Using the ConGolog and CASL Formal Agent Specification Languages for the Analysis, Verification, and Simulation of i* Models

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@cse.yorku.ca

Abstract. This chapter describes an agent-oriented requirements engineering approach that combines informal *i** models with formal specifications in the multiagent system specification formalisms ConGolog and its extension 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. ConGolog or CASL can be used to model complex processes formally. We introduce an intermediate notation to support the mapping between *i** models and ConGolog/CASL specifications. In the combined *i**-CASL framework, agents' goals and knowledge are represented as their subjective mental states, which allows for the formal analysis and verification of, among other things, complex agent interactions and incomplete knowledge. Our models can also serve as high-level specifications for multiagent systems.

This volume is dedicated to John Mylopoulos. Yves was fortunate to have John as his Master's thesis supervisor 30 years ago and John is Alexei's current Ph.D. thesis supervisor. The work described in this paper fits perfectly in the model-based approach to software/systems engineering that John developed and promoted throughout his career. His vision, with its roots in knowledge representation research, its embrace of ideas from social science, and its insights into the "model-based" future of software/systems engineering continues to inspire us. Thanks John, for all the inspiration and mentoring.

1 Introduction

*i** [29] is an informal diagram-based language for early-phase requirements engineering that supports the modeling of social and intentional dependencies between agents and how process design choices affect the agents' goals, both functional and non-functional. It has become clear that such social and organizational issues play an important role in many domains and applications. However, *i** is not a formal language,

A.T. Borgida et al. (Eds.): Mylopoulos Festschrift, LNCS 5600, pp. 483–503, 2009.

has inadequate precision, and thus provides limited support for describing and analyzing complex processes. While it is possible to informally analyze small systems, formal analysis is needed for realistically-sized ones.

To alleviate this, we first propose an approach that integrates *i** with a formal multiagent system specification language, *ConGolog* [5, 13], in the context of agent-oriented requirements engineering. ConGolog is an expressive formal language for process specification and agent programming. It supports the formal specification of complex multiagent systems, but lacks features for modeling the rationale behind design choices available in *i**. In this paper, we show how *i** and ConGolog can be used in combination. The *i** framework will be used to model different alternatives for the desired system, to analyze and decompose the functions of the different actors, and to model the dependency relationships between the actors and the rationale behind process design decisions. The ConGolog framework will be used to formally specify the system behaviour described informally in the *i** model. The ConGolog model will provide more detailed information about the actors, tasks, processes, and goals in the system, and the relationships between them. Complete ConGolog models are executable and this will be used to validate the specifications by simulation. To bridge the gap between *i** and ConGolog models, an *intermediate notation* involving the use of process specification annotations in *i** SR diagrams will be introduced [26, 27]. We will describe how such *annotated SR (ASR) diagrams* can be systematically mapped into ConGolog formal specifications that capture their informal meaning, and support validation through simulation and verification. The annotations are not used to capture design-level information, but to obtain a more complete and precise model of the domain.

Its support for modeling intentional notions such as goals makes the *i** notation especially suited for developing multiagent systems, e.g., as in the Tropos agent-oriented development framework [2]. *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 objectives [28]. Furthermore, most problems are too complex to be solved by just one agent — one must create a multiagent system (MAS) with several agents working 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. Agents in MAS frequently represent individuals, companies, etc. This means that there is an "underlying organizational context" [8] 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 interactions are mostly viewed not at the syntactic level, but "at the knowledge level, in terms of goal delegation, etc." [8]. Therefore, modeling and analyzing agents' mental states helps in the specification and analysis of multiagent systems.

In requirements engineering (RE), goal-oriented approaches, e.g., KAOS [4] 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. Their 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 components in the software-to-be and agents in the environment is the main outcome of the RE process [10]. 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 requirements engineering approach with a formal component that supports rigorous formal analysis, including reasoning about agents' goals (and knowledge). This would allow for rigorous formal analysis of the requirements expressed as the objectives of the agents in a MAS.

Ordinary ConGolog does not support the specification of the intentional features of *i** models, that is, the *mental states* of the agents in the system/organization modeled; these must be operationalized before they are mapped into ConGolog. But there is an extension of ConGolog called the *Cognitive Agents Specification Language* (*CASL*) [22, 23, 24] that supports formal modeling of agent mental states, incomplete agent knowledge, etc. Mapping *i** models into CASL gives the modeler the flexibility and intuitiveness of the *i** notation as well as the powerful formal analysis capabilities of CASL. So we will extend the *i**-ConGolog approach to combine *i** with CASL and accommodate formal models of agents' mental states. Our intermediate notation will be generalized to support the intentional/mental state modeling features of CASL [11, 12], in what we will call *intentional annotated SR (iASR) diagrams*. 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 [22, 24] verification 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 [2] and KAOS [4]. This allows for the modeling of conflicting goals, agent negotiation, information exchange, complex agent interaction protocols, etc.

The rest of the chapter is organized as follows. Section 2 briefly introduces *i** and a case study that we will refer to throughout the chapter, and gives an overview of the ConGolog framework. Section 3 presents our approach to map *i** diagrams into ConGolog formal specifications and discusses the use of simulation to validate the models. Section 4 discusses our second approach where *i** models are mapped into CASL, to preserve the intentional features of the models in the formal specifications; we also discuss verification. We conclude in Section 5 by summarizing our results, comparing our approach to related work, and discussing possible extensions.

2 Background

2.1 The *i** Framework and a Case Study

*i** [29] is an agent-oriented modeling framework that can be used for requirements engineering, business process reengineering, etc. *i** centers on the notion of *intentional actor* and *intentional dependency*. In the approaches described here, we use *i** as a graphical requirements modeling notation. We will assume a basic knowledge of *i** in the remainder; to learn about *i** see [30] or the chapter by Yu in this book. We will add various new notational elements to SR diagrams to produce our ASR and iASR diagrams; we will discuss these in detail in later sections. Note also that we do not use softgoals or resource dependencies in ASR and iASR (we will explain why later).

To illustrate the approach that we propose, we will use a variant of the meeting scheduling problem, which has become a popular exemplar in RE [9]. In the context of the *i** modeling framework a meeting scheduling process was first analyzed in [30]. We introduce a number of modifications to 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 assume that in the environment of the system-to-be there is a legacy software system called the Meeting Room Booking System (MRBS) that handles the booking of meeting rooms. Complete case studies are presented in [11, 12].

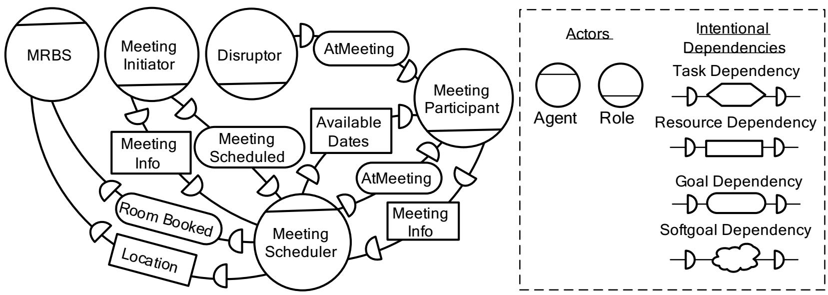

Fig. 1. The Meeting Scheduler in its environment

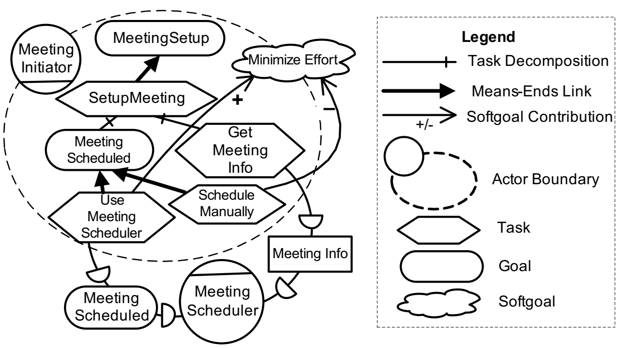

Fig. 2. SR model for the meeting initiator

Fig. 1 is a Strategic Dependency 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 Meeting Participant (MP) 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.

Fig. 2 is a simple SR models showing some details of the MI process. To schedule meetings, the MI can either do it manually, or delegate it to the scheduler. Softgoal contribution links specify how process alternatives affect quality requirements (softgoals), and so softgoals such as `MinimizeEffort` in Fig. 2 are used to evaluate these alternatives.

2.2 The Formal Foundations: The Situation Calculus and ConGolog

ConGolog [5] is a framework for process modeling and agent programming. It is based on the situation calculus [15], a language of predicate logic for representing dynamically changing worlds. The ConGolog framework can be used to model *complex processes* involving loops, concurrency, multiple agents, etc. Because it is logic-based, the framework can accommodate incompletely specified models, either in the sense that the initial state of the system is not completely specified, or that the processes involved are non-deterministic and may evolve in any number of ways.

A ConGolog specification includes two components. First, to support reasoning about the processes executing in a certain domain, that domain must be formally specified: what predicates describe the domain, what primitive actions are available to agents, what the preconditions and effects of these actions are, and what is known about the initial state of the system. The other component of a ConGolog specification is the model of the process of interest, i.e. the behaviour of the agents in the domain.

In ConGolog and in the situation calculus, a dynamic domain is modeled in terms of the following entities:

- *Primitive actions*: all changes to the world are assumed to be the result of named primitive actions that are performed by some agent; primitive actions are represented by terms, e.g. *acceptAgreementReq(participant,MS,reqID, date)*, i.e. the *participant* agent accepts the request *reqID* from the *MS* agent to attend a meeting on *date*.
- *Situations*: these correspond to possible world histories viewed as sequences of actions. The actual initial situation (where no actions have yet been executed) is represented by the constant S_0. There is a distinguished binary function symbol *do* and a term *do(a,s)* denotes the situation that results from action *a* being performed in situation *s*. For example, $do(a_3,do(a_2,do(a_1,S_0)))$ represents the situation where first a_1, then a_2, and then a_3 have been performed starting in the initial situation S_0. Thus, situations are organized in tree structures rooted in some initial situation; the situations are nodes in the tree and the edges correspond to primitive actions.
- *Fluents*: these are properties, relations, or functions of interest whose value may change from situation to situation; they are represented by predicate and function symbols that take a situation term as their last argument, e.g. *agreementReqRcvd(participant,MS,reqID, date,s)*, i.e. *participant* has received a request *reqID* from *MS* to agree to hold a meeting on *date* in situation *s*. Non-fluent predicates/functions may also be used to represent static features of the domain.

The dynamics of a domain are specified using four kinds of axioms:

- *Action precondition axioms*: these state the conditions under which an action can be performed; they use the predicate *Poss(a,s)*, meaning that action *a* is possible in situation *s*. E.g., in our meeting scheduling domain, we have:

$$\begin{aligned} &Poss(acceptAgreementReq(participant,MS,reqID,date),s) \equiv \\ &\qquad agreementReqRcvd(participant,MS,reqID,date,s) \wedge \\ &\qquad dateFree(participant,date,s) \end{aligned}$$

  This says that in situation *s*, *participant* may perform the action of accepting a request *reqID* from *MS* to hold a meeting on *date* if and only if he has received a request to that effect and the date is free for him.

- *Successor state axioms (SSA)*: these specify how the fluents are affected by the actions in the domain. E.g., in our meeting scheduling domain, we have:

$$\begin{aligned} &agreementReqRcvd(participant,MS,reqID,date,do(a,s)) \equiv \\ &\qquad a = requestAgreement(MS,participant,date) \wedge \\ &\qquad\qquad requestCounter(s) = reqID \vee \\ &\qquad agreementReqRcvd(participant,MS,reqID,date,s) \end{aligned}$$

  This says that *participant* has received a request *reqID* from *MS* to agree to hold a meeting on *date* in situation *do(a,s)* if and only if the action *a* is such a request and the value of the request counter is *reqID* or if she had already received such a request in situation *s*.

  Successor state axioms were introduced by Reiter [19] and provide a solution to the frame problem. They can be generated automatically from a specification of the effects of primitive actions if we assume that the specification is complete. Lespérance et al. [13] described a convenient high-level notation for specifying the effects (and preconditions) of actions and a tool that compiles such specifications into successor state axioms.

- *Initial situation axioms*: these specify the initial state of the modeled system. E.g., in our meeting scheduling domain, we might have the following initial situation axiom: $participantTimeSchedule(Yves, S_0) = [10,12]$, representing the fact that agent *Yves* is busy on the 10^{th} and 12^{th} in the initial situation.
- *Other axioms*: these include unique name axioms for actions, axioms specifying the agent of each type of action, and domain independent foundational axioms as described in [19].

The process of a system is specified procedurally in the ConGolog framework. We define a *main* procedure that specifies the behaviour of the whole system. Every agent has an associated ConGolog procedure to represent its behaviour in the system. The behaviour of agents is specified using a rich high-level programming language with recursive procedures, while loops, conditionals, non-determinism, concurrency, and interrupts [5]. The available constructs include:

a,	primitive action
$\varphi?$,	wait for condition
$\delta_1;\delta_2$,	sequence
$\delta_1 \vert \delta_2$,	nondeterministic branch
δ^*,	nondeterministic iteration
$\pi v.\delta$,	nondeterministic choice of argument
if φ ***then*** δ_1 ***else*** δ_2 ***endIf***,	conditional
while φ ***do*** δ ***endWhile***,	while loop
$\delta_1 \Vert \delta_2$,	concurrency with equal priority
$\delta_1 \rangle\rangle \delta_2$,	concurrency with *δ1* at higher priority
guard φ ***do*** δ ***endGuard***	guard
$<v: \varphi \rightarrow \delta$ **until** $\alpha>$	interrupt
$\beta(p)$,	procedure call

Note the presence of several non-deterministic constructs. For instance, *δ1|δ2* nondeterministically chooses between executing *δ1* or *δ2*. *πv.δ* non-deterministically picks a binding for the variable *v* and performs the program *δ* for that binding. *δ** performs *δ* zero or more times. A test/wait action *φ?* blocks until the condition φ becomes true. *<v: φ → δ* ***until*** *α>* represents an interrupt; when the trigger condition *φ* becomes true for some value of *v*, the interrupt triggers and the body, *δ*, is executed; the interrupt may trigger repeatedly as long as its cancellation condition *α* does not hold. The guard construct blocks the execution of a program *δ* until the condition *φ* becomes true.

A formal semantics based on transition systems (structural operational semantics) has been specified for ConGolog [5]. It defines a special predicate ***Do****(program,s,s')* that holds if there is a successful execution of *program* that ends in situation *s'* after starting in *s*. Communication between agents can be represented by actions performed by the sender agent, which affect certain fluents that the recipient agent has access to.

A process simulation and validation tool for ConGolog has been implemented [5]. It uses an interpreter for ConGolog implemented in Prolog. This implementation requires that the precondition axioms, successor state axioms, and axioms about the initial situation be expressed as Prolog clauses, and relies on Prolog's closed world assumption and negation as failure. Thus with this tool, simulation can only be performed for completely specified initial states.

A verification tool has also been developed [22, 24]. We discuss verification in Section 4.3. De Giacomo et al. [5] describe applications of ConGolog in different areas, such as robot programming, personal assistants, etc. Lespérance et al. [13] discuss the use of ConGolog (without combining it with *i**) for process modeling and requirements engineering.

3 Using ConGolog for the Analysis, Simulation, and Verification of *i** Models

While the informal *i** notation can be successfully used for modeling and analysing relatively small systems, formal analysis is very helpful with larger systems. Thus, formal analysis of *i** models is one of the goals of the approaches presented here. Another aim is to allow for a smooth transition from requirements specifications to high-level design for agent-based systems. While the *i** SR diagram notation allows many aspects of processes to be represented, it is somewhat imprecise and the models produced are often incomplete. For instance, it is not specified whether the subtask in a task decomposition link has to be performed once or several times. In a ConGolog model, on the other hand, the process must be completely and precisely specified (although non-deterministic processes are allowed). We need to bridge this gap. To do this, we will introduce a set of *annotations* to SR diagrams that allow the missing information to be specified. We also want to have a tight mapping between this *Annotated SR (ASR) diagram* and the associated ConGolog model, one that specifies which parts of each model are related. This allows us to identify which parts of the ConGolog model need to be changed when the SR/ASR diagram is modified and vice versa. The *i**-ConGolog approach that we describe in this section is largely based on [26, 27].

3.1 Annotated SR Diagrams

The main tool that we use for disambiguating SR diagrams is *annotations*. Annotations allow analysts to model the domain more precisely and capture data/control dependencies among goals and other details. Annotations, introduced in [26, 27] and extended in [11, 12], are textual constraints on ASR diagrams and can be of three types: composition, link, and applicability conditions. *Composition annotations* (specified by σ in Fig. 3) are applied to task and means-ends decompositions and specify how the subtasks/subgoals are to be combined to execute the supertask and achieve the goal respectively. Four types of composition are allowed: sequence (";"), which is the 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. E.g., in Fig. 3, if the "»" annotation is applied, n_1 has the highest priority, while n_k has the lowest. The choice of composition annotations is based on the ways actions and procedures can be composed in ConGolog.

Link annotations (γ_i in Fig. 3) are applied to subtasks/subgoals (n_i) and specify how/under which condition they are supposed to be achieved/executed. There are six types of link annotations (corresponding to ConGolog operators): *while* loop, *for* loop (introduced in [22]), the *if* condition, the pick, the interrupt, and the guard (introduced in [11, 12]). The difference between the *if* annotation and the guard is that the guard blocks execution until its condition becomes true while the task with the *if* link annotation is skipped if the condition is not true. The pick annotation (π*(VariableList,Condition)*) non-deterministically picks values for variables in the

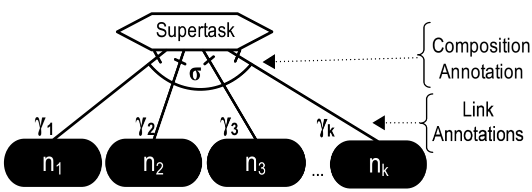

Fig. 3. Composition and link annotations

subtask that satisfy the condition. The interrupt (*whenever(varList, Condition, CancelCondition)*) fires and executes the subtask whenever there is a binding for the variables that satisfies the condition until 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.

If alternative means of achieving a certain goal exist, the designer can specify under which circumstances it makes sense to try each alternative. We call these *applicability conditions* and introduce a new annotation *ac(condition)* to be used with means-ends links to specify them. The presence of an applicability condition (AC) annotation specifies that only when the condition is true may the agent select the associated alternative in attempting to achieve the parent goal. E.g., one may specify that phoning participants to notify them of the meeting details is applicable only for important participants, while the email option is applicable for everyone (see Fig. 6). When there is no applicability condition, an alternative can always be selected.

3.2 Increasing Precision with ASR Models

The starting point for developing an ASR diagram for an actor is the regular SR diagram for that actor (e.g., see Fig. 2). It then can be appropriately transformed to become an ASR diagram every element of which can easily be mapped into ConGolog. The steps for producing ASR diagrams from SR ones include the addition of model annotations, the removal of softgoals, the deidealization of goals [9], and the addition of details of agent interaction to the model. Since an ASR diagram is going to be mapped into a ConGolog specification consisting of parameterized procedures, parameters for annotations/goals/tasks capturing the details of events as well as what data or resources are needed for goal achievement or task execution can be specified in ASR diagrams (see Fig. 6) to simplify the generation of ConGolog code. However, we sometimes omit the parameters in ASR diagrams for brevity.

Softgoals. Softgoals represent non-functional requirements [3] and are imprecise and difficult to handle in a formal specifications language such as ConGolog. Therefore in this approach, we use softgoals to choose the best process alternatives and then remove them before ASR diagrams are produced. Alternatively, softgoals can be operationalized or metricized, thus becoming hard goals. The removal of softgoals in ASR diagrams is a significant deviation from the standard *i** framework.

Deidealization of goals. Goals in ASR diagrams that cannot always be achieved are replaced by weaker goals that can. This involves identifying various possible failure conditions and guarding against them.

Providing agent interaction details. *i** usually abstracts from modeling any details of agent interactions. In ASR diagrams, we specify the interactions through which intentional dependencies are realized by the actors involved. Interactions are specified as processes involving various "communication" primitive actions that change the state of the system. The effects of these actions are modeled using ordinary fluents/properties. This supports simulation, but does not capture the fact that these actions operate on the mental states of the communicating agents. We address this in Section 4. Agent interaction details include tasks such as requests for services or information from agents in the system, tasks that supply information or communicate about success or failure in providing services, etc. Arbitrarily complex interaction protocols can be specified. We assume that the communication links are reliable.

In ASR diagrams, all resource dependencies are modeled more precisely using either goal or task dependencies according to the level of freedom that the dependee has in supplying the resource.

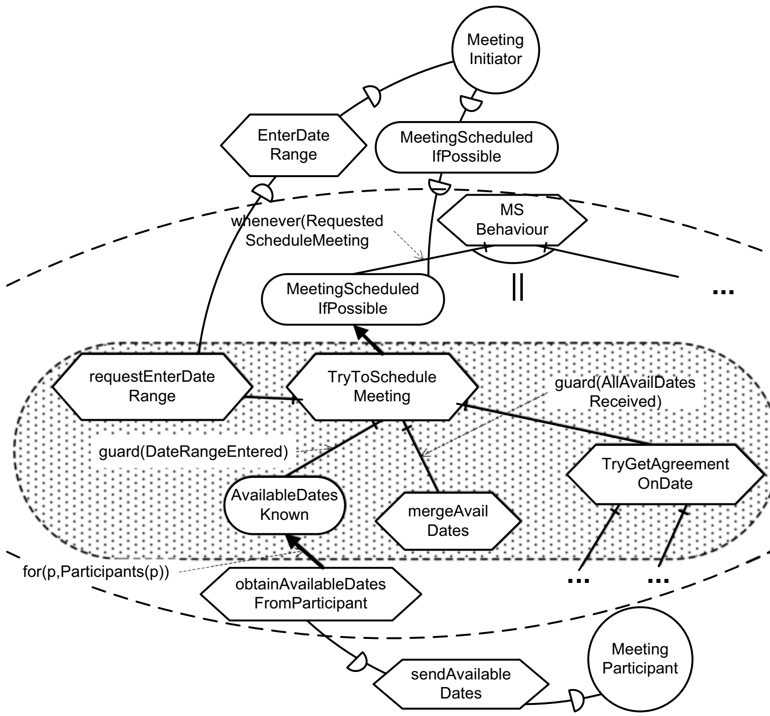

Fig. 4. A fragment of the ASR diagram for the MS agent

Fig. 4 shows a small fragment of the ASR diagram for the Meeting Scheduler agent. This model shows a very high-level view of the achievement of the goal TryToScheduleMeeting. Here, the MS must get the suggested meeting dates from the MI, get the available dates from the participants, find agreeable dates (potential dates for the meeting), and try to arrange the meeting on one of those days. Various annotations have been added to the model. The absence of a composition annotation for the TryToScheduleMeeting task indicates that it is sequentially decomposed. There are interrupt/guard annotations that let the MS agent monitor for incoming requests and for replies to its queries about the meeting date range and available dates for participants. The *for* annotation indicates that the querying for the available dates is iterated for all the participants. Note that the goal TryToScheduleMeeting in Fig. 4 is a deidealized (weakened) goal.

3.3 Mapping ASR Diagrams into ConGolog

Once all necessary details have been introduced into an ASR diagram, it can be mapped into a corresponding formal ConGolog model, thus making the model amenable to formal analysis. The modeler must define a *mapping* ***m*** that maps every element (except for intentional dependencies) of an ASR diagram into ConGolog. This mapping associates ASR diagram elements with ConGolog procedures, primitive actions, and formulas so that a ConGolog program can be generated from an ASR diagram. Specifically, agents are mapped into constants that serve as their names and ConGolog 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. So, when an agent plays several roles or occupies several positions, it executes the procedures that correspond to these roles/positions concurrently. Leaf-level task nodes are mapped into ConGolog procedures or primitive actions. Composition and link annotations are mapped into the corresponding ConGolog operators, and conditions present in the annotations map into ConGolog formulas.

Mapping Task Nodes. A non-leaf task node with its decomposition is automatically mapped into a ConGolog procedure that reflects the structure of the decomposition and all the annotations.

Consider the shaded part of Fig. 4, where the task TryToScheduleMeeting is decomposed into a number of subtasks/subgoals. This task will be mapped into the following ConGolog procedure (it contains parts still to be mapped into ConGolog; they are the parameters of the mapping ***m***). Here, the parameter *mid* stands for “meeting ID”, a unique meeting identifier:

proc *TryToScheduleMeeting(mid,mi)*
  requestEnterDateRange(MS,mi,mid);
  guard m*(DateRangeEntered)* ***do***
    m*(AvailableDatesKnown).achieve;*
  endGuard*;*
  guard m*(AllAvailDatesReceived)* ***do***
    mergeAvailDates(MS,mid);
  endGuard*;*
  TryToGetAgreementOnDate(MS,mid);
endProc

Notice that the mapping of tasks into ConGolog procedures is compositional. We have defined a set of mapping rules that formally specify this part of the mapping process.

Mapping Goal Nodes. In the *i**-ConGolog approach, goal nodes are mapped into a ConGolog formula that represents the desired state of affairs associated with the goal and a procedure that encodes means for achieving the goal. The achievement procedure is generated from the decomposition of the goal into means for achieving it, which is modeled in the ASR diagram through means-ends links. This is similar to the mapping of task decompositions as seen above and can be performed automatically. The achievement procedure for a goal *G* can be referenced as ***m****(G).achieve* (e.g., see the code fragment above). Fig. 5 shows a generic goal decomposition together with the generated achievement procedure. At the end of the achievement procedure, there is typically a test that makes sure that the goal is achieved: ***m****(G).formula)?*.

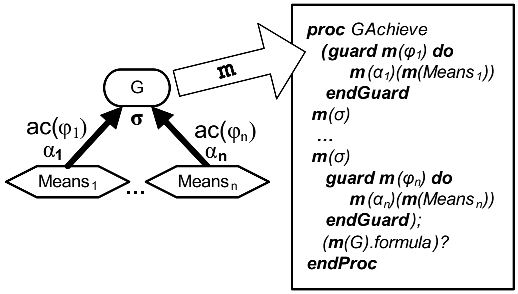

Fig. 5. Generating a goal achievement procedure

The default composition annotation for means-ends decompositions (represented by σ in Fig. 5) is alternative ("|"). This indicates that the means for achieving the goal is selected non-deterministically. As shown in Fig. 5, each goal achievement alternative is wrapped in a guard operator with the guard condition being the result of mapping the corresponding applicability condition annotation. This ensures that an alternative will only be selected when it can begin execution and its applicability condition holds. Other composition annotations (e.g. concurrency or sequence) can also be used. Note that neither ConGolog nor CASL currently provides built-in language constructs for sophisticated handling of alternative selection, execution monitoring, failure handling, retries, etc.; this is an area for future work.

Since in this approach, softgoals are removed from ASR diagrams, 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, while the email option is applicable for any number of participants (see Fig. 6). This may be due to the softgoal Minimize Effort that has been removed from the model before the ASR diagram was produced.

In addition to applicability conditions, other link annotations can be used with means-ends decompositions to specify extra control information. These are represented by α_i in Fig. 5 and are exemplified by the *for* loop annotations in Fig. 6. Note that these annotations are applied after applicability conditions.

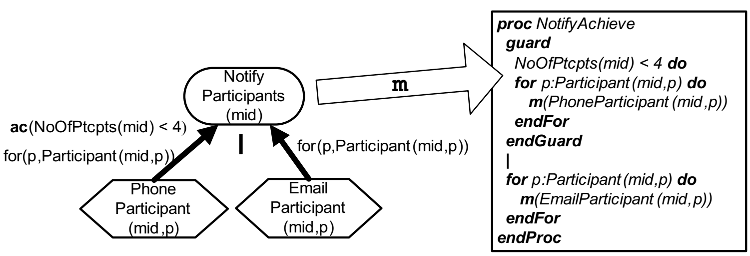

Fig. 6. Goal achievement procedure example

Specifying Domain Dynamics. To obtain a complete ConGolog specification, one needs to provide the declarative part of the specification, namely an action precondition axiom for every primitive action, a successor state axiom for every fluent, and initial state axioms, as described in Section 2.2.

3.4 Simulation

ConGolog models can be executed to run process simulation experiments. To do this, the modeler must first specify an instance of the overall system. We do this by defining a `main` procedure. Here is how this looks in the ConGolog simulation tool notation (`#=` is the concurrent execution operator):

```
proc(main,[
     initiator_behavior(mi,ms)#=
     meetingScheduler_behavior(ms,mi)#=
     participant_behaviour(yves,ms)#=
     participant_behaviour(alexei,ms)#=
]).
```

Here, there are the Meeting Initiator agent, `mi`, the Meeting Scheduler `ms`, and two participants, `yves` and `alexei`. The modeler must also provide a *complete* specification of the initial state of the system. Here, the possible meeting dates are represented as integers in order to simplify the explanation. Initially the schedule for the participant `alexei` is `[11,12,14]`, i.e., `alexei` is busy on the 11th, 12th, and 14th of some month. The schedule for the participant `yves` is `[10,12]`, i.e. `yves` is busy on the 10th and 12th. The Meeting Initiator `mi` wants to schedule a meeting with `alexei` and `yves` on the 12th or 14th. Then the modeler can execute the `main` procedure to obtain a simulation trace. The simulation obtained from this instance of the system is as follows:

```
// start interrupts in initial situation
startInterrupts
// mi requests ms to schedule a meeting with alexei and yves
requestScheduleMeeting(mi,ms,[alexei,yves])
// ms requests mi to enter the possible date range for meeting with id = 1
requestEnterDateRange(ms,mi,1)
// mi enters 12, 14 as possible meeting dates
enterDateRange(mi,ms,1,[12,14])
// ms requests available dates from all participants
obtainAvailDatesFromParticipant(ms,yves,1)
obtainAvailDatesFromParticipant(ms,alexei,1)
// yves sends his available dates
sendAvailDates(yves,ms,1,[…])
// alexei sends his available dates
sendAvailDates(alexei,ms,1,[…])
mergeAvailableDates(ms,1)
// ms finds the list of common available dates empty
setAllMergedlist(ms,1,[])
```

```
// ms notifies both participants and the initiator that it failed to schedule
// meeting 1
notifyFail(ms,mi,1,[alexei,yves])
notifyFail(ms,alexei,1,[alexei,yves])
notifyFail(ms,yves,1,[alexei,yves])
```

Generally, this validation step of the process involves finding gaps or errors in the specification by simulating the processes. The ConGolog code can be instrumented with tests (using the *"?"* operator) to verify that desired properties hold, e.g., during or at the end of the execution of the program. Alternative specifications can be also compared. A graphical user interface tool for conducting such simulation experiments is available, see [13]. As mentioned, the simulation tool requires a *complete* specification of the initial state. This limitation comes from the fact that the tool uses Prolog and its closed world assumption to reason about how the state changes. The tool (like ConGolog itself) does not provide support for modeling agent mental states and how they are affected by communication and other actions. As we saw in the examples, it is possible to model limited aspects of this using ordinary actions and fluent predicates, but this does not capture the full logic of mental states and communication. Work is underway to relax these limitations and develop techniques for efficient reasoning about limited types of incomplete knowledge and knowledge-producing actions in ConGolog [20]. ConGolog models can also be verified using the CASLve tool discussed in Section 4.3.

4 Modeling Mental States in Requirements Engineering

4.1 Motivation

Suppose that we are employing an approach like Tropos [2, 6] to model a simple goal delegation involving two agents. Fig. 7 shows a goal dependency where the Meeting Scheduler depends on the Meeting Participant for attending a meeting. We would like to

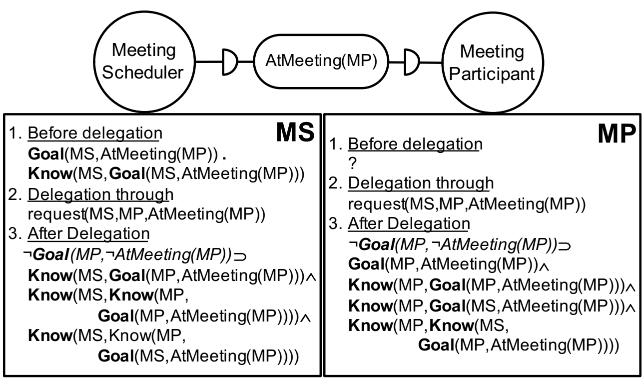

Fig. 7. A motivating example

be able to analyze this interaction and predict how it will affect the agents' goals and knowledge. Using the *i**-CASL approach presented in this section [11, 12], one can create a formal model based on the diagram, analyze it, and conclude that, e.g., before the goal delegation, the MS has the goal AtMeeting(MP) and knows about this fact. After the delegation (and provided that the MP did not have a conflicting goal), the MS knows that the MP has acquired the goal, that the MP knows that it has the goal, and that the MP knows that the MS has the same goal, etc. Similar questions can be asked about MP.

Note that the change in the mental state of the requestee agent is the core of goal delegation. One of the main features of the *i**-CASL 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., [4]). This allows for the modeling of conflicting goals, agent negotiation, information exchange, complex agent interaction protocols, etc. In CASL, the full logic of these mental states and how they change is formalized. The *i**-CASL approach thus allows for creating richer, more expressive specifications with precise modeling of agents' mental states. However, the more complex CASL models currently require the use of a theorem-prover-based verification tool such as CASLve and cannot be used with the ConGolog simulation tool.

4.2 The Cognitive Agents Specification Language

The Cognitive Agents Specification Language (CASL) [22, 23] is a formal specification language that extends ConGolog to incorporate *models of mental states* expressed in the situation calculus [21]. CASL uses modal operators to formally represent agents' knowledge and goals; communication actions are provided to update these mental states and ConGolog is then employed to specify the behaviour of agents. The logical foundations of CASL allow it to be used to specify and analyze a wide variety of MAS as shown in [22, 23]. For instance, it can model non-deterministic behaviours and systems with an incompletely specified initial state. Similar to ConGolog (see Section 2.2), 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 formal representation for both goals and knowledge in CASL is based on a *possible worlds semantics* incorporated into the situation calculus, where situations are viewed as possible worlds [16, 21]. 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. In this case, the situation s' is called *K-accessible*. When an agent does not know the truth 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 that φ in situation s if φ is true in all its K-accessible situations in s: $\boldsymbol{Know}(agt,\varphi,s) = \forall s'(K(agt,s',s) \supset \varphi[s'])$. 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. Built-in communication actions such as *inform* are used for exchanging information among agents. The precondition for the *inform* action ensures that no false information is transmitted. The changes to agents' knowledge due to communication and other actions are specified by the successor state axiom for the *K* relation. The specification ensures that agents are aware of the execution of all actions. Enhanced accounts of knowledge change and communication in the situation calculus have also been proposed to handle, for instance, encrypted messages [23] or belief revision [25].

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. We use the formula ***Goal****(agt,ψ,s)* to indicate 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 ensures that while agents may want something they know is impossible to achieve, the goals of agents must be consistent with what they currently know. There are constraints on the *W* and *K* relations that ensure that agent's goals are consistent and that agents introspect their goals. 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 built-in communication actions *request* and *cancelRequest* are used by agents to request services from other agents and to cancel such requests respectively. Requests are used to establish intentional dependencies among actors and lead to changes in goals of the requested agent. The dynamics of the *W* relation are specified, as usual, by a successor state axiom that guarantees that no inconsistent goals are adopted.

4.3 The *i**-CASL Notation and Process

Increasing Precision with Intentional Annotated Strategic Rationale Models. Our aim in this approach is to tightly associate *i** models with formal specifications in CASL. As was the case with the *i**-ConGolog approach presented in Section 3, we use an intermediate notation, *Intentional Annotated SR (iASR) diagrams,* to bridge the gap between SR diagrams and CASL specifications.

When developing an iASR diagram, one starts with the corresponding SR diagram (e.g., see Fig. 2). The steps for producing iASR diagrams from the corresponding SR ones are similar to the ones presented in Section 3.

Agent Goals in iASR Models. A CASL agent has procedural (behaviour) and declarative (mental state) components. iASR diagrams only model agent processes and thus are used to represent the procedural component of CASL agents. 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 if and how to achieve it. For the agent to modify its behaviour in response to the changes to its mental state, it must synchronize its procedural and declarative components (see Fig. 8A). Agent mental states usually change as a result of communication acts that realize goal delegation and information exchange. So, the procedural component of the agent must monitor for these changes. The way to do this is to use interrupts or guards with their conditions being the presence of certain goals or knowledge in the mental state of the agent (Fig. 8B). Procedurally, the goal node is interpreted as invoking the means to achieve it.

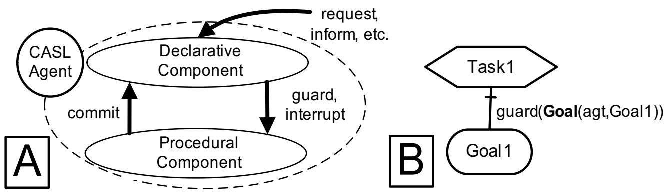

Fig. 8. Synchronizing declarative and procedural components of CASL specifications

In CASL, only communication actions have effects on the mental state of the agents. However, we also would like the agents to be able to change their mental state on their own by executing the action *commit(agent,φ)*, where *φ* is a formula that the agent (or the 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 existing or potential alternatives exist in these processes and makes sure the mental state of agents reflect this. This allows agents to reason about their goals and ways to attain them at runtime.

Modeling agent interactions. We take an intentional stance towards modeling agent interactions. We are modeling them with built-in generic communication actions (e.g., *request*, *inform*) that modify the mental states of the agents. In iASR models, these generic communication actions are used to request services, provide information, etc. Also, the conditions in annotations and communication actions (as well as the *commit* action) may refer to the agents' mental states, knowledge and goals. Because of the importance of agent interactions in MAS, in order to formally verify multiagent system specifications in CASL, all high-level aspects of agent interaction must be provided in the corresponding iASR models.

Fig. 9A and Fig. 9B illustrate how an intentional goal dependency RoomBooked (see Fig. 1) can be modeled in SR and iASR models respectively. It is established by the MS's execution of the *request* action (with that goal as the argument) towards the MRBS agent. This will cause the MRBS to acquire the goal RoomBooked (if it is consistent with its existing goals). The interrupt in the iASR model for the MRBS monitors

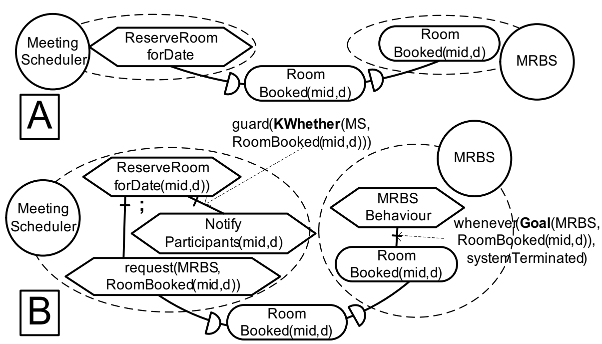

Fig. 9. Adding iASR-level agent interaction details

its mental state for the goal and triggers the behaviour for achieving it (i.e. booking a room, which is not shown) when the goal is acquired. Also, once the MS's knowledge state is updated and it knows whether (formally, ***KWhether***) the room has been booked (note the guard condition), the task for notifying participants will be triggered.

From iASR Models to CASL Specifications. Once an iASR model has been produced, it can be mapped into a CASL specification for formal analysis.

As previously, the modeler defines a mapping ***m*** that associates iASR model elements (except for dependencies) with CASL procedures, primitive actions, and formulas, so that a CASL program can be generated from an iASR model. Specifically, actors are mapped into CASL procedures, leaf-level tasks are mapped into procedures or primitive actions, while annotations are mapped into CASL operators. Conditions in the annotations map into CASL formulas that can refer to agents' mental states.

Mapping Goal Nodes. An iASR goal node is mapped into a CASL formula (the formal definition for the goal) and an achievement procedure that is based on the means-ends decomposition for the goal in the iASR diagram (see Fig. 5). E.g., 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 was booked and all participants accepted to meet. Often, an initial goal definition is too ideal and needs to be *deidealized* [9] or weakened. See [12] for an example.

CASL's support for reasoning about agents' goals gave us the ability not to maintain meeting participants' schedules explicitly. Rather, we relied on the presence of goals *AtMeeting(participant,mid,date,s)* in their mental states together with an axiom that made sure that they could only attend one meeting per time slot (see [12]).

The achievement procedures for goals are automatically constructed based on the modeled means for achieving them as described in Section 3.

Modeling Dependencies. Intentional dependencies are not mapped into CASL per se — they are established by the associated 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. For task dependencies, we use *request(FromAgt, ToAgt, **DoAL**(SomeProcedure))* to request that a known procedure be executed while allowing other actions to occur (***DoAL*** stands for "do at least").

In order for a dependency to be established, we also need a commitment from a dependee agent to act on the request from the depender. It must monitor its mental state for the newly acquired goals, which is done using interrupts that trigger whenever unachieved goals of certain types are in their mental states. The bodies of interrupts specify appropriate responses to the messages. Also, 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). For details, see [11, 12].

Analysis and Verification. Once an iASR model is mapped into the corresponding CASL specification, it is ready to be formally analyzed. One tool that can be used is CASLve [24, 22], 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. In addition to physical executability of agent programs, one can also check for the epistemic feasibility of agent plans [14], i.e., whether agents have enough knowledge to successfully execute their processes. Alternative verification approaches based, for instance, on simulation or model checking can be used. However, they require much less expressive languages, so CASL specifications need to be simplified for these approaches.

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.

5 Discussion and Future Work

In this chapter, we have presented an approach to requirements engineering that involves the combined use of *i** and some multiagent system specification formalisms, ConGolog and its extension CASL. This allows the requirements engineer to exploit the complementary features of the frameworks. The *i** framework can be used to model social dependencies between agents, perform an analysis of opportunities and vulnerabilities, explore alternatives and trade-offs. These models are then gradually made more precise with the use of *annotated models*. ConGolog or CASL can then be used to model complex processes formally with subsequent verification or simulation (for ConGolog only). Additionally, CASL supports the explicit modeling of agent mental states and reasoning about them. In our approach, both graphical/informal and textual/formal notations are used, which supports a progressive specification process and helps in communicating with the clients, while providing traceability.

There have been a few other proposals for using *i** with formal specification languages for RE. The Trust-Confidence-Distrust (TCD) approach combining *i** and ConGolog to model/analyze trust in social networks was proposed in [7]. TCD is focused on a specific type of applications and has an extended SR notation that is quite different from our proposal in terms sequencing of elements, explicit preconditions, etc.

Formal Tropos (FT) [6] is another approach that supports formal analysis of *i** models though model checking. Its specifications use temporal logic and it can be used at the SD level, unlike our approaches, which use procedural notations that are more suitable for SR models. Unlike CASL, the formal components of FT and the *i**-ConGolog approach do not support reasoning about goals and knowledge and thus require that goals be abstracted out of the specifications. However, most agent interactions involve knowledge exchange and goal delegation. The ability of CASL to formally model and reason about mental states as properties of agents is important and supports new types of analysis (e.g., of conflicting goals).

In future work, we would like to develop tool support for representing ASR/iASR diagrams and mapping them into ConGolog/CASL and for supporting the co-evolution of the two representations. We expect that our RE toolkit will be able to significantly simplify the specification of the declarative component of ConGolog/CASL models. We plan to explore how different types of agent goals (e.g., maintenance) as well as privacy, security, and trust can be handled in CASL. There are also a number of limitations of CASL's formalization of mental state change and communication that should be addressed in future work. One such limitation is that agents cannot send false information. Removing this limitation requires modeling belief revision, which adds a lot of complexity (see [25]). However, this will support modeling of, e.g., malicious and untruthful agents.

We also note that CASL assumes that all agents are aware of all actions being executed in the system. Often, it would be useful to lift this restriction, but dealing with the resulting lack of knowledge about agents' mental states can be challenging.

Finally, there is also ongoing work on supporting limited forms of incomplete knowledge and information acquisition actions in a logic programming-based ConGolog implementation [20]. This may eventually lead to an executable version of CASL where simulation can be performed on models of agents with mental states.

Bibliography

1. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-agent programming: Languages, platforms and applications. Springer, Heidelberg (2005)
2. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: TROPOS: An agent-oriented software development methodology. Journal of Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
3. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-functional requirements in software engineering. Kluwer, Dordrecht (2000)
4. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
5. De Giacomo, G., Lespérance, Y., Levesque, H.: ConGolog, a concurrent programming language based on the Situation Calculus. Artificial Intelligence 121(1-2), 109–169 (2000)
6. Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in Tropos. Requirements Engineering Journal 9(2), 132–150 (2004)
7. Gans, G., Jarke, M., Kethers, S., Lakemeyer, G.: Continuous requirements management for organisation networks: a (Dis)trust-based approach. Requirements Engineering Journal 8(1), 4–22 (2003)
8. Jennings, N.R.: Agent-oriented software engineering. In: Garijo, F.J., Boman, M. (eds.) MAAMAW 1999. LNCS, vol. 1647, pp. 1–7. Springer, Heidelberg (1999)
9. van Lamsweerde, A., Darimont, R., Massonet, P.: Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In: Proc. RE 1995, York, UK (1995)
10. van Lamsweerde, A.: Requirements engineering in the year 00: a research perspective. In: Proc. ICSE 2000, Limerick, Ireland (2000)
11. 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)
12. Lapouchnian, A., Lespérance, Y.: Modeling mental states in agent-oriented requirements engineering. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 480–494. Springer, Heidelberg (2006)
13. Lespérance, Y., Kelley, T.G., Mylopoulos, J., Yu, E.S.K.: Modeling dynamic domains with conGolog. In: Jarke, M., Oberweis, A. (eds.) CAiSE 1999. LNCS, vol. 1626, pp. 365–380. Springer, Heidelberg (1999)
14. Lespérance, Y.: On the epistemic feasibility of plans in multiagent systems specifications. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS, vol. 2333, pp. 69–85. Springer, Heidelberg (2002)

15. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. Machine Intelligence 4, 463–502 (1969)
16. Moore, R.C.: A formal theory of knowledge and action. In: Hobbs, J.R., Moore, R.C. (eds.) Formal Theories of the Common Sense World, pp. 319–358. Ablex Publishing, Greenwich (1985)
17. van Otterloo, S., van der Hoek, W., Wooldrige, M.: Model checking a knowledge exchange scenario. Applied Artificial Intelligence 18(9-10), 937–952 (2004)
18. Reiter, R.: The frame problem in the Situation Calculus: a simple solution (sometimes) and a completeness result for goal regression. In: Lifschitz, V. (ed.) Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy, pp. 359–380. Academic Press, London (1991)
19. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems. MIT Press, Cambridge (2001)
20. Sardina, S., Vassos, S.: The Wumpus world in IndiGolog: A Preliminary Report. In: Proc. Sixth Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC 2005) at IJCAI 2005, Edinburgh, UK (2005)
21. Scherl, R.B., Levesque, H.: Knowledge, action, and the frame problem. Artificial Intelligence 144(1-2), 1–39 (2003)
22. Shapiro, S.: Specifying and Verifying Multiagent Systems Using CASL. Ph.D. Thesis. Dept. of Computer Science, University of Toronto (2004)
23. Shapiro, S.: Modeling Multiagent Systems with CASL - A Feature Interaction Resolution Application. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS, vol. 1986, pp. 244–259. Springer, Heidelberg (2001)
24. Shapiro, S., Lespérance, Y., Levesque, H.: The Cognitive Agents Specification Language and verification environment for multiagent systems. In: Proc. AAMAS 2002, Bologna, Italy, pp. 19–26 (2002)
25. Shapiro, S., Pagnucco, M., Lespérance, Y., Levesque, H.: Iterated belief change in the Situation Calculus. In: Proc. KR-2000, Breckenridge, Colorado, USA (2000)
26. Wang, X.: Agent-oriented requirements engineering using the ConGolog and i* frameworks. M.Sc. Thesis. Department of Computer Science, York University, Toronto (2001)
27. Wang, X., Lespérance, Y.: Agent-oriented requirements engineering using ConGolog and *i**. In: Proc. AOIS 2001 (2001)
28. Wooldridge, M.: Agent-based software engineering. IEE Proceedings on Software Engineering 144(1), 26–37 (1997)
29. Yu, E.: Modeling strategic relationships for process reengineering. Ph.D. Thesis. Department of Computer Science, University of Toronto (1995)
30. Yu, E.: Towards modeling and reasoning support for early requirements engineering. In: Proc. RE 1997, Annapolis, MD, USA (1997)