Proc. Natl. Sci. Counc. ROC(A)
Vol. 22, No. 3, 1998. pp. 344-357

An Integrated Combination of JSD and Formal Notations

JONATHAN LEE, JIANN-I PAN, NIEN-LIN XUE, AND WEI-TEH HUANG

Department of Computer Science and Information Engineering
National Central University
Chungli, Taiwan, R.O.C.

(Received February 5, 1997; Accepted September 17, 1997)

ABSTRACT

In this paper, we propose the use of Z as the formal notation to express Jackson System Development (JSD) specifications (called JSDZ), where JSD serves as a mechanism to aid analysis of problem domains. A function process in JSD specifications is manifested by an operation schema. A model process is treated as an active entity that requires an operation on its data store to add a new instance to the collection of existing instances. The model process is, thus, translated into a state schema, and its related operations are converted into the operation schemas with instances set that can be modified by the operations. A structured diagram is transformed into a state transition diagram and then converted into its Z specifications. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merge. These transformation rules are illustrated using the problem domain of the Car-Rental System. The bringing together of diagrammatical and text elements of JSD specifications in Z notations offers two major benefits. First, JSD specifications can be seen both as a structuring mechanism that helps in deriving Z specifications and as a preliminary step that assists in ascertaining the clients requirements. Second, Z specifications make it easier to identify omissions or errors.

Key Words: formal specifications, informal methods, Jackson System Development (JSD), methods integration, Z

I. Introduction

Recently, there has been growing interest in combining different methods in formal methods research, especially in terms of the marriage of an informal method with a formal one (Wing, 1990). An informal method is developed to help users and analysts to informally define systems and provides a systematic procedure for analyzing systems. On the other hand, a formal method is a precise specification language which uses mathematical notations to specify a system.

It has been widely recognized that informal methods have advantages for elicitation of requirements, ease of understanding, communication, and support for the software development process through structuring mechanisms. Meanwhile, formal methods provide conciseness, clarity and precision, and are more suitable for detailed analysis and verification. Therefore, as advocated by Meyer (1985), informal requirements specifications and formal ones are complementary, not competing. Gehani (1986) has further suggested that system specifications, ideally, should include both formal and informal specifications, and that a method for combining both specifications is needed.

As a result, a number of researchers have reported progress towards the successful integration of formal and informal methods, such as by combining structured analysis (SA) with formal specifications (e.g., France, 1992; Fraser *et al.*, 1991; Liu, 1993; Polack *et al.*, 1991; Randell, 1991; Semmens and Allen, 1990), by integrating object-oriented analysis (OOA) and formal specifications (e.g., Giovanni and Iachini, 1990; Hammond, 1994; Lee *et al.*, 1997), by transforming Jackson System Development (JSD) to formal specifications (e.g., Sridhar and Hoare, 1989; Smith *et al.*, 1991), and by utilizing task-based specification methodology to derive specifications expressed in conceptual graphs (Lee *et al.*, 1996).

We propose the use of JSD (Cameron, 1986) as a mechanism to aid analysis of problem domains. As pointed out in Renold (1988) and Sridhar and Hoare (1989), JSD offers several benefits that are useful for constructing informal requirements specifications: First, JSD is applicable to a wide range of problems (e.g., data processing systems, real time systems etc. (Renold, 1988)). Second, JSD begins with a model of real world entities (i.e., a kernel), and functions are considered in the network phase in an incremental fashion. This

helps considerably in keeping the kernel intact, which in turn reduces the time needed for testing and maintenance (Renold, 1988).

The Z specification language has been widely used and successfully applied to industrial usage (Craigen *et al.*, 1995). For example, IBM has used Z successfully in specifying parts of its Customer Information Control System (CICS) transaction processing system; Z has been used to clarify an IEEE floating point standard; and Z is recommended by the UK Ministry of Defense for certain project classes. The benefits of using Z can be summarized as follows (Semmens and Allen, 1990): (1) It provides a specification which is precise and unambiguous. (2) It allows the possibility of mathematical reasoning about the specification. (3) It shows evidence of a clear understanding of the requirements.

The focus of this paper is on the use of Z (Spivey, 1992) as a formal notation to express JSD requirements specifications. A function process in JSD specifications is manifested by an operation schema. A model process is treated as an active entity that requires an operation on its data store to add a new instance to the collection of existing instances. The model process is, thus, translated into a state schema, and its related operations are converted into the operation schemas with instances set that can be modified by the operations. A structured diagram is transformed into a state transition diagram (STD) and then converted into its Z specifications. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merge. These transformation rules are illustrated using the problem domain of the Car-Rental System (Storer, 1987).

In the following, we will first give an overview of JSD in the next section. The proposed approach, which guides mapping from JSD specifications to Z notations, is fully discussed in Section III. Related work is introduced in Section IV. Finally, we summarize the benefits of our approach and outline our future research plans in Section V.

II. JSD Overview

JSD is a methodology for software development first proposed by M. Jackson (Cameron, 1986). Variations of JSD have been proposed, and we will base our discussion on the features discussed in Sridhar and Hoare (1989). JSD specifications are mainly composed of a distributed network of sequential processes. Each process can have its own local data. Communication between processes is achieved by reading and writing messages (data stream) and by read-only access to another process's data (state vector). A JSD specification is initiated from a particular set of model processes (or entities). New processes are added to the specification by connecting them to the model. There are three main phases in JSD methodology:

(1) Modeling phase: Model processes (or entities) and their actions are selected and defined.
(2) Network phase: The relationships between model processes and function processes are established and are represented by the system specification diagram (SSD).
(3) Implementation phase: The processes and their data are fitted to the available processor and storage devices.

In the following sections, we will only concentrate on the modeling and network phases as well as some notations and their meaning as they are used in these two phases. The "Car-Rental System" is used as an example to illustrate both the JSD methodology and the proposed approach to transforming JSD specifications to Z. The problem description of the "Car-Rental System" is summarized below (Storer, 1987).

The "Car-Rental System" concerns renting cars to renters. Cars are rented and returned by customers, and customers are invoiced for each transaction. The system produces customer invoices, responds to inquiries about whether a particular car is rented or not, and produces a weekly report of car rental details.

Entity	attribute	type	Action	attribute	type
Car	reg_no	CAR_ID	Purchase	reg_no	CAR_ID
	bought_date	DATE		bought_date	DATE
	price	N		price	N
	last_renter	RENTER_ID	rent	reg_no	CAR_ID
	rent_date	DATE		renter	RENTER_ID
	in-company	BOOLEAN	return	reg_no	CAR_ID
	on_rent	BOOLEAN		return_date	DATE
	rent_price	N	sell	renter	RENTER_ID
	time_on_rent	N		reg_no	CAR_ID
				sold_date	DATE
Renter	id	RENTER_ID	start	renter	RENTER_ID
	name	TEXT		name	TEXT
	address	ADDRESS		address	ADDRESS
	lastaction_date	DATE	invoice	renter	RENTER_ID
				name	TEXT
				address	ADDRESS
				reg_no	CAR_ID
				invoice	DATE
				rent_period	N
				amount	N

Fig. 1. Entity-action list.

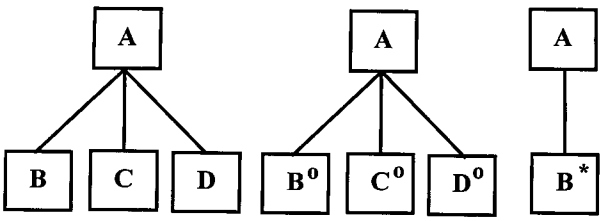
1. Modeling Phase

The first step in the modeling phase is to find entities and actions related to each entity. An entity-action list is used to specify the entities, related actions and their attributes needed in the system. The entity-action list for the Car-Rental System is shown in Fig.

1.

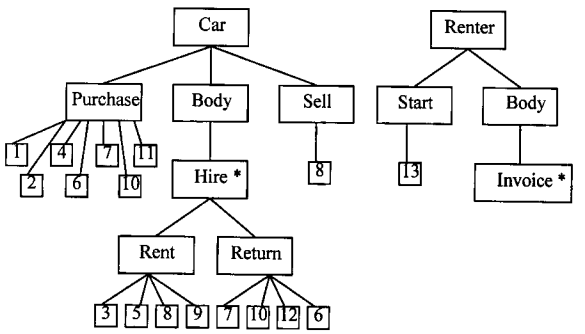
In JSD specifications, a model process is composed of a set of ordering actions and is denoted by a structured diagram. A structured diagram has a tree structure, in which the leaves are actions, and all other components describe either sequential, iterative or selective relationships between actions or between group of actions. Iterations are denoted by "*" in the top right corner of the constituent box, and selections are denoted by "o". Figure 2(a) shows that A is a sequence component, which means that A contains one B action, followed by one C action, followed by one D action. Figure 2(b) shows that A is a selection component, which means that A consists of either exactly one B action or exactly one C action or exactly one D action. Figure 2(c) shows that A is an iteration component, which means that A contains zero or more B actions.

Having built a model process as a framework, we need to define data items and basic operations to describe in detail the meaning of the model process. The actions define what happens, and the data define what is to be remembered about what has happened. The model processes can then be transformed into the equivalent



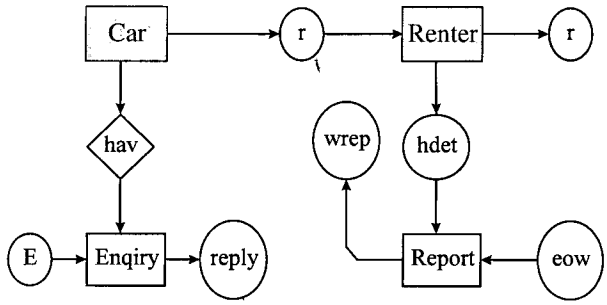
(a) *Sequence* (b) *Selection* (c) *Iteration*

Fig. 2. Structured diagram.



1. bought_date := purchase.bought_date?
2. price := purchase.price?
3. last_renter := rent.renter?
4. last_renter := UNDEFINED
5. rent_date := rent.rent_date/
6. rent_date := UNDEFINED
7. in_company := true
8. in_company := false
9. on_rent := true
10. n_rent := false
11. time_on_rent := 0
12. time_on_rent := return_date-rent_date
13. name := start.name
    address := start.address

Fig. 3. Structured diagram for the car-rental system.



hsv: hire state vector, r: renter_id, inv: invoice
hdet: hire detail, eow: end of week

Fig. 4. SSD for car-rental system.

structure text. Figure 3 shows the result of the basic operations that are added into the car and renter entities in the Car-Rental System.

2. Network Phase

The result of the modeling phase is a set of disconnected processes. In the network phase, input and output processes are added and connected to the model processes (e.g., *Car* and *Renter* in Fig. 4) to build a network of potentially concurrent processes. The network is described in an SSD. In this phase, three different types of function processes can be included into the network: input function processes, information function processes, and interactive function processes.

Input function processes collect data from the real world, check them for errors, and pass them on to the model process if they are correct or reject them by producing an error message. All the input processes together constitute the input subsystem. In particular, an input process has one of the following tasks: (1) to handle message and context error, (2) to replicate action messages for common actions, or (3) to serve as the dialog processes of on-line parts.

Information function processes extract information from the model in order to compute the required system outputs. The system's outputs must be definable in terms of the model, namely, as actions, action attributes and entity attributes. If this is not possible, the model must be elaborated by defining new actions and entity attributes. *Enquiry* and *Report* in Fig. 4 are information function processes.

Interactive function processes generate system actions. System actions can be considered as external actions that are created by the system itself. In most cases, the interactive function process needs information from the model processes to generate the system actions. Interactive function processes are like infor-

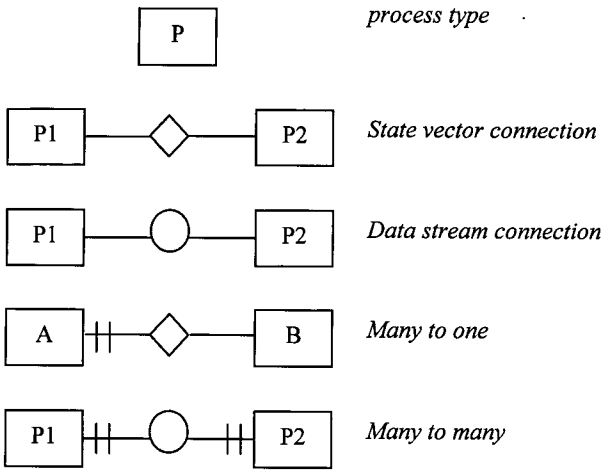


Fig. 5. SSD notations.

mation function processes except that they produce inputs for the model processes instead of system outputs. There are two communication mechanisms between processes in JSD.

(1) State Vector: a state vector (SV) connection constitutes a read-only access from a process to the local data of another process.

(2) Data Stream: a data stream (DS) is a first-in-first-out data queue with infinite capacity.

The cardinality relationship between processes can be either one-to-one, one-to-many or many-to-many (Fig. 5) to capture the notion of the multiplicity of a process (i.e. multiple instances). By completing all function processes and communicating them with modeling processes using SV and DS, an SSD is established.

In the next section, the proposed approach will be fully discussed and illustrated step-by-step using the Car-Rental System.

III. Expressing JSD Specifications in Z Notations

The focus of this section will be on the use of Z as a formal notation to express JSD requirements specifications. A function process in JSD specifications is manifested by an operation schema. A model process is treated as an active entity that requires an operation on its data store to add a new instance to the collection of existing instances. The model process is, thus, translated into a state schema, and its related operations are converted into the operation schemas with instances set that can be modified by the operations. A structured diagram is transformed into a state transition diagram and then converted into its Z specifications. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merge. However, the transformation from JSD to Z focuses not only on the conversion of each JSD artifact to its corresponding Z specifications, but also on the transformation of the relationships between these artifacts. An overview of the proposed approach depicting the mapping from JSD specifications to Z notations is shown in Fig. 6.

1. Modeling Phase

(1) Define given sets for all types of attributes based on the entity-action list. Consider the Car-Rental example; the types needed in the system are described as follows:

$[DATE, TEXT, RENTER\_ID, BOOLEAN, CAR\_ID, ADDRESS]$.

(2) Define a state schema for each entity, where its signature is depicted using the attributes in the entity-action list. Using the same example, we define *Car* and *Renter* state schemas for the entities car and renter:

Car

$reg\_no : CAR\_ID$
$bought\_date : DATE$
$price : N$
$last\_renter : RENTER\_ID$
$rent\_dat : DATE$
$in\_company : BOOLEAN$
$on\_rent : BOOLEAN$
$rent\_price : N$
$time\_on\_rent : N$

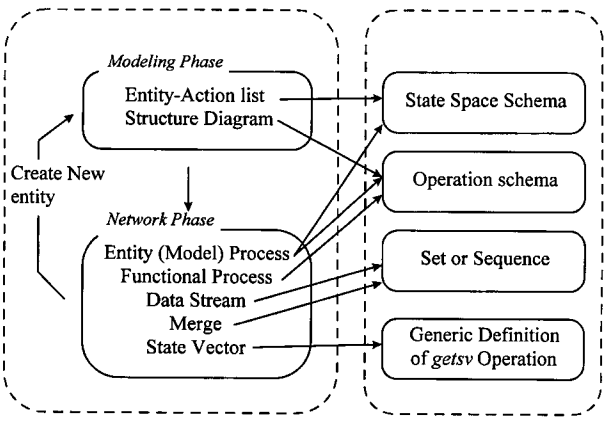


Fig. 6. An overview of the mapping from JSD to Z.

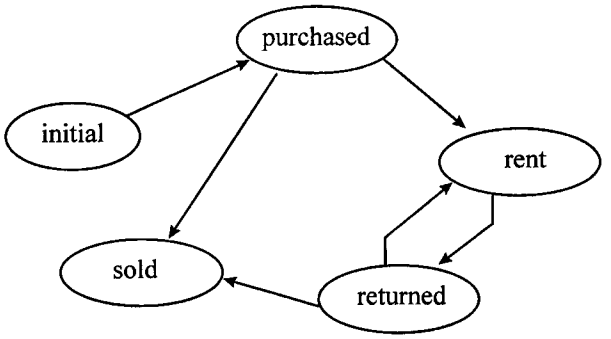


Fig. 7. State transition diagram for the structured diagram of the car entity.

Renter

id:RENTER_ID
name:TEXT
address:ADDRESS
lastaction_date:DATE

(3) Create a data store state schema to represent an instances set for each entity. In each data store, an identity is required to uniquely identify each instance. In the same example, there are two data store state schemas (*Car_SET* and *Renter_SET*):

Car_SET

cars:**F** *Car*
idcar:CAR_ID $\rightarrowtail\!\!\!\!\!\!\!\!\rightarrow$ *Car*

$\forall c$:*Car* • $c \in cars \wedge c.reg\_no = idcar^{-1}(c)$
ran *idcar*=*cars*

Renter_SET

renters:**F** *Renter*
idrenter:RENTER_ID $\rightarrowtail\!\!\!\!\!\!\!\!\rightarrow$ *Renter*

$\forall r$:*Renter* • $c \in renters \wedge r.id = idrenter^{-1}(r)$
ran *idrenter*=*renters*

(4) Add an attribute, *currentstate*, to indicate the possible states the entity can go through for each entity state schema. The type of *currentstate* can be composed of an *initial* state and other states that result from performing each action on the entity. The possible states for car (denoted as *C_STATE*) can be: *initial*, *purchased*, *rent*, *returned*, and *sold*. The initial state means the state of the car before any action is performed, and the purchased state means that the car has been purchased but not yet rented. The schemas and type definitions below are the result after adding the *currentstates* for the *Car* and *Renter* state schemas, respectively:

$C\_STATE ::= initial \mid purchased \mid rent \mid returned \mid sold$

$R\_STATE ::= initial \mid started \mid invoiced$

Car

reg_no:CAR_ID
bought_date:DATE
price:N
last_renter:RENTER_ID
rente_date:DATE
in_company:BOOLEAN
on_rent:BOOLEAN
rent_price:N
time_on_rent:N
car_currentstate:C_STATE

---

car_currentstate=*initial*

Renter

id:RENTER_ID
name:TEXT
address:ADDRESS
lastaction_date:DATE
renter_currentstate:R_STATE

---

renter_currentstate=*initial*

(5) Convert each structured diagram into a STD by arranging each state (represented by an oval) defined in step 4, based upon the time order in the structured diagram, and by connecting the states through actions that are viewed as events (represented by directed arcs) to change the states in a STD. Jackson has pointed out that the time-ordering structure in JSD can also be specified using regular expressions. Recently, Bass and Ratcliff (1994) have further elaborated on this claim and have provided automatic support for the implementation of JSD process specifications. We have adopted a similar view in describing the time-ordering structure. Referring to the same example, the structured diagram of the car entity in Fig. 3 can be converted into the STD given below.

(6) Define an operation schema for each action, and define its signature according to the entity-action list. If an action changes its associated data store, attach the *Delta* notation to its associated data store to indicate the change. Otherwise, use the Ξ notation. If an action is shared by more than one entity, include all their related data

stores in the declaration part. The operation schema for the *purchase* and *rent* actions of car entity are described below:

purchase
Δ *Car_SET*
reg_no:*CAR_ID*
bought_date?:*DATE*
price?:*N*
car:*Car*

rent
Δ *Car_SET*
reg_no?:*CAR_ID*
rent_date?:*DATE*
renter?:*RENTER_ID*

(7) Include the state changes using state variables in the predicate part of each operation schema based on the STD obtained in step 5. For example, *car.car_currentstate= purchased | returned* shows the state before the operation *rent*, where there are two mutual exclusive states, *purchased* and *returned*; *car.car_currentstate' =rent* indicates the state after the operation:

rent
Δ *Car_SET*
reg_no?:*CAR_ID*
rent_date?:*DATE*
renter?:*RENTER_ID*
---
car_currentstate=initial
reg_no?∈ *Car_SET.idcar*
.
.
.
Car.car_currentstate=purchased | returned
.
.
.
car.car_currentstate'=rent

(8) Treat all the entities in the modeling phase as active entities. An active entity is defined as an entity which has instances created as a part of the normal system operations, thus requiring an operation on its data store to create a new instance to be added to the collection of existing instances. For example, the *purchase* action is treated as an operation that can create new instances to be added into the existing instances set, *Car_SET*. That is, *Car_SET.cars'= Car_SET.cars* ∪ {*car*}. We then include basic operations in the predicate part of the operation schema (e.g. *bought_date*):

purphase
Δ *Car_SET*
reg_no?:*CAR_ID*
bought_date?:*DATE*
price?:*N*
car:*Car*
---
reg_no? ∉ **dom** *Car_SET.idcar*
car.car_currentstate=initial
car.reg_no'=reg_no?
car.bought_date'=bought_date?
car.price'=price?
car.in_company'=true
car.on_rent'=false
car.time_on_rent'=0
car car_currentstate'=purphased
Car_SET.cars'=Car_SET.cars ∪ {*car*}
Car_SET.idcar'=Car_SET.idcar ⊕ {*reg_no?* ↦ *car*}

2. Network Phase

A. Function Process

Function processes in an SSD are transformed into operation schemas. Generally, both input and information function processes will not change the related state space while interactive function processes will. Therefore, we usually attach the Ξ notation to the related data store for input and information function processes, and attach Δ for the interactive function process. Referring to the same example, two operation schemas are described below for the function processes in the SSD. *Report* is a function process that creates a renting transaction list for that week. The *Enquiry* function process receives a customer's query and answers with the rental status of a car:

RENTSTAUS::= rent | available

Enquiry
Ξ *Car_SET*
reg_no?:*CAR_ID*
status!:*RENTSTATUS*
---
reg_no? ∈ *dom Car_SET.idcar*
∃ *car*:*Car* • (
car.reg_no=reg_no?
(*getsv*(*car.on_rent*)=*true*) ⇒ *status!=rent*
(*getsv*(*car.on_rent*)=*false*) ∧
(*getsv*(*ca.in_company*)=*true*) ⇒ *status!=available*

B. Data Stream

A data stream can be considered as a first-in-first-out data buffer with infinite capacity and two opera-

tions: *read* and *write* (Renold, 1988). Each data stream can be expressed in Z through the following steps: (1) Define a schema for each data stream entry. (2) Denote a data stream as a sequence of such entries. (3) Define the *write* operation schema for the data stream. (4) Define the *read* operation schema for the data stream. If a data stream merges with others, we use a merge reading process instead of a read operation. In our example, the data stream *hdet* can be expressed as follows. Because *hdet* merges with *eow*, we will use other the reading process instead of *read_hdet*:

hdet

renter:*RENTER_ID*
name:*TEXT*
address:*ADDRESS*
reg_no:*CAR_ID*
invoice_date:*DATE*
rent_period:*N*
amount:*N*

hdet_DS:**seq** *hdet*

write_hdet

Δ *hdet_DS*
entry?:*hder*

entry?.*renter*=*invoice.renter*?
entry?.*name*=*invoice.name*!
entry?.*address*=*invoice.address*!
entry?.*reg_no*=*invoice.reg_no*?
entry?.*invoice_date*=*invoice.invoice_date*?
entry?.*rent_period*=*invoice.rent_period*?
entry?.*amount*=*invoice.amount*!
hdet_DS'=<*a*>⌢ *hdet_DS*

read_hdet_ok

Δ *hdet_DS*
entry!: **seq** *hdet*
r!:*Response*

hdet_DS ≠ <>
entry!=*hdet_DS*
hdet_DS'=<>
r!=*success*

read_hdet_suspend

Ξ *hdet_DS*
r!:*Response*

hdet_DS=<>
r!=*suspend*

read_hdet$\hat{=}$*read_hdet_ok* ∨ *read_hdet_suspend*

C. State Vector

For all the state vectors in an SSD, we predefine a *getsv* generic operation schema (defined below). Note that *getsv* will not change any of the state variables; therefore, any process (either model or function) that performs the operation needs to attach the Ξ notation to the related data stores. In our example, the *Enquiry* function process should include Ξ*Car_SET* in its signature and use *getsv* to get *on_rent* and *in_company* in the predicate part:

[*X*, *Y*]

getv:*X* → *Y*

getsv(x)=={*y* | *x* ∈ *X*, *y* ∈ *Y*}

Enquiry

Ξ *Car_Set*
reg_no?: *CAR_ID*
status!:*RENTSTATUS*

reg_no? ∈ *dom Car_SET.idcar*
∃ *car*:*Car* • (
car.reg_no=*reg_no*?
(*getsv*(*car.no_rent*)=*true*) ⇒ *status*!=*rent*
(*getsv*(*car.no_rent*)=*false*) ∧ (*getsv*(*ca.in_company*)=*true*)
⇒ *status*!=*available*

D. Merge

In a data stream connection, the initiative for communication comes from the writing process. The reading process must consume the complete stream of data records (Renold, 1988). If a process reads data streams from different processes, the developer must specify how the data streams are merged. The merge strategy defines the sequence in which the receiving process will read the data records.

There are two types of merges: fixed and rough merges. In a fixed merge, the reading sequence is defined by the reading process alone. In a rough merge, the strategy is indeterminate.

For rough merge, we use a set as an internal buffer in the reading process to take in the data from each data stream. If all the data streams are empty, the reading process is suspended; otherwise, the set collects the data from all the data streams and clears each data stream. The output sequence of the data in the set is not determined until implementation.

For fixed merges, we use a sequence as an internal buffer instead of a set to collect data. The output sequence of the data in the buffer sequence is determined by the reading process. In general, if one of the data streams is empty, the reading process is suspended. In particular, whether the reading process is

to be suspended or not depends on the problem domain. In general, if one of the data streams is empty, the reading process is suspended.

Note that due to the fact that data collected from data streams may be of different types, it is necessary that the internal buffer (i.e., a set) be able to assume different types as well (that is, the notion of polymorphism).

Referring to our example, *Report* is treated as a fixed merge, and its Z counterpart is described below:

Buffer::=hdet | *DATE*

fix_read_ok

Δ *hdet_DS*
Δ *eow_DS*
buffer:**seq** *Buffer*
entry!:**seq** *Buffer*
r!:*Response*

---

eow_DS ≠ <>
buffer=<>
(*eow_DS* ≠ <>)⇒(*buffer*=*buffer* ⌢ *eow_DS*)∧(*eow_DS'*= <>)
(*hdet_DS*≠<>)⇒(*buffer*=*buffer* ⌢ *hdet_DS*)∧ (*hdet_DS'*= <>)
entry!=**tail** *buffer*
r!=*success*

fix_read_suspend

Ξ *hder_DS*
Ξ *eow_DS*
r!:*Response*

---

eow_DS=<>
r!=*suspend*

Report≙*fix_read_ok* ∨ *fix_read_suspend*

E. Cardinality

The notion of cardinality in JSD, is different from that of the structured analysis and object-oriented approaches. Cardinality in the structured analysis and object-oriented approaches refers to the number of instances that an entity or an object can participate in. In JSD, cardinality specifies the communication between instances of a process. In Fig. 4, the data streams from the type A process are roughly merged to form a single input DS in the case of a many-to-one situation or a multiple input DS for the many-to-many case into process B.

In our example, the cardinality relation between *Renter* and *Report* is many-to-one and is described as many writing processes which write data into *hdet_DS* and read data by means of one reading process:

write_hdet_i

Δ *hdet_DS*
entry?:*hdet*

---

entry?.*renter*=*invoice.renter*?
entry?.*name*=*invoice.name*!
entry?.*address*=*invoice.address*!
entry?.*reg_no*=*invoice.invoice_date*?
entry?.*invoice_date*=*invoice.reg_no*?
entry?.*rent_period*=*invoice.rent_period*?
entry?.*amount*=*invoice.amount*!
hdet_DS'=<*a*>⌢ *hdet_DS*

write_hdet≙*write_hdet_*1 ∨...∨ *write_hdet_i* i ∈ **N**

read_hdet_ok

Δ *hdet_DS*
entry!: **seq** *hdet*
r!:*Response*

---

hdet_DS ≠ <>
entry!=*hdet_DS*
hdet_DS'=<>
r!=*success*

read_hdet_suspend

Ξ *hdet_DS*
r!=*Response*

---

hdet_DS=<>
r!=*suspend*

Read_hdet≙*read_hdet_ok* ∨ read*f_read_suspend*

IV. Related Work

A number of researchers have reported progress towards the successful integration of formal and informal methods. Semmens *et al.* (1992) have conducted a comparative study of related work. However, their study only concentrated on approaches combining structured analysis and formal specification techniques. To provide a more comprehensive view of the state-of-the-art in this line of work, we have classified integration methods based upon the different informal analysis approaches used.

(1) From structured analysis to formal specifications: for example, Polack (Mander and Polack, 1995; Polack, 1992; Polack *et al.*, 1991) and Semmens's (Semmens and Allen, 1990) integrated SA and Z; Randell (1991) translated a data flow diagram into Z ; Frances combined SA and algebraic specifications; Fraser *et al.* (1991) integrated structured analysis with Vienna Development Method (VDM); Liu (1993)

combined an extended data flow diagram and VDM; Moulding and Smith (1995) combined Controlled Requirement Expression (CORE) with VDM and CSP.

(2) From object-oriented analysis to formal specifications: typical examples are Giovanni's inclusion of Z in Hierarchical Object-Oriented Design (HOOD) (Giovanni and Iachini, 1990); Hammond (1994) integrated Shaler Mellor's approach into Z; and Lee *et al.* (1997) combined Bailin's Object-Oriented requirement Specification (OOS) with Z.

(3) From Jackson System Development to formal specifications; Sridhar and Hoare (1989) transformed JSD into Communicating Sequential Processes (CSP).

An important idea for developing a JSDZ approach is that most of the related works outlined above focused mainly on the integration of structured analysis or object-oriented analysis and formal methods. JSDZ provides an alternative option, in particular for those who apply JSD methodology to development of software requirements.

1. SA to Formal Specifications

In Semmens' work (Semmens and Allen, 1990), there were in general, two phases involved in the analysis. First, a data model of the system was developed, expressed first as an entity relationship diagram (ERD) and then as a Z state schema which was systematically derived from the ERD. In the second phase, the process model was built. A semiformal model was expressed using a data flow diagram (DFD); then, the semantics of each of the processes was specified using Z operation schemas. Entity was represented in an entity type schema and an entity instance set schema. Attributes of an entity could be referred to in an entity dictionary. A relationship could be represented as a relationship schema. Partial function, partial injection, or relation could be used to specify the cardinality and connectivity relationships between the two entities. These schemas built the state space of the system in Z. Data flows between processes could be shown as input and output variables in an operation schema.

Polack and her colleagues took a different view, seeing the systems analysis as an important but partially independent precursor to the formal definition (Mander and Polack, 1995; Polack, 1992; Polack *et al.*, 1991). In their approach, the Z specifications must not be a straight translation of the informal analysis if it is to find requirements or errors overlooked by the analysts. It must be guided by the systems analysis rather than be exclusively derived from it. The basic components of the Z definition, state and process specifications, are considered separately. In the state specifications of Z, which are derived from ERD, constraints on data or conditions on relationships are included in the formal state model, and data attributes may be modeled in the entity definitions. In the process specifications, function definitions are derived from DFD.

Randell (1991) explained how to translate a data flow diagram into Z. The Z specification is generated by carrying out a sequence of steps: taking external entities, data stores and processes in turn to produce parts of the Z specification. Since ERD is not considered in the data modeling, the state specifications are directly translated from data stores. The operation schemas are translated from a process. Input and output variables indicate the data flow between the external entity and the process. Predicate information about the process does not appear in a DFD and must be obtained elsewhere.

France (1992) provided a semantically extended DFD as a control-extended DFD (C-DFD) associated with an algebraic specification technique. A C-DFD's data domain is defined by specifiers using a data description language. State specifications and label specifications are defined so as to describe the data flow, and DataStore state specifications are used to describe the data stores. Operators used in accessing these data stores were also discussed. Data transform specification is used to specify the behavior of a data transform. Algebraic state transition systems corresponding to the state transition diagram can be used to specify the behavior of an external entity and the system behavioral.

Fraser *et al.* (1991) integrated VDM and structured analysis. Two approaches were described in their paper: (1) structured analysis as a cognitive guide in developing VDM specifications, and (2) automated generation of VDM specifications from SA models using a rule-based approach. The main difference in these two approaches is that the former is manual while the latter is automated. Information from an entity dictionary is used to build the information domain. Operators *rd* and *wr* indicate the data flow between processes. Pre-condition and post-condition are used to specify the functionality of the process.

Moulding and Smith (1995) focused on the use of the VDM with CORE and explained the role of CORE in CSP. That is, a VDM specification was used to define the processing semantics of the actions, and a CSP specification was used to identify the real-time collective behavior of these actions. In their approach, the data flows between viewpoints which are identified

in a CORE model form of the global state of the VDM specification, and the VDM data types for these state variables are derived from the data-structuring information within the CORE model. The target system and all other viewpoints are expressed as Vienna Development Method-Specification Language (VDM-SL) modules, in which the actions of the viewpoints are modeled as VDM operations. Internal data flows within a viewpoint are considered as the local state of the viewpoint module, and the triggering conditions for an action are expressed in the precondition of the corresponding operation. The control and sequencing behavior of operations is specified in single viewpoint modeling (SVM) and combined viewpoint modeling (CVM) stages of CORE. Each CORE action, channel and pool are represented by a CSP process. Processes are combined using the CSP parallel composition operator (||) to specify the overall behavior of the SVM. A CVM is a process composed of channels, pools and actions. Similarly, the compositional features of CSP are used to describe a CVM.

2. OOA/D to Formal Specifications

Giovanni and Iachini (1990) combined HOOD and Z, where HOOD was used as a structuring mechanism to guide the construction of Z specifications. In their paper, WHAT, With WHAT, and HOW specifications were used to build the formal specification. A WHAT specification could consist of more than one schema and be used to describe an object. These schemas could be partitioned into state schemas and operation schemas. A state schema was used to specify an object's data model. However, information for specifying an object was insufficient. Each method of an object was described in a separate operation schema.

Hammond (1994) integrated Shaler-Mellor's OOA and Z. In their approach, there are three phases: (1) information modeling, (2) state transition modeling, and (3) process modeling. ERD is used to express the information model in OOA. Objects and relationships are specified in an entity type schema, entity instance set schema, and relationship schema. In state transition modeling, a schema event is used to specify the generation or receipt of any event, and a state transition schema is used to specify state transitions. The predicates in a state transition schema should describe the pre-condition of an input event and the initial state, as well as the post-condition of the components' values of the after state and output events. An active object is defined as an object that requires an operation on its data store to add a new instance to the collection of existing instances. In the process modeling phase, process accessors and event generators are identified be means of Z transition specifications.

Bourdeau and Cheng (1995) presented formal semantics for OMT object model notations. Object models and instance diagrams were formalized as algebraic specifications and algebras, respectively. In their approach, instance diagrams can be used to provided semantics for object models, and algebraic specifications have algebras as their semantics. Finally, the set of algebras can be treated as the semantics of an object model. There are two ways to determine algebras: (1) compute the algebraic specifications of the object model and look at a set of algebras that satisfies this specification, and (2) determine the instance diagrams consistent with the object model and compute their corresponding algebras. If the design is consistent, then either method for determining the algebras will yield the same results. The Larch Shared Language (LSL) is used as the algebraic specification language. A trait in LSL is used to represent an abstract data type in an object model, and Σ-algebra is generated by means of this trait. Traits can be used to depict classes and associations. The related classes within an association are included in an association trait. The multiplicity constraints can be described in terms of four relational properties: functional, injective, surjective and total.

Lee *et al.* (1997) proposed an integration of Bailin's object-oriented specification and Z, called OOSZ. In OOSZ, OOS is used as a structuring mechanism to guide the derivation of Z specifications. An entity process can be considered in two dimensions: (1) as an active entity that requires an operation on its data store to add a new instance to the collection of existing instances, and (2) as an abstract operation which can be decomposed into operations related to the entity process. Therefore, an entity process is manifested through a state schema, an instance creation operation schema and an abstract operation schema; meanwhile, a function process is converted into an operation schema in Z specifications.

3. JSD to Formal Notations

Sridhar and Hoare showed how to express JSD in CSP through several examples (Sridhar and Hoare, 1989). In their approach, only the SD is considered in the transformation process. An entity modeled in the SD is described by its life history and can be defined by two formulas in CSP: the first one depicts the creation of an entity, and the second one describes the remainder of the history. Adding new actions later into the second formula is permitted. An iterative mechanism between CSP and JSD is provided. That

is, a new entity process can be defined in CSP and then feedback to modify the SD. Sridhar and Hoare only used several examples to show how JSD could be expressed in CSP. That is, no heuristics were provided to guide the transformation. In our work, the corresponding transformation rules between JSD and Z are given based on the steps introduced in JSD.

Compared with our work, JSDZ provides more systematic transformation heuristics. That is, the corresponding transformation rules between JSD and Z are given based on the steps introduced in JSD and Z.

Compared with all these approaches, JSDZ offers two important advantages: (1) artifacts (entity-action lists, structured diagrams and system specification diagrams) generated from the informal method (JSD) are tightly coupled, and (2) the notion of an active entity/object is manifested through a state schema, an instance creation operation schema and its operation schemas in the Z specifications. Table 1 summarizes the comparison of JSDZ and related work.

V. Conclusion

In this paper, we have proposed an integration of an informal method (JSD) with a formal notation (Z). A function process in JSD specifications is manifested by an operation schema. A model process is treated as an active entity that requires an operation on its data store to add a new instance to the collection of existing instances. The model process is, thus, translated into a state schema, and its related operations are converted into the operation schemas with instances set, which can be modified by means of the operations. A structured diagram is transformed into an STD and then converted into its Z specifications. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merge. These steps have been illustrated using the problem domain of the Car-Rental System.

The bringing together of diagrammatical and text elements of JSD specifications in Z notations offers two major benefits: (1) JSD specifications can be seen both as a structuring mechanism that aids the derivation of Z specifications and as a preliminary step that assists in ascertaining the clients requirements; (2) Z specifications make it easier to identify omissions or errors.

Various researchers have also remarked on the complementary nature of informal and formal methods and the benefits to be gained from combining the two. These benefits form the main motivation behind our work. In addition, we believe it is important to look at the relationships between different models (those generated from either informal methods or formal methods) in order to develop more powerful and comprehensive methods to support system analysis and design. This belief is also shared by many researchers such as France (1992), Fraser *et al.* (1991), Polack (1992), Semmens *et al.* (1992) and others. JSDZ has been developed based on this belief.

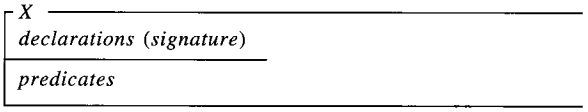
Our future research plans consist of several tasks: (1) to incorporate a knowledge base to assist the task of analysis, (2) to investigate the criteria for evaluating formal specifications derived from different informal methods, and (3) to utilize CASE tools (e.g., MicroStep (Yeh, 1990)) to transform Z specifications into prototypes.

Acknowledgment

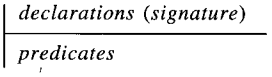
This project was supported by the National Science Council, R.O.C., under grant NSC 85-2213-E-008-005.

Appendix
A Summary of Z Notations

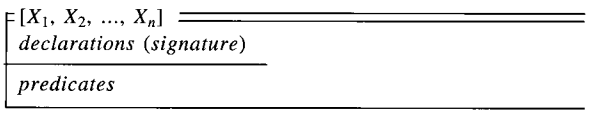
In this brief summary, we will only introduce those notations of Z used in this paper. Z is based on typed set theory and first-order logic. Z provides a construct, called a schema, to describe a specification's state space and operations. A schema groups variable declarations into a list of predicates that constrain the possible values of variables. In Z, the X schema is defined in the form:



Global functions and constants or axiomatic definitions are defined in the form:



Generic definitions are defined in the form:



The declaration gives the function or constant type while the predicate gives its value.

Declarations

x:**P** X	x is declared as a set of X's
x:**F** X	x is declared as a finite set of X's
$[X_1, X_2, ..., X_n]$	Introduction of free types called $X_1, X_2, ..., X_n$
ΔS	Change of state schema S
ΞS	No change of state schema S

Table 1. Integrating Formal and Informal Methods: A Comparison

•Structured Analysis

		Polack (Z)	Semmens (Z)	Randell (Z)	France (Algebra)	Fraser (VDM)	Moulding (VDM&CSP)
ERD	Entity	state schema - type - instance set	state schema - type - instance set	No	No	No	No
	Relationship	state schema - relation	state schema - relation				
DFD	Process	operation schema	operation schema	operation schema	data transform specification	pre-/post-condition -decision	VDM module
	Data Flow	input/output variables	input/output variables	input/output variables	state, label specification	rd/wr operation	global state & local state
	Data Store	entity	entity	entity	data store state specification	data store	data store
CFD	Behavior	No	No	No	ASTS, STD - transition spec. - state spec. - label spec. - transition relation transition axioms	No	SVM & CVM expressed in CSP

• Object-Oriented Analysis

	Giovanni (HOOD to Z)	Hammond (OOA to Z)	Cheng (OMT to Larch)	Lee (OOS to Z)
Object	WHAT spec. - state schema - operation schema	state schema - type schema - instance set schema - creation operation schema (for active entity)	trait	active entity process - state schema - abstract operation schema - instance creation schema
Use-relationship	with "WHAT" specification	data store	by the "includes" word which declared in a trait	data store
State Transition	No	event schema state transition schema	No	No

Sets

$x \in X$	x is a member of X
$x \notin X$	x is not a member of X
$S \subseteq T$	S is a subset of T
$S \cup T$	The union of S and T
$S \cap T$	The intersection of S and T
$S \setminus T$	The difference between S and T
$\emptyset$	Empty set: It contains no member
$\{x\}$	Singleton: It contains only x

Functions

$f: X \rightarrowtail Y$	f is declared as a partial injection from X to Y
$f: X \nrightarrow Y$	f is declared as a partial function from X to Y
dom f	The domain of f: The set of values x of which $f(x)$ is defined
ran f	The range of f: The set of values taken by $f(x)$ as x varies over the domain of f
$f \oplus \{x \mapsto y\}$	A function that agrees with f except that x is mapped to y

Logic

$s=t$	Equality between terms
$s \neq t$	Inequality between terms
$P \wedge Q$	P and Q: It is true if both P and Q are true
$P \vee Q$	P or Q: It is true if P or Q is true
$P \Rightarrow Q$	P implies Q: It is true if either Q is true or P is false
$\forall x: T \bullet P$	Universal quantification: "for all x of type T, P

	holds"
$\exists x{:}T \bullet P$	Existential quantification: "there exists an x of type T such that P holds"

Sequence

seq X	The set of finite and infinite sequences whose elements are drawn from X
$< >$	The empty sequence
$<a_1, \ldots, a_n>$	$= = \{1 \mapsto a_1, \ldots, n \mapsto a_n\}$
$\frown$	Concatenation
head X	The first element of a non-empty sequence
tail X	All but the head of a non-empty sequence
last X	The final element of a non-empty sequence
front X	All but the last element of a non-empty sequence

Schema Operators

$S \wedge T$	The schema formed from schemas S and T by merging their declarations and conjuncting their predicates
$S \vee T$	The schema formed from schemas S and T by merging their declarations and disjuncting their predicates

References

Bass, A. and B. Ratcliff (1994) Automated dismemberment of jsd process specifications. *Information and Software Technology*, **36**(8), 515-523.

Bourdeau, R. H. and B. H. C. Cheng (1995) A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, **SE-21**(10), 799-821.

Cameron, J. R. (1986) An overview of jsd. *IEEE Transactions on Software Engineering*, **SE-12**(2), 222-240.

Craigen, D., S. Gerhart, and T. Ralston (1995) Formal methods reality check: industrial usage. *IEEE Transactions on Software Engineering*, **SE-21**(2), 90-98.

France, R. B. (1992) Semantically extended data flow diagrams: a formal specification tool. *IEEE Transactions on Software Engineering*, **SE-18**(4), 329-346.

Fraser, M. D., K. Kumar, and V. K. Vaishnavi (1991) Informal and formal requirements specification languages: bridging the gap. *IEEE Transactions on Software Engineering*, **SE-17**(5), 454-466.

Gehani, N. (1986) Specifications: formal and informal–a case study. In: *Software Specification Techniques*, pp. 173-185. N. Gehani and A.D. McGettrick Eds. Addison-Wesley, Reading, MA, U.S.A.

Giovanni, R. D. and P. L. Iachini (1990) Hood and z for the development of complex software systems. In: *VDM and Z-Formal Methods in Software Development*, pp. 262-289. D. Bjorner *et al.* Springer-Verlag, Berlin, Germany.

Hammond, A. R. (1994) Producing z specifications from object-oriented analysis. 8th Z User Meeting, Cambridge, U.K.

Lee, J., L. F. Lai, and W. T. Huang (1996) Task-based specifications through conceptual graphs. *IEEE Expert*, **11**(4), 60-70.

Lee, J., J. I. Pan, and W. T. Huang (1997) Oosz: an integration of Bailin's objected-oriented analysis and formal specifications. *Journal of Information Science and Engineering*, **13**(4), 517-542.

Liu, S. (1993) A formal requirements specification method based on data flow analysis. *Journal of Systems and Software*, **21**, 141-149.

Mander, K. C. and F. A. Polack (1995) Rigorous specifications using structured systems analysis and z. *Information and Software Technology*, **37**(5-6), 285-291.

Meyer, B. (1985) On formalism in specifications. *IEEE Software*, **2**(1), 6-26.

Moulding, M. and L. Smith (1995) Combining formal specification and core: an experimental investigation. *Software Engineering Journal*, **10**(2), 31-42.

Polack, F. A. (1992) Integrating formal notations and system analysis: using entity relationship diagrams. *Software Engineering Journal*, **7**(5), 363-371.

Polack, F. A., M. Whiston, and P. Hitchcock (1991) Structured analysis-a draft method for writing z specifications. 6th Annual Z User Meeting, York, U.K.

Randell, G. (1991) Data flow diagrams in z. 5th Annual Z User Meeting, Oxford, U.K.

Renold, A. (1988) Jackson system development for real time systems. In: *JSP and JSD: the Jackson Approach to Software Development*, pp. 235-268. J. Gameron Ed. IEEE Computer Society Press, Washington, D.C., U.S.A.

Semmens, L. T. and P. Allen (1990) Using yourdon and z: an approach to formal specification. 5th Annual Z User Meeting, Oxford, U.K.

Semmens, L. T., R. B. France, and T. W. Docker (1992) Integrating structured analysis and formal specification techniques. *Computer Journal*, **35**(6), 600-610.

Smith, P., W. L. Yeung, and G. Topping (1991) A formalisation of jackson system development. 3rd International Conference on Software Engineering for Real Time Systems, Cirencester, U.K.

Spivey, J. M. (1992) *The Z-Notation: A Reference Manual*, 2nd Ed. Prentice-Hall, Englewood Cliffs, NJ, U.S.A.

Sridhar, K. T. and C. A. R. Hoare (1989) JSD expressed in csp. In: *JSP and JSD: the Jackson Approach to Software Development*, pp. 334-363. J. Cameron Ed. IEEE Computer Society Press, Washington, D.C., U.S.A.

Storer, R. (1987) *Practical Program Development Using JSP*. Blackwell Scientific, Oxford, U.K.

Wing, J. M. (1987) A specifier's introduction to formal methods. *IEEE Computer*, **23**(9), 8-21.

Yeh, R. T. (1990) Microstep: a business definition language system. In: *Modern Software Engineering: Foundation and Current Perspective*, pp. 502-536. P.A. Ng and R.T. Yeh Eds., Van Nostrand Reihold, New York, NY, U.S.A.

結合傑克森系統發展方法與正規語言之研究

李允中　潘健一　薛念林　黃爲德

國立中央大學資訊工程研究所

摘　要

在本論文中，我們提出了使用Z當作正規語言來表達傑克森系統發展方法之需求規格（稱之爲JSDZ），其中JSD可視爲一個可幫助分析問題領域的機制。在JSDZ方法中，傑克森系統發展方法中的功能處理元（function process）是由運算框架（operation schema）來表示。把一個模式處理元（model process）當作是一個主動實體（active entity）來看待。而主動實體需要一個運算（operation），這個運算能夠產生出新的成員（instance）而加入現有的成員集中。因此，我們把一個模式處理元轉換成一個狀態框架，而且其相關的運算轉換成爲運算框架，並且加上一個能夠由運算框架所更改的成員集。結構圖方面則先轉換成狀態轉移圖再轉換成相關的Z規格。程序之間基數（cardinality）關係的表示方法是根據合併（rough merge）表示成Z符號的觀念。我們將以一個租車系統來說明這些轉換。這種結合傑克森系統發展方法與Z正規化規格，使其兼具了圖形表示與正規語意表示的方式，主要是提供兩個優點。首先，JSD規格可視爲幫助推導出Z正規化規格的結構化機制貢獻是提供有系統的方法來產生兼具有結構化及正規化的需求規格。