iASR diagram allows us to understand how that goal can fail and thus iASR models can be used to come up with a correct formal definition for the goal. The following is one possibility for deidealizing the goal `MeetingScheduled`:

```
MeetingScheduledIfPossible(mid,s)=
//1. The meeting has been successfully scheduled
SuccessfullyScheduled(mid,s) ∨
//2. No agreeable (suitable for everybody) dates
∀d[IsDate(d) ⊃ ¬ AgreeableDate(mid,d,s)] ∨
//3. For every agreeable date at least one participant declined
∀d[AgreeableDate(mid,d,s) ⊃ SomeoneDeclined(mid,d,s)] ∨
//4. No rooms available
∀d[SuggestedDate(mid,d,s) ∧ AllAccepted(mid,d,s)] ⊃
   RoomBookingFailed(mid,date,s)]
```

The ability of CASL agents to reason about their goals gives us an opportunity to avoid maintaining agents' schedules explicitly in the meeting scheduler example. Instead, we rely on the presence of goals `AtMeeting(participant,mid,date,s)` in the agents' mental states as indications of the participants' willingness and intention to attend meetings on specific dates (the absence of meeting commitments indicates an available time slot). Then, we can use the initial state axiom below (which can be shown to persist in all situations) to make the agents know that they can only attend one meeting per time slot (day):

```
∀agt[Know(agt,∀p,mid1,mid2,date[
  AtMeeting(p,mid1,date,now) ∧ AtMeeting(p,mid2,date,now)
   ⊃ mid1=mid2],S0)]
```

Since CASL prevents agents from acquiring conflicting goals, requests from the Meeting Scheduler that conflict with already acquired `AtMeeting` goals will not be accommodated.

Generating Goal Achievement Procedures. The achievement procedures for goals are automatically constructed based on the means for achieving them and the associated annotations (see Fig. 5). By default, the alternative composition annotation is used, which means that some applicable alternative will be non-deterministically selected (other approaches are also possible). Note that the applicability condition (`ac`) maps into a guard operator to prevent the execution of an unwanted alternative.

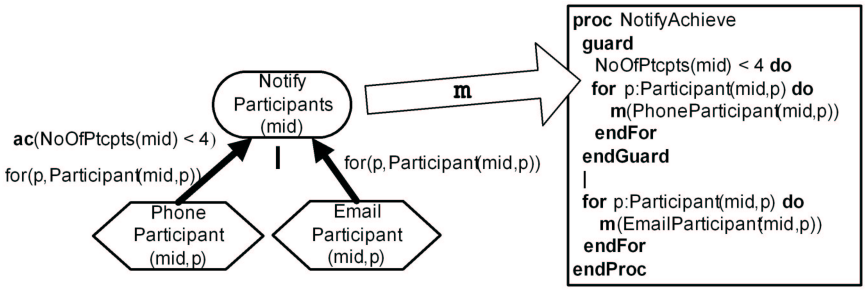

Fig. 5. Generating achievement procedures

Generally, an agent's means to achieve its goals typically work, but it is rare that they will always work. Thus, we cannot guarantee that the means for achieving the goal that are represented in the achievement procedure for that goal are always capable of achieving it. We therefore state that the achievement procedure will sometimes achieve the goal, instead of saying that it will, in fact, always achieve the goal. This semantic constraint is expressed in the following formula. It says that there exist situations `s` and `s'` such that the achievement procedure starts executing in `s`, terminates successfully in `s'`, and the CASL formula representing the agent's goal holds in `s'`:

$$\exists s, s' . \mathbf{Do}(\mathtt{AchievementProcedure}, s, s') \wedge \mathtt{GoalFormula}[s']$$

However, if one needs more assurance that the goal will, in fact, be achieved, then one must use agent *capabilities* in place of the regular procedures. There has been a lot of work on capabilities in the agent community (e.g., [13]). Here, we view goal capabilities as goal achievement procedures that are guaranteed to succeed under certain circumstances. Goal capabilities (task capabilities are also discussed in [8]) are represented in iASR models by special nodes (see Fig. 6) that are mapped into a CASL formula that represents the goal of the capability, the achievement procedure that is constructed similarly to a goal achievement procedure, a context condition that must hold when the capability is invoked, and the specification of the behaviour of the other agents in the environment that is compatible with the capability. The following formula describes the constraints on goal capabilities:

$$\forall s . \mathtt{ContextCond}(s) \supset \mathbf{AllDo}((\mathtt{AchieveProc}; \mathtt{GoalFormula}?) \parallel \mathtt{EnvProcessesSpec}, s)$$

The formula states that if we start the execution in a situation where the context condition of the capability holds, then all possible executions of the goal achievement procedure in parallel with the allowable environment behaviours terminate successfully and achieve the goal. The designer needs to determine what restrictions must be placed on the processes executing concurrently with the capability. One extreme case is when no concurrent behaviour is allowed (`EnvProcessesSpec = nil`). The other extreme is to allow the execution of any action. Of course, the specification for most capabilities will identify concrete behaviours for the agents in the environment, which assure the successful execution of the achievement procedure. For example, suppose an agent has a goal capability to fill a tank with water. It is guaranteed to succeed unless the tank's valve is opened. Here is the corresponding specification for the outside processes compatible with the capability (pick an action; if the action is not `openValve`, execute it; iterate):

$$\mathtt{EnvProcessesSpec} = (\pi\, \mathtt{action} . (\mathtt{action} \neq \mathtt{openValve})? \; ; \; \mathtt{action})^{\leq k}$$

Here, $\delta^{\leq k}$ represents a bounded form of nondeterministic iteration where δ can be executed up to k times (k is a large constant). We must bound the number of environment actions, otherwise the process will have nonterminating executions.

Fig. 6 shows the graphical notation for capabilities. Here, `T1Cap` is a task capability that executes `Task_1`, while `G1Cap` is a goal capability that achieves the goal `G1`.

Note that `G1Cap` shows the internals of the capability, while `T1Cap` is an abbreviated form that hides all the details of the capability. Detailed discussion of capabilities in this approach is presented in [8].

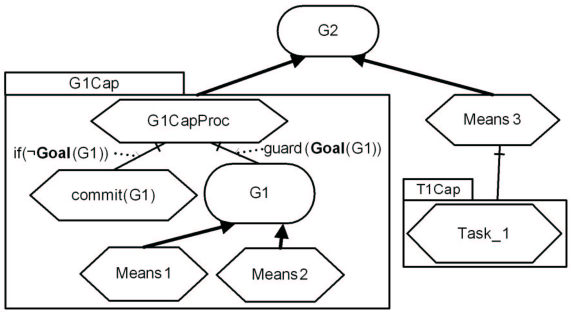

Fig. 6. Using goal and task capability nodes in iASR models

Modeling Dependencies. Intentional dependencies are not mapped into CASL per se — they are established by agent interactions. iASR tasks requesting help from agents will generally be mapped into actions of the type `request(FromAgt,ToAgt,` **`Eventually`**`(`ϕ`))` for an achievement goal ϕ. We add a special abbreviation **`DoAL`**`(`δ`,s,s'`) (Do At Least) to be used when establishing task dependencies. It stands for **`Do`**`(`$\delta \parallel (\pi$`a.a)*,s,s'`), which means that the program δ must be executed, but that other actions may occur. Thus, to ask an agent to execute a certain known procedure, the depender must request it with: `request(FromAgt,ToAgt,` **`DoAL`**`(SomeProcedure))`.

In order for an intentional dependency to be established we also need a commitment from a dependee agent to act on the request from the depender. Thus, the dependee must monitor its mental state for newly acquired goals. Here is an interrupt that is used by the MP to check for a request for the list of its available dates:

$\langle$`mid:`
`Goal``(mp,`**`DoAL`**`(InformAvailableDates(mid,MS),now,then))` $\wedge$
`Know``(mp,` $\neg\exists$`s,s'(s` $\preceq$ `s'` $\preceq$ `now` $\wedge$
  `DoAL``(InformAvailDates(mid,MS),s,s'))` $\rightarrow$
`InformAvailDates(mid,MS)`
`until` `SystemTerminated` $\rangle$

Here, if the MP has the goal to execute the procedure `InformAvailDates` and knows that it has not yet executed it, the agent sends the available dates. The cancellation condition `SystemTerminated` indicates that the MP always monitors for this goal. Requesting agents use similar interrupt/guard mechanism to monitor for requested information or confirmations. Cancellation conditions in interrupts allow the agents to monitor for certain requests/informs only in particular contexts (e.g., while some interaction protocol is being enacted).

The i^* notation, even if modified as presented in this paper, may not be the most appropriate graphical notation for representing agent interaction protocols since it usually concentrates on strategic dependencies and does not have facilities for modeling low-level agent interaction details. A notation like AgentUML [12] may be more suitable for this task. However, iASR models still can be used for modeling agent interactions. To illustrate this, we present a simplified version of an interaction protocol called NetBill [26]. The protocol describes the interactions between a customer and a merchant. In NetBill, once a customer accepts a product quote, the merchant sends the encrypted goods (e.g., software) and waits for payment from the customer. Once the payment is received, the merchant sends the receipt with the decryption key (see Fig. 7).

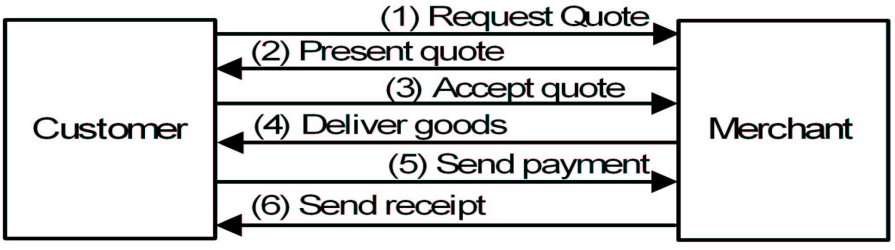

Fig. 7. The NetBill interaction protocol

Fig. 8 shows a fragment of a high-level iASR representation of the NetBill protocol (note that we omit goal/task parameters for brevity) centered on the customer side. Here, we specify a particular context for the use of the protocol by including the actor Customer that has the goal `PurchaseProduct`, which can be achieved by using the NetBill interaction protocol. Since, i^* allows for creation of modular diagrams where agents can exhibit certain behaviour by playing appropriate roles, we make use of the protocol by delegating the task `PurchaseProduct` to a role NetBill Customer, which in turn interacts with another role NetBill Merchant. While Customer and Netbill Customer are two separate roles in Fig. 8, they will most likely be played by the same agent (by concurrently executing the procedures for the two roles).

Inside the NetBill Customer role, the task `PurchaseProduct` is decomposed into two tasks, `request(KnowPrice)` and `EvaluateQuote`, and the goal `SendPayment`. These tasks and goal represent the important chunks of the customer's behaviour in the NetBill protocol: the request of a product quote, the evaluation and possible acceptance of the quote, and the payment for the product. We use guard annotations to make sure that these sub-behaviours are only executed when appropriate. Note that the conditions in the guards refer to the mental state of the agent. Thus, the request for product quote is executed only when the agent does not know the quote's value (¬**`KRef`**`(Quote)`), while the evaluation of the quote only takes place once the agent knows it. Similarly, a payment is made only after the agent knows that it has received the desired product. The parallel decomposition (note the concurrency annotation) together with the use of the guard annotations allow for the possibility of agents

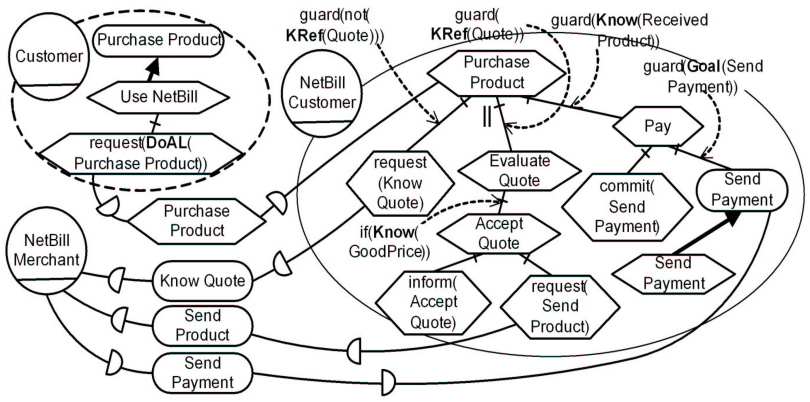

Fig. 8. iASR model for NetBill

executing their protocols flexibly. For example, in case of NetBill, if the agent already knows the price for a product, the quote request step will be skipped. Similarly, if the customer agent knows that some particular merchant always has the best price for certain products, it can request the delivery of the product right away. We now show how the customer part of the NetBill protocol modeled in Fig. 8 is represented in CASL:

```
proc PurchaseProduct(cust,merch,product)
  KRef(cust,Quote(product)) do
    request(cust,merch,Eventually(
      KRef(cust,Quote(product))))
  endGuard
   ||
  guard KRef(cust,Quote(product)) do
    if Know(cust,GoodPrice(Quote(product))) then
      AcceptQuote(cust,merch,product)
    endIf
  endGuard
   ||
  guard Know(cust,ReceivedProduct(merch,product)) do
    Pay(cust,merch,product)
  endGuard
endProc,
```

where the procedure `Pay` is defined as follows:

```
proc Pay(cust,merch,product)
  commit(cust,Eventually(SendPayment(merch,product)));
  guard Goal(cust,Eventually(SendPayment(merch,product))) do
    SendPayment(merch,product)
  endGuard
```

`endProc`

3.3 Formal Verification

Once an iASR model is mapped into the procedural component of the CASL specification and after its declarative component (e.g., precondition axioms, SSAs, etc.) has been specified, it is ready to be formally analyzed. One tool that can be used for that is CASLve [20, 18], a theorem prover-based verification environment for CASL. CASLve provides a library of theories for representing CASL specifications and lemmas that facilitate various types of verification proofs. [18] shows a proof that there is a terminating run for a simplified meeting scheduler system as well as example proofs of a safety property and consistency of specifications. In addition to physical executability of agent programs, one can also check for the *epistemic feasibility* [9] of agent plans, i.e., whether agents have enough knowledge to successfully execute their processes.

Other approaches could be used as well (e.g., simulation or model checking). However, tools based on these techniques work with much less expressive languages than CASL. Thus, CASL specifications must be simplified before these methods can be used on them. For example, most simulation tools cannot handle mental state specifications; these would then have to be operationalized before simulation is performed. The ConGolog interpreter can be used to directly execute such simplified specifications, as in [24]. Model checking methods (e.g. [5]) are restricted to finite state specifications, and work has only begun on applying these methods to theories involving mental states (e.g., [23]).

If expected properties of the system are not entailed by the CASL model, it means that the model is incorrect and needs to be fixed. The source of an error found during verification can usually be traced to a portion of the CASL code and to a part of its iASR model since our systematic mapping supports traceability.

3.4 Discussion

Our choice of CASL as the formal language is based on the fact that compared to other formalisms (e.g., [14]), it provides a richer language for specifying agent behaviour (with support for concurrency), so it makes it easier to specify complex MAS. We use a version of CASL where the precondition for the `inform` action requires that the information being sent by an agent be known to it. This prevents agents from transmitting false information. The removal of this restriction allows the modeling of systems where agents are not always truthful. This can be useful when dealing with security and privacy requirements. However, dealing with false information may require belief revision (see [21]). Similarly, the precondition for `request` makes sure that when requesting services from other agents, the sender does not itself have goals that conflict with the request. Relaxing this constraint also allows for the possibility of modeling malicious agents. Other extensions to CASL to accommodate various characteristics of application domains are possible (e.g., a simple way of modeling trust is discussed in [8]). We also note that in CASL all agents are aware of all actions being executed in the system. Often, it is useful to lift this restriction, but dealing with the resulting lack of knowledge about agents' mental states can be challenging.

4 Conclusion

In the approach presented in this paper and in [8], we produce CASL specifications from i^* models for formal analysis and verification. The approach is related to the Tropos framework in that it is agent-oriented and is rooted in the RE concepts. Our method is not the first attempt to provide formal semantics for i^* models. For example, Formal Tropos (FT) [5], supports formal verification of i^* models through model checking. Also, in the i^*-ConGolog approach [24], on which our method is based, SR models are associated with formal ConGolog programs for simulation and verification. The problem with these methods is that goals of the agents are abstracted out and made into objective properties of the system in the formal specifications. This is because the formal components of these approaches (the model checker input language for FT and ConGolog for the i^*-ConGolog approach) do not support reasoning about agent goals (and knowledge). However, most agent interactions involve knowledge exchange and goal delegation since MAS are developed as social structures, so formal analysis of goals and knowledge is important in the design of these systems. We propose a framework where goals are not removed from the agent specifications, but are modeled formally and can be updated following requests. This allows agents to reason about their objectives. Information exchanges among agents are also formalized as changes in their knowledge state. In our approach, goals are not system-wide properties, but belong to concrete agents. The same applies to knowledge. This subjective point of view provides support for new types of formal analysis. Our method is more agent-oriented and allows for precise modeling of stakeholder goals. Modeling of conflicting stakeholder goals, a common task in RE, and agent negotiations is also possible. In future work, we plan to develop a toolkit to support requirements engineering using our approach, to look into handling quality constraints (softgoals) in our approach, as well as to test the method on more realistic case studies.

References

[1] Castro J., Kolp M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. Information Systems, 27(6) (2002) 365–389

[2] Chung, L.K., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer (2000)

[3] Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-Directed Requirements Acquisitions. Science of Computer Programming, 20 (1993) 3–50

[4] De Giacomo, G., Lespérance, Y., Levesque, H.: ConGolog, A Concurrent Programming Language Based on the Situation Calculus. Artificial Intelligence, 121 (2000) 109–169

[5] Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and Analyzing Early Requirements in Tropos. RE Journal, 9(2) (2004) 132–150

[6] Gans, G., Jarke, M., Kethers, S., Lakemeyer, G., Ellrich, L., Funken, C., Meister, M.: Requirements Modeling for Organization Networks: A (Dis-)Trust-Based Approach. In Proc. RE'01 (2001) 154–163

[7] Jennings, N.R.: Agent-Oriented Software Engineering. In Proc. MAAMAW-99 (1999) 1–7

[8] Lapouchnian, A.: Modeling Mental States in Requirements Engineering — An Agent-Oriented Framework Based on i^* and CASL. M.Sc. Thesis. Department of Computer Science, York University, Toronto (2004)

[9] Lespérance, Y.: On the Epistemic Feasibility of Plans in Multiagent Systems Specifications. In Proc. ATAL-2001, Revised papers, LNAI 2333, Springer, Berlin (2002) 69–85
[10] McCarthy, J., Hayes, P.: Some Philosophical Problems From the Standpoint of Artificial Intelligence, Machine Intelligence, Vol. 4, Edinburgh University Press (1969) 463–502
[11] Moore, R.C.: A Formal Theory of Knowledge and Action. Formal Theories of the Common Sense World, J.R. Hobbs, R.C. Moore (eds.). Ablex Publishing (1985) 319–358
[12] Odell, J., Van Dyke Parunak, H. and Bauer, B.: Extending UML for Agents. In Proc. AOIS-2000, Austin, TX, USA (2000) 3–17
[13] Padgham, L., Lambrix, P.: Agent Capabilities: Extending BDI Theory. In Proc. AAAI-2000, Austin, TX, USA (2000) 68–73
[14] Rao, A.S., Georgeff, M.P.: Modeling Rational Agents within a BDI Architecture. In Proc. KR'91 (1991) 473–484
[15] Reiter, R.: The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, V. Lifschitz (ed.), Academic Press (1991) 359–380
[16] Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge MA (2001)
[17] Scherl, R.B., Levesque, H.: Knowledge, Action, and the Frame Problem. Artificial Intelligence, 144(1–2) (2003) 1–39
[18] Shapiro, S.: Specifying and Verifying Multiagent Systems Using CASL. Ph.D. Thesis. Department of Computer Science, University of Toronto (2004)
[19] Shapiro, S., Lespérance, Y.: Modeling Multiagent Systems with the Cognitive Agents Specification Language — A Feature Interaction Resolution Application. In Proc. ATAL-2000, LNAI 1986, Springer, Berlin (2001) 244–259
[20] Shapiro, S., Lespérance, Y., Levesque, H.: The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems. In Proc. AAMAS'02, Bologna, Italy, ACM Press (2002) 19–26
[21] Shapiro, S., Pagnucco, M., Lespérance, Y., Levesque, H.: Iterated Belief Change in the Situation Calculus. In Proc. KR-2000 (2000) 527–538
[22] van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt. Proc. RE'95, York, UK, (1995) 194–203
[23] van Otterloo, S., van der Hoek, W., Wooldrige, M.: Model Checking a Knowledge Exchange Scenario. Applied Artificial Intelligence, 18:9-10 (2004) 937–952
[24] Wang, X., Lespérance, Y.: Agent-Oriented Requirements Engineering Using ConGolog and i^*. In Proc. AOIS-01 (2001) 59–78
[25] Wooldridge, M.: Agent-Based Software Engineering. IEE Proceedings on Software Engineering, 144(1) (1997) 26–37
[26] Yolum, P., Singh, M.P.: Commitment Machines. In Proc. ATAL-2001, Revised Papers, LNAI 2333, Springer, Berlin (2002) 235–247
[27] E. Yu. Towards modeling and reasoning support for early requirements engineering. In Proc. RE'97, Annapolis, USA (1997) 226–235