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

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
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-
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 sub-tasks 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 intention 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.
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, fulfillment 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.

\[
\text{all_actors, all_depend : } \mathbb{P}_1 \text{ NAME}
\]

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:
In contrast to other values, they have some common patterns. That is why we use the SR elements (internal inside actors) but at the same time (external between actors) varies from a general structure of by its own schema. A general structure of SD dependencies (as well as every element in SD dependencies and SR elements; are described in the SD model). 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:

- \( \Phi \text{Dependency} \) schema which describes a general structure of all the SD dependencies and includes \( \Phi \text{Depend} \) schema as one of the component parts;
- A detailed schema for every SD dependency using \( S\text{Dependency} \) schema as a basis.

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

\[
\Phi \text{Depend} \\
\text{dependum} : \text{NAME} \\
\text{type} : \text{TYPE} \\
\text{degree} : \text{DEGREE} \\
\text{result}! : \text{STATE} \\
\text{result}! \neq \text{unresolved} \\
\text{result}! = \text{satisfied} \lor \text{result}! = \text{denied} \lor \\
\text{result}! = \text{undetermined} \Rightarrow \text{type} = \text{softgoal}
\]

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 \( \Phi \text{Depend} \) describes the fact that the resulting state cannot be unresolved. The second line of the predicate part of \( \Phi \text{Depend} \) reflects that the resulting state can take the satisfied, denied or undetermined value only for softgoals.

The following \( S\text{Dependency} \) 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 (\( \Delta SD \)). \( S\text{Dependency} \) schema includes the components \( \Phi \text{Depend} \) 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 \( i^\ast \) model for a specific example.

\[
S\text{Dependency} \\
\Delta SD \\
\Phi \text{Depend} \\
\text{dependum} \in \text{all-depend} \\
\text{dependee} \in \text{all-actors} \\
\text{dependee} \in \text{all-actors} \\
\text{SD_state}' = \text{SD_state} \oplus \{\text{dependum} \mapsto \text{result}!\}
\]

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

\[\text{STATE} ::= \text{inapplicable} | \text{unresolved} | \text{fulfilled} \\
| \text{violated} | \text{satisfied} | \text{denied} | \text{undetermined}\]
the value of the SD model’s state function $SD_{\text{state}}'$ after the dependency realization differs from its value $SD_{\text{state}}$ 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 $\Phi_{\text{Depend}}$ 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 $actor_{\text{name}}$, set $actor_{\text{element}}$ of names of all internal elements, and state function $actor_{\text{state}}$.

\[
\text{Actor} \\
actor_{\text{name}} : \text{NAME} \\
actor_{\text{element}} : \mathbb{P}_1 \text{NAME} \\
actor_{\text{state}} : \text{NAME} \rightarrow \text{STATE} \\
\]

\[
\begin{align*}
\text{actor}_{\text{name}} &\in \text{all}_\text{actors} \\
\text{dom} \text{actor}_{\text{state}} &\leq \text{actor}_{\text{element}}
\end{align*}
\]

The $actor_{\text{state}}$ function is similar to the SD model’s state function $SD_{\text{state}}'$ 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.

\[
\text{LINK_TYPE ::= NA} \mid \text{task}_{\text{decomp}} \mid \text{means}_{\text{ends}} \\
\mid \text{contrib}
\]

Type NA (Non-Applicable) is used for elements which have no means for attaining them and have no components. Type $\text{task}_{\text{decomp}}$ represents task decomposition links. Types $\text{means}_{\text{ends}}$ and $\text{contrib}$ describe means-ends links. Type $\text{means}_{\text{ends}}$ is used for Goal-Task, Task-Task, Resource-Task, and Goal-Goal links. Type $\text{contrib}$ 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 $\text{int}_{\text{components}}$ which are linked with the considered element;
- names of external (from SD model) dependencies $\text{ext}_{\text{components}}$ which are linked with the considered element;
- type of the link;
- names of elements which give positive ($\text{contrib}_p$) and negative ($\text{contrib}_n$) contribution to the softgoals.

\[
\text{Link} \\
\Phi_{\text{Depend}} \\
\text{int}_{\text{components}}, \text{ext}_{\text{components}} : \mathbb{P}_1 \text{NAME} \\
\text{contrib}_p, \text{contrib}_n : \mathbb{P}_1 \text{NAME} \\
\text{link} : \text{LINK_TYPE}
\]

\[
\begin{align*}
\text{link} &\Rightarrow \text{type} = \text{task} \\
\text{link} &\Rightarrow \text{type} = \text{softgoal} \\
\text{contrib}_p \cup \text{contrib}_n &\neq \emptyset \Rightarrow \text{link} = \text{contrib} \\
\text{link} &\Rightarrow \text{task}_{\text{decomp}} \\
\text{link} \Rightarrow \text{NA} \Rightarrow \text{cint}_{\text{components}} \cup \text{ext}_{\text{components}} = \emptyset
\end{align*}
\]

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 ($\Delta\text{Actor}$). Similarly $\text{SDependency}$ schema, $\text{AElement}$ one includes as components $\Phi_{\text{Depend}}$ schema. Inclusion of **Link** schema brings all the information concerning links between the elements.

\[
\text{AElement} \\
\Delta\text{Actor} \\
\text{Link}
\]

\[
\begin{align*}
\text{dependum} &\in \text{actor}_{\text{element}} \\
\text{int}_{\text{components}} &\subseteq \text{actor}_{\text{element}} \\
\text{ext}_{\text{components}} &\subseteq \text{all}_\text{depend} \\
\text{actor}_{\text{name}}' &\leq \text{actor}_{\text{name}} \\
\text{actor}_{\text{element}}' &\leq \text{actor}_{\text{element}} \\
\text{actor}_{\text{state}}' &\leq \text{actor}_{\text{state}} \cup \{ \text{dependum} \rightarrow \text{result} \}
\end{align*}
\]
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:

$$
\text{initiator, scheduler, participant : NAME} \\
\text{scheduled, date\_range, avail\_dates,} \\
\text{proposed\_date, agreement, attend : NAME}
$$

The next step is to create six Z schemas ($Scheduled$, $Date\_Range$, $Avail\_Dates$, $Proposed\_Date$, $Agreement$, 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 $Date\_Range$ schema which describes the following task dependency - the scheduler expects the meeting initiator to enter the data range.

<table>
<thead>
<tr>
<th>$Date_Range$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SDependency$</td>
</tr>
<tr>
<td>$dependum = date_range$</td>
</tr>
<tr>
<td>$dependee = scheduler$</td>
</tr>
<tr>
<td>$type = task$</td>
</tr>
<tr>
<td>$degree = committed$</td>
</tr>
</tbody>
</table>

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 $dependee = scheduler$ and $dependee = initiator$. The importance of the dependency is not marked in the SD diagram hence we consider $degree = committed$.

Thus, $Date\_Range$ 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:

$$
correspond : \text{NAME} \rightarrow SDependency \\
\text{dom correspond = all\_depend} \\
\forall x : \text{NAME} \mid x \in \text{all\_depend} \bullet \ (correspond(x)).dependum = x
$$

The schemas for all the other dependencies are similar to $Date\_Range$ schema so we present only one of them without comments.

<table>
<thead>
<tr>
<th>$Avail_Dates$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SDependency$</td>
</tr>
<tr>
<td>$dependum = avail_dates$</td>
</tr>
<tr>
<td>$dependee = scheduler$</td>
</tr>
<tr>
<td>$dependee = participant$</td>
</tr>
<tr>
<td>$type = task$</td>
</tr>
<tr>
<td>$degree = committed$</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>$Initiator$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Actor$</td>
</tr>
<tr>
<td>$org_meeting$, $meeting_be_sch$, $low_effort$, $let_scheduler$ : $\text{NAME}$</td>
</tr>
</tbody>
</table>

The schemas for $scheduler$ and $participant$ actors are similar:
The next step is the creation of Z schemas for all the internal elements using \textit{AElem} schema as the basis. In this way, we need to create seventeen schemas - four for internal elements of \textit{initiator} actor, five for \textit{scheduler} actor, and eight for \textit{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 \textit{using\_sched} and friendly of \textit{participant} actor.

The benefits of using concepts introduced in Section 2.1 state function \textit{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 \textit{Scheduler} actor receives and sends information about the possible dates of the meeting. We use the basic type \textit{DATE} to describe this information and introduce more detailed schema \textit{Scheduler1} which contains all these dates.

$$[\text{DATE}]$$

This gives us an opportunity to create a more detailed schema for dependency \textit{AvailDates} (now \textit{AvailDates1}). First of all, intermediate (partial) schema $\Phi_{\text{Avail1}}$ shows that \textit{Scheduler1} changes but only for $d\_avail$ component.

$$\Phi_{\text{Avail1}} = \Delta\text{Scheduler1} \land (\exists\text{Scheduler1} \setminus (d\_avail))$$

\textit{AvailDates1} schema includes $\Phi_{\text{Avail1}}$ and describes the way now $d\_avail$ changes.

$$\text{SD\_state}(d\_range) = \text{fulfilled}$$

$$d\_avail = \emptyset \Rightarrow d\_avail' = \text{input}?$$

$$d\_avail \neq \emptyset \Rightarrow d\_avail' = d\_avail \cap \text{input}?$$

This change is possible only if the previous dependency $d\_range$ is realized (fulfilled) successfully. The scheduler collects data from several participants. Hence the condition of $d\_range$ 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 \textit{SD\_state} represents the snapshot state of the system. To describe the behaviour of the
system in time, consider all the possible sequences of system states

\[
\begin{align*}
SD_{\text{scenarios}} &: \mathcal{P}(\text{seq} SD) \\
SD_{\text{future}}, SD_{\text{past}} &: SD \rightarrow \mathcal{P}(\text{seq} SD) \\
\text{ran} SD_{\text{future}} \cup \text{ran} SD_{\text{past}} &\subseteq \mathcal{P} SD_{\text{scenarios}} \\
\forall s &: SD \ni SD_{\text{future}}(s) = \{ f : \text{seq} SD \mid \text{head } f = s \} \land \text{SD}_{\text{past}}(s) = \{ p : \text{seq} SD \mid \text{last } p = s \}
\end{align*}
\]

For each state function \( s \) consider all the behaviours \( SD_{\text{future}} \) which are started in \( s \) (the future of the system) and behaviours \( SD_{\text{past}} \) 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 \( \square \phi \) always in the future [7] for state \( s \) can be modelled as

\[
\forall c : \text{seq} SD; \ st : SD \mid c \in SD_{\text{future}}(s) \land \langle st \rangle \subseteq c \cdot \phi
\]

Correspondingly, the operator \( \circ \phi \) next state for state \( s \) can be modelled as

\[
\forall c : \text{seq} SD; \ st : SD \mid c \in SD_{\text{future}}(s) \land st = c(2) \cdot \phi
\]

and the operator \( \diamond \phi \) eventually in the future for state \( s \) can be modelled as

\[
\forall c : \text{seq} SD \mid c \in SD_{\text{future}}(s) \cdot \exists st : SD \mid \langle st \rangle \subseteq c \cdot \phi
\]

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 \( i^\varphi \) 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 \( i^\varphi \) 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 \( i^\varphi \) does not.

We have considered in detail the first step of the methodology - one-to-one mapping \( i^\varphi \) diagrams into the \( Z \) notation. It allows us to formalize \( i^\varphi \) diagrams without adding or suppressing information. The next step in the approach is the refinement of the methodology by considering additional information from the \( i^\varphi \) 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


