Specifying Complex Systems in Object-Z: A Case Study of Petrol Supply Systems

Yangping Li[a*], Xiaoheng Pan[a], Tianming Hu[a], Sam Yuan Sung[b], Huaqiang Yuan[a]
[a]Dongguan University of Technology, China
[b]South Texas College, USA
*Contact author. Email: yvonne-lee@qq.com

***Abstract*— As modern complex systems become increasingly large, sophisticated, feature-rich and data-intensive, people have recognized the importance of precisely and unambiguously specifying them with formal methods for a number of years. This paper advocates the use of Object-Z, a formal specification language, in the description of complex systems. Object-Z is an extension to the Z language to facilitate specification in an object-oriented style. The notation Object-Z builds on Z's strengths in modeling complex data and algorithms, and on its new class structuring's strengths in succinctly specifying the various relationships and communication between objects in a large system. In detail, first we describe informally the syntax and semantics of Object-Z, highlighting those features that facilitate decomposing a large system into a collection of interacting objects and thus separating concerns. Then, we demonstrate the use of Object-Z by presenting a case study of a petrol supply system, illustrating how the system runs by communicating the constituent objects. Finally, we discuss several issues we encountered in this exercise, which may serve as feedback to the development of Object-Z.**

***Index Terms*— Object-Z; object-oriented modeling; formal methods; system specification**

I. Introduction

With the rapid development of information technology, modern complex systems are getting increasingly large, sophisticated, feature-rich and data-intensive. Ranging from gigantic ones like space shuttles to everyday services like Automatic Teller Machines, these systems are regularly required to accomplish more, faster and on a broader scale, to adapt dynamically to changing workloads, scenarios and objectives, and to achieve guaranteed levels of performance and dependability. Satisfying such demanding requirements in the presence of the variability and heterogeneity that characterize complex systems poses numerous challenges to both their developers and their users.

Among the various technologies that support complex systems, formal methods have been advocated as a principled approach due to its power of precise and unambiguous specification of key features and standards [1], [2]. In particular, Z is a model-oriented specification language with powerful features for describing complex data structures and their operations [3], [4]. Object-Z extends the Z language to facilitate specification in an object-oriented style [5]. In such a style, the system comprises a collection of underlying objects, each of which has predefined structure and behavior. The system runs by communicating the objects. Hence object-oriented specification languages clarify the large specification and facilitate separation of concerns.

This paper presents a case study of specifying a petrol supply system in Object-Z. Such a system involves objects like the petrol company, petrol stations, pumps, customers, etc. Typical behaviors include that the company offers petrol to its stations and the customer fuels his vehicle at a pump in a station. Such complex object structures and their operations naturally lend themselves to the specification of Object-Z, which is good at decomposing large systems.

The rest of the paper is organized as follows. Section II reviews formal specification languages and Object-Z's applications. Section III gives an overview of Object-Z. In particular, the major constructs used in the case study are introduced, including class constructs, secondary attributes, object containment and several operation expressions. Section IV provides a detailed specification of the petrol supply system, illustrating how the above items can be employed to capture the structural relationship between objects and to specify communication between objects. Finally, Section V concludes this paper with some remarks.

II. Related Work

A. Formal Specification Languages

In general, formal specification languages can be divided into the following categories based on the particular specification paradigm they rely on.

History-based specification languages specify a system by characterizing the admissible histories over time. The properties of interest are specified by temporal logic assertions about system objects in past, current and future states. Time structures can be discrete [6] or continuous [7].

State-based specification languages characterize the admissible system states at some arbitrary snapshot. The properties of interest are specified by invariants constraining the system objects at any snapshot, as well as pre-/post-assertions constraining the application of system

Manuscript received August 10, 2011; revised January 2, 2012; accepted April 16, 2012.

doi:10.4304/jsw.9.7.1707-1717

operations at any snapshot. Languages such as Z [8], VDM [9] or B [10] rely on this paradigm. Object-oriented variants, such as Object-Z, have been proposed as well.

Transition-based specification languages characterize the required transitions from state to state. The properties of interest are specified by a set of transition functions in the state machine transition. For each input state and triggering event (unlike the precondition which is necessary, triggering event is sufficient), the transition function gives the corresponding output state. Languages such as Statecharts [11] or SCR [12] rely on this paradigm.

Functional specification languages specify a system as a structured collection of mathematical functions and they come in two classes. In algebraic specification, the functions are grouped by object types that appear in their domain. The properties of interest are then specified as conditional equations that capture the effect of composing functions. Languages such as PLUSS [13] rely on this paradigm. In higher-order specification, the functions are grouped into logical theories. Languages such as PVS [14] rely on this paradigm.

Operational specification languages characterize a system as a structured collection of processes that can be executed by some abstract machine. Early languages such as Petri nets or process algebras [15], [16] rely on this paradigm.

B. Object-Z's Applications

As long ago as the early days of Z and Object-Z, both of them have been applied to a wide range of applications, such as programming languages, communication protocols, and mobile phone systems [17], [18], [19]. Their recent applications include specifying oil and gas seismic survey, multi-agent systems, wireless network routing protocol, train control systems, control architectures for remotely operated vehicles, and software development [20], [21], [22], [23], [24], [25], [26]. In the following, we review some of latest representative applications, illustrating how they help make the system reliable.

In [21], Object-Z was used to specify asynchronous multi-agent systems (AMAS). Since such systems consist of multiple autonomous agents with asynchronous updates and communications, they are often designed from the point of view of local computations and the interactions of autonomous agents. However, some of the system functionality can only be proposed from the global point of view. To guarantee the system's functionality by the local behavior of the agents, the authors employed Object-Z with bounded fairness constraints for the specification framework.

In [22], Object-Z was used to specify and reason about Ad hoc On-Demand Distance Vector (AODV) routing protocol in wireless networks. To formalize the route discovery process, the network is decomposed into a collection of nodes and the network topology is defined by relations between the nodes. Thus the broadcast communication can be modeled by operations which change local variables of the sender and receivers. In this way, the authors proved the loop freedom property for the established routes based on the specification.

class name
visibility list
inherited classes
local types
state
initial state
operations

Figure 1. The class construct of Object-Z.

Although formal specifications help develop reliable software, lack of enough knowledge and high cost often force developers to mainly use semi-formal (visual) methods in practical large-scale software development. In [26], the authors proposed to transform formal Object-Z and semi-formal UML into each other using a set of bidirectional formal rules. UML facilitates the interactions among stakeholders and makes the software development flexible. Since Object-Z are able to detect the inconsistencies of software, as demonstrated in the multi-lift case study, the transformation of formal and visual models into each other through the iterative and evolutionary process helps develop the highly reliable yet flexible software applications.

III. OBJECT-Z

In this section, we brief the major constructs of Object-Z. We assume the reader has basic knowledge of Z.

A. Class and Object

In the strongly typed object Z, every object is of a type, called "class". The class construct encapsulates all relevant features and acts as a template from which objects can be produced. The class construct is defined in Figure 1, where

- *class name* is the name of the class.
- *visibility list* specifies the interface of the objects of the class and their environment. When it is omitted, it means all features are visible.
- *inherited classes* denote those classes whose features are inherited in this class.
- *local types* have the syntax of Z global type.
- *state* is an anonymous (with the same name as the class) constant state schema that must be satisfied throughout the life time of the object. Constants and variables declared in state schema are attributes of the class.
- *initial state* specifies the conditions to be met by the object when it is born.
- *operations* denote those operations through which object can change its state. They have the syntax of Z operation schema.

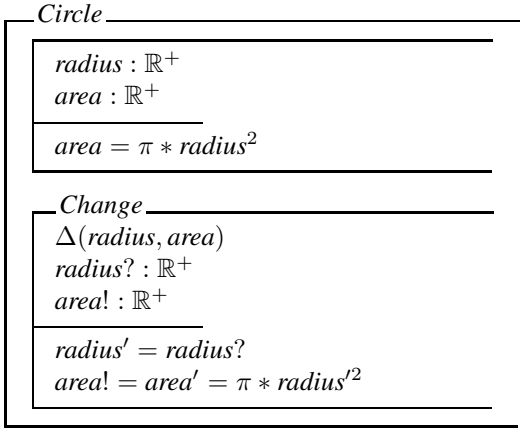


Figure 2. The *Circle* class.

Attributes, initial states and operations are the key features of the class. At any particular time, any object of the class has a state, i.e., an assignment of some values to the attributes. This state must conform to the constant state schema in the class definition. The state transition of the object can only occur through those specified operations.

An exemplar class construct of *Circle* is shown in Figure 2, which has two attributes *radius* and *area*, and one operation *Change*. The constant state schema, $area = \pi * radius^2$, is maintained throughout the life time of any object of class *Circle*. With operation *Change*, class *Circle* changes its attributes and communicates with the environment. As to the first goal, all attributes to be changed, namely, *radius* and *area*, are explicitly included in the operation's Δ list. We use the unprimed identifier (e.g. *radius*) to denote the attribute value before the operation and the primed identifier (e.g. $radius'$) to denote the attribute value after the operation. As to the second goal, we use the decoration "base name+?" (e.g. *radius*?) to denote the input from the environment, which will be assigned to the class attribute with the same base name (e.g. $radius' \leftarrow radius?$). Similarly, we use the decoration "base name+!" (e.g. *radius*!) to denote the output to the environment, which will receive a value from the class attribute with the same base name after the operation (e.g. $radius! \leftarrow radius'$).

Finally, it is worth noting that $a : A$ only declares a variable reference a to some object of class A. a's value is the identity of that object. If that object has changed, e.g., its attribute's value has changed, a does not change. a changes only when it refers to another object. This is particularly significant for modeling aggregates. For instance, $rooms : \mathbb{P}\, Room$ declares that *rooms* is a set of references to objects of class *Room* ($\mathbb{P}$ denotes power set). *rooms* will not change when its member room changes. *room* will change only when it refers to a different set of rooms.

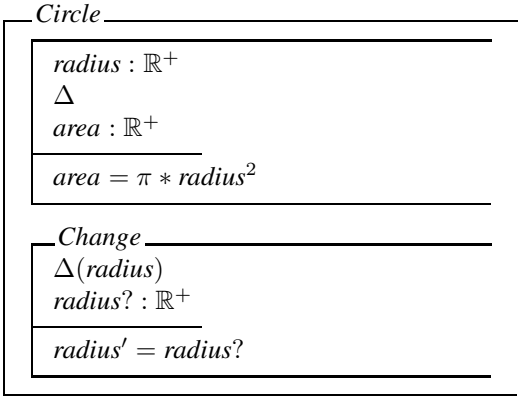


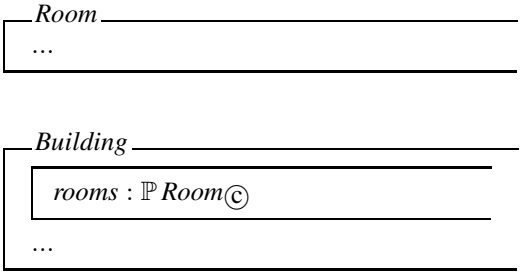
Figure 3. The *Circle* class with *area* as secondary attribute.

B. Secondary Attributes

There may be some dependency among the class attributes [27]. For example, as shown in Figure 2, class *Circle* has two attributes *radius* and *area*, for which $area = \pi * radius^2$ is always maintained. In this case, *area* is a secondary attribute that depends on *radius*. Attributes like *area* add no further information to the specification in that it is derivable from the primary attributes like *radius*. In principle, class attributes/state variables can only change via those operations whose Δ list explicitly includes them. However, to improve readability and convenience, secondary attributes are implicitly included in every Δ list, denoting they will be changed in such operation schema. As illustrated in Figure 3, *area* is introduced by a Δ declarator, suggestive of implicit inclusion in every Δ list. When the input *radius*? is available in the environment, the attribute $radius'$ is equated to it, and implicitly $area'$ is updated according to $area' = \pi * radius'^2$ as well.

C. Object Containment

Sometimes we want to emphasize that an object can only be contained by another object. For example, on campus, one room can only be in one building and cannot be shared by two or more buildings. We can denote this notion by



This means that *rooms*, a set of references to objects of class *Room*, is directly contained in the object of class *Building*. In general, two properties are implied in the use of symbol © to denote object containment [28].

- Any object cannot be directly contained by two distinct objects.
- Any object cannot directly or indirectly contain itself.

Formally, let $\mathbb{O}$ denote the class of all object identities, then we can define the relation of directly-contain, *dcon*, as

$$dcon : \mathbb{O} \leftrightarrow \mathbb{O}$$

$$dcon \in \mathbb{O} \nrightarrow \mathbb{O}$$

$$\forall o : \mathbb{O} \bullet \neg\, o \, \underline{dcon}^{+} \, o$$

The first constraint says that *dcon* is a partial function in that for any object a in $\mathbb{O}$, we can have at most one image $dcon(a)$ that directly contains a. The second constraint says that for any object o in $\mathbb{O}$, $(o, o) \notin dcon^{+}$, where $dcon^{+}$ is the transitive closure of *dcon*.

D. Operation Expressions

In the specification for the petrol supply system, four binary operators are frequently used to construct new operations from old ones. They are

- Conjuction operator $\wedge$: It conjoins constraints and equates variables with the same name.
- Parallel operator $\|$: Like conjuction, it is communicative and supports communication in both directions. It conjoins constraints and equates input and output variables with the same base name. The inputs with matching output in the other operation are hidden. However, to make this operator associative, those matching outputs are not hidden. As in conjuction, outputs with the same name are equated, so are those residual inputs with the same name.
- Sequential operator $\fatsemi$: It is one-way operator, i.e., communication is from left to right. The outputs from the left operand are equated with the inputs with the same base names from the right operand and both are hidden. Residual inputs with the same base names are hidden, so are those residual outputs with the same base names. Hence, sequential operator is non-communicative and non-associative.
- Choice operator $\Box$: It indicates a non-deterministic choice of operation. If neither precondition of the operand is met, the operation fails. If only one operand's precondition is met, the choice is deterministic. If both operands' preconditions are met, the environment selects an operand to proceed. There is no communication between the two operands, because at any time at most one operand is performed.

IV. Specifying Petrol Supply Systems

A. The System Overview

This section demonstrates the use of Object-Z to the description of a petrol supply system. The class constructs, object containment and operation operators are very useful in specifying the relations among different objects in such a system.

Figure 4. A typical petrol station.

As illustrated in Figure 4, a petrol supply system can be informally described as follows. A petrol company owns a number of petrol stations. Each petrol station consists of a number of petrol pumps. Each pump can be reset by a supervisor, after which it can pump out petrol from a common store, recording at each stage the volume and cost of petrol pumped so far. When pumping is finished, the total cost is recorded by the supervisor and subsequently paid by the customer. Each pump in a petrol station sells petrol at the same price, although this price may vary between stations. The common store can be restocked by the petrol company at any time. The company supplies petrol at the same cost to each station. Records are kept of the total volume of petrol supplied to each station, as well as the outstanding amount owed by each station. As the city becomes increasingly crowded, more and more stations are established in densely populated areas like business and residential districts. Hence the station is discouraged from maintaining a high outstanding amount. The amount owed to the company by the station is paid when requested by the company.

Some additional requirements are

- New petrol pumps can be added to a station.
- New stations can be added to the system.
- The wholesale price of petrol can change.
- A station can change its retail price of petrol.

After preliminary examination, it is intuitive to extract the following underlying objects, which communicate with one another as in Figure 5.

- *Customer*: At a pump in a station, he asks for a certain volume of petrol. While stock lasts, he knows the cost after pumping.
- *Supervisor*: He resets the pump and perhaps monitors pumping. He is not included in our system, because we assume there is one supervisor in each station and his role can be replaced by the station.
- *Pump*: It needs at least two attributes, *volume* and *cost* about the pumped petrol. If pumping fails, e.g., the customer's demand is greater than the stock in the common store, the pump is reset to zero, informing the customer of the failure.
- *Station*: Each station contains some pumps that are directly contained only by that station. It can change its retail price and must pay the outstanding amount owed to the company when requested. New pumps can be added to it.

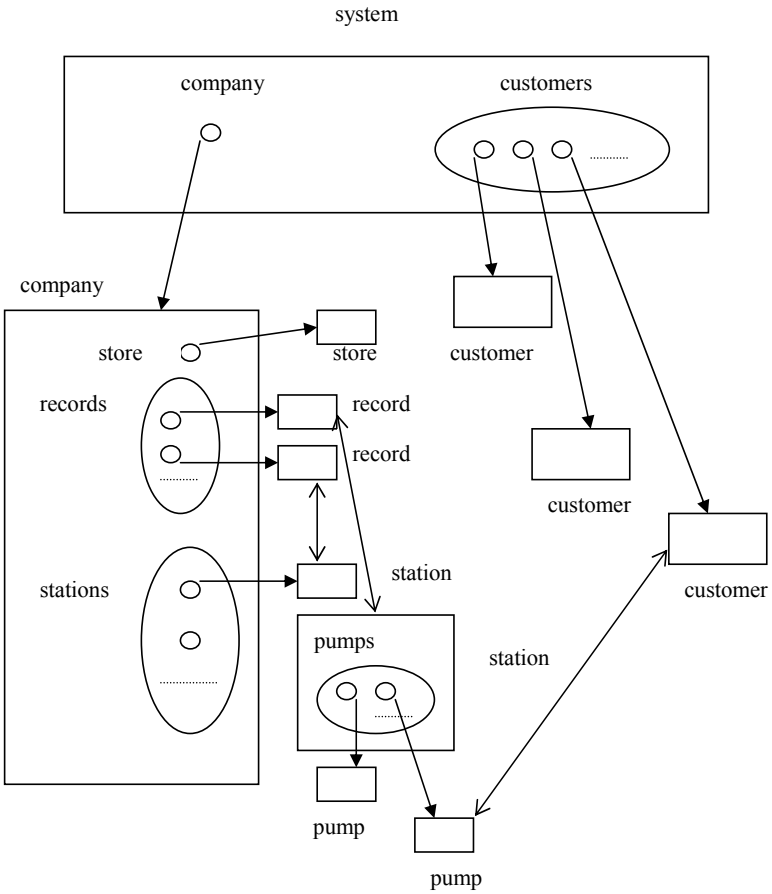


Figure 5. The overview of a petrol supply system.

- *Company*: The company directly contains some stations which cannot be shared. It also contains a store that supplies petrol to its stations. It keeps records, defined below, for each station. It can change the whole sale price and ask any station for the amount owned. New stations can be added to it.
- *Record*: There is exactly one record object for each station. It has three attributes, the station identity, the total volume provided to that station so far, and the outstanding amount owed by that station.

B. Pump

Class *Pump* needs at least two attributes: *vol*, recording the pumped volume, and *cost*, recording the cost of the pumped petrol. It also needs an operation *Pumping*, which takes the input *vol*? from the customer, and outputs the cost *cost*! equated with *cost'* obtained from the constraint

$$cost = rprice * vol$$

where *rprice* denotes the retail price determined by the station. Because all pumps in the station share the same value of the retail price, to avoid storing one price copy in each *Pump* object, we can specify *cost* as a secondary attribute depending on *vol* and store only one price copy in class *Station*. This gives classes *Pump* and *Station* in Figure 6.

At first glance, this concise specification works fine; each time a customer asks for pumping, the pump will provide the petrol and output the cost according to the constraint defined in class *Station*. However, there is a pitfall that would cause contradictions. On one hand, if the station changes the price via operation *ChangRprice*, *cost* will also change instantly according to the new price in the constraint, even without any new pumping. This is inconsistent with our definition that *cost* stores the cost of the last pumping. On the other hand, since *cost* is defined as an attribute of class *Pump* recording the cost of the last pumped petrol, it will only change via operation *Pumping* of class *Pump*. Besides, such requirement is more object-oriented in that attributes can only change via the operations in the same class. After further investigation, we find that such a contradiction is caused by the improper definition of *cost* as secondary attribute. As introduced previously, a secondary attribute, derivable from the primary attributes, adds no further information to the specification. In our case, however, when the price is changed by the station, *cost* becomes primary in that it stores the last pumping cost which is no longer derivable from the current values of *rprice* and *vol*.

Therefore, we redefine *cost* as primary attribute and add an attribute *rprice* for each pump, as shown in Figure 7.

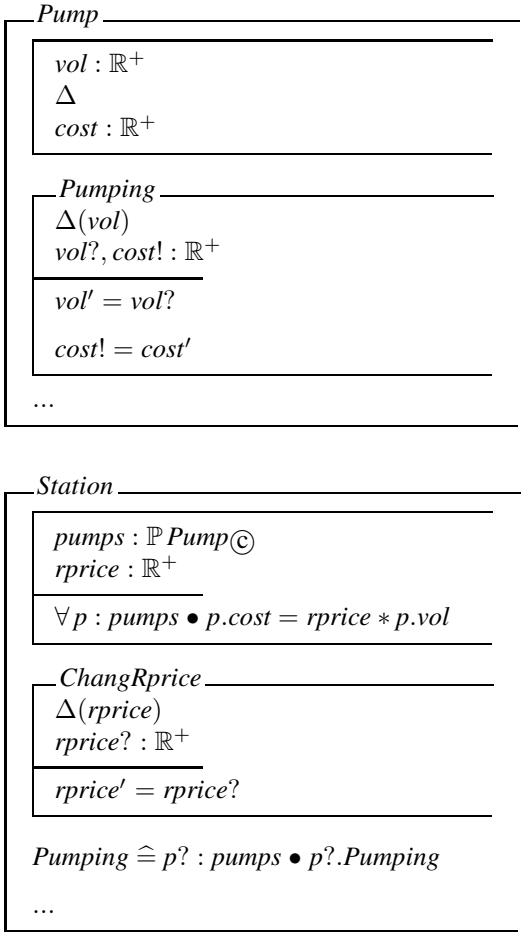


Figure 6. The *Pump* and *Station* classes with *cost* as secondary attribute.

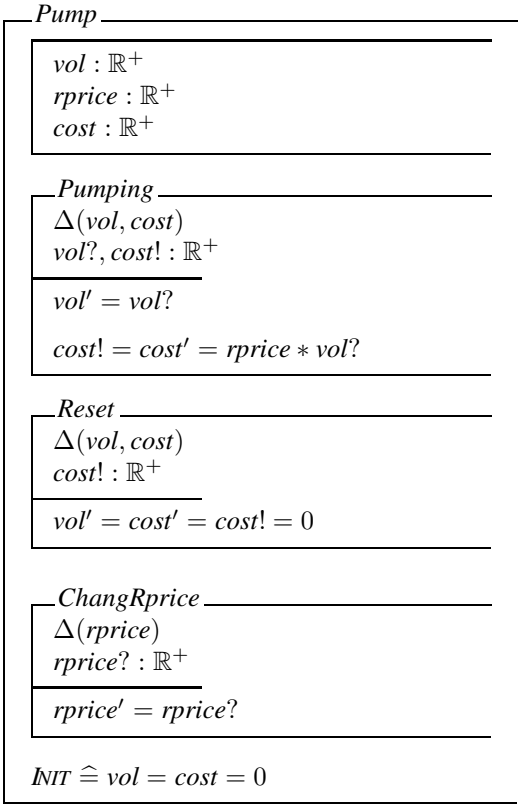


Figure 7. The *Pump* class.

When the stock in the store meets the demand of the customer (specified in class *Company*), *Pumping* will be performed, outputting the cost based on the current price. Otherwise, operation *Reset* is performed, setting both *vol* and *cost* to zero, indicating such failure. Therefore, at any time instant, *vol* and *cost* denote the last successful or failed transaction. If they are positive, they denote actual volume and cost of the last pumping. If they are both zero, it means the last customer's demand is greater than the stock, provided that the customer never asks for petrol of zero volume and the station never sets its retail price to zero. Operation *ChangRprice* changes the local copy of the price. As we will see later, this operation will only be activated when the station decides to change the price. Finally, *INIT* initializes the pump by setting *vol* and *cost* to zero. As for initial price, it will be equated with the price stored in the station when the new pump is added.

C. Station

A station directly contains some pumps. As shown in Figure 8, it has two attributes, *pumps*, representing its pumps, and *rprice*, representing its retail price. The notion of direct containment is denoted by *pumps* : $\mathbb{P}$ *Pump*©. Operation *AddPump* adds a pump to the station, given that the selected pump, *p*?, is not currently in the station. Besides, it also initializes the new pump and equates the new pump's *rprice* with the station's *rprice*. Operation *ChangRprice* changes the station's retail price. Also, it can change all pumps' prices via their own operations *ChangRprice*, for all input variables with the same name *rprice* are equated. When the station is asked by the company to pay the outstanding money, it performs *PayOwed* and outputs *owed*! that must be equal to the amount in its record (defined in class *Record*). *INIT* sets *pumps* empty. *Pumping* selects a pump (actually the pump is selected by the customer, see class *Customer*) and promotes *Pumping* of class *Pump* to class *Station*. *Reset* is similar to *Pumping* in that it promotes *Reset* of class *Pump* to class *Station* and it happens when transaction fails.

D. Store

The store provides petrol to every station when requested, provided its stock meets the demand. As shown in Figure 9, it has one attribute, *stock*, denoting the current stock. A successful transaction is represented by *Pumping* that receives the demand, *vol*?, and deduces it from the stock. There is no explicit check, i.e., checking whether stock meets demand. This check is defined in the class *Company*. Actually, this check is implicitly required in the declaration of *stock* : $\mathbb{R}^+$. Because we declare *stock* is a non-negative real number, such constrain also applies to *stock*′, which implies

$$stock - vol? \geqslant 0$$

Station

$pumps : \mathbb{P}\, Pump©$
$rprice : \mathbb{R}^+$

AddPump

$\Delta(pumps)$
$p? : Pump$

$p? \notin pumps \land pumps' = pump \cup \{p?\}$
$\land p?.\textit{INIT} \land p?.rprice = rprice$

ChangRprice

$\Delta(rprice)$
$rprice? : \mathbb{R}^+$

$rprice' = rprice? \parallel \forall p : pumps \bullet p.ChangRprice$

PayOwed

$owed! : \mathbb{R}^+$

$\textit{INIT} \mathrel{\widehat=} pumps = \varnothing$

$Pumping \mathrel{\widehat=} p? : pumps \bullet p?.Pumping$

$Reset \mathrel{\widehat=} p? : pumps \bullet p?.Reset$

Figure 8. The *Station* class.

Store

$stock : \mathbb{R}^+$

Pumping

$\Delta(stock)$
$vol? : \mathbb{R}^+$

$stock' = stock - vol?$

Restock

$\Delta(stock)$
$vol? : \mathbb{R}^+$

$stock' = stock + vol?$

$\textit{INIT} \mathrel{\widehat=} stock = 0$

Figure 9. The *Store* class.

Although for readability and clarity, we add such check in class *Company*, there is a choice here. In the extreme, we can give the specification in the most "tedious" form where there exists much redundancy in constraints. At the other extreme, we can do it in the most "concise" form where the set of constraints is minimal in that no constraint can be inferred by others. In practice, we may need some balance. Ultimately this may depend on the specification reader. *Restock* adds to the store some petrol of volume *vol*? determined by the company. *INIT* initializes the store by setting *stock* to zero.

Record

$station : Station$
$total, owed : \mathbb{R}^+$

Trans

$\Delta(total, owed)$
$vol?, wprice? : \mathbb{R}^+$

$total' = total + vol?$

$owed' = owed + wprice? * vol?$

PayOwed

$\Delta(owed)$
$owed? : \mathbb{R}^+$

$owed? = owed \land owed' = 0$

$\textit{INIT} \mathrel{\widehat=} total = owed = 0$

Figure 10. The *Record* class.

E. Record

As shown in Figure 10, class *Record* records the total volume pumped so far and the outstanding amount of money owed by the *station*. The former is represented by *total* and the latter by *owed*. When a successful transaction completes, the corresponding modification is made to the record. That is, adding the volume of the current transaction to the original *total* and the corresponding cost $wprice? * vol?$ to the original *owed*. *wprice*? is input here but it is renamed later with the actual whole sale price *wprice* defined in class *Company*. *PayOwed* corresponds to the same name operation in class *Station*. It receives *owed*? that must be equal to the money owed by that station and sets the new *owed* to zero. *INIT* just initializes the record by setting *owed* to zero.

Obviously, every station can only appear in at most one record. This is not embodied in the class construct of *Record*. We cannot modify the declaration of *station* as $station : Station©$, since stations are already directly contained in the company (defined later in class *Company*). Alternatively, instead of *Record*, we can define a new class *Records* that contains attribute *records* as a partial function

$$records : Station \pfun (Total \times Owed)$$

where $Total ::= \mathbb{R}^+$ and $Owed ::= \mathbb{R}^+$. This property is captured by the requirement of partial function, though it becomes less object-oriented and inconvenient to reference attributes *total* and *owed*. Eventually, we decide to specify this one to one mapping between stations and records in the state schema of the *Company* class.

F. Company

Now we come to the most complex class of *Company*. As shown in Figure 11, it directly contains a set of

Company

$\lceil(\textit{INIT}, \textit{AddStaRec}, \textit{ChangeRprice}, \textit{ChangWprice}, \textit{AddPump}, \textit{PayOwed}, \textit{Restock}, \textit{Trans})$

$\textit{records} : \mathbb{P}\, \textit{Record}\textcircled{c}$
$\textit{stations} : \mathbb{P}\, \textit{Station}\textcircled{c}$
$\textit{store} : \textit{Store}\textcircled{c}$
$\textit{wprice} : \mathbb{R}^+$

$\cup\{r : \textit{records} \bullet r.\textit{station}\} = \textit{stations}$

$\# \cup \{r : \textit{records} \bullet r.\textit{station}\} = \#\textit{records}$

AddStaRec

$\Delta(\textit{stations}, \textit{records})$
$s? : \textit{Station}$
$r? : \textit{Record}$

$s? \notin \textit{stations} \land \textit{stations}' = \textit{stations} \cup \{s?\}$

$r?.\textit{station} = s? \land r?.\textit{INIT} \land \textit{records}' = \textit{record} \cup \{r?\}$

ChangeWprice

$\Delta(\textit{wprice})$
$\textit{wprice}? : \mathbb{R}^+$

$\textit{wprice}' = \textit{wprice}?$

$\textit{INIT} \mathrel{\widehat=} \textit{stations} = \textit{records} = \varnothing \land \textit{store}.\textit{INIT}$

$\textit{AddPump} \mathrel{\widehat=} s : \textit{stations} \bullet s.\textit{AddPump}$

$\textit{ChangeRprice} \mathrel{\widehat=} s : \textit{stations} \bullet s.\textit{ChangeRprice}$

$\textit{PayOwed} \mathrel{\widehat=} s : \textit{stations} \bullet [r : \textit{records} \mid r.\textit{station} = s] \bullet s.\textit{PayOwed} \fatsemi r.\textit{PayOwed}$

$\textit{Restock} \mathrel{\widehat=} \textit{store}.\textit{Restock}[\textit{vol} : \mathbb{R}^+ / \textit{vol}?]$

$\textit{Reset} \mathrel{\widehat=} s? : \textit{stations} \bullet s?.\textit{Reset}$

$\textit{Pumping} \mathrel{\widehat=} \textit{store}.\textit{pumping} \parallel s? : \textit{stations} \bullet s?.\textit{pumping} \parallel [r : \textit{records} \mid r.\textit{station} = s?] \bullet r.\textit{Trans}[\textit{wprice}/\textit{wprice}?]$

$\textit{Success} \mathrel{\widehat=} \textit{vol}? : \mathbb{R}^+ \bullet \textit{vol}? \leqslant \textit{store}.\textit{stock}$

$\textit{Failure} \mathrel{\widehat=} \textit{vol}? : \mathbb{R}^+ \bullet \textit{vol}? > \textit{store}.\textit{stock}$

$\textit{Trans} \mathrel{\widehat=} \textit{Failure} \parallel \textit{Reset} \Box \textit{Success} \parallel \textit{Pumping}$

Figure 11. The *Company* class.

stations, a set of corresponding records, a store and a changeable whole price. Attribute *stations* and *records* denote the set of stations and the set of records directly contained by the company. The notion of directly contain is again captured by the symbol ©. So is attribute *store*. *wprice* represents the whole price. All stations contained in the company must have corresponding records in the *records* and this is embodied in the constraint in state schema

$$\cup\{r : \textit{records} \bullet r.\textit{station}\} = \textit{stations}$$

Any station occurs exactly once in the records and this is captured by

$$\# \cup \{r : \textit{records} \bullet r.\textit{station}\} = \#\textit{records}$$

Otherwise,

$$\# \cup \{r : \textit{records} \bullet r.\textit{station}\} < \#\textit{records}$$

Operation *AddStaRec* adds a station and a corresponding record to *stations* and *records* respectively, provided this station is not contained in the company. *ChangWprice* simply changes the whole sale price to *wprice*?. *INIT* initializes the company by setting *stations* and *records* empty. *AddPump* lets the company select a station which then adds a pump. *Restock*, defined as $\textit{store}.\textit{Restock}[\textit{vol} : \mathbb{R}^+ / \textit{vol}?]$, promotes such operation from the store to the company level, in which variables are renamed. That is, the original *vol*?, the input to the store, is renamed with $\textit{vol} : \mathbb{R}^+$, a value provided by the company. *PayOwed* enables the company to select a

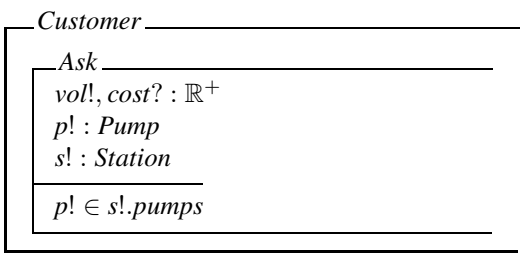


Figure 12. The *Customer* class.

station *s*, ask it to pay the owed money and then modify its record.

Operation *Reset*, *Pumping*, *Success*, *Failure*, and *Trans* together specify a successful or failed transaction on the company's part. *Reset* promotes such operation from the level of station to the level of company and it is performed when the customer's demand exceeds the stock. Otherwise, *Pumping* is performed at the company's level, which runs three operations in parallel, *Pumping* in the store, *Pumping* in a selected station and *Tran* in the corresponding record that updates itself. *Success* and *Failure* just check whether the stock meets the demand and only one of them holds at each transaction. A transaction is defined by *Trans* that makes a choice with symbol $\Box$ between a successful transaction *Success* $\|$ *Pumping* and a failed one *Failure* $\|$ *Reset*, depending on the current stock in the store and the customer's demand.

G. System

On the level of the petrol supply system, there are two parties, a company and a set of customers. First, we define class *Customer* in Figure 12 that is on the other side of the transaction. The only operation is *Ask*. It outputs *vol*!, the volume to pump, *p*!, the pump that performs pumping, and *s*!, the station that owns the pump. It receives an input *cost*?, which is either equal to zero, indicating a failed transaction, or equal to some positive value, indicating the cost of the pumped petrol of volume *vol*! . The only constraint is that *p*! must be in the *pumps* in that station *s*!. In fact, this is not necessary, for it is implicitly required in *Pumping* and *Reset* defined in the class *Station*. It is rewritten here for clarity.

With both classes *Company* and *Customer* ready, as defined in Figure 13, the petrol supply system communicates them through operation *Trans*. The selected customer indicates a pump of a station and asks for a certain amount of petrol through *Ask* . On the other side, the company performs *Trans*. By the definition of $\|$, inputs are equated with the outputs with the same base name on the other side. Hence *p*?, *s*? and *vol*? in the expanded version of *Trans* on the company's side are equated with the corresponding variables on the customer's side. *cost*? on the customer's side is equated with *cost*! on the company's side that is ultimately defined in the state schema constraint of class *Pump*. Since there is no need to communicate with the environment, *p*!, *s*!, *vol*!, and *cost*! are hidden. Other operations just promote the corresponding operations in *Company* to *System*.

V. CONCLUDING REMARKS

In this paper, we presented a brief overview of Object-Z and then demonstrated its use on a case study of a petrol supply system. Considering the complexity of modern systems, structured specification is essential in that it decomposes the concerns to its constituent objects. Object-Z extends the Z language by including a class construct that enables it to specify in object-oriented environment. The notion of object reference makes it simple to express object aggregates. Furthermore, its expressive power is greatly enhanced by introducing notions like object containment and operation expressions like conjunction operator.

Although the case study is not large, our exercise of Object-Z on it has provided some feedback to the development of Object-Z. Below, we highlight some of them, which are worth further investigation.

- What is the suitable degree of constraint redundancy in such a specification? If we write the specification in the most succinct mode, it may be unreadable to humans, not to mention model checking by humans. Therefore, we need computer programs to automatically check models by executing the specification and deriving implicit constraints when necessary [29], [30]. On the other hand, if we write the specification too tedious, it may cause a waste of computation, for the same constraint is checked multiple times in multiple places by the machine [31], [32]. Apparently we need some tradeoff between human-readability and machine-efficiency.
- In the visibility list, it is more appropriate to let some features accessible to a limited set of classes, rather than all of other classes. For instance, *Restock* in class *Store* should be visible only to class *Company*. So we need another construct, whose function is similar to "friend list" in C++.
- The process semantics of Object Z means that process execution is single-threaded, operations are atomic in that the duration of operations is zero. For instance, the operation *Pumping* of class *Pump* is either initiated and executed successfully or never initiated at all. In practice, however, the pumping may fail halfway. Because the process control logic is tightly coupled with class structure, it is difficult to use Object-Z to model real-time reactive systems. Hence we may need to combine Object-Z with other process-oriented languages like Communicating Sequential Processes (CSP) [15], where data and algorithmic concerns are handled by Object-Z style and process control, timing, and communication concerns are treated in the CSP style [33], [34].
- In the case study, specification is written manually and may vary much from person to person. Besides, in practical large-scale software development, few people grasp enough knowledge in the complex mathematical concepts of formal methods. Hence,

System

$\uparrow(\mathit{INIT}, \mathit{AddStaRec}, \mathit{AddPump}, \mathit{ChangeRprice}, \mathit{ChangWprice}, \mathit{PayOwed}, \mathit{Restock}, \mathit{Trans})$

$company : Company$
$customers : \mathbb{P}\, Customer$

$\mathit{INIT} \mathrel{\widehat=} company.\mathit{INIT}$

$\mathit{AddStaRec} \mathrel{\widehat=} company.\mathit{AddStaRec}$

$\mathit{AddPump} \mathrel{\widehat=} company.\mathit{AddPump}$

$\mathit{ChangeRprice} \mathrel{\widehat=} company.\mathit{ChangeRprice}$

$\mathit{ChangeWprice} \mathrel{\widehat=} company.\mathit{ChangeWprice}$

$\mathit{PayOwed} \mathrel{\widehat=} company.\mathit{PayOwed}$

$\mathit{Restock} \mathrel{\widehat=} company.\mathit{Restock}$

$\mathit{Trans} \mathrel{\widehat=} cust : customers \bullet (cust.Ask \parallel company.Trans) \setminus \{p!, s!, vol!, cost!\}$

Figure 13. The *System* class.

we need tool support that aims at providing translation between human modeling languages and formal methods. Ideally, the tool accepts the user's specification in natural languages and then automatically generates an "optimal" formal specification. Furthermore, the tool could even allow the user to describe their models graphically without knowing the underlying syntax and knowledge of Object-Z. Of course, many issues arise here, such as natural language understanding [35] and specification refinement [36], [34]. Unlike the above translation tool between the two extremes (e.g. least-formal methods like natural language and formal methods like Object-Z), a more practical approach may start with bridging the gap between semi-formal methods like UML and Object-Z [37], [26]. The translation should be two-way in that it not only transforms the description in UML to Object-Z specification to ensure the reliability of software, it also transforms Object-Z specification back to UML to facilitate the interactions among stakeholders.

VI. Acknowledgment

This work was partially supported by Natural Science Foundation of China (Grant No. 61100136).

References

[1] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626–643, 1996.

[2] J. Bicarregui, J. Fitzgerald, P. Larsen, and J. Woodcock, *FM 2009: Formal Methods*. Springer, 2009, ch. Industrial Practice in Formal Methods: A Review, pp. 810–813.

[3] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*. Prentice Hall International, 1990.

[4] M. Henson, S. Reeves, and J. Bowen, "Z logic and its consequences," *Computing and Informatics*, vol. 22, no. 3-4, pp. 381–415, 2012.

[5] R. Duke, G. Rose, and G. Smith, "Object Z: A specification language advocated for the description of standards," *Computer Standards and Interfaces*, vol. 17, pp. 511–533, 1995.

[6] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, 1994.

[7] K. Hansen, A. Ravn, and H. Rischel, "Specifying and verifying requirements of real-time systems," in *Proceedings of the ACM SIGSOFT Conference on Software for Critical Systems*, 1991.

[8] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, 2nd ed. Prentice Hall, 1996.

[9] C. B. Jones, *Systematic Software using VDM*, 2nd ed. Prentice Hal, 1990.

[10] J. R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[12] C. Heitmeyer, R. Jeffords, and B. Labaw, "Automated consistency checking of requirements specificatons," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 231–261, 1996.

[13] M. C. Gaudel, "Structuring and modularizing algebraic specifications: the pluss specification language, evolutions and perspectives," in *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, 1992, pp. 3–18.

[14] S. Owre, J. Rushby, and N. Shankar, "Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, 1995.

[15] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[16] R.Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[17] *Working Document on Topic 9.1 - ODP Trader*, ISO Std.

[18] J. Dong, R. Duke, and G. Rose, "An object-oriented approach to the semantics of programming languages," in *Proceedings of the 17-th Annual Computer Science Conference*, 1994, pp. 767–775.

[19] G. Rose and R. Duke, *Object-Oriented Specification Case*

Studies. Prentice Hall International, 1993, ch. An Object-Z Specification of a Mobile Phone System, pp. 110–129.

[20] S. Asif, S. Khan, and F. Ahmad, "Formalization of oil and gas seismic survey using z-notation," *Journal of American Science*, vol. 8, no. 12, pp. 820–826, 2012.

[21] Q. Li and G. Smith, "Using bounded fairness to specify and verify ordered asynchronous multi-agent systems," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, 2013, pp. 111–120.

[22] X. Wu, J. W. Sanders, and H. Zhu, "Formal modelling and analysis of aodv," in *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems*, 2013, pp. 93–100.

[23] B. Xu and L. Zhang, "Specification of train control systems using formal methods," in *Proceedings of the FTRA 4th International Conference on Mobile, Ubiquitous, and Intelligent Computing*, ser. LNEE 274, 2013, pp. 131–136.

[24] B. Meng, W. Wang, and W. Chen, "Verification of resistance of denial of service attacks in extended applied pi calculus with proverif," *Journal of Computers*, vol. 7, no. 4, pp. 890–899, 2012.

[25] F. H. de Assis, F. K. Takase, N. Maruyama, and P. E. Miyagi, "Developing an rov software control architecture: A formal specification approach," in *Proceedings of the 38th Annual Conference on IEEE Industrial Electronics Society*, 2012, pp. 3107–3112.

[26] A. Rasoolzadegan and A. A. Barforoush, "Reliable yet flexible software through formal model transformation (rule definition)," *Knowledge and Information Systems*, pp. 1–48, 2013, online.

[27] J. Dong, G. Rose, and R. Duke, "The role of secondary attributes in formal object modeling," in *Proceedings of the 1st IEEE International Conference on Engineering of Complex Computer Systems*, 1995.

[28] J. S. Dong and R. Duke, "The geometry of object containment," *Object-Oriented Systems*, vol. 2, no. 1, pp. 41–63, 1995.

[29] J. Sun, H. Wang, and T. Hu, "Design software architecture models using ontology," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, 2011, pp. 191–196.

[30] X. Ren, X. Xu, and Y. Li, "An one-way hash function based lightweight mutual authentication rfid protocol," *Journal of Computers*, vol. 8, no. 9, pp. 2405–2412, 2013.

[31] A. Felfernig, F. Reinfrank, and G. Ninaus, *Foundations of Intelligent Systems*. Springer-Verlag, 2012, ch. Resolving Anomalies in Configuration Knowledge Bases, pp. 311–320.

[32] A. Felfernig, C. Zehentner, and P. Blazek, "Corediag: Eliminating redundancy in constraint sets," in *Proceedings of the 22nd International Workshop on Principles of Diagnosis*, 2010, pp. 219–224.

[33] B. Mahony and J. S. Dong, "Timed communicating object z," *IEEE Transactions on Software Engineering*, vol. 26, no. 2, pp. 150–177, 2000.

[34] J. Derrick and E. A. Boiten, *Refinement in Z and Object-Z*. Springer, 2014.

[35] T. Hu, C. L. Tan, Y. Tang, S. Y. Sung, H. Xiong, and C. Qu, "Co-clustering bipartite with pattern preservation for topic extraction," *International Journal on Artificial Intelligence Tools*, vol. 17, no. 1, pp. 87–107, 2008.

[36] M. Shahbaz, K. C. Shashidhar, and R. Eschbach, "Iterative refinement of specification for component based embedded systems," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 276–286.

[37] H. Du and W. Yu, "A measur and rup combined business modeling method," *Journal of Computers*, vol. 6, no. 6, pp. 1086–1093, 2011.

Yangping Li received her B.A. degree from Tamkang University, Taiwan. She is currently a technical staff member at Dongguan University of Technology, China. Her research interest includes information systems and formal languages.

Xiaoheng Pan received his M.E. degree from University of Chinese Academy of Sciences, China. He is currently a software engineer at Dongguan University of Technology, China. His research interest includes information systems and software engineering.

Tianming Hu received his Ph.D. degree from National University of Singapore, Singapore. He is currently a professor at Dongguan University of Technology, China. His research interest includes software testing and data mining.

Sam Yuan Sung received his Ph.D. degree from University of Minnesota, USA. He is currently a professor at South Texas College, USA. His research interest includes information systems and data mining.

Huaqiang Yuan received his Ph.D. degree from Shanghai Jiaotong University, China. He is currently a professor at Dongguan University of Technology, China. His research interest includes software verification and information security.