SNet: A Modeling and Simulation Environment for Agent Networks Based on i* and ConGolog

Günter Gans[1], Gerhard Lakemeyer[1], Matthias Jarke[1,2], and Thomas Vits[1]

[1] RWTH Aachen, Informatik V
Ahornstr.55, 52056 Aachen, Germany
[2] Fraunhofer FIT
Schloss Birlinghoven, 53754 Sankt Augustin, Germany
{gans,lakemeyer,jarke}@cs.rwth-aachen.de

Abstract. SNet is a prototype environment supporting the representation and dynamic evaluation of designs for social networks comprising human, hardware, and software agents. The environment employs metadata management technology to integrate an extended version of the i* formalism for static network modeling with the ConGolog logic-based activity simulator. The paper defines the formal mappings necessary to achieve the integration and describes an operational prototype demonstration. SNet's intended application domain is requirements management and mediation support for inter-organizational and embedded process systems, as well as simulation support for inter-organizational studies e.g. in hightech entrepreneurship networks.

1 Introduction

The modeling of business processes has been an important aspect of information systems engineering for many years. In this research, a progress from pure drawing facilities towards a more formal semantics can be observed. This formal understanding enables consistency and completeness analysis of models as well as their semi-automatic transformation.

A further step in this progression is the modeling and simulation of dynamic business aspects. For many well-known business process formalisms, such as the event-process chains of the ARIS modeling formalism [Sch94], timed Petri nets ([OSS94], meanwhile commercialized by PROMATIS AG for the Oracle Designer environment), or simply automata-based mechanisms [PJ96], simulation environments have been developed from which the impact of different business strategies on operational efficiency and, in some cases, organizational memory and similar long-term factors can be assessed.

The modeling of dynamic inter-organizational relationships, especially for complex social networks involving many human, organizational, and possibly technological agents, is in a much less mature stage. While extensions of traditional business models do cover some important aspects of modern business concepts such as supply chain management, they often ignore the autonomy of the members within network settings, thus underestimating the independent

A. Banks Pidduck et al. (Eds.): CAISE 2002, LNCS 2348, pp. 328–343, 2002.

evaluation of different agent goals, modulated by the strategic interdependences among them, and the resulting complex dynamics of negotiation and trust-based (or distrust based) activity. In this paper, we report on SNet, a modeling and simulation environment for agent networks that attempts to remedy some of these shortcomings.

The social networks we have in mind are in particular those created among independent organizations or individuals to pursue some shared strategic goals, but always at the risk of falling apart. To formalize such networks the agent-oriented requirements modeling language i* [Yu95] seems particularly suited since it explicitly allows one to capture the mutual dependencies among actors, which are key ingredients of such networks. However, representing the structural relationships among actors alone is not sufficient. As was argued in [GJK+01a], it is equally important to model the network dynamics because the interactions among the actors are to a large extent trust-based, and to understand the impact of trust one needs to consider interactions and their effects over time. To do so, the authors propose a multi-perspective modeling approach which includes an extension of i*, speech acts, and the action language ConGolog. While it was suggested already then to translate i* into executable ConGolog programs (see also [GJK+01b]), the mapping was preliminary and only partially developed. In this paper we consider a complete and automatic translation for a large fragment of extended i* models. Moreover, we report on a fully implemented prototype implementation. Having a system which takes as input graphical network representations based on (extended) i*, which can then be turned into executable programs, is valuable because it provides a tool for network participants to simulate various network scenarios whose outcome may give valuable information regarding the risks and benefits of taking certain actions.

The rest of the paper is organized as follows. In the next section we introduce i* and the extensions necessary to facilitate the translation into executable programs. In Section 3, we introduce ConGolog and present the translation from extended i* networks into ConGolog programs. In Section 4 we briefly go over the implemented system. Finally, we end with some conclusions and open issues.

2 Representing Social Networks in Extended i*

We begin this section by introducing the i* modeling language [Yu95], originally devised for early requirements engineering. i* is then extended to facilitate the automatic translation into executable programs. We also show that, by representing i* diagrams in the conceptual modeling language Telos [MBJK90], it becomes possible to perform a static analysis of a network or enforce integrity constraints with the help of the Telos query language.

2.1 The i* Modeling Language

i* is firmly based on the notions of *actor* and *goal* and assumes that social settings involve actors who depend on each other for goals to be achieved, tasks to be

performed, and resources to be furnished. The framework includes the *strategic dependency (SD) model* for describing the network of relationships among actors, as well as the *strategic rationale (SR) model* for describing and supporting the reasoning that each actor performs concerning her relationships with other actors.

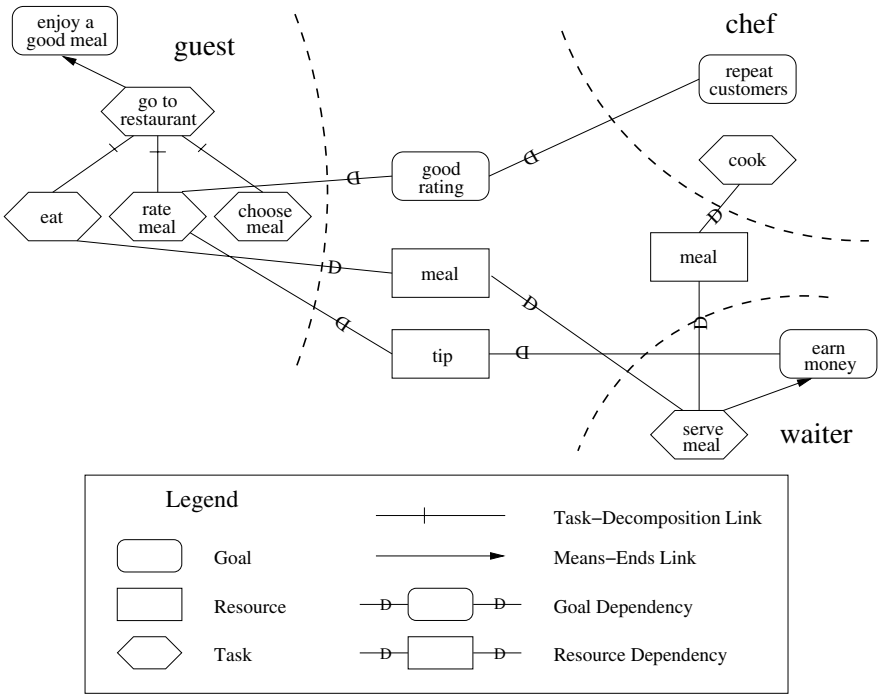

Fig. 1. Strategic rationale model of the "restaurant" example

We will not go over SD models here, but instead focus on SR models and illustrate some of their key features by way of a restaurant example shown in Figure 1 involving three actors: a *guest*, a *waiter*, and a *chef*.[1] The guest's main component is a *go_to_restaurant* task which serves to bring about the goal of enjoying a good meal and is decomposed into three subtasks. The guest depends on the waiter for the meal to be served. Conversely, to obtain a tip the waiter depends on the guest's favorable rating of the meal, which the chef also depends on in his goal to have repeat customers.

[1] While the example, which we use throughout the paper, is rather simple and the actors involved may not even share long-term strategic goals, it nevertheless serves to illustrate the main ingredients of our approach and has the advantage of being familiar to most readers.

2.2 Enriching Task Description

Our goal is to turn specifications of SR diagrams into agent programs ready for simulation. However, in their current form, i* models are not expressive enough for this purpose. For example, consider the decomposition of the *go_to_restaurant* task. Clearly, the subtasks need to be ordered, but the ordering is not determined by the model. In fact, in i* the semantics of this decomposition is left open. Here we interpret it as an *and-decomposition*, that is, all subtasks need to be performed, and we will give it a precise semantics in Section 3.2. Also, when tasks are decomposed, it is implicitly assumed that *all* subtasks are (eventually) performed. However, as we will see below in an example, there are cases where an or-decomposition is needed, that is, where only one of the subtasks should be performed. Finally, it is not clear at all when and how any of the tasks involved are actually activated.

To see how all this can be achieved, let us consider Figure 2, which enriches the task descriptions of Figure 1 in an appropriate way. From a syntactical point of view, the main new graphical feature are triangles. These are labeled with logical formulas whose predicates are so-called *fluents* whose truth value may vary during the execution of tasks. When a triangle points to a task like *trust_high_enough* pointing to *choose_meal*, then it serves as a *precondition*. When a task points to a triangle, then it denotes a *postcondition* or effect of that task, which in turn can be preconditions of other tasks.

Precondition triangles in bold face have a special meaning in that they serve as triggers or *interrupts* for the execution of the task they are pointing to. For example, *order_received* triggers the execution of *cook*. We require that only top-level tasks can have interrupts and that there can be at most one per task. Tasks without interrupts that trigger their execution are considered to be *exogenous*, like the *start* task in the example. Each graphical object has to be assigned to one actor. So, an exogenous task belongs to a special actor called *Exogenous*. Intuitively, these kinds of tasks are under external control and may be used, for example, to start a simulation run. Note that there may be more than one exogenous task present.

The decomposition of *rate_meal* is an example of an *or-decomposition*. In particular, we want to make sure that only one of *rate_good* or *rate_bad* is performed, not both. We will see in Section 3 that the semantics of or-decompositions is actually quite subtle.

Note that there are preconditions without any arrows directed to them. Where does, for example, the fluent *trust_high_enough* obtain its value from? The complete specification of this precondition refers to another fluent *confidence_guest* which, say, takes as value a positive real number and is initialized appropriately. It is then updated by the effects of *good_rating* and *bad_rating*, respectively. For example, when *good_rating* is activated, it raises the value of "guest-confidence" by some small amount.[2]

[2] We have not included these details in the diagram to avoid clutter. In the implementation discussed in Section 4, these details are usually also hidden, but can be made visible on demand (see Figure 3).

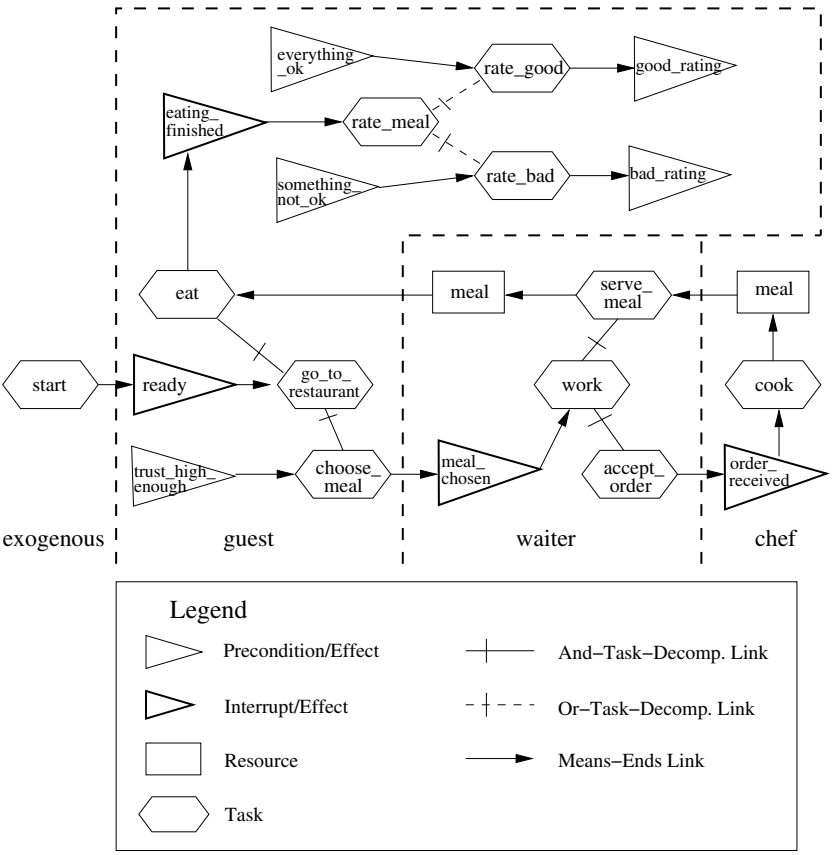

Fig. 2. Extended strategic rationale model of the "restaurant" example

Finally, note that the synchronization of tasks via pre- and postconditions gives rise to dependencies among actors in an implicit fashion. For example, the chef can only cook a meal after an order was accepted by the waiter, thus expressing a natural dependency between the waiter and the chef.

To summarize, the new features introduced into SR models are preconditions, postconditions, interrupts as a special kind of preconditions, and or-decompositions.

2.3 Static Analysis in ConceptBase

We use the *ConceptBase* metadata manager [JEG+95] based on the conceptual modeling language Telos [MBJK90] as the representation language of extended i* diagrams. This has the added benefit of providing us with a powerful query language which allows us to perform a static analysis of our networks. Syntactical checks are one application. For example it is useful to know whether there

are network nodes with neither in- nor outgoing links, because isolated elements make no sense, which can be formulated as follows in the *ConceptBase Query Language*:

```
QueryClass elements_without_link isA SNetElement with
retrieved_attribute
   name: String
constraint
   rule: $not exists l/SNetLink (l from this) or (l to this)$
end
```

Another example describes that it is not allowed for subtasks to be a decomposition of more than one supertask. If there are tasks with more than one in-going task-decomposition-link (tdl) they will be found by the following query:

```
QueryClass subTask_with_n_tdls isA SNetTaskElement with
constraint
  rule: $exists tdl1,tdl2/SNetTaskDecompLink
         (tdl1 to this) and (tdl2 to this) and (not (tdl1==tdl2))$
end
```

As we will see in Section 3.2, queries to ConceptBase also play an important role to supply the relevant information for the translation from extended i* diagrams into ConGolog programs.

3 Simulation of Social Networks

3.1 ConGolog – A Short Introduction

In our methodology, plans are expressed in the logic-based language ConGolog. This section describes both ConGolog and its formal foundation, the situation calculus.

The *situation calculus* is an increasingly popular language for representing and reasoning about the preconditions and effects of actions [McC63]. It is a variant of first-order logic[3], enriched with special function and predicate symbols to describe and reason about dynamic domains. We will not go over the language in detail except to note the following features: all terms in the language are one of three sorts, ordinary objects, actions or situations; there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation are called *relational fluents*, and are denoted by predicate symbols taking a situation term as their last argument; similarly, functions varying across situations are called *functional fluents* and are denoted analogously; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

[3] Strictly speaking, a small dose of second-order logic is required as well, an issue which should not concern us here.

Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [LPR98]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$. For example, the fact that a robot can only pick up an object if it is next to the object and it is not holding anything can be formalized as follows: $Poss(pickup(r, x), s) \equiv NextTo(r, x, s) \wedge \forall y.\neg Holding(r, y, s)$. We use the convention that free variables are implicitly universally quantified.
- Successor state axioms, one for each fluent F, stating under what conditions $F(x, do(a, s))$ holds as a function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [Reiter 1991]. As an example, consider a simple model of time which progresses in a discrete fashion by 1 unit as a result of a special action *clocktick*. The time of a situation can then be specified with the help of a fluent $time(s)$ and the following successor state axiom:

$$time(do(a, s)) = t \equiv a = clocktick \wedge t = time(s) + 1 \\ \vee a \neq clocktick \wedge t = time(s)$$

- Domain closure and unique-name axioms for actions.

ConGolog [dGLL00], an extension of Golog [LRL+97], is a language for specifying complex actions (high-level plans). It comes equipped with an interpreter which maps these plans into sequences of atomic actions assuming a description of the initial state of the world, action precondition axioms and successor state axioms for each fluent. Complex actions are defined using control structures familiar from conventional programming languages such as sequence, while-loops, and recursive procedures, but also non-deterministic actions like choosing non-deterministically between two actions or performing an action an arbitrary number of times. In addition, parallel actions with or without priorities are possible as well.

α	primitive action
$\phi?$	test action
$[\sigma_1, \sigma_2]$	sequence
if ϕ then σ_1 else σ_2	conditional
while ϕ do σ	loop
$or(\sigma_1, \sigma_2)$	nondeterministic choice of actions
$pi(x, \sigma)$	nondeterministic choice of arguments
$star(\sigma)$	nondeterministic iteration
$conc(\sigma_1, \sigma_2)$	concurrent execution
$pconc(\sigma_1, \sigma_2)$	prioritized concurrent execution
$tryAll(\sigma_1, \sigma_2)$	concurrent execution until one terminates
$interrupt(\phi, \sigma)$	interrupts
$proc(\beta(\vec{x}), \sigma)$	procedure definition

When translating extended i* into ConGolog, the most important constructs are *conc*, *pconc*, *tryAll*, and *interrupt*. While the intuitive meaning of *conc* is the obvious, *pconc* says that σ_1 should be preferred over σ_2 whenever possible. *tryAll*[4] means that both programs σ_1 and σ_2 start executing concurrently, but the whole *tryAll*-construct terminates as soon as one of the two terminates. This is in contrast to *conc* and *pconc*, where both parts need to terminate. As we will see later, *tryAll* is needed to give semantics to the or-decomposition of subtasks in i*. Finally, $interrupt(\phi, \sigma)$ says that σ should be executed whenever the condition ϕ becomes true. In other words, interrupts serve as triggers to initiate actions.

We will not go over the formal semantics of ConGolog here except to note that it uses a conventional transition semantics defining single steps of computation and where concurrency is interpreted as an interleaving of primitive actions and test actions. For details see [dGLL00, GL00].

3.2 Transformation of Extended i* into Executable Programs

In this section we show how to automatically translate a large fragment of extended SR models, which includes all the new features added in Section 2.2, into executable ConGolog specifications. We will get back to the parts of SR models that are not dealt with at the end of this section.

The translation needs to specify two parts, the description of the application domain and the complex tasks operating on this domain. For the application domain we need to describe the fluents and primitive actions together with their preconditions and effects. Complex tasks correspond to ConGolog procedure definitions. Each fluent, primitive action, and procedure is also assigned a unique actor to which it belongs.

In what follows we use the Prolog syntax of the IndiGolog interpreter used for our implementation. IndiGolog [dGL99] is a variant of ConGolog developed for *on-line* execution, where the choice of the next primitive action alternates with its execution.[5]

Generating the Application Domain Description

To describe the application domain we use clauses (in the sense of Prolog) of the form `prim_fluent(F)`, `prim_action(a)`, and `exog_action(e)` for primitive fluents, primitive actions, and exogenous actions, respectively. The clauses `initially(F,true)` and `poss(a,`ϕ`)` are needed to initialize a fluent's value in situation S_0 resp. to define an action precondition axiom. A convenient feature of IndiGolog is that it suffices to declare the effects of actions, which are then

[4] We remark that the *tryAll*-construct was not present in the original ConGolog, but was later added in [GL00].

[5] IndiGolog can also be used in an off-line modus, where a whole action sequence is computed before execution. The on-line modus, however, seems more appropriate for simulation purposes.

automatically converted into successor state axioms introduced in Section 3.1. Effect axioms have the form `causes_val(a,F,newVal,cond)` with the reading that after performing an action `a`, the fluent `F` will obtain the new value `newVal`, if the condition `cond` holds.

Besides fluents which are explicitly specified in the extended SR model like *meal_chosen* or *confidence_guest*, interrupts and resources are also represented as relational fluents. The former are used to trigger the corresponding ConGolog interrupts inside procedures (see below). The latter are needed to describe the owner of a resource, who, unlike other components of the model, may change over time. In the restaurant example the *meal*'s owner initially obtains the default value "notexist." After the chef performs *cook* he owns the *meal*-resource. The waiter can only perform *serve_meal* if this resource is owned by the chef in that situation. Thus, in a sense, resources are formally treated like special task preconditions. There is one system-defined fluent *time* which keeps track of time and whose meaning is given by the successor state axiom in Section 3.1. Finally, initial values of fluents are generated from corresponding annotations in the SR model (see below for an example).

The primitive actions are generated by extracting the names of all tasks that are neither decomposed further nor an exogenous task. For example, *choose_meal* is considered a primitive action, while *go_to_restaurant* is not. There is one system-defined primitive action *clocktick* which advances time and which is executed with lowest priority.

A `poss(a,`ϕ`)`-clause for a primitive action `a` is generated by collecting the corresponding preconditions in the SR model and then computing the appropriate ϕ. For example, the task *rate_good* will not be performed unless the precondition *everything_ok* holds. Also, *everything_ok* held and *rate_good* has been performed, the *rate_bad* or-branch will be pruned, because or-decomposition applies the *tryAll*-construct (see above).

Exogenous actions are special primitive actions that are controlled by the user. In our model, an action is exogenous if it belongs to a special actor called "Exogenous." During a simulation, the user is able to invoke exogenous actions interactively. Because they are regarded as primitive, the corresponding task in the SR model may not be decomposed further. In our example, the task *start* is the only exogenous action and does the obvious. In general, exogenous tasks are not restricted to initiating simulation runs. For example, we could replace the *rate_meal* task by an exogenous *rate_meal_exog* task, which would give the user the chance to influence the meal-rating process from the outside.

How a fluent is affected by actions is determined in one of four ways: 1) it can be read off explicitly from the postconditions of tasks corresponding to primitive actions, as in *meal_chosen*, which is satisfied by the action *choose_meal*; 2) interrupt-fluents like *meal_chosen* are automatically reset at the end of the task they are pointing to; 3) resource-fluents are affected by changes in ownership; 4) the special fluent *time* is only affected by the action *clocktick*. In any case, appropriate `causes_val`-clauses are generated.

Here is an excerpt of the definitions generated for the restaurant example.

```
initially(confidence_guest, 0.5).
prim_action(choose_meal).
prim_fluent(meal_chosen).
prim_fluent(confidence_guest).
poss(choose_meal,confidence_guest > 0.45).
causes_val(rate_good,confidence_guest,X,X is confidence_guest+0.1).
causes_val(choose_meal , meal_chosen, true, true).
causes_val(work, meal_chosen, false, true).
```

Note that these clauses are generated completely automatically from the SR model. Since the SR model is stored in ConceptBase, we can easily collect the relevant information for clause generation by posing appropriate queries. For example, in order to define fluents for all interrupts in the model, we would use the following query:

```
QueryClass fluents isA SNetInterruptElement with
   retrieved_attribute
      name : String
end
```

Given the result of the query, it is then a simple matter to emit the right `prim_fluent`-definitions.

Generating Procedural Descriptions

For the generation of procedures from complex tasks, we only need a subset of the IndiGolog instructions, namely sequential and (prioritized) concurrent execution of actions (`[A,B]` resp. `conc(A,B)`) (`pconc(A,B)`), as well as control of actions by (prioritized) interrupts (`interrupt(condition,body)` resp. `prioritized_interrupts(list_of_interr.)`). The latter is an abbreviation of the prioritized concurrent execution of interrupts, where precedence in the list means higher priority.

For each actor in our model we need a procedure that describes its behavior. In particular, all interrupts and their following actions form the body of the procedures. Additionally, we need a start-procedure which both concurrently starts the procedures of all involved agents and provides other services for our simulation. (Again, the relevant information can be extracted automatically using ConceptBase queries.):

```
proc(agent_guest,
     conc(interrupt(eating_finished=true, decomp_rate_meal),
          interrupt(ready=true, decomp_go_to_restaurant))).
proc(agent_chef,
     interrupt(order_received=true, cook)).
proc(agent_waiter,
     interrupt(meal_chosen=true, decomp_work)).
proc(start_sim,
     prioritized_interrupts([interrupt(sim_running=true,
```

```
pconc(conc(conc(agent_guest,agent_chef),agent_waiter),
interrupt(true,noOp))),
interrupt(sim_running=false, noOp)])).
```

While primitive actions like *cook* are used directly, decomposed tasks like *go_to_restaurant* lead to auxiliary procedures which initiate the execution of the corresponding subtasks in a concurrent fashion. In other words, our interpretation of and-decomposition is that all subtasks are started concurrently and they all need to terminate successfully for the supertask to terminate. In the case of the and-decomposed task *work*, we obtain

```
proc(decomp_work,[conc(accept_order, serve_meal), work]).
```

Note that each procedure `decomp_[taskname]` ends with a primitive action `[taskname]`, which is used for things like switching off the interrupt or performing other direct effects of the task.

As for or-decompositions, recall that the intuition is that the termination of one of the subtasks should lead to the termination of the whole task. At first glance, one might be tempted to use the $or(\sigma_1, \sigma_2)$-construct provided by ConGolog, which nondeterministically chooses one of the σ_i for execution. However, this is problematic for on-line execution. After all, the interpreter needs to commit right away to one of the choices, but it may not have enough information to make the right choice. This is why we need the $tryAll(\sigma_1, \sigma_2)$-construct first proposed in [GL00], which starts executing both σ_1 and σ_2 concurrently and stops as soon as one of them reaches a final state. In the restaurant example, *rate_meal* is or-decomposed and the translation results in:

```
proc(decomp_rate_meal,[tryAll([rate_bad , rate_meal],
                              [rate_good, rate_meal])]).
```

This ends the description of the translation from extended SR diagrams into IndiGolog. Note that, since the mapping is completely automatic and since IndiGolog has a precise declarative semantics, we have also given extended SR models a precise meaning. The only caveat is that we have not yet considered all features of i*. *Goals* and *subgoals*, perhaps the most notable omissions, will be dealt with in the near future.[6] *Soft-goals* and *positive* resp. *negative contribution links* are more problematic because they are introduced as vague concepts, and it is not clear at all how to formally represent them.

4 SNet: A Software Environment for Modeling, Analysis, and Simulation of Social Networks

Extended i* diagrams are composed using a modification of the editor OME3 (Object Modeling Environment) [LY]. While OME3 was developed for the original i*, we extended it to cover the new features like preconditions and or-decompositions of tasks.

[6] If a task satisfies a goal, one idea would be to insert a test action corresponding to the goal as the final action of the procedure representing the task.

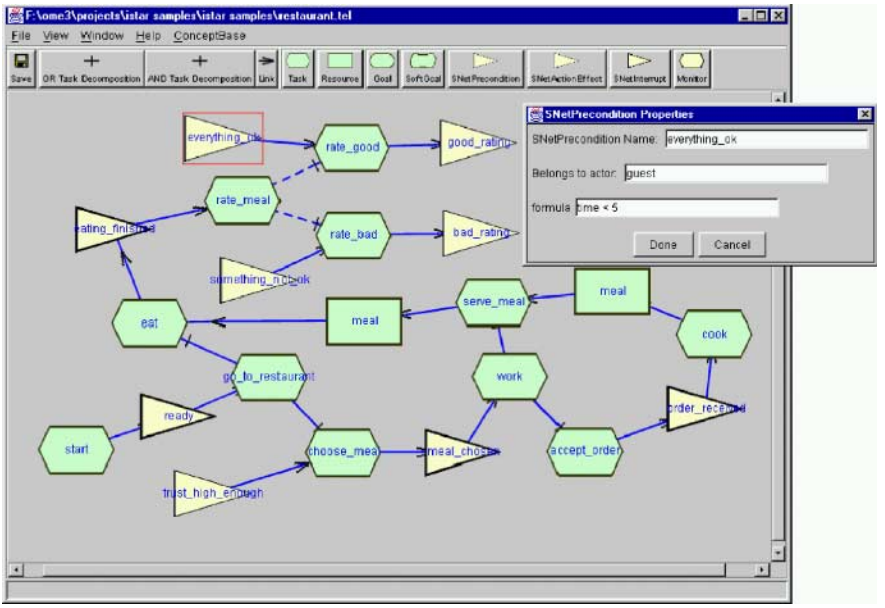

Fig. 3. OME(Object Modeling Environment)

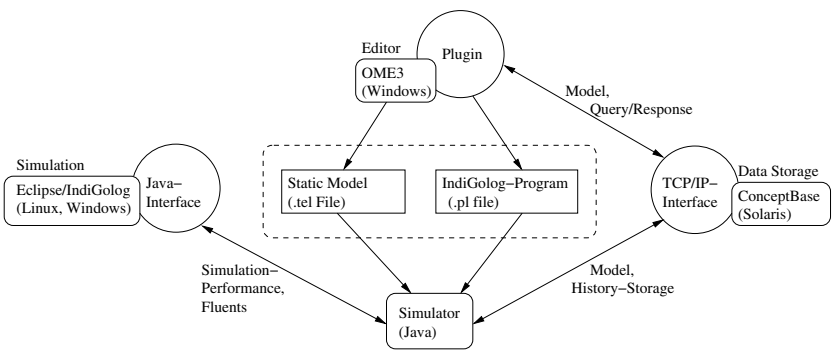

Fig. 4. SNet Software Architecture

In addition, we have attached Java plugins to OME3 to maintain a connection to a ConceptBase server (see below), initiate the static analysis of extended i* diagrams, and to perform the translation process to ConGolog programs. An OME-snapshot is shown in Figure 3.

In the context of static analysis in 2.3 we mentioned that we use *ConceptBase*, a deductive object manager for meta databases (for details see [JEG+95]). The role of ConceptBase will become clearer if we have a look at our implementation's architecture (Figure 4). By establishing a TCP/IP connection we are able to transfer the SNet framework as well as the current application model

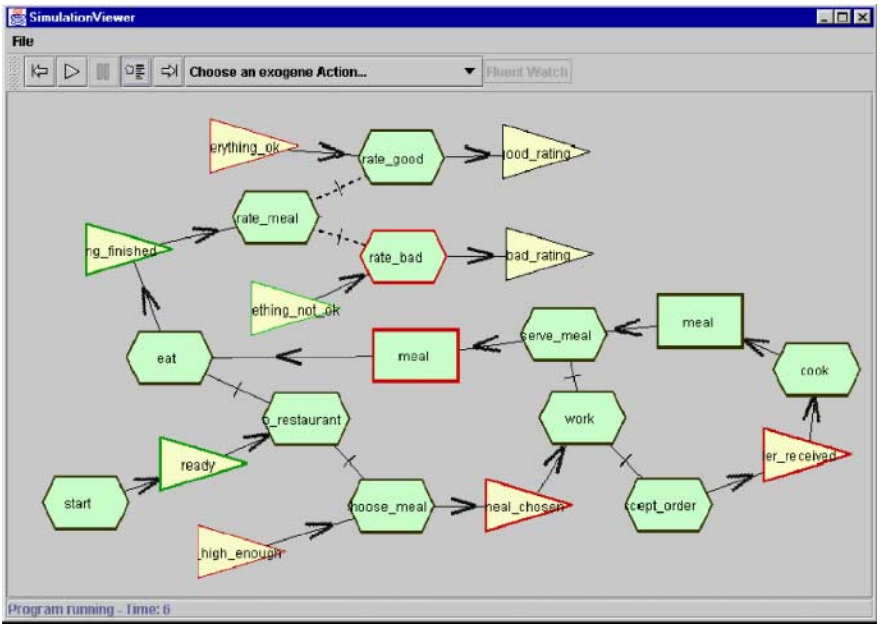

Fig. 5. SNet Simulator[7]

to ConceptBase. This way we are able to use the ConceptBase Query Language to perform certain queries for static analysis purposes as described in 2.3. Also, suitable queries are issued to ConceptBase to obtain the information necessary for the translation process to generate ConGolog programs.

A third component (Fig. 4), the simulator, written in Java, retrieves the SNet framework, the application, and the position of each graphical object from ConceptBase. While the interpreter written in Prolog processes the IndiGolog program, the simulator controls the execution, monitors the changing fluents, initiates exogenous actions, and shows a step by step view of the simulation run.

Figure 5 shows a snapshot of our simulator running the restaurant example. The user creates specific simulation runs by exogenous actions. Preconditions that hold are depicted with a green border, otherwise they have red borders. In the snapshot *something_not_ok* holds, which leads to the task *rate_bad* being performed. As a consequence the precondition *trust_high_enough* gets a red border. The red-bordered resource *meal* illustrates that the guest is it's owner.

5 Related Work

Lespérance et al. [LKMY99] are the first to demonstrate that ConGolog can be applied to model and simulate business processes. Similar in spirit to our work,

[7] For those labels in the screenshot which are not legible the reader is referred to Figure 2 instead, which shows the same network, only hand-drawn.

Wang and Lespérance [WL01] also propose to integrate i* and ConGolog, but in a way quite different from ours. Roughly, while we introduce a small number of new node and link types like task preconditions and or-decomposition into the graphical representation of SR models, they annotate the original SR diagrams with ConGolog constructs like while-loops, sequential task decompositions, and the like. While this allows very fine-grained control-flow specifications at the i*-level, it comes at the expense of burdening the user with choosing among the various control alternatives. Also, while we strive for automatic translations from extended i* into ConGolog, they still need considerable user interaction. Fuxman et al. [FPMT01] also start with i* and enrich it with constraints formalized in a linear time logic inspired by the KAOS language [DvLF93].

They then use model checking techniques [CCGRar] to verify the consistency of the specification. Models are presented by listing which fluents are true at certain time points and hence can be thought of as a form of simulation. However, the main motivation for model checking is finding counterexamples, that is, bugs in the specification. Our system, on the other hand, is mainly intended for the simulation of different scenarios for consistent specifications. Hence, the two approaches seem to complement each other well, an issue we want to explore further in the future.

6 Conclusions

In this paper we proposed a framework for modeling and simulating the complex interactions found in social networks. A graphical language based on Yu's i* is introduced for the specification of the network and the internal structure of the agents involved. The models are stored in the ConceptBase system, whose query language can be used for the static analysis of the models. These are then automatically translated into executable ConGolog programs and the prototype implementation provides a graphical interface to visualize simulation runs.

This work was originally motivated by our desire to understand and model the role of trust and distrust in social networks [GJK+01a]. Now that we have an implemented modeling and simulation environment, we are in a position to test existing theories about how trust and distrust evolve over time. Doing so will require collecting data from extensive simulation runs and storing the history of interactions in a suitable form. Storing the history in ConceptBase will have the advantage of making it accessible not only to the user for analysis but also to the agents in the network who can use it for further decision making.

Another important issue is to test our modeling and simulation environment on real data. One such application scenario is our ongoing case study in trans-Atlantic entrepreneurship networks. Using such a realistic application will likely lead to refinements and extensions of our methodology.

Acknowledgment

This work was supported in part by the Deutsche Forschungsgemeinschaft in its Focussed Research Programme on Socionics.

References

[CCGRar] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Int. Journal on Software Tools for Technology Transfer (STTT)*, To appear. 341

[dGL99] G. de Giacomo and H. J. Levesque. An incremental interpreter for high-level programs with sensing. *Logical Foundations for Cognitive Agents*, pages 86–102, 1999. 335

[dGLL00] G. de Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000. 334, 335

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993. 341

[FPMT01] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering (RE01), Toronto, Canada*, August 27-31 2001. 341

[GJK+01a] G. Gans, M. Jarke, S. Kethers, G. Lakemeyer, L. Ellrich, C. Funken, and M. Meister. Requirements modeling for organization networks: A (dis-)trust-based approach. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE01)*, pages 154–163, Toronto, Canada, August 2001. Los Alamitos: IEEE Computer Society Press 2001, ISBN 0-7695-1125-2. 329, 341

[GJK+01b] G. Gans, M. Jarke, S. Kethers, G. Lakemeyer, L. Ellrich, C. Funken, and M. Meister. Towards (dis)trust-based simulations of agent networks. In *Proceedings of the 4th Workshop on Deception, Fraud, and Trust in Agent Societies*, pages 49–60, Montreal, May 2001. 329

[GL00] H. Grosskreutz and G. Lakemeyer. Towards more realistic logic-based robot controllers. In *Proc. of AAAI-00*, 2000. 335, 338

[JEG+95] Matthias Jarke, Stefan Eherer, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases*, 4(2):167–192, 1995. 332, 339

[LKMY99] Y. Lespérance, T. G. Kelley, J. Mylopoulos, and E. Yu. Modeling dynamic domains with congolog. In *Proceedings of CAiSE-99*, June 1999. 340

[LPR98] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. 334

[LRL+97] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1):59–84, 1997. 334

[LY] L. Liu and E. Yu. OME (Object Modeling Environment), http://www.cs.toronto.edu/km/ome/. 338

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos - representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990. 329, 332

[McC63] John McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted 1968 in Minsky, M.(ed.): Semantic Information Processing, MIT Press. 333

[OSS94] A. Oberweis, G. Scherrer, and W. Stucky. INCOME/STAR: Methodology and tools for the development of distributed information systems. *Information Systems*, 19(8):643–660, 1994. 328

[PJ96] P. Peters and M. Jarke. Simulating the impact of information flows on networked organizations. In *Proceedings of the 17th International Conference on Information Systems, Cleveland, Ohio, USA*, pages 421–439, Dezember 1996. 328

[Sch94] A.-W. Scheer. *Business Process Engineering - Reference Models for Industrial Companies*. Springer Verlag, Berlin, 2 edition, 1994. 328

[WL01] Xiyun Wang and Yves Lespérance. Agent-oriented requirements engineering using ConGolog and i*. In *Working Notes of the Agent-Oriented Information Systems (AOIS-2001) Workshop, Montreal, QC*, May 2001. 341

[Yu95] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, 1995. 329