A Method for Requirements Elicitation and Formal Specification

Maritta Heisel[1] and Jeanine Souquières[2]

[1] Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, D-39016 Magdeburg, Germany, email: heisel@cs.uni-magdeburg.de
[2] LORIA—Université Nancy2, B.P. 239 Bâtiment LORIA, F-54506 Vandœuvre-les-Nancy, France, email: souquier@loria.fr

Abstract. We propose a method for the elicitation and the expression of requirements. The requirements are then transformed in a systematic way into a formal specification. The approach – which distinguishes between requirements and specifications – gives methodological support for requirements elicitation and specification development. It avoids introducing new notations but builds on known techniques.

1 Introduction

The usefulness of formal specification is more and more accepted by researchers and practical software engineers. Consequently, this subject is treated in the majority of software engineering text books, and many formal specification languages have been developed.

But formal specification techniques still suffer from two drawbacks. First, research spends more effort to develop new languages than to provide methodological guidance for using existing ones. Often, users of formal techniques are left alone with a formalism for which no explicit methodology has been developed.

Second, formal specification techniques are not well integrated with the analysis phase of software engineering. The starting point from which the development of a formal specification should begin is not well elaborated. Often, formal specification begins with a very short description of the system to be implemented, and detail is added during the development of the formal specification. Such a procedure does not adequately take into account the need to thoroughly analyze the system to be implemented and the environment in which it will operate *before* a detailed specification is developed. The elicitation of the requirements for the system and their transformation into a formal specification are separate tasks.

This paper treats both of these issues. It introduces an explicit requirements elicitation phase, the result of which is an adequate starting point for the development of the formal specification. Furthermore, it provides substantial methodological guidance for requirements elicitation as well as for specification acquisition. The different phases provide feedback to one another: not only is the specification based on the requirements, but the specification phase may also reveal omissions and errors in the requirements.

Our approach opts for a formal expression of requirements. After some informal brainstorming steps, the requirements are formalized as constraints on sequences of events that can happen or operations that can be invoked in the context of the system. Thus, the transition from informal to formal means of expression is performed very early in the software development process. This has the advantage that requirements can be analyzed, e.g., for consistency, and that it is possible to define a formal notion of correctness of a specification with respect to given requirements.

In the following, we describe the methods for requirements elicitation (Section 2) and specification acquisition (Section 3), which are illustrated by an example, an automatic teller machine. Section 4 discusses the state of the art, and a summary of our contributions concludes the paper.

2 Requirements elicitation

Our approach to requirements engineering is inspired by the work of Jackson and Zave [10, 17] and by the first steps of object oriented methods. The starting point is a brainstorming process where the application domain and the requirements are described in natural language. This informal description is then transformed into a formal representation.

Our approach is suitable for a large class of systems: transformational as well as reactive systems. Transformational systems perform data transformations. They offer a set of system operations that can be invoked by the user. Reactive systems, on the other hand, have to react to events that happen in their environment.

The difference between requirements and a specification is that requirements refer to the entire system to be realized, whereas a specification refers only to the part of the system to be implemented by software. To express requirements formally, we use *traces* of the system, i.e., sequences of events that happen in certain states and at a certain time. For transformational systems, events can be identified, too, namely the invocation and the termination of the system operations. In this way, systems that are partially reactive and partially transformational can also be treated.

2.1 Agenda for requirements elicitation

Requirements elicitation is performed in six steps. To express our method, we use the *agenda* concept [7].

An agenda is a list of steps to be performed when carrying out some task in the context of software engineering. The result of the task will be a document expressed in some language. Agendas contain informal descriptions of the steps, which may depend on each other. Usually, they will have to be repeated to achieve the goal, because later steps will reveal errors and omissions in earlier steps. Agendas are presented as tables, see Table 1.

Agendas are not only a means to guide software development activities. They also support quality assurance because the steps may have validation conditions associated with them. These validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly. Validation conditions that can in principle be checked mechanically are marked $\vdash$, validation conditions that can only be checked informally are marked $\circ$.

No	Step	Validation Conditions
1	Fix the domain vocabulary.	$\circ$ The vocabulary must contain exactly the notions occurring in the facts, assumptions, requirements, operations and events.
2	State the facts, assumptions and requirements concerning the system in natural language.	
3	List the possible system operations that can be invoked by the users, together with their input and output parameters.	
4	List all relevant events that can happen in connection with the system, together with their parameters.	
5	Classify the events.	$\vdash$ There must not be any events controlled by the software system and not shared with the environment.
6	Formalize facts, assumptions and requirements as constraints on the admissible traces of system events.	$\circ$ Each requirement of Step 2 must be expressed. $\vdash$ The constraints must be consistent. $\vdash$ For each predicate introduced, the events that modify it must be shared with the software system.

Table 1. Agenda for requirements elicitation

The starting point of the requirements elicitation phase is an informal requirements document provided by the customer. The goal of the first five steps of our method is to understand the problem, to fix the domain vocabulary, and to state the requirements more precisely. They usually will need communication with the customer. The result of these brainstorming steps forms the starting point for the formal expression of the requirements.

We now explain the steps of the agenda one by one.

Step 1. Fix the domain vocabulary.

In informal requirements documents, the used vocabulary is often ambiguous. Several names may be used to refer to the same thing or concept, and even the same word may be used to refer to different concepts. In this step, the domain vocabulary should be fixed in such a way the for each concept exactly one name is used.

To represent the domain vocabulary, we use a notation similar to entity-relationship diagrams [1].

Step 2. State the facts, assumptions and requirements concerning the system.

It does not suffice to just state requirements for the system. Often, facts and assumptions must be introduced to make the requirements satisfiable. *Facts* express phenomena that always hold in the application domain, regardless of the implementation of the software system. These are often hardware properties, e.g., that a card reader can hold at most hold one card. Other requirements cannot be enforced because, e.g., human users might violate regulations. In such a case, it may be possible that the system can fulfill its purpose properly only under the condition that users behave as required. Such conditions are expressed as *assumptions*, which can be used to show that the software system is correctly implemented. *Requirements* constrain the specification and implementation of the software system. It must be *demonstrated* that they are fulfilled, once the specification and the implementation are finished. In the specification phase, it must be shown that the modeling of the software system takes the facts into account and that the requirements are fulfilled, whereas the assumptions can be taken for granted.

If it is not already done in the initial requirements document, the facts, assumptions and requirements should be stated as *fragments* as small as possible. Such fragments can correspond to (parts of) independent scenarios of system behavior. For reasons of traceability, each fragment should be given a unique name or number.

Step 3. List the possible system operations that can be invoked by the users, together with their input and output parameters.

This step is concerned with the non-reactive part of the system to be described. For purely reactive systems, it can be empty. System operations are usually independent of the physical components of the system, but refer to the information that is stored in the part of the system to be realized by software.

With each system operation *Op*, the events *OpInvocation* and *OpTermination* are associated, where *OpInvocation* is controlled by the environment and shared with the software system, whereas the *OpTermination* is controlled by the software system and shared with the environment.

Step 4. List all relevant events that can happen in connection with the system, together with their parameters.

This step only concerns the reactive part of the system. For purely transformational systems, the set of events may be empty. An event usually has a relation with a physical part of the system, such as a button or a light.

Step 5. Classify the events.
According to Jackson and Zave[1] [10], events can be classified as follows:
1. events controlled by the environment and not shared with the software system,
2. events controlled by the environment but observable by the software system,
3. events controlled by the software system and observable by the environment,
4. events controlled by the software system and not shared with the environment.

[1] They use the term “machine” for what we call software system.

If system design is not yet finished, classifying the events means deciding which part of the task to be implemented will be fulfilled by the software system. If system design is already completed, then the classification of the events must be consistent with the system design. Phenomena that are internal to the software system cannot be part of the requirements but only of the software specification.

The first five steps of our method can be carried out in any order or in parallel. These steps result in several documents:

- an entity-relationship diagram to fix the domain vocabulary, expressing both static and dynamic aspects of the system,
- a list of system operations, concerning the transformational part of the system,
- a list of events, together with their classification, concerning the reactive part of the system,
- an updated informal requirements document, where the fixed vocabulary is used and the results of the communications with the customer are taken into account.

These documents must be consistent: the vocabulary should contain exactly the notions occurring in the facts, assumptions, requirements, operations and events.

Step 6. Formalize facts, assumptions and requirements as constraints on the admissible traces of system events.

We express requirements, assumptions and facts referring to the current state of the system, events that happen, and the time an event happens:

$$S_1 \xrightarrow[t_1]{e_1} S_2 \xrightarrow[t_2]{e_2} \dots S_n \xrightarrow[t_n]{e_n} S_{n+1} \dots$$

The system is started in state S_1. When event e_1 happens at t_1, then the system enters state S_2, and so forth. An element of a trace of the system thus contains a state, an event and a time. We require $t_i \leq t_j$ for $i \leq j$. Hence, concurrent events are possible.

Sometimes, it may be necessary to introduce predicates on the system state to be able to express the constraints formally. For example, an automatic teller machine can grant money to a customer only if the available amount is high enough. Expressing such a requirement makes it necessary to introduce a predicate *amount_available*. Such predicates, however, are only *declared* in the requirements elicitation phase. Their *definition* is part of the specification phase. For each predicate, the events that establish it and the events that falsify it must be given. These events must be shared with the software system.

Using constraints to express assertions on the behavior of the system has the following advantages:

- It is possible to express *negative* requirements, i.e., to require that certain things do not happen.
- It is possible to give scenarios, i.e., example behaviors of the system.
- Giving constraints does not fix the system behavior entirely. The constraints do not restrict the specification unnecessarily. Any specification that fulfills them is permitted.

In practice, requirements will often be inconsistent. We advocate to resolve conflicts at the requirements level, because no adequate specification can be derived from inconsistent requirements and because the requirements are the basis of the contract between customers and suppliers of software products. A formal representation of requirements is much more suitable to detect inconsistencies than an informal one. We have defined a heuristic approach to detect conflicting requirements [8] that cannot be presented here for reasons of space.

2.2 Case study : an automatic teller machine

We carry out the agenda of Section 2.1 for an automatic teller machine. Because the example is fairly simple, we describe the system and fix the domain vocabulary at the same time.

Step 1: Fix the domain vocabulary. A *client* has a *card* with which a *PIN* is associated. The card can be *inserted* into a *card reader*, which is part of the *automatic teller machine* (*ATM*). The client will be asked to *enter the PIN*. If three times a wrong PIN is entered, the card will be *kept*. Otherwise, the client will be asked to *enter an amount*. The ATM can either *refuse* or *grant* the amount. In the latter case, the client *takes the money*, and the ATM *debits* the *account* of the client, which is *managed* by a *bank*. Before money is granted, the client can *cancel the transaction* at any time. When the transaction is terminated, the card reader *ejects* the card, and the customer will *take the card*. The bank that runs the ATM can *query* the amount available in the ATM, and it can *recharge* the ATM.

Step 2: State the facts, assumptions and requirements concerning the system. We present a selection of relevant facts, assumptions and requirements.

$fact_1$ The card reader can hold only one card at a time.
$fact_2$ If the machine is out of service, it does not read an inserted card but ejects it immediately. This is a fact because the chosen hardware behaves that way.

ass_1 When the machine grants money to the customer, the customer will take the money.
ass_2 When the machine ejects the card, the customer will take it.

req_1 The inserted card must be valid. Otherwise, it is ejected immediately.
req_2 A transaction can always be canceled by the customer, as long as money is not yet granted.
req_3 To withdraw money, customers have to insert their card and enter their PIN.
req_4 A customer has only three trials to type the right PIN. If three times a wrong PIN is entered, the card is kept by the teller machine.
req_5 A user can only withdraw money if a weekly limit is not exceeded and if the demanded amount does not exceed the amount currently available in the teller machine.
req_6 All valid withdrawal transactions entail a withdraw order to the bank of the client.
req_7 The bank can recharge the machine with money between two withdrawal transactions.
req_8 The bank can query the amount of money available in the teller machine.

Step 3: List the possible system operations that can be invoked by the users. As far as the non-reactive aspects of the system are concerned, we have two system operations that can be invoked by the bank to maintain the teller machine, one to recharge the machine with money, the other to query the amount of money available in the machine.

Step 4: List all relevant events that can happen in connection with the system, together with their parameters. For the teller machine, we can identify the following events: $insert\_card$, $enter\_PIN(p : PIN)$, $enter\_amount(n : \mathbb{N})$, $grant\_amount(n : \mathbb{N})$, $refuse\_amount$, $debit\_account(c : CARD, n : \mathbb{N})$, $take\_money$, $eject\_card$, $keep\_card$, $take\_card$, and $cancel\_transaction$.

Step 5: Classify the events. We classify the events as follows:

– Environment controlled and shared with software system are the events $insert\_card$, $enter\_PIN(p : PIN)$, $enter\_amount(n : \mathbb{N})$, $take\_card$, $cancel\_transaction$.
– We assume that the hardware of the automatic teller machine does not have a sensor to detect if the client really takes the money. Hence, the event $take\_money$ is environment controlled and not shared with software system.
– Controlled by software system and shared with environment are the events $grant\_amount(n : \mathbb{N})$, $refuse\_amount$, $debit\_account(c : CARD, n : \mathbb{N})$, $eject\_card$ and $keep\_card$.
– As required by the validation condition associated with this step, there are no events that are controlled by the software system but not shared with the environment.

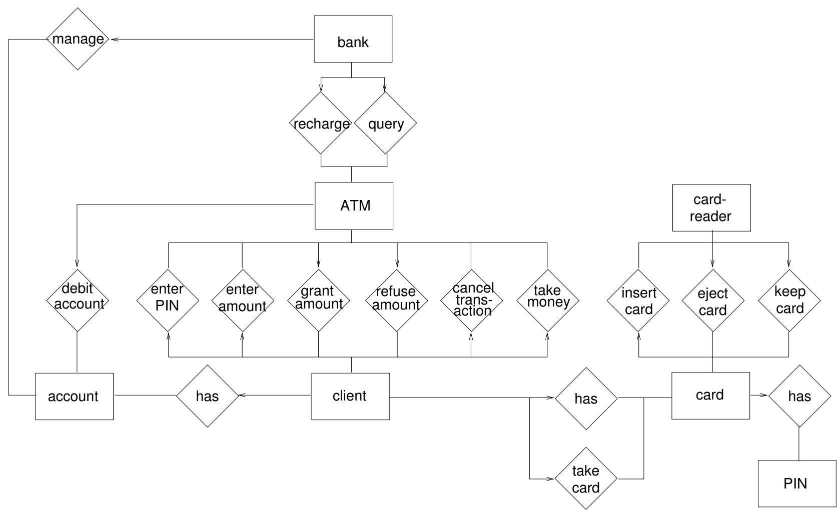

Fig. 1. Domain vocabulary for the teller machine

An entity-relationship-like diagram that summarizes the results of the informal steps of the agenda is given in Figure 1. Boxes denote entities, and diamonds denote relations. Arrows indicate the direction of the relations. The relation *has* denotes static aspects of the system, whereas all other relations concern dynamic aspects.

The reader can easily check the validation condition associated with Steps 1–4: the diagram contains exactly the notions used to express the facts, assumptions, requirements, operations and events.

Step 6: Formalize facts, assumptions and requirements as constraints on the admissible traces of system events. For each system, we call the set of possible traces *Tr*. Constraints are expressed as formulas restricting the set *Tr*. For a trace $tr \in Tr$, $tr(i)$ denotes the i-th trace element, $tr(i).s$ denotes the i-th state that is reached, and $tr(i).e$ denotes the event that happens in that state. For each possible trace, its prefixes are also possible traces. The definitions of the specification macros *immediately_followed_by* and *subtraces* used in the following are obvious.

For reasons of space, we only present the formalizations of $fact_1$, ass_1, req_1 and req_4.

$\underline{fact_1}\ \forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \bullet card\_inside(tr(i).s) \Rightarrow tr(i).e \neq insert\_card)$

To formalize $fact_1$, we have introduced a predicate *card_inside* on the system state. The predicate is established by the event *insert_card*, and it is falsified by the events *eject_card* and *keep_card*.

$\underline{ass_1}\ \forall\, n : \mathbb{N}_1 \bullet grant\_amount(n)\ immediately\_followed\_by\ take\_money$

$$\begin{aligned}
&\underline{req_1}\ \forall\, tr : Tr \bullet (\forall\, i : \mathrm{dom}\, tr \mid i \neq \#tr \bullet \\
&\qquad\qquad (tr(i).e = insert\_card \wedge tr(i+1).e \neq eject\_card \\
&\qquad\qquad\quad \Rightarrow valid\_card(tr(i+1).s)))
\end{aligned}$$

To formalize req_1, we have introduced a predicate *valid_card* on the system state. If an inserted card is not ejected immediately, the state reached after insertion of the card must satisfy the predicate *valid_card*. The predicate *valid_card* is established by the event *insert_card* and falsified by the events *eject_card* and *keep_card*.

$$\begin{aligned}
&\underline{req_4}\ \forall\, tr : Tr \bullet \\
&\quad (\forall\, tr' : subtraces(tr, insert\_card, grant\_amount) \bullet \\
&\qquad \#\{trit : \mathrm{ran}\, tr' \mid \exists\, p : PIN \bullet trit.e = enter\_PIN(p)\} \leq 3 \wedge \\
&\qquad \exists\, i : \mathrm{dom}\, tr';\ p : PIN \mid ti.e = enter\_PIN(p) \bullet \\
&\qquad\qquad right\_PIN(tr'(i+1).s)) \\
&\quad \wedge \quad (\forall\, tr' : subtraces(tr, insert\_card, insert\_card) \mid \\
&\qquad (\#\{trit : \mathrm{ran}\, tr' \mid \exists\, p : PIN \bullet trit.e = enter\_PIN(p)\} = 3 \wedge \\
&\qquad (\forall\, i : \mathrm{dom}\, tr' \mid \exists\, p : PIN \bullet tr'(i).e = enter\_PIN(p) \bullet \\
&\qquad\qquad wrong\_PIN(tr'(i+1).s))) \bullet \\
&\qquad\quad (\exists\, j : \mathrm{dom}\, tr' \bullet tr'(j).e = keep\_card))
\end{aligned}$$

The first conjunct of the formula states that if money is granted, then at most three times a PIN has been entered, and one of them must have been valid[2].

[2] Note that we choose not to prescribe that after a correct PIN has been entered, no more *enter_PIN* events will occur. This will be decided in the specification.

The second conjunct of the formula states that if three times the wrong PIN is entered, then a *keep_card* event occurs before the next *insert_card* event. To express this constraint, have introduced predicates *right_PIN* and *wrong_PIN* on states that express whether an entered PIN is valid or not.

Not only the formal constraints, but also the documents produced during the first five steps of the agenda form the starting point for the specification phase.

3 Specification development

Jackson and Zave [10] consider a specification to be a special kind of requirement. A requirement is a specification if all events constrained by the requirement are controlled by the software system, and all information it relies on is shared with the software system and refers only to the past, not the future. Requirements (and thus specifications) do not make statements about the state of the software system. In contrast to this view, we consider a specification to be a *model* of the software system to be built in order to satisfy the requirements. It forms the basis for refinement and implementation. Therefore, in our approach, a specification may - in contrast to the requirements - make statements about the software system that are not directly observable by the environment.

While requirements elicitation is independent of the specification language that is used, the development of a specification depends to a certain extent on the specification language and its means of expression. In the following, we use the specification language Z [16]. If other specification languages are used, our method is applicable, too. In this case, slight changes of the agenda might be necessary.

3.1 Agenda for specification development from requirements

The starting point of the specification development is the whole material obtained by the requirements elicitation phase presented in Section 2. Again, our method for specification acquisition is expressed as an agenda, given in Table 2. The steps have to be performed in the given order.

Step 1. Define a first approximation of the software system state.

This approximation should be defined in such a way that as many as possible of the predicates on the system state that were introduced in the requirements elicitation process can be defined. The initial states are specified in parallel with the legal states.

Step 2. Augment the specification, incorporating the requirements one by one.

For each constraint, this step should be performed in two sub-steps. The basic idea is to define a Z operation for each event identified in Step 4 of the requirements elicitation phase that is shared with the software system (according to the classification made in Step 5), and for each system operation identified in Step 3.

Step 2.1 can be performed by a simple syntactic inspection of the constraint in question. Step 2.2, however, can be complex with several revisions of the current

No	Step	Validation Conditions
1	Define a first approximation of the software system state and the initial states.	
2	Augment the specification, incorporating the requirements one by one. For each constraint: **2.1** List the events occurring in the constraint. **2.2** For each event in the list, set up a first definition of the corresponding Z operation, or adjust an already existing operation.	$\vdash$ The constraints expressing facts must not be violated. $\circ$ All events introduced in the requirements elicitation phase must be taken into account. $\circ$ The preconditions of the operations must be defined appropriately. $\vdash$ The post-states of the operations must be completely defined. $\vdash$ Each predicate on states used in the requirements must be definable with the final definition of the system state.

Table 2. Agenda for specification acquisition

version of the specification [12]. The state of the system and the operations have to be re-considered in order to take into account the evolution introduced by new constraints. Propagation of modifications is important here [15]. Incorporating a new constraint may involve the following modifications: (i) adding or modifying state components, (ii) adding or modifying data types, (iii) adding or modifying the state invariant and (iv) propagating those modifications into the current state of the specification.

The first two validation conditions require that all definitions and modifications must respect the domain properties as they are expressed by the facts, and that no event may be "forgotten".

To explain the third validation condition, we note that the preconditions for system operations (see Step 3 of the agenda for requirements elicitation) should be as weak as possible to make the software system more robust. For operations corresponding to the reactive part of the system, however, the situation is different. Strong preconditions may enforce a certain order in which operations can be applied. This is a possible way to encode behavioral descriptions in Z. In our case study (see Section 3.2), we will make use of this possibility.

Fourth, in most cases, an operation should give a condition for all state components that specifies their value after the operation has terminated. If non-determinism is introduced deliberately, this should be justified.

The last validation condition refers to the final definition of the system state. It must contain enough information to define the predicates introduced to express constraints in the requirements elicitation phase.

The agendas for requirements elicitation and specification acquisition provide an integrated approach that introduces formality as early as possible in the software engineering process. The formal expression of requirements and facts as constraints guide the development of the formal specification.

Our approach even allows to define a *notion of correctness* of a specification with respect to requirements, facts and assumptions: the set of possible traces of the specification is the set of traces where each operation is executed only if its precondition is satisfied. Assuming that the assumption constraints are satisfied, it must be demonstrated that the set of possible traces induced by the specification fulfills the constraints stated as requirements and facts.

3.2 Case study : the automatic teller machine

Taking the result of the requirements elicitation phase of Section 2.2 as the starting point, we now use the agenda presented in Section 3.1 to develop a formal Z specification of the automatic teller machine.

Step 1: Define a first approximation of the software system state. In formalizing the requirements, we have introduced the predicates *card_inside* $out\_of\_service(st : STATE)$, $valid\_card(st : STATE)$, $right\_PIN(st : STATE)$, $wrong\_PIN(st : STATE)$, $amount\_available(st : STATE, n : \mathbb{N})$ and $customer\_limit\_not\_exceeded(st : STATE, c : CARD, n : \mathbb{N})$. These motivate the following definitions:

$$OP\_MODES ::= out\_of\_service \mid in\_service$$

$$[CARD, PIN]$$

$$\begin{array}{|l}
valid\_cards : \mathbb{P}\,CARD \\
null\_card : CARD \\
\hline
null\_card \notin valid\_cards
\end{array}$$

$$\begin{array}{|l}
get\_PIN : CARD \rightarrowtail PIN \\
possible\_withdraw : CARD \rightarrowtail \mathbb{N} \\
\hline
valid\_cards \subseteq \operatorname{dom} get\_PIN \\
valid\_cards \subseteq \\
\qquad \operatorname{dom} possible\_withdraw
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Teller\_machine}} \\
\hline
current\_card : CARD \\
op\_mode : OP\_MODES \\
available\_amount : \mathbb{N} \\
\hline
(current\_card = null\_card \Leftrightarrow op\_mode \in \\
\qquad \{in\_service, out\_of\_service\}) \\
(current\_card \in valid\_cards \Leftrightarrow op\_mode \notin \\
\qquad \{in\_service, out\_of\_service\}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init\_Teller\_machine}} \\
\hline
Teller\_machine' \\
given? : \mathbb{N} \\
\hline
op\_mode' = in\_service \\
current\_card' = null\_card \\
available\_amount' = given? \\
\hline
\end{array}$$

With this preliminary definition of the system state, we cannot define the predicates $right\_PIN(st : STATE)$ and $wrong\_PIN(st : STATE)$. However, we decide that the PIN should only be an input, and not a state component. We have chosen to introduce a state component that stores the introduced card, because we will need more information about this card in the following. The predicate *card_inside* can be defined as $card \neq null\_card$.

Step 2: Augment the specification, incorporating the requirements one by one. For each event which is shared with the software system, we introduce an operation. The definition of each operation is guided by analyzing each requirement constraint, taking also into account the domain properties expressed by the facts.

Goal: $\underline{req_1}$ The inserted card must be valid. Otherwise, it is ejected immediately.

List of events: *insert_card*, *eject_card*

To define first approximation of the *insert_card* operation derived from req_1, we introduce a new value *valid_card* for the operational modes of the teller machine, which must be added to the type *OP_MODES*. We also must define a second operation, *eject_card*, that returns the card to the customer:

$$\begin{array}{|l}
\hline
\textit{insert\_card} \\
\Delta \textit{Teller\_machine} \\
\textit{card}? : \textit{CARD} \\
\hline
\textit{op\_mode} \in \{\textit{in\_service}, \textit{out\_of\_service}\} \\
(\textit{op\_mode} = \textit{out\_of\_service} \lor \\
\qquad \textit{card}? \notin \textit{valid\_cards}) \Rightarrow \textit{eject\_card} \\
(\textit{op\_mode} = \textit{in\_service} \land \textit{card}? \in \textit{valid\_cards} \\
\qquad \Rightarrow \textit{current\_card}' = \textit{card}? \\
\qquad \land\ \textit{op\_mode}' = \textit{valid\_card} \land \\
\qquad \textit{available\_amount}' = \textit{available\_amount}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{eject\_card} \\
\Delta \textit{Teller\_machine} \\
\hline
\textit{current\_card}' = \textit{null\_card} \\
\textit{op\_mode} \neq \textit{out\_of\_service} \\
\qquad \Rightarrow \textit{op\_mode}' = \textit{in\_service} \\
\textit{op\_mode} = \textit{out\_of\_service} \\
\qquad \Rightarrow \textit{op\_mode}' = \textit{op\_mode} \\
\textit{available\_amount}' = \textit{available\_amount} \\
\hline
\end{array}$$

Note that $\textit{card} \neq \textit{null\_card}$ is not a precondition for the operation *eject_card*, because $fact_2$ states that an invalid card is ejected immediately, as defined in *insert_card*. To take $fact_1$ into account, we have given *insert_card* the precondition $\textit{op\_mode} \in \{\textit{in\_service}, \textit{out\_of\_service}\}$. In the following, we must guarantee that this precondition is established only by the operations *eject_card* and *keep_card*.

Goal: $\underline{req_4}$ A customer has only three trials to type the right PIN. If three times a wrong PIN is entered, the card is kept by the teller machine.

List of events: *insert_card*, *grant_amount*, *enter_PIN*(p), *keep_card*

We have to add new values to the *op_mode* state component to count the number of trials of the customer and to express if the customer is granted the money or the card is kept.

$$\begin{array}{l}
\textit{OP\_MODES} ::= \textit{out\_of\_service} \mid \textit{in\_service} \mid \textit{valid\_card} \mid \textit{PIN\_entered} \\
\qquad\qquad\qquad \mid \textit{incorrect\_PIN}1 \mid \textit{incorrect\_PIN}2 \mid \textit{failure} \mid \textit{success}
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{enter\_PIN} \\
\Delta \textit{Teller\_machine} \\
\textit{pin}? : \textit{PIN} \\
\hline
\textit{op\_mode} \in \{\textit{valid\_card}, \textit{incorrect\_PIN}1, \textit{incorrect\_PIN}2\} \\
\textit{get\_PIN}(\textit{current\_card}) = \textit{pin}? \Rightarrow \textit{op\_mode}' = \textit{PIN\_entered} \\
\textit{get\_PIN}(\textit{current\_card}) \neq \textit{pin}? \Rightarrow \\
\qquad (\textit{op\_mode} = \textit{valid\_card} \Rightarrow \textit{op\_mode}' = \textit{incorrect\_PIN}1) \land \\
\qquad (\textit{op\_mode} = \textit{incorrect\_PIN}1 \Rightarrow \textit{op\_mode}' = \textit{incorrect\_PIN}2) \land \\
\qquad (\textit{op\_mode} = \textit{incorrect\_PIN}2 \Rightarrow \textit{op\_mode}' = \textit{failure}) \\
\textit{current\_card}' = \textit{current\_card} \\
\textit{available\_amount}' = \textit{available\_amount} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{keep\_card}} \\
\Delta \mathit{Teller\_machine} \\
\hline
\mathit{op\_mode} = \mathit{failure} \\
\mathit{current\_card}' = \mathit{null\_card} \\
\mathit{op\_mode}' = \mathit{in\_service} \\
\mathit{kept\_cards}' = \mathit{kept\_cards} \\
\qquad \cup \{\mathit{current\_card}\} \\
\mathit{available\_amount}' = \\
\qquad \mathit{available\_amount} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{grant\_amount}} \\
\Delta \mathit{Teller\_machine} \\
\hline
\mathit{op\_mode} = \mathit{PIN\_entered} \\
\mathit{op\_mode}' = \mathit{success} \\
\hline
\end{array}$$

With these definitions, $fact_1$ is not violated, because only *keep_card* establishes the precondition of *insert_card*. The operation *grant_amount* will be defined further when taking into account req_5 and req_6.

A new state component $\mathit{kept\_cards} : \mathbb{P}\, CARD$ must be added to the schema *Teller_machine*, and the current specification has to be revised by adding the equation $\mathit{kept\_cards}' = \mathit{kept\_cards}$ to each operation schema except *keep_card*.

Note that in the requirements we have no statement about what happens with the kept cards. As it is now, the set *kept_cards* can only be augmented. Thus, we have detected a missing requirement.

The validation conditions of the agenda are fulfilled, as can be checked by inspection of the specification. As an application-dependent validation of the teller machine specification, we can check if all operational modes that were introduced during the specification acquisition phase are indeed reachable from an initial state. It turns out that we have introduced a mode, *out_of_service*, without any possibility to access it. Again, we have detected a missing requirement, which should cause us to add a requirement describing when the teller machine goes out of service.

4 Related work

Separating domain knowledge from requirements and checking consistency between these is frequent in the literature. For example, when Parnas describes the mathematical content of a requirements document for a nuclear shutdown system [14], he introduces different mathematical relations, one describing the environment of the computer system, and one describing the requirements of the computer system. The two relations must fulfill certain feasibility conditions. Parnas also notes that a critical step in documenting the requirements concerns the identification of the environmental quantities to be measured or controlled and the representation of these quantities by mathematical variables. He proposes to characterize environmental quantities as either monitored or controlled. This corresponds to the classification of events in our and Jackson and Zave's approach [10]. Reading a monitored quantity corresponds to an event observable by the software system, and controlling a quantity corresponds to events controlled by the software system.

Whereas assumptions are not used by Parnas and by Jackson and Zave, they do play a role in KAOS [2, 3]. KAOS supports the design of composite systems (see also [5]). Such a system consists of human, software and hardware agents, each of them being assigned responsibility for some goals. The KAOS approach is goal-oriented: goals are stated and then elaborated into KAOS specifications in several consecutive steps, starting with elaborating the goals in an AND/OR structure and ending with the assignment of responsibilities to agents. KAOS is similar to our approach in that it provides heuristics for requirements elicitation and specification development. It is distinguished from our approach in that it uses its own language and in that it takes a much broader perspective: not only the software system, but also its environment are modeled in detail. This results in a very rich terminology. In contrast, our approach is focussed on developing a formal specification for a software component. We only model those aspects of the environment that are necessary for an adequate specification.

Easterbrook and Nuseibeh [4] do not distinguish different phases for requirements elicitation and specification development. They elicit more requirements when they detect inconsistencies in their specifications. On the one hand, their approach makes it possible to delay the resolution of conflicts. On the other hand, this kind of "lazy" requirements elicitation delays the point where a definite contract between customers and providers of the software product can be made. Moreover, it is harder to validate a specification with respect to the requirements when no separate requirements document is set up that is checked for consistency and completeness in its own right.

We have already contrasted our approach to the one of Jackson and Zave [10, 17]: first, we see a difference in requirements and specifications. As a result, our method leads to a formal specification that is expressed in a conventional specification language, whereas their approach stops when the requirements have been transformed in such a way that they can be regarded as a specification. The language in which the requirements are expressed (formulas of first-order predicate logic) is not changed during this process. Second, the most important part of our approach is the methodological guidance that is given to analysts and specifiers in form of agendas, as well as the validation of the developed products. These issues are not addressed explicitly by Jackson and Zave.

Of the object-oriented methods, we have only borrowed the very first steps, where the relevant vocabulary is introduced in a systematic way. Our approach is not biased toward object-oriented development.

5 Conclusions

Requirements define a set of conditions that must be met by a system or a system component to satisfy a contract, standard or other imposed document or description. For example, the IEEE Standard 1498 [9] defines a requirement as a characteristic that a system or a software item must possess in order to be acceptable to the acquirer. Requirements should be completely and unambiguously stated. In our approach, achieving completeness is supported by the first steps of the requirements elicitation agenda, where brainstorming processes are performed, and by feedback from the specification phase. In our example, we found missing requirements by analyzing the formal specification. Formalizing the requirements as we do eliminates ambiguities.

Adequately integrating formal methods into the whole development process is still an important challenge in software engineering. Integration strategies have been classified by Fraser et al. [6] with respect to the following factors:

1. Does the strategy lead directly from the informal requirements to the formalized specification, or does it introduce intermediate and increasingly formal models of the requirements?
2. If the strategy introduces intermediate models, is the process one of parallel, successive refinement of the requirements and the formal specification, or are the formal specifications derived after the requirements models have been finalized in a sequential strategy?
3. To what extend does the strategy offer mechanized support for requirements capture and formalization?

In terms of these classification criteria, our approach can be classified as *transitional*, because we introduce predicates over system event traces as an intermediate representation of the requirements. Our approach is *sequential*, because the specification phase is only entered when the requirements are deemed complete and correct. However, as already mentioned, feedback between the two phases is possible. Mechanized support for our approach is not yet available. Such support is conceivable for checking validation conditions, performing conflict analyses, and for developing the specification. First steps in this direction have already been performed: there is a prototypical implementation of our algorithm to detect conflicting requirements [8], and, furthermore, with Proplane [13] a specification support system already exists that can be adjusted to support the method presented here.

Requirements traceability is an important issue in requirements engineering. Jarke [11] defines requirements traceability as the ability to describe and follow the life of a requirement, in both a forward and backward direction. In our method, traceability is guaranteed in the following way:

– Single requirements are fragments as small as possible. The smaller the requirements, the better traceable they are, because their realization does not distribute over large parts of the system.
– For each event and each predicate that is introduced, it is noted in which requirements it is used.
– For each part of the formal specification, we can name the requirements that lead us to define it in the way we did.

In summary, our approach can be characterized as follows: requirements/requirements elicitation on the one hand and specifications/specification acquisition on the other hand are clearly distinguished. We give substantial methodological guidance for the two activities, which are integrated smoothly. Our approach does not introduce a new language or a new formalism. The requirements elicitation phase is independent of the specification language to be used, and the specification development phase can be adjusted to support the usage of other specification languages than Z. The method is suitable for transformational as well as reactive systems. Also real-time considerations can be taken into account.

We propose a standardized way of expressing facts, assumptions and requirements. Constraints on the set of possible traces are a very flexible and powerful means of describing a system and its interaction with the environment. Expressing requirements as constraints on traces makes it possible to systematically detect conflicting requirements and to define a formal notion of correctness of a specification with respect to a set of requirements.

References

1. P. Chen. The entity-relationship model – towards a unified view of data. *ACM Transactions on Database Systems*, 1(1), 1976.
2. A. Dardenne, A.v. Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.
3. R. Darimont and A.v. Lamsweerde. Formal Refinement for Patterns for Goal-Driven Requirements Elaboration. In *Proc FSE-4, ACM Symposium on the Foundation of Software Engineering*, pages 179–190, 1996.
4. S. Easterbrook and B. Nuseibeh. Using ViewPoints for inconsistency management. *Software Engineering Journal*, pages 31–43, January 1996.
5. M.S. Feather, S. Fickas, and R.B. Helm. Composite System Design : the Good News and the Bad News. In *Proc. 6th Knowledge-Based Software Engineering Conference*, pages 16–25. IEEE Computer Society Press, 1991.
6. M.D. Fraser, K. Kumar, and V.K. Vaisnavi. Strategies for Incorporating Formal Specifications in Software Development. *CACM*, 37(10):74–86, Oct. 1994.
7. M. Heisel. Agendas – a concept to guide software development activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.
8. M. Heisel and J. Souquières. A heuristic approach to detect feature interactions in requirements. In K. Kimbler and W. Bouma, editors, *Proc. 5th Feature Interaction Workshop*, pages 165–171. IOS Press Amsterdam, 1998.
9. IEEE94. Software development. IEEE publications office, IEEE Standard 1498, Los Alamitos, CA, March 1994.
10. M. Jackson and P. Zave. Deriving specifications from requirements: an example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.
11. M. Jarke. Requirements tracing. *Communications of the ACM*, pages 32–36, December 1998.
12. N. Lévy and J. Souquières. A "Coming and Going" Approach to Scenario. In W. Schafer, J. Kramer, and A. Wolf, editors, *Proc. 8th Int. Workshop on Software Specification and Design*, pages 115–158. IEEE Computer Society Press, 1996.
13. N. Lévy and J. Souquières. Modelling Specification Construction by Successive Approximations. In M. Johnson, editor, *6th International AMAST conference*, pages 351–364. Springer Verlag LNCS 1349, 1997.
14. D.L. Parnas. Using Mathematical Models in the Inspection of Critical Systems. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*, pages 17–31. Prentice Hall, 1995.
15. S. Sadaoui and J. Souquières. Quelques approches de la réutilisation dans le modèle Proplane. In *Conférence AFADL, Approches formelles dans l'assistance au développement de logiciels*, pages 85–96, Toulouse, 1997. Onera–Cert.
16. J. M. Spivey. *The Z Notation – A Reference Manual.* Prentice Hall, 2nd edition, 1992.
17. P. Zave and M. Jackson. Four dark corners for requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.