Precise Scenarios - A Customer-Friendly Foundation for Formal Specifications

Oliver Au, Roger Stone, and John Cooke

Loughborough University, England,
{o.t.s.au, r.g.stone, d.j.cooke}@lboro.ac.uk

Abstract. A formal specification is beyond the comprehension of the average software customer. As a result, the customer cannot provide useful feedback regarding its correctness and completeness. To address this problem, we suggest the formalism expert to work with the customer to create precise scenarios. A precise scenario describes an operation by its effects on the system state with only a few simple Z concepts. The customer would find a concrete precise scenario easier to understand than its corresponding abstract schema. The Z expert derives schemas based on the precise scenarios. Precise scenarios affords user involvement that improves the odds of a formal specification fully capturing the user requirements.

Key words: precise scenario, formal method, requirements specification, use case, Z

1 Introduction

The most important software project success factor is user involvement [1]. Formal specification languages, being hard to read [2], discourage user involvement. The resulting formal specification may not truly reflect user's requirements. This could explain the limited adoption of formal methods [3]. We hope to broaden the appeal of formal specification by increasing user involvement with precise scenarios.

Use cases and scenarios may be used to involve customers in requirements elicitation. There should be a use case for each user task. A use case consists of several scenarios, one scenario for a particular situation. The most common way to describe the details of a scenario is by listing a sequence of steps. Each step describes an actor and its action in a natural language. Due to the ambiguous nature of natural languages, the formal specification cannot be verified against the scenario descriptions it was based on.

We propose to describe a scenario by its precise effects on a state. A state is represented by its actual data expressed in a small subset of the specification language Z. For the layman, actual data are easier to understand than their abstract descriptions. The small number of Z symbols being used make our precise scenarios easier to understand than the corresponding schemas. The customers can participate in the creation of the precise scenarios. But scenarios only partially describe the infinite behaviour of a software application. A Z expert will need to generalise the scenarios into schemas that form a complete Z specification.

The idea of creating formal specifications from scenarios is not new. Amyot et al. expressed scenarios in Use Case Maps (UCM) to be translated into high-level LOTOS specifications [4]. Whittle and Schumann started out with UML sequence diagrams and created statechart diagrams from them [5]. Uchitel et al. [6] and Damas et al. [7] used message sequence charts (MSC) to synthesize labeled transition system (LTS). Their work has been limited to system behaviour made up of sequences of parameterless events. Being based on Z, we can extend sequences of simple events to general computation in the precise scenarios.

In another strand of research, Grieskamp and Lepper combined use cases with Z [8]. Use cases benefit from Z for its added precision while a Z specification benefits from use cases for relating its operations in actual usage. Test dialogues are built using nondeterministic choice, repetition and interruption. Executing a test dialogue in ZETA, the final state can be determined [9]. Their focus is on *black-box test evaluation.* The added capability has worsened the accessibility by software customers which contrasts with our research objective.

We use a simple ordering and invoicing problem to demonstrate our approach [10]. Section 2 describes the problem and a representation of the state space in Z. Each of sections 3 to 6 describes a user task, its precise scenarios and their use in the derivation of schemas. Section 7 briefly discusses validation, underspecification, overspecification, nondeterminism and tool support. Our conclusions are stated in section 8.

2 Ordering Problem and State Space

There are four user tasks in our ordering problem: create a new order, invoice an order, cancel an order and refill the stock. The following statement introduces basic types *OrderId* and *Product* for the identification of individual orders and products. Once defined, we can use them in the specification without worrying about their implementation.

$$[\mathit{OrderId}, \mathit{Product}]$$

When an order is newly created, it will be in the state *pending*. After the order has left the warehouse, its state changes to *invoiced*. These are the only two order states that concern us regarding the scenarios to be discussed. The following definition could be modified by adding the new state *paid* to deal with payment scenarios in an iterative development process which is beyond the scope of this paper.

$$\mathit{OrderState} ::= \mathit{pending} \mid \mathit{invoiced}$$

We declare the state space with schema *OrderSystem* which has four variables, and after a horizontal dividing line, two invariants.

$$
\begin{array}{|l}
\hline
\textit{OrderSystem} \\
\textit{stock} : \mathrm{bag}\,\textit{Product} \\
\textit{orders} : \textit{OrderId} \pfun \mathrm{bag}\,\textit{Product} \\
\textit{orderStatus} : \textit{OrderId} \pfun \textit{OrderState} \\
\textit{freeIds} : \mathbb{P}\,\textit{OrderId} \\
\hline
\mathrm{dom}\,\textit{orders} = \mathrm{dom}\,\textit{orderStatus} \\
\mathrm{dom}\,\textit{orders} \bigcap \textit{freeIds} = \varnothing \\
\hline
\end{array}
$$

A bag of *Product* is equivalent to a partial function from *Product* to the set of positive natural numbers $\mathbb{N}_1$. We use the function to keep track of a product's quantity in stock or in an order. The symbols $\pfun$ and $\mathbb{P}$ represent partial function and power set respectively.

In an arbitrary state, we have 5 nuts and 6 bolts in stock. Order 1 was placed for 2 nuts and 2 bolts. Order 2 was placed for 3 bolts. Order 1 has been invoiced and order 2 is still pending. Ids 3 and 4 are free for future use. The state could be expressed with the following values in the variables of schema *OrderSystem*.

$$
\begin{array}{l}
\textit{stock} = \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 6\} \\
\textit{orders} = \{1 \mapsto \{\textit{nut} \mapsto 2, \textit{bolt} \mapsto 2\}, 2 \mapsto \{\textit{bolt} \mapsto 3\}\} \\
\textit{orderStatus} = \{1 \mapsto \textit{invoiced}, 2 \mapsto \textit{pending}\} \\
\textit{freeIds} = \{3, 4\}
\end{array}
$$

The keyword dom stands for domain. The first invariant ensures that an order id in use must appear in both *orders* and *orderStatus* for an order cannot exist without its status. The second invariant prevents an order id from being used and at the same time available for new orders. Often, we don't know all the constraints until we have explored the scenarios.

After an operation, we want to report whether it was successful. If not, what was the error?

$$
\begin{array}{lll}
\textit{Report} & ::= & \textit{OK} \mid \textit{no\_more\_ids} \mid \textit{order\_not\_pending} \\
 & \mid & \textit{id\_not\_found} \mid \textit{not\_enough\_stock}
\end{array}
$$

3 New Order

The scenario *NewOrder* documents a successful order creation using four Z concepts. They are input with the symbol ?, output with !, maplet with $\mapsto$, and set with { } and commas. The input parameter *order*? places a new order for 4 *nuts* and 4 *bolts*. Below the table headings are the pre-state followed by the post-state. In the post-state, the 3-dot symbol ... is used to denote the unchanged function *stock*. Functions *orders* and *orderStatus* are extended by a map for *OrderId* 3. The element 3 is removed from the set *freeIds*. The output parameters *id*! and *report*! return 3 and *OK* respectively.

scenario *NewOrder*

$order? = \{nut_a \mapsto 4_b, bolt_c \mapsto 4_d\}$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced\}$	$\{3_e, 4\}$
...	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3_e \mapsto \{nut_a \mapsto 4_b, bolt_c \mapsto 4_d\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3_e \mapsto pending\}$	$\{4\}$

$id! = 3_e, report! = OK$

Values referred in input/output parameters are subscripted allowing us to relate them to the state. When two pieces of data have the same subscript, for example, 3_e in the post-state of *orders* and *orderStatus*, they must be identical. If two pieces of data have identical value but different subscripts, for example 4_b and 4_d, their equality is merely a coincidence. Value 4_d could have been 5_d throughout the scenario. The values allowed in a scenario are confined by earlier definitions. For example, 4_b and 4_d must be taken from the set of positive natural numbers $\mathbb{N}_1$.

To generalise the above scenario to an operation schema, we need a type for the new order that maps *Product* to a positive integer.

$$Order == \{order : \mathrm{bag}\, Product \mid order \neq \varnothing\}$$

The scenario *NewOrder* can be converted to the equivalent Z schema below. The declaration part declares the variables, input/output parameters and their types. The symbol Δ alerts us that the state of *OrderSystem* is changed by this operation. The predicate part lists the constraints on the variables and parameters. The trailing symbol $'$ is used to denote a value after the operation.

NewOrderScenario

$\Delta OrderSystem$
$order? : Order$
$id! : OrderId$
$report! : Report$

$order? = \{nut_a \mapsto 4_b, bolt_c \mapsto 4_d\}$
$3_e \in freeIds$
$stock' = stock$
$orders' = orders \cup \{3_e \mapsto \{nut_a \mapsto 4_b, bolt_c \mapsto 4_d\}\}$
$orderStatus' = orderStatus \cup \{3_e \mapsto pending\}$
$freeIds' = freeIds \setminus \{3_e\}$
$id! = 3_e$
$report! = OK$

The first predicate specifies the value of the input parameter *order*?. The membership of 3_e in set *freeIds* gives rise to the second predicate. The third predicate states that *stock* is unchanged after the operation. The new maplets for 3_e, adding to *orders* and *orderStatus*, are captured in the fourth and fifth predicates. The removal of 3_e from the set *freeIds* is expressed next. The values for the output parameters are specified by the last two predicates.

Subscripted values, used to represent input/output parameters, are not fixed. For example, instead of picking 3_e in the pre-state, we could have picked 4_e. By replacing the subscripted values with the input/output variables they represent, for example, "3_e" with "$id!$" and "$\{nut_a \mapsto 4_b, bolt_c \mapsto 4_d\}$" with "$order?$", we have the generalised schema below.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NewOrderGeneralised}} \\
\Delta \mathit{OrderSystem} \\
\mathit{order}? : \mathit{Order} \\
\mathit{id}! : \mathit{OrderId} \\
\mathit{report}! : \mathit{Report} \\
\hline
\mathit{order}? = \mathit{order}? \\
\mathit{id}! \in \mathit{freeIds} \\
\mathit{stock}' = \mathit{stock} \\
\mathit{orders}' = \mathit{orders} \cup \{\mathit{id}! \mapsto \mathit{order}?\} \\
\mathit{orderStatus}' = \mathit{orderStatus} \cup \{\mathit{id}! \mapsto \mathit{pending}\} \\
\mathit{freeIds}' = \mathit{freeIds} \setminus \{\mathit{id}!\} \\
\mathit{id}! = \mathit{id}! \\
\mathit{report}! = \mathit{OK}
\end{array}$$

The generalised version of the Z schema can be simplified by removing the two identity predicates that always evaluate to *true*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{NewOrder}} \\
\Delta \mathit{OrderSystem} \\
\mathit{order}? : \mathit{Order} \\
\mathit{id}! : \mathit{OrderId} \\
\mathit{report}! : \mathit{Report} \\
\hline
\mathit{id}! \in \mathit{freeIds} \\
\mathit{stock}' = \mathit{stock} \\
\mathit{orders}' = \mathit{orders} \cup \{\mathit{id}! \mapsto \mathit{order}?\} \\
\mathit{orderStatus}' = \mathit{orderStatus} \cup \{\mathit{id}! \mapsto \mathit{pending}\} \\
\mathit{freeIds}' = \mathit{freeIds} \setminus \{\mathit{id}!\} \\
\mathit{report}! = \mathit{OK}
\end{array}$$

We now turn our attention to an unsuccessful attempt to create a new order. A separate post-state is not shown in the scenario because the state is unchanged. No subscripts are used because the input/output parameters do not relate to any piece of data in the state. The pre-condition is the absence of any *OrderId*s in *freeIds*.

scenario *NoMoreIdsError*
$order? = \{nut \mapsto 7\}$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3 \mapsto \{nut \mapsto 4, bolt \mapsto 4\},$ $4 \mapsto \{bolt \mapsto 8\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3 \mapsto pending,$ $4 \mapsto pending\}$	$\{\ \}$

$report! = no\_more\_ids$

The symbol Ξ indicates that the state *OrderSystem* is unchanged by the schema.

```
NoMoreIdsError
  $\Xi OrderSystem$
  $order? : Order$
  $report! : Report$
  ----
  $freeIds = \varnothing$
  $report! = no\_more\_ids$
```

The pre-condition of *NewOrder* is that there is some element in *freeIds*. Conversely, the pre-condition of *NoMoreIdsError* is that the set *freeIds* is empty. The disjunction of the two pre-conditions is *true*. Therefore *NewOrderOp* can handle all situations.

$$NewOrderOp == NewOrder \lor NoMoreIdsError$$

4 Invoice Order

The invoicing operation updates the order status from *pending* to *invoiced* and reduces the stock accordingly. In the next scenario, we use subscripts to help us express constraints on data values in the state. The two pre-conditions, shown before the table, require the state to have sufficient stock to fill the order. The two post-conditions, shown after the table, determine the updated stock quantities.

scenario *InvoiceOrder*
$id? = 2_a, 4_f \leq 5_c, 3_h \leq 9_e$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut_b \mapsto 5_c,$ $bolt_d \mapsto 9_e\}$	$\{1 \mapsto \{nut \mapsto 2\},$ $2_a \mapsto \{nut_b \mapsto 4_f, bolt_d \mapsto 3_h\}\}$	$\{1 \mapsto invoiced,$ $2_a \mapsto pending\}$	$\{3, 4\}$
$\{nut_b \mapsto 1_i,$ $bolt_d \mapsto 6_j\}$	...	$\{1 \mapsto invoiced,$ $2_a \mapsto invoiced\}$	...

$report! = OK, 1_i = 5_c - 4_f, 6_j = 9_e - 3_h$

The Z mathematical toolkit provides the sub-bag symbol $\sqsubseteq$ that concisely expresses multiple $\leq$ relationships between corresponding quantities in two bags.

Likewise, the bag difference symbol $\uminus$ expresses multiple pairwise subtractions. The updating of the order status to *invoiced* is expressed with the override symbol $\oplus$. Round brackets represent function application, for example, $orders(2_a)$ returns $\{nut_b \mapsto 4_f, bolt_d \mapsto 3_h\}$.

$$\begin{array}{|l}
\textit{InvoiceOrderScenario} \\
\Delta OrderSystem \\
id? : OrderId \\
report! : Report \\
\hline
id? = 2_a \\
orders(2_a) \sqsubseteq stock \\
orderStatus(2_a) = pending \\
stock' = stock \uminus orders(2_a) \\
orders' = orders \\
orderStatus' = orderStatus \oplus \{2_a \mapsto invoiced\} \\
freeIds' = freeIds \\
report! = OK \\
\hline
\end{array}$$

After substitution and simplification, we have the following Z schema.

$$\begin{array}{|l}
\textit{InvoiceOrder} \\
\Delta OrderSystem \\
id? : OrderId \\
report! : Report \\
\hline
orders(id?) \sqsubseteq stock \\
orderStatus(id?) = pending \\
stock' = stock \uminus orders(id?) \\
orders' = orders \\
orderStatus' = orderStatus \oplus \{id? \mapsto invoiced\} \\
freeIds' = freeIds \\
report! = OK \\
\hline
\end{array}$$

There are three unsuccessful scenarios for this operation. We will skim the intermediate steps and explanations of the remaining derivations due to their similarities to the previous ones.

scenario *IdNotFoundError*
$id? = 3_a, 3_a \neq 1_b, 3_a \neq 2_c$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 9\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1_b \mapsto invoiced,$ $2_c \mapsto pending\}$	$\{3, 4\}$

$report! = id\_not\_found$

IdNotFoundErrorScenario

$\Xi State$
$id? : OrderId$
$report! : Report$

$id? = 3_a$
$3_a \notin \{1_b, 2_c\}$
$report! = id\_not\_found$

IdNotFoundError

$\Xi State$
$id? : OrderId$
$report! : Report$

$id? \notin \mathrm{dom}\ orderStatus$
$report! = id\_not\_found$

scenario *OrderNotPendingError*
$id? = 1_a, invoiced_b \neq pending$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt \mapsto 9\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1_a \mapsto invoiced_b,$ $2 \mapsto pending\}$	$\{3, 4\}$

$report! = order\_not\_pending$

OrderNotPendingError

$\Xi State$
$id? : OrderId$
$report! : Report$

$orderStatus(id?) \neq pending$
$report! = order\_not\_pending$

scenario *NotEnoughStockError*
$id? = 2_a, 9_b < 77_c$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut \mapsto 5,$ $bolt_d \mapsto 9_b\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2_a \mapsto \{bolt_d \mapsto 77_c\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$

$report! = not\_enough\_stock$

NotEnoughStockError

$\Xi State$
$id? : OrderId$
$report! : Report$

$\neg(orders(id?) \sqsubseteq stock)$
$report! = not\_enough\_stock$

We define *InvoiceOrderOp* to deal with all situations. When multiple errors happen at the same time, the definition is unspecific about which error report to return. We will discuss nondeterminism in subsection 7.4.

$$\begin{aligned} \mathit{InvoiceOrderOp} == {} & \mathit{InvoiceOrder} \lor \mathit{IdNotFoundError} \lor \\ & \mathit{OrderNotPendingError} \lor \mathit{NotEnoughStockError} \end{aligned}$$

5 Cancel Order

A *pending* order may be cancelled. Its order id is returned to the pool of free id's for future use.

scenario *CancelOrder*
$id? = 2_a$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5, bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\}, 2_a \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced, 2_a \mapsto pending_b\}$	$\{3, 4\}$
...	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\}\}$	$\{1 \mapsto invoiced\}$	$\{2_a, 3, 4\}$

$report! = OK$

The domain restriction symbol $\ntriangleleft$ is used to remove maplets for order id 2_a from *orders* and *orderStatus*.

CancelOrderScenario
$\Delta OrderSystem$
$id? : OrderId$
$report! : Report$

$id? = 2_a$
$orderStatus(2_a) = pending$
$stock' = stock$
$orders' = \{2_a\} \ntriangleleft orders$
$orderStatus' = \{2_a\} \ntriangleleft orderStatus$
$freeIds' = \{2_a\} \cup freeIds$
$report! = OK$

We generalise the scenario schema by replacing 2_a with *id?*. After simplification, we have schema *CancelOrder*.

$$\begin{array}{|l}
\textit{CancelOrder} \\
\Delta \textit{OrderSystem} \\
id? : \textit{OrderId} \\
report! : \textit{Report} \\
\hline
\textit{orderStatus}(id?) = \textit{pending} \\
stock' = stock \\
orders' = \{id?\} \lhd orders \\
orderStatus' = \{id?\} \lhd orderStatus \\
freeIds' = \{id?\} \cup freeIds \\
report! = OK \\
\hline
\end{array}$$

It is an error trying to cancel an order that does not exist or have already been *invoiced*. We can reuse error detecting schemas to handle all situations.

$$\begin{aligned}\textit{CancelOrderOp} == {} & \textit{CancelOrder} \lor \textit{IdNotFoundError} \lor \\ & \textit{OrderNotPendingError}\end{aligned}$$

6 Enter Stock

Entering stock is the task of replenishing depleted stock. By assuming that there is always sufficient storage space, we don't worry about detecting an error for this task. The postconditions concerning the updated stock quantities are expressed with the bag addition symbol $\uplus$.

scenario *EnterStock*

$newStock? = \{nut_a \mapsto 80_b, bolt_c \mapsto 70_d\}$

stock	*orders*	*orderStatus*	*freeIds*
$\{nut_a \mapsto 5_e,$ $bolt_c \mapsto 9_f\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto pending\}$	$\{3, 4\}$
$\{nut_a \mapsto 85_g,$ $bolt_c \mapsto 79_h\}$	...	...	...

$report! = OK, 85_g = 5_e + 80_b, 79_h = 9_f + 70_d$

$$\begin{array}{|l}
\textit{EnterStockScenario} \\
\Delta \textit{OrderSystem} \\
newStock? : \textit{Order} \\
report! : \textit{Report} \\
\hline
newStock? = \{nut_a \mapsto 80_b, bolt_c \mapsto 70_d\} \\
stock' = stock \uplus \{nut_a \mapsto 80_b, bolt_c \mapsto 70_d\} \\
orders' = orders \\
orderStatus' = orderStatus \\
freeIds' = freeIds \\
report! = OK \\
\hline
\end{array}$$

$$
\begin{array}{|l}
\hline
\textit{EnterStock} \\
\Delta \textit{OrderSystem} \\
\textit{newStock}? : \textit{Order} \\
\textit{report}! : \textit{Report} \\
\hline
\textit{stock}' = \textit{stock} \uplus \textit{newStock}? \\
\textit{orders}' = \textit{orders} \\
\textit{orderStatus}' = \textit{orderStatus} \\
\textit{freeIds}' = \textit{freeIds} \\
\textit{report}! = \textit{OK} \\
\hline
\end{array}
$$

7 Analysis

7.1 Validation

We can apply the values of input parameters and pre-states in a scenario to its operation schema. If the post-state and output parameters obtained are the same as in the original scenario, we know that the operation schema works correctly for the scenario.

Another type of validation we can perform is to apply new input parameters and pre-state values to an operation schema. This exercise in essence creates new scenarios. If the customer is satisfied with the newly created post-states and output parameters, we gain confidence that our Z schemas meet the user requirements. For instance, we can validate the *InvoiceOrder* schema with the following input parameter and pre-state different from our scenarios.

$$
\begin{aligned}
&\textit{id}? = 3 \\
&\textit{stock} = \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 9\} \\
&\textit{orders} = \{1 \mapsto \{\textit{nut} \mapsto 2\}, \\
&\qquad\qquad 2 \mapsto \{\textit{nut} \mapsto 4, \textit{bolt} \mapsto 3\}, \\
&\qquad\qquad 3 \mapsto \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 6\}\} \\
&\textit{orderStatus} = \{1 \mapsto \textit{invoiced}, 2 \mapsto \textit{invoiced}, 3 \mapsto \textit{pending}\} \\
&\textit{freeIds} = \{4\}
\end{aligned}
$$

The first two predicates in the *InvoiceOrder* schema specify the preconditions of the schema. They both evaluate to *true*. The next four predicates specify the resulting values of the four tables. The last predicate specifies the output parameter. To save space, we only show the evaluation of two predicates below.

$$
\begin{aligned}
&\textit{orders}(\textit{id}?) \sqsubseteq \textit{stock} \\
&\textit{orders}(3) \sqsubseteq \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 9\} \\
&\{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 6\} \sqsubseteq \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 9\} \\
&\textit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\textit{stock}' = \textit{stock} \uminus \textit{orders}(\textit{id}?) \\
&\textit{stock}' = \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 9\} \uminus \{\textit{nut} \mapsto 5, \textit{bolt} \mapsto 6\} \\
&\textit{stock}' = \{\textit{bolt} \mapsto 3\}
\end{aligned}
$$

After evaluating the schema predicates, we have the following new scenario. Though without subscripts, pre- and post-conditions, there is enough information for the customer to decide that the new scenario matches the expectation.

scenario *InvoiceOrderNew*
$id? = 3$

$stock$	$orders$	$orderStatus$	ids
$\{nut \mapsto 5,$ $bolt \mapsto 9\}$	$\{1 \mapsto \{nut \mapsto 2\},$ $2 \mapsto \{nut \mapsto 4, bolt \mapsto 3\}$ $3 \mapsto \{bolt \mapsto 5, bolt \mapsto 6\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3 \mapsto pending\}$	$\{4\}$
$\{bolt \mapsto 3\}$	$\ldots$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3 \mapsto invoiced\}$	$\ldots$

$report! = OK$

7.2 Underspecification

Underspecification happens when a required condition is missing. Suppose we had omitted the following precondition of checking for sufficient stock in schema *InvoiceOrder*.

$$orders(id?) \sqsubseteq stock$$

If we validate the schema with excessive ordered quantities, the operation will still succeed. The customer can easily spot that the new scenario should have caused an error.

7.3 Overspecification

Overspecification happens when unnecessary conditions are included in a scenario. Recall scenario *NewOrder*. The quantities of the nuts and bolts in the input parameter were both 4 by coincidence. In our original scenario, they have different subscripts b and d to indicate that they need not be the same. Suppose we had made a mistake by using the same subscript b on both occurrences of 4. We would have a slightly different scenario.

scenario *OverSpecifiedNewOrder*
$order? = \{nut_a \mapsto 4_b, bolt_c \mapsto 4_b\}$

$stock$	$orders$	$orderStatus$	$freeIds$
$\{nut \mapsto 5,$ $bolt \mapsto 6\}$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced\}$	$\{3_e, 4\}$
$\ldots$	$\{1 \mapsto \{nut \mapsto 2, bolt \mapsto 2\},$ $2 \mapsto \{bolt \mapsto 3\},$ $3_e \mapsto \{nut_a \mapsto 4_b, bolt_c \mapsto 4_b\}\}$	$\{1 \mapsto invoiced,$ $2 \mapsto invoiced,$ $3_e \mapsto pending\}$	$\{4\}$

$id! = 3_e, report! = OK$

The equality implied by the identical subscripts gives rise to the first predicate in the following schema. When we validate the schema with $order? = \{nut \mapsto 7, bolt \mapsto 9\}$, the new predicate would evaluate to *false*. Overspecification is caught when the operation fails on legitimate input and pre-state.

$$
\begin{array}{|l}
\textit{OverSpecifiedNewOrder} \\
\hline
\Delta \textit{OrderSystem} \\
order? : Order \\
id! : OrderId \\
report! : Report \\
\hline
\forall p, q : Product \mid p \in \operatorname{dom} order? \land q \in \operatorname{dom} order? \bullet \\
\qquad\qquad\qquad order?(p) = order?(q) \\
id! \in freeIds \\
stock' = stock \\
orders' = orders \cup \{id! \mapsto order?\} \\
orderStatus' = orderStatus \cup \{id! \mapsto pending\} \\
freeIds' = freeIds \setminus \{id!\} \\
report! = OK \\
\hline
\end{array}
$$

7.4 Nondeterminism

Recall that we have defined *InvoiceOrderOp* to catch errors.

$$
\begin{aligned}
InvoiceOrderOp == {} & InvoiceOrder \lor IdNotFoundError \lor \\
& OrderNotPendingError \lor NotEnoughStockError
\end{aligned}
$$

What if the preconditions of *OrderNotPendingError* and *NotEnoughStockError* are *true* at the same time? Which error report are we going to get? In this nondeterministic definition of *InvoiceOrderOp*, we could get either one. If it is necessary to distinguish the two errors, we can strengthen their preconditions so that their conjunction will always be *false*. This is the general strategy we could use to eliminate nondeterminism.

7.5 Tool Support

The success of a methodology relies heavily on the availability of good tools. There are many Z tools around, for example, CADiZ, CZT(Community Z Tools), Z/EVES, and ZETA. They can be used to check for type conisistency and prove theorems about a specification. Some allow you to validate operation schemas against scenarios as we just did [11]. But to use precise scenarios in a practical setting, we need native support to keep track of subscripts, validate scenarios against schemas and maintain traceability of schemas to their sourcing scenarios.

8 Conclusion

The Z specification we derived in this paper is almost identical to the one found in [10]. One notable difference is that our specification catches more errors. It is premature to conclude how the use of precise scenarios would shape a Z specification. However we do not seem to have lost any capability to create a generic Z specification.

What evidence do we have in support of our claim that precise scenarios bridge the gap between customers and formal specifications? Our precise scenario descriptions use simple Z concepts, namely input, output, maplet, set enumeration, pre-state and post-state. On the other hand, schemas use additional Z concepts like set membership, set union, set difference, sub-bag, bag difference, override, domain restriction, and bag addition. Due to the smaller number of symbols used, the customers would have an easier time learning to read precise scenarios. Our experience on other problems taken from the areas of numerical computation, sorting and telephone system has been similar in that precise scenarios are representable concisely with significantly less number of concepts than schemas. Biologists and educationists also suggest that understanding begins with concrete examples [12]. We expect the majority of customers to prefer dealing with concrete examples rather than abstract descriptions.

The use of precise scenarios for the development and validation of schemas relates to a software testing technique called *equivalence partitioning* [13]. The technique ensures that a scenario is selected from every class of similar situations. For example, the operation *InvoiceOrder* can be partitioned into at least two classes of situations, one for ordered quantities being a sub-bag of stock quantities and one for otherwise. If necessary, we can divide them into more partitions. For example, we can add a class of situations that the ordered quantities are identical to the stock qunatities. Why is testing necessary in the application of formal methods? Refinement can produce an implementation guaranteed to be correct according to a formal specification. Therefore the research community has long held the attitude that formal methods reduce or eliminate the need of testing. But how do we know the formal specification is complete in the first place? Even for toy problems, it is hard to be certain that a formal specification is complete. Though the use of precise scenarios cannot guarantee completeness, the improved customer involvement can only help.

Formal methods have been focusing on the initial creation of an abstract specification and its gradual refinement to concrete implementation. Our proposal challenges the validity of this conventional wisdom. We suggest an approach to precede the current formal approaches by concrete examples used as a basis for the abstract specification. The precise scenarios has a dual purpose; serving as a solid foundation for a formal specification and a bridge between the customers to the formal specification.

The current paper shows only scenarios with pre- and post-states. Work in progress includes longer scenarios with intermediate states. We have started working on alternative representations for the precise scenarios to facilitate the derivation of specifications in languages other than Z. Eventually, we would like to field-test the approach when suitable tool support becomes available.

References

1. The Standish Group: The CHAOS Report (1994) 5
2. Zimmerman, M.K., Lundqvist, K., Leveson, N.: Investigating the Readability of State-Based Formal Requirements Specification Languages. ICSE'02: 24th International Conference on Software Engineering, May (2002) 33-43
3. Glass, R.L.: The Mystery of Formal Methods Disuse. Communications of the ACM, Vol. 47, No. 8 (2004) 15-17
4. Amyot, D., Logrippo, L., Buhr, R.J.A., Gray, T.: Use Case Maps for the Capture and Validation of Distributed Systems Requirements. RE'99: 4th IEEE International Symposium on Requirements Engineering, June (1999) 44-54
5. Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. ICSE'00: 22nd International Conference on Software Engineering, June (2000) 314-323
6. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Transactions on Software Engineering, vol. 29, no. 2 (2003) 99-115
7. Damas, C., Lambeau, B., Dupont, P., van Lamsweerde, A.: Generating Annotated Behaviour Models from End-User Scenarios. IEEE Transactions on Software Engineering, vol. 31, no. 12 (2005) 1056-1073
8. Grieskamp, W., Lepper, M.: Using Use Cases in Executable Z. Liu, S., McDermid, J.A., Hinchey, M.G. (ed.) ICFEM 2000: 3rd IEEE International Conference on Formal Engineering Methods, September (2000) 111-119
9. B*ü*ssow, R., Grieskamp, W.: A Modular Framework for the Integration of Heterogenous Notations and Tools. Araki, K., Galloway, A., Taguchi, K. (ed.) IFM99: 1st International Conference on Integrated Formal Methods, June (1999) 211-230
10. Bowen, J.P.: Chapter 1 - Z. Habrias, H., Frappier M. (ed.): Software Specification Methods, ISTE (2006) 3-20
11. Saaltink, M.: The Z/EVES 2.0 User's Guide. TR-99-5493-06a, ORA Canada (1999) 31-32
12. Zull, J.E.: The Art of Changing the Brain. Stylus Publishing (2002) 102-103
13. Sommerville, I.: Software Engineering, 6th Edition. Addison Wesley (2001) 444-447