University of Wollongong
Research Online

Faculty of Informatics - Papers (Archive)

Faculty of Engineering and Information Sciences

2004

Combining agent-oriented conceptual modelling with formal methods

S. A. Vilkomir
University of Wollongong

Aditya K. Ghose
University of Wollongong, aditya@uow.edu.au

A. Krishna
University of Wollongong, aneesh@uow.edu.au

Publication Details

This paper originally appeared as: Vilkomir, SA, Ghose, A & Krishna, A, Combining agent-oriented conceptual modelling with formal methods, Proceedings. Australian Software Engineering Conference, 2004, 147-155. Copyright IEEE 2004

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

Combining agent-oriented conceptual modelling with formal methods

Abstract
Agent-oriented conceptual modelling notations such as i* have received considerable recent attention as a useful approach to early-phase requirements engineering. Agent-oriented conceptual modelling notations are highly effective in representing requirements from an intentional stance and answering questions such as what goals exist, how key actors depend on each other and what alternatives must be considered. Formal methods such as those based on the Z notation offer a complementary set of representational facilities. We explore how these two otherwise disparate approaches might be used in a synergistic fashion.

Keywords
formal specification, software agents

Disciplines
Physical Sciences and Mathematics

Publication Details
This paper originally appeared as: Vilkomir, SA, Ghose, A & Krishna, A, Combining agent-oriented conceptual modelling with formal methods, Proceedings. Australian Software Engineering Conference, 2004, 147-155. Copyright IEEE 2004

This conference paper is available at Research Online: http://ro.uow.edu.au/infopapers/89

Combining Agent-Oriented Conceptual Modelling with Formal Methods

Sergiy A. Vilkomir, Aditya K. Ghose and Aneesh Krishna
Decision Systems Lab, School of IT and Computer Science
University of Wollongong, NSW 2522, Australia
{*sergiy, aditya, ak86*}*@uow.edu.au*

Abstract

Agent-oriented conceptual modelling notations such as i have received considerable recent attention as a useful approach to early-phase requirements engineering. Agent-oriented conceptual modelling notations are highly effective in representing requirements from an intentional stance and answering questions such as what goals exist, how key actors depend on each other and what alternatives must be considered. Formal methods such as those based on the Z notation offer a complementary set of representational facilities. This paper explores how these two otherwise disparate approaches might be used in a synergistic fashion.*

1 Introduction

Some common questions that need to be addressed in early-phase requirements engineering are the following: what are the main goals of the system, how the stakeholders depend from each other, and what alternatives exist [16]. The *i** framework [12–15] is an agent- and goal-oriented modelling language that has been specifically designed for early-phase requirements engineering and that is well-suited for answering questions such as these. The central concept in the i* framework is that of an intentional *actor (agent)*. Intentional properties of an agent such as goals, beliefs, abilities and commitments are used in modelling requirements [17]. The i* framework is particularly useful for:

- making explicit (and in the process gaining) a deeper understanding of the organisational relationships between various actors in the target environment;
- understanding of the rationale behind the existing practices and structures; and
- representing, at an intentional level, the internals of actors populating the target system, and relating these explicitly to organizational objectives and inter-actor relationships.

The i* notation consists of two main modelling components: the Strategic Dependency Model (SD) and the Strategic Rationale Model (SR).

The SD and SR models are graphical representations of the dependencies between actors and internal intentional characteristics of actors respectively. A SD model is a graph consisting of *nodes* and *links* among the nodes. Each node represents an actor, and each link between the actors represents how one actor (*depender*) depends on another (*dependee*) for something in order to accomplish a *goal* or *task*. The object around which the dependency relationship centres is called the *dependum*. An SD model represents *goals, task, resource, and soft goal* dependencies between actors. The first three of these dependency types are relatively straightforward to understand (an actor depends on another to fulfill a goal, execute a task and supply a resource, respectively). Softgoals are effectively non-functional requirements, i.e., statements of objectives that the target system should eventually meet. The SR model provides a more detailed level of modelling by looking "inside" actors to model internal intentional relationships.

As an example, consider a simplified version of the well-known meeting scheduler scenario [11, 16, 17]. This example will be used to illustrate both the i* notation and our proposed methodology for transforming i* models into Z specifications. The SD modelling process (see Figure 1) begins with identifying the actors involved with the meeting scheduling system and their mutual dependency relationships.

The *MeetingInitiator* agent depends on *Participant* agents to achieve its *AttendsMeeting* goal. The *MeetingInitiator's* dependency on the *MeetingScheduler* to schedule a meeting can be modeled as a goal dependency *MeetingBeScheduled*. The resource dependency *Agreement* and task dependency *EnterAvailDates* are examples of other kinds of dependencies between actors.

In the SR model (see Figure 2) intentional elements like *goals*, *tasks*, *resources*, and *softgoals* appear not only as ex-

Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)
1530-0803/04 $ 20.00 © 2004 IEEE

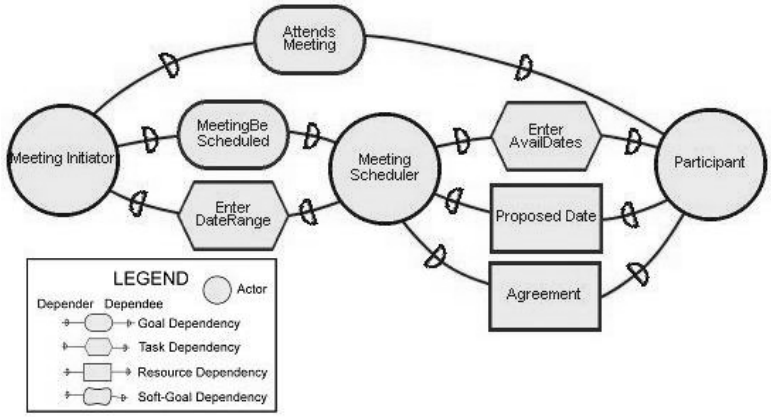

Figure 1. The Strategic Dependency Model

ternal dependencies, but also as internal elements which are connected by task-decomposition links and means-ends relationships.

For example, the *Participant* has an internal task to *ParticipateInMeeting*. This task can be performed by subtasks *AttendMeeting* and *ArrangeMeeting* (these are related to the parent task via task decomposition links). For the *MeetingInitiator*, the goal of *MeetingBeScheduled* is an internal goal. In the case of *Participant*, the internal tasks *FindAgreeableDateUsingScheduler* and *FindAgreeableDateByTalkingToInitiator* are alternative means to achieve the goal *Agreeable (Meeting, Date)*. How the alternatives contribute to softgoals is also represented. These are represented as means-ends link relationships. The SR model thus provides a way of modelling stakeholder interests, how they might be met, and the stakeholders' evaluation of various alternatives with respect to their interests.

Several proposals exist for integrating i* modelling with late-phase requirements analysis and the downstream stages of the software life-cycle. The TROPOS methodology [4] explores how i* models might be refined to form the basis for late-phase requirements specifications, and subsequently architecture specifications. The i* notation alone is not adequate for representing the level of detail necessary for late-phase requirements specifications. To address this, formal languages such as Formal Tropos [7] have been developed. An alternative approach has been to define methodologies for transforming i* models into agent programs in formal agent programming languages such as ConGOLOG [11].

Our thesis in this paper is that the Z formal notation and the i* modelling framework can function in a complementary and synergistic fashion and that a conceptual modelling methodology that supports their co-evolution is of interest.

Z [10] is a formal notation for computer systems and software specification based on set theory and first order predicate logic. This mature formal method is widely used both in theoretical investigations [1] and in practice [3]. The main elements of the Z notation are *schemas* which are used to specify states and operations for the modelling of systems. While Z can be used for early-phase requirements modelling, the necessary level of formalization, precision and detail, the lack of a diagrammatic notation to support the visualization of requirements and the inability to represent the *intentional* elements all suggest that an alternative notation such as *i** might be better suited for this phase.

Our proposal for a synergistic combination of *i** and Z offers several advantages:

- *i** and Z can be viewed as a pair of complementary representation languages that can be jointly brought to bear on the requirements engineering exercise. The *i** notation permits us to make explicit the intentional aspects of the requirements specification, including an understanding of the organizational context of the proposed system, the alternatives that may be considered in making design decisions as well as the rationale behind these decisions (these latter features support process reengineering). The Z notation permits us to specify late-phase requirements with a degree of precision and formality that *i** does not.

- The *i** notation allows us to represent and reason with softgoals (representations of non-functional requirements or objectives).

- We propose a mapping from *i** models into Z schemas

IEEE COMPUTER SOCIETY

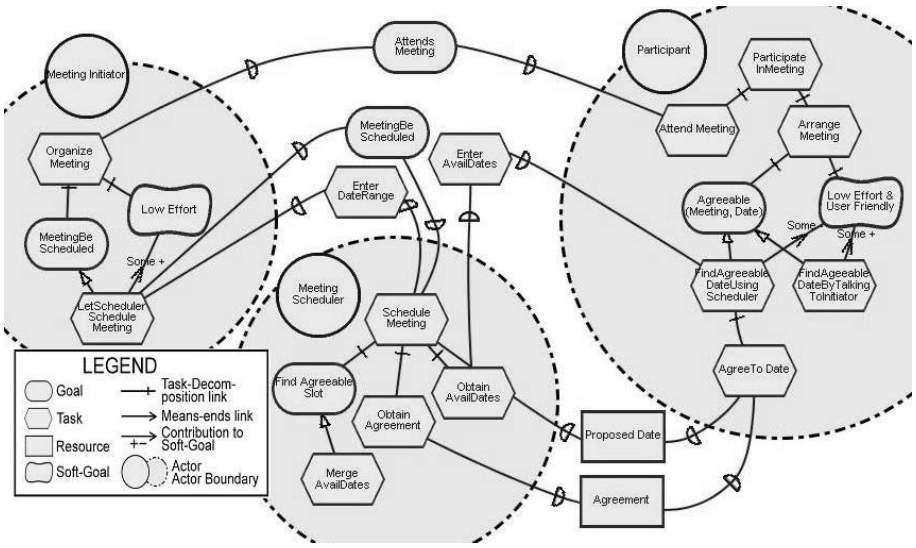

Figure 2. The Strategic Rationale Model

that does not result in any information loss, nor the introduction of information extraneous to the original $i*$ model (this is distinct from proposals such as the one involving mapping $i*$ models to ConGolog agent programs [11], where aspects of the $i*$ model are ignored in the translation).

- The mapping of $i*$ models to Z schemas enables the refinement of these schemas with additional information, such as invariant properties, fulfilment conditions etc. (note that these cannot be represented in the original $i*$ model).

- Current approaches to the use of formal methods in conjunction with $i*$ models are unduly complies. Formal Tropos [7], for instance, is an intermediate language in which $i*$ models must be defined before an eventual translation into a state machine model on which model checkers can be deployed to verify systems properties (the process also assumes a significant amount of refinement of the original model with additional information). Existing tool support for Z, on the other hand, allows analysis of specifications without any of this additional effort.

In Section 2, below, we presented the mapping between $i*$ models and Z schemas. In Section 3, we present an example of such a mapping. The example is specially interesting on account of the pointers we provide on how an initial set of Z schemas obtained from an $i*$ model might be refined in useful ways. This paper may be viewed as a first step in defining a complete methodology for supporting the co-evolution of $i*$ models and Z specifications.

2 Mapping i^* into Z

2.1 Mapping a general SD model into Z

All elements (actors and dependencies) of a SD model differ in names. For describing their names in the Z notation we introduce the basic type (given set) *NAME*. Given an SD model, one can refer to distinct subsets of *NAME*. The subset *all_actors* contains the names of all actors while the subset *all_depend* contains the names of all dependencies in the SD model.

$$[NAME]$$

$$\mid \; all_actors, all_depend : \mathbb{P}_1\, NAME$$

It is necessary to mention that names of internal intentional elements of a SR model are also members of the given set *NAME* but do not belong to subset *all_depend*. Formalization of these internal elements is considered later in the paper.

Both SD and SR models provide a description of the intentional relationships among actors of a process and do not directly address the dynamics of this process. But exactly the dynamics are the most important for process or system specification. To reflect it, we use the fact that all dependencies in SD and internal elements in SR are realized dynamically: a goal is achieved, task is performed or resource becomes available. We consider different states of the dependencies (elements) before and after realization using the following free type definition:

$$STATE ::= inapplicable \mid unresolved \mid fulfilled \mid violated \mid satisficed \mid denied \mid undetermined$$

State *inapplicable* is held before the creation of a new instance of a dependency (element). State *unresolved* conforms to a dependency (element) after the creation but before realization and all other states are conforming to a dependency (element) after realization. The dependency (element) is in state *fulfilled* if realization is successful and in state *violated* if realization is unsuccessful. With the idea of keeping uniform terminology with other researchers (e.g., [5]) in the area, for softgoals we use two states *satisficed* and *denied*. The last state *undetermined* can also be used only for a softgoal. Softgoals are often identified with quality criteria and sometimes it is impossible to conclude immediately after realization whether a quality criterion is satisfied. It means that it may not be clear whether the realization had been successful or not. In this case we consider the softgoal is in the *undetermined* state.

The state of a whole SD model is a collection of states of all dependencies for this SD model that is reflected in *SD* schema[1]:

$$\begin{array}{|l} \hline \textit{SD} \\ SD_state : NAME \rightarrowtail STATE \\ \hline \mathrm{dom}\, SD_state = all_depend \\ \hline \end{array}$$

Thus, the realization of a dependency changes its state and at the same time changes a state of the whole SD model.

Each SD dependency or SR element has its own specific features and differs first in types and degrees.

$$TYPE ::= goal \mid softgoal \mid task \mid resource \mid ISA$$
$$DEGREE ::= open \mid committed \mid critical$$

In contrast to other values, the *ISA* type does not represent a dependency. It means that one actor can be considered as a special instance of other actor. Since, *ISA* is a relationship between two actors it is convenient for us to consider them together as a different values of *TYPE*. All other values of free type definitions *TYPE* and *DEGREE* are standard for the i^* framework.

All the dependencies in SD (as well as every element in SR model - see the next section of the paper)are described by its own schema. A general structure of SD dependencies (external between actors) varies from a general structure of SR elements (internal inside actors) but at the same time they have some common patterns. That is why we use the following steps of formalization, creating consecutively:

- Φ*Depend* schema which describes a common pattern of SD dependencies and SR elements;
- *SDependency* schema which describes a general structure of all the SD dependencies and includes Φ*Depend* schema as one of the component part;
- A detailed schema for every SD dependency using *SDependency* schema as a basis.

Common patterns for SD dependencies and SR elements are represented in Φ*Depend* schema. Here, Φ is a part of the schema name, not an operator. It is just a naming convention used to indicate a partial (incomplete) specification [2].

$$\begin{array}{|l} \hline \Phi Depend \\ dependum : NAME \\ type : TYPE \\ degree : DEGREE \\ result! : STATE \\ \hline result! \neq unresolved \\ result! = satisficed \lor result! = denied \lor \\ \quad result! = undetermined \Rightarrow type = softgoal \\ \hline \end{array}$$

Except for the above-mentioned type and degree, specific features of every dependency are its name (*dependum*) and resulting state, which is represented by the output variable *result*!. The first line of the predicate part of Φ*Depend* describes the fact that the resulting state cannot be *unresolved*. The second line of the predicate part of Φ*Depend* reflects that the resulting state can take the *satisficed*, *denied* or *undetermined* value only for softgoals.

The following *SDependency* schema is a result of one-to-one mapping of the general structure of a SD dependency into the Z notation. This schema is an *operation* schema and changes the state of the SD model (ΔSD). *SDependency* schema includes the components Φ*Depend* schema as well as names of actors (*depender* and *dependee*) which are linked by the dependency. While, this schema represents a *general* structure, its name, type, degree and names of actors are not specified. It could be done later on during the consideration of an i^* model for a specific example.

$$\begin{array}{|l} \hline SDependency \\ \Delta SD \\ \Phi Depend \\ depender, dependee : NAME \\ \hline dependum \in all_depend \\ depender \in all_actors \\ dependee \in all_actors \\ SD_state' = SD_state \oplus \{dependum \mapsto result!\} \\ \hline \end{array}$$

The most significant information is contained in the last line of the predicate part of this schema, which describes how the realization of the dependency changes the state of the SD model. Using the *override* operator $\oplus$ shows that

[1] All schemas in this paper were checked using the ZTC type-checker package [8].

the value of the SD model's state function SD_state' after the dependency realization differs from its value SD_state before the realization only in the part of the considered dependency and coincides for all other dependencies.

2.2 Mapping a general SR model into Z

Our approach of mapping a SR model into the Z notation is similar to the approach for SD diagrams which were considered in the last section. The mapping consists in consecutively creating:

- *Actor* schema which describes a general structure of all the actors in SR diagrams;
- *AElement* schema which describes a general structure of all the SR internal intentional elements and includes $\Phi Depend$ schema as one of the component part;
- A detailed schema for every actor in the specific SR model using *Actor* schema as a basis;
- A detailed schema for every internal element of every actor using *AElement* schema as a basis.

The following schema describes a general structure of all the actors. An actor is characterized by his name *actor_name*, set *actor_element* of names of all internal elements, and state function *actor_state*.

$$
\begin{array}{|l}
\hline
\textit{Actor} \\
actor_name : NAME \\
actor_element : \mathbb{P}_1\, NAME \\
actor_state : NAME \rightarrowtail STATE \\
\hline
actor_name \in all_actors \\
\mathrm{dom}\, actor_state = actor_element \\
\hline
\end{array}
$$

The *actor_state* function is similar to the SD model's state function SD_state' and represents a collection of states of all internal elements of the actor.

For formalizing a general structure of all SR elements, we need to introduce a new free type, which describes possible types of links between the elements.

$$
\begin{aligned}
LINK_TYPE ::= {} & NA \mid task_decomp \mid means_ends \\
\mid {} & contrib
\end{aligned}
$$

Type *NA* (Non-Applicable) is used for elements which have no means for attaining them and have no components. Type *task_decomp* represents task decomposition links. Types *means_ends* and *contrib* describe means-ends links. Type *means_ends* is used for Goal-Task, Task-Task, Resource-Task, and Goal-Goal links. Type *contrib* represents special kinds of means-ends links for softgoal (Softgoal-Task and Softgoal-Softgoal links).

For convenience, we allocate all conditions connected with links into a separate schema *Link*. This schema includes:

- names of internal (inside the actor) elements *int_components* which are linked with the considered element;
- names of external (from SD model) dependencies *ext_components* which are linked with the considered element;
- type of the link;
- names of elements which give positive (*contrib_p*) and negative (*contrib_m*) contribution to the softgoals.

$$
\begin{array}{|l}
\hline
\textit{Link} \\
\Phi Depend \\
int_components, ext_components : \mathbb{P}\, NAME \\
contrib_p, contrib_m : \mathbb{P}\, NAME \\
link : LINK_TYPE \\
\hline
link = task_decomp \Rightarrow type = task \\
link = contrib \Rightarrow type = softgoal \\
contrib_p \cup contrib_m \neq \varnothing \Rightarrow link = contrib \wedge \\
\quad \langle contrib_p, contrib_m \rangle \text{ partitions } int_components \\
ext_components \neq \varnothing \Rightarrow link = task_decomp \\
link = NA \Leftrightarrow cint_components \cup ext_components = \varnothing \\
\hline
\end{array}
$$

The predicate part describes the following constraints between types of links and types of elements:

- Task decomposition links are used only for tasks;
- Positive or negative contribution is possible only for softgoals;
- Only task decomposition links are used for connection with external components.

The following schema describes a general structure of all the SR internal elements. This operational schema changes the state of the general model of an actor ($\Delta Actor$). Similarly *SDependency* schema, *AElement* one includes as components $\Phi Depend$ schema. Inclusion of *Link* schema brings all the information concerning links between the elements.

$$
\begin{array}{|l}
\hline
\textit{AElement} \\
\Delta Actor \\
Link \\
\hline
dependum \in actor_element \\
int_components \subset actor_element \\
ext_components \subseteq all_depend \\
actor_name' = actor_name \\
actor_element' = actor_element \\
actor_state' = actor_state \oplus \{dependum \mapsto result!\} \\
\hline
\end{array}
$$

IEEE COMPUTER SOCIETY

The predicate part of *AElement* schema formalizes the changes of *Actor* schema under the realization of the internal element. Only one component of *Actor* schema namely the actor's state function $actor_state'$ is changed. Similar to the SD model's state function *SD_state*, the difference between values of *actor_state* before and after the element realization exists only in the state of the considered element.

3 Mapping a specific i^* model into Z: an example

3.1 Mapping the SD model

Consider the example of mapping the i^* model into Z for the meeting scheduling system (see above Section 1). The SD model of the meeting scheduling system includes three actors (*initiator*, *scheduler*, and *participant*) and six dependencies (*scheduled*, *date_range*, *avail_dates*, *proposed_date*, *agreement*, and *attend*). First of all it is necessary to describe their names in Z using the following axiomatic definition:

$$
\begin{array}{l}
initiator, scheduler, participant : NAME \\
scheduled, date_range, avail_dates, \\
\quad proposed_date, agreement, attend : NAME \\
\hline
all_actors = \{initiator, scheduler, participant\} \\
all_depend = \{scheduled, date_range, \\
\quad avail_dates, proposed_date, agreement, attend\}
\end{array}
$$

The next step is to create six Z schemas (*Scheduled*, *DateRange*, *AvailDates*, *ProposedDate*, *Agreement*, and *Attend*) for each of six dependencies using *Sdependency* schema as a basis. In other words, we use inclusion of *Sdependency* schema and then additionally specify the following information: the names of the dependum, depender and dependee, the type and the degree of the dependency.

As an example, consider *DateRange* schema which describes the following task dependency - the scheduler expects the meeting initiator to enter the data range.

$$
\begin{array}{l}
\textit{DateRange} \\
\hline
SDependency \\
\hline
dependum = date_range \\
depender = scheduler \\
dependee = initiator \\
type = task \\
degree = committed \\
\hline
\end{array}
$$

Line $dependum = date_range$ shows the name of the dependency. It is a *task* dependency so $type = task$. The scheduler depends on the meeting initiator so $depender = scheduler$ and $dependee = initiator$. The importance of the dependency is not marked in the SD diagram hence we consider $degree = committed$.

Thus, *DateRange* schema corresponds to the *date_range* dependency. It is intuitively obvious because of the similarity of names (we use this similarity only for clarity purpose). The formal correspondence between schemas and dependencies is established by using variable *dependum* inside the schemas without explicitly using the names of schemas. The formal rule of correspondence is described below:

$$
\begin{array}{l}
correspond : NAME \rightarrowtail SDependency \\
\hline
\mathrm{dom}\, correspond = all_depend \\
\forall x : NAME \mid x \in all_depend \bullet \\
\quad (correspond(x)).dependum = x
\end{array}
$$

The schemas for all the other dependencies are similar to *DateRange* schema so we present only one of them without comments.

$$
\begin{array}{l}
\textit{AvailDates} \\
\hline
SDependency \\
\hline
dependum = avail_dates \\
depender = scheduler \\
dependee = participant \\
type = task \\
degree = committed \\
\hline
\end{array}
$$

3.2 Mapping the SR model

The first step of formalization of the SR model in Z is creating Z schemas *Initiator*, *Scheduler*, and *Participant* for each actor using *Actor* schema as the basis. In such schema we specify the name of the actor and names of all the internal elements of this actor. For example, consider the SR diagram of the meeting scheduling system (see above Section 1) *initiator* actor has four internal elements. We reflect it in the following Z schema:

$$
\begin{array}{l}
\textit{Initiator} \\
\hline
Actor \\
org_meeting, meeting_be_sch, low_effort, \\
\quad let_scheduler : NAME \\
\hline
actor_name = initiator \\
actor_element = \{org_meeting, meeting_be_sch, \\
\quad low_effort, let_scheduler\} \\
\hline
\end{array}
$$

The schemas for *scheduler* and *participant* actors are similar:

IEEE COMPUTER SOCIETY

$$
\begin{array}{l}
\textit{Scheduler} \\
\hline
\textit{Actor} \\
\textit{schedule}, \textit{obt_avail}, \textit{obt_agreement}, \textit{slot}, \\
\quad \textit{merge} : \textit{NAME} \\
\hline
\textit{actor_name} = \textit{scheduler} \\
\textit{actor_element} = \{\textit{schedule}, \textit{obt_avail}, \\
\quad \textit{obt_agreement}, \textit{slot}, \textit{merge}\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{Participant} \\
\hline
\textit{Actor} \\
\textit{participate}, \textit{attend}, \textit{arrange}, \textit{agreeable}, \textit{friendly}, \\
\quad \textit{using_sched}, \textit{talking_init}, \textit{agree_date} : \textit{NAME} \\
\hline
\textit{actor_name} = \textit{participant} \\
\textit{actor_element} = \{\textit{participate}, \textit{attend}, \textit{arrange}, \\
\quad \textit{agreeable}, \textit{friendly}, \textit{using_sched}, \\
\quad \textit{talking_init}, \textit{agree_date}\} \\
\hline
\end{array}
$$

The next step is the creation of Z schemas for all the internal elements using *AElement* schema as the basis. In this way, we need to create seventeen schemas - four for internal elements of *initiator* actor, five for *scheduler* actor, and eight for *participant* actor. It is necessary to specify the name of the dependee, the type and the degree of the element (similar external dependencies) but also the kind of the link and names of external and internal components of the considered element. We are describing an example (without comments) of two internal elements *using_sched* and *friendly* of *participant* actor.

$$
\begin{array}{l}
\textit{UsingSched} \\
\hline
\textit{AElement} \\
\textit{Participant} \\
\hline
\textit{dependum} = \textit{using_sched} \\
\textit{type} = \textit{task} \\
\textit{degree} = \textit{committed} \\
\textit{int_components} = \{\textit{agree_date}\} \\
\textit{ext_components} = \{\textit{avail_dates}\} \\
\textit{link} = \textit{task_decomp} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{Friendly} \\
\hline
\textit{AElement} \\
\textit{Participant} \\
\hline
\textit{dependum} = \textit{friendly} \\
\textit{type} = \textit{softgoal} \\
\textit{degree} = \textit{committed} \\
\textit{int_components} = \{\textit{using_sched}, \textit{talking_init}\} \\
\textit{contrib_p} = \{\textit{talking_init}\} \\
\textit{contrib_m} = \{\textit{using_sched}\} \\
\textit{ext_components} = \varnothing \\
\textit{link} = \textit{contrib} \\
\hline
\end{array}
$$

3.3 Refinement

The benefits of using concepts introduced in Section 2.1 state function *SD_state* becomes apparent on the step of the information refinement. The task of refining with additional information from the existing SD and SR models requires a separate investigation and is beyond the scope of this paper. We are providing essence of possible approaches with two examples.

The first example is concerned with the data exchange between actors. The *Scheduler* actor receives and sends information about the possible dates of the meeting. We use the basic type *DATE* to describe this information and introduce more detailed schema *Scheduler*1 which contains all these dates.

$$[\textit{DATE}]$$

$$
\begin{array}{l}
\textit{Scheduler}1 \\
\hline
\textit{Scheduler} \\
\textit{d_range}, \textit{d_avail} : \mathbb{P}\,\textit{DATE} \\
\textit{d_propos} : \textit{DATE} \\
\hline
\textit{d_avail} \subseteq \textit{d_range} \\
\textit{d_propos} \in \textit{d_avail} \\
\hline
\end{array}
$$

This gives us an opportunity to create a more detailed schema for dependency *AvailDates* (now *AvailDates*1). First of all, intermediate (partial) schema Φ*Avail*1 shows that *Scheduler*1 changes but only for *d_avail* component.

$$\Phi\textit{Avail}1 == \Delta\textit{Scheduler}1 \wedge (\Xi\textit{Scheduler}1 \setminus (\textit{d_avail}))$$

*AvailDates*1 schema includes Φ*Avail*1 and describes the way now *d_avail* changes.

$$
\begin{array}{l}
\textit{AvailDates}1 \\
\hline
\textit{AvailDates} \\
\Phi\textit{Avail}1 \\
\textit{input}? : \mathbb{P}\,\textit{DATE} \\
\hline
\textit{SD_state}(\textit{date_range}) = \textit{fulfilled} \\
\textit{d_avail} = \varnothing \Rightarrow \textit{d_avail}' = \textit{input}? \\
\textit{d_avail} \neq \varnothing \Rightarrow \textit{d_avail}' = \textit{d_avail} \cap \textit{input}? \\
\hline
\end{array}
$$

This change is possible only if the previous dependency *date_range* is realized (fulfilled) successfully. The scheduler collects data from several participants. Hence the condition of *date_range* realization is the selection of available dates which are suitable for all the participants.

The second example is concerning temporal features and operators. The state function *SD_state* represents the snapshot state of the system. To describe the behaviour of the

IEEE COMPUTER SOCIETY

system in time, consider all the possible sequences of system states

$$
\begin{array}{l}
SDscenarios : \mathbb{P}(\operatorname{seq} SD) \\
SDfuture, SDpast : SD \rightarrowtail \mathbb{P}(\operatorname{seq} SD) \\
\hline
\operatorname{ran} SDfuture \cup \operatorname{ran} SDpast \subseteq \mathbb{P}\, SDscenarios \\
\forall s : SD \bullet SDfuture(s) = \{f : \operatorname{seq} SD \mid head\, f = s\} \wedge \\
\quad SDpast(s) = \{p : \operatorname{seq} SD \mid last\, p = s\}
\end{array}
$$

For each state function s consider all the behaviours *Sdfuture* which are started in s (the future of the system) and behaviours *SDpast* that are finished in s (the past of the system). Now we can formalize all the main temporary operators such as *sometimes in the past*, *always in the past*, *sometimes in the future*, *always in the future*, etc., which are used in different techniques of requirements engineering, for example, KAOS [6], Formal Tropos [7]. Thus, the operator $\Box\ \phi$ *always in the future* [7] for state s can be modelled as

$$\forall c : \operatorname{seq} SD;\ st : SD \mid c \in SDfuture(s) \wedge \langle st \rangle \subseteq c \bullet \phi$$

Correspondingly, the operator $\circ\ \phi$ *next state* for state s can be modelled as

$$\forall c : \operatorname{seq} SD;\ st : SD \mid c \in SDfuture(s) \wedge st = c(2) \bullet \phi$$

and the operator $\Diamond\ \phi$ *eventually in the future* for state s can be modelled as

$$\forall c : \operatorname{seq} SD \mid c \in SDfuture(s) \bullet \exists\, st : SD \mid \langle st \rangle \subseteq c \bullet \phi$$

If we consider a system which demands special timing requirements (for example, a concurrent real-time reactive system), then it is possible to use special extensions of Z like Timed Communicating Object Z (TCOZ) [9] designed for modelling real-time.

4 Conclusions

Our proposal in this paper is that the Z formal notation and the i* modelling framework can function in a complementary and synergistic fashion. A conceptual modelling methodology that supports their co-evolution is of interest. This approach makes use of the advantages of i^* for the early-phase of requirements engineering (visualization of requirements, possibility of easy modifications, etc.) and then continues with the specification of requirements in Z. The Z notation permits us to specify late-phase requirements with a degree of precision and formality that i^* does not.

We have considered in detail the first step of the methodology - one-to-one mapping i^* diagrams into the Z notation. It allows us to formalize i^* diagrams without adding or suppressing information. The next step in the approach is the refinement of the methodology by considering additional information from the i^* diagrams. This information (invariant properties, fulfilment conditions, etc.) can be easily incorporated into the Z schemas and allows us to consider the dynamic changes in the system states. The complete methodology of refinement with additional information from the existing SD and SR models requires a separate investigation and forms the main direction of our future research.

References

[1] Bert, D., Bowen, J. P., King, S., Waldén, M., editors. *ZB2003: Formal Specification and Development in Z and B*, 3rd International Conference of B and Z Users, Turku, Finland, 4-6 June 2003. Springer-Verlag, LNCS 2651, 2003. ISBN 3-540-40253-5.

[2] Bowen, J. P. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press, 1996.

[3] Bowen, J. P., Hinchey, M. G., editors. *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, FACIT series, 1999.

[4] J. Castro, M. Kolp and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. To appear in *Information Systems*, Elsevier, Amsterdam, The Netherlands, 2002.

[5] Chung, L., Nixon, B. A., Yu, E., Mylopoulos J. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[6] Dardenne, A., van Lamsweerde, A., Fickas, S. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, Vol. 20, North Holland, 1993, pp. 3-50.

[7] Fuxman, A., Pistore, M., Mylopoulos, J., Traverso, P. Model checking early requirements specifications in Tropos. *Proceedings of Fifth IEEE International Symposium on Requirements Engineering*, Toronto, Canada, August 27-31, 2001, pp. 174 -181.

[8] Jia, X. *ZTC: A Type Checker for Z Notation. User's Guide. Version 2.03, August 1998.* Division of Software Engineering, School of Computer Science, Telecommunication, and Information Systems, DePaul University, USA, 1998.

[9] Mahony, B., Dong, J S. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, Vol. 26, Issue 2, February 2000, pp. 150–177.

[10] Spivey, J. M. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[11] Wang, X., Lesprance, y. Agent-Oriented Requirements Engineering Using ConGolog and i*. *Proceedings of 3rd International Bi-Conference Workshop Agent-Oriented Information Systems (AOIS-2001)*, Berlin, Germany, 2001, pp. 59-78.

[12] Yu, E., Mylopoulos, J. An Actor Dependency Model of Organizational Work - with Application to Business Process Reengineering. *Proceedings of Conference on Organizational Computing Systems*, Milpitas, California, USA, Nov. 1-4, 1993, pp. 258-268.

[13] Yu, E., Mylopoulos, J. Understanding ”Why” in Software Process Modelling, Analysis, and Design. *Proceedings of 16th International Conference on Software Engineering*, Sorrento, Italy, May 16-21, 1994, pp. 159-168.

[14] Yu, E. Towards Modelling Strategic Actor Relationships for Information Systems Development – With Examples from Business Process Reengineering. *Proceedings of 4th Workshop on Information Technologies and Systems*, Vancouver, B.C., Canada, December 17-18, 1994, pp. 21-28.

[15] Yu, E. Modelling Strategic Relationships for Process Reengineering. *PhD Thesis, Graduate Department of Computer Science, University of Toronto*, Toronto, Canada, 1995, pp. 124.

[16] Yu, E. Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. *Proceedings of 3rd IEEE International Symposium on Requirements Engineering*, Washington D.C., USA, January 6-8, 1997, pp. 226-235.

[17] Yu, E. Agent Orientation as a Modelling Paradigm. *Wirtschaftsinformatik*, April 2001, pp. 123-132.

IEEE COMPUTER SOCIETY