A Rule-Based Approach to Producing Z Specifications from Jackson System Development

Jonathan Lee* and Jiann-I Pan†
Software Engineering Lab., Department of Computer Science and Information Engineering, National Central University, Chungli, Taiwan

In this paper, we propose a rule-based approach (called JSDZ) to producing Z specifications from Jackson system development (JSD) specifications automatically. In JSDZ, JSP is to serve as the structuring mechanism to help the analysis of problem domains, and Z is to express the formal specifications of JSD artifacts. Several criteria are identified for comparing specifications generated from JSDZ and Z. The bringing together of diagrammatical and text elements of JSD specifications in Z notations offers two major benefits. First, JSD can be seen both as a structuring mechanism that helps in deriving Z specifications, and as a preliminary that assists in ascertaining the clients requirements. Second, Z specifications make it easier to identify omissions or errors.

1. INTRODUCTION

Because formal methods find increasing acceptance within the software industry, there is a growing body of research and user interest in the integration of informal and formal methods.[31,15,3,17,16] Informal methods have advantages for requirements elicitation, ease of understanding, communication, and supporting software development process through structuring mechanisms. Meanwhile, formal methods provide conciseness, clarity, and precision, and are more suitable for analysis and verification. Therefore, as are advocated by Gehani,[9] Bowen and Hinchey,[3] system specifications, ideally, should include both formal and informal specifications, and that a method for combining both specifications is needed.

However, as a result, a number of researchers have reported progress toward the successful integration of formal and information methods such as combining structured analysis (SA) with formal specifications (e.g., Refs. 7, 8, 19,

*Author to whom correspondence should be addressed. E-mail: yjlee@se01.csie.ncu.edu.tw.
†E-mail: pan@se01.csie.ncu.edu.tw.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 13, 587–611 (1998)
 CCC 0884-8173/98/070587-25

24, 25), integrating object-oriented analysis (OOA) and formal specifications (e.g., Refs. 10, 12, 2, 18), as well as transforming JSD to formal notations (e.g., Refs. 28).

The focus of this paper is on the use of Z[27] as the formal notation to express JSD requirements specifications, called JSDZ. The transformation in JSDZ is achieved using a rule-based approach (in CLIPS Ref. 5) that enables the derivation of Z specifications from JSD artifacts automatically. The transformation heuristics are outlined below.

In JSDZ, a model process in the modeling phase 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. State invariants are captured in the entity-action list as constraints, which are then described as state invariants in the state schema. The time-ordering relationship in the structure diagram is transformed into a collection of transition constraints using the axiomatic description to serve as a global constraint to restrict the ordering relationship of the related operations.

In the network phase, a function process in the network phase is manifested through an operation schema. State vectors and their related functions are treated as a collection of query operation schemas. Data streams are implemented by read and write operations. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merges. These are illustrated using the problem domain of the library system.[30]

We first give an overview of JSD in the next section. The proposed approach that guides the mapping from JSD specifications to Z notations is fully discussed in Section 3. Related work is introduced in Section 4. In Section 5, several important issues such as structuring mechanism, criteria for comparing specifications, etc. are discussed. Finally, we summarize the benefits of our approach and outline our future research plans.

2. JACKSON SYSTEM DEVELOPMENT OVERVIEW

Jackson system development (JSD)[14] is a method originated by Michael Jackson for software development. Variations of JSD have been proposed. We will base our discussion on the features discussed in Ref. 4. JSD specifications are mainly composed of a distributed network of sequential processes. Each process can have its own local data. The communication between processes is achieved by reading and writing messages (data stream) and by read-only access to one another's data (state vector). The 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 the JSD method.

- Modeling phase: Model processes (or entities) and their actions are selected and defined.

- Network phase: The relationships between model processes and function processes are established, and are represented by the system specification diagram (SSD).
- Implementation phase: The processes and their data are fitted onto the available processor and storage devices.

In the following sections, we will only concentrate on the modeling and network phases. The library system is used as an example to illustrate both the JSD methodology and the proposed approach for transforming JSD specifications to Z. The problem description of the library system is summarized below.[30]

Consider a small library database with the following transactions:

(1) Check out a copy of a book. Return a copy of a book.
(2) Add a copy of a book to the library. Remove a copy of a book from the library.
(3) Get the list of books by a particular author or in a particular subject area.
(4) Find out the list of books currently checked out by a particular borrower.
(5) Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

(c_1) All copies in the library must be available for check-out or be checked out.
(c_2) No copy of a book may be available and checked out at the same time.
(c_3) A borrower may not have more than a predefined number of books checked out at one time.

2.1. 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 a system. To extend JSD, we incorporate constraints into the entity-action list so that the state invariant can be included in the state schema.‡ The second constraint (c_2), for example, is defined as the exclusive or of INLIB and ONLOAN to indicate that no copy of a book can be both in the library and on loan at the same time. The entity-action lists for the entities *Book* and *User* in the library system are shown in Figure 1.

In JSD specifications, a model process is composed of a set of ordering actions and is denoted by a structure diagram. A structure diagram is a tree structure, the leaves are actions, all other components describe either sequence, iteration, or selection relationships between actions or between group of actions. Iterations are denoted by the "*" in the top right corner of the constituent box, and the selections are denoted by the "°". Figure 2(a) shows that A is a sequence

‡Similar ideas about the inclusion of constraints in an object representation can be found in BON.[22]

Entity	Attributes	Type
Book	ID	BOOK_ID
	DATE_ADDED	DATE
	TITLE	TEXT
	SUBJECT	TEXT
	AUTHOR	SET OF TEXT
	LAST_BORROWER	USER_ID
	DATE_LENT	DATE
	DATE_RETURNED	DATE
	INLIB	BOOLEAN
	ONLOAN	BOOLEAN
	DATE_REMOVED	DATE

Action	Attributes	Type
Add	ID	BOOK_ID
	DATE_OF_ADDED	DATE
	TITLE	TEXT
	SUBJECT	TEXT
	AUTHOR	SET OF TEXT
Check_out	ID	BOOK_ID
	BORROWER	USER_ID
	DATE_OF_LEND	DATE
Give_back	ID	BOOK_ID
	DATE_OF_RETURN	DATE
Remove	ID	BOOK_ID
	DATE_OF_REMOVE	DATE

Constraints	Definition
$\{x \mid x \in BOOK\_ID\} = \{x \mid x \in BOOK\_ID \wedge x.INLIB=TRUE\} \cup \{x \mid x \in BOOK\_ID \wedge x.ONLOAN=TRUE\}$	All copies in the library must be available or be checked out.
$INLIB=TRUE \wedge ONLOAN=FALSE \vee INLIB=FALSE \wedge ONLOAN=TRUE$	No copy of a book may be both available and checked out at the same time.

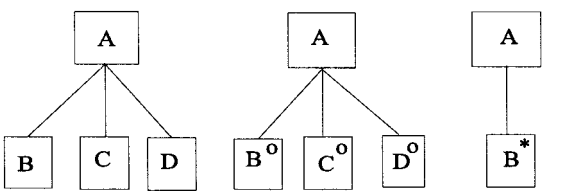
Entity	Attributes	Type
User	ID	USER_ID
	NAME	TEXT
	LIMIT	N
	LOAN_COUNT	
	DATE_OF_LAST_ACTION	DATE
	BOOK_LIST	SET OF BOOK_ID

Action	Attributes	Type
Join	ID	USER_ID
	NAME	TEXT
	LIMIT	N
	DATE_OF_JOIN	DAT
Lend	ID	USER_ID
	DATE_OF_LEND	DATE
Return	ID	USER_ID
	DATE_OF_RETURN	DATE
Leave	ID	USER_ID
	DATE_OF_LEAVE	DATE

Constraints	Definition
LIMIT ≤ LOAN_COUNT	A borrower may not have more than a predefined number of books checked out at one time.

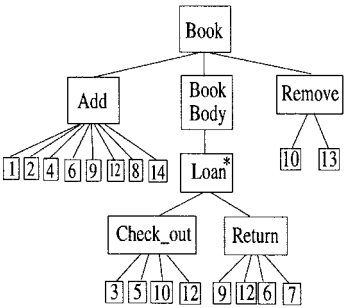
Figure 1. Entity-action lists of the library problem.

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.



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

Figure 2. Structured diagrams: (a) sequence, (b) selection, (c) iteration.

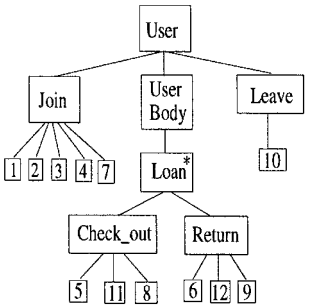


1. DATE_ADDED:= add.DATE_OF_ADDED
2. TITLE:=add.TITLE
   AUTHOR:=add.AUTHOR
   SUBJECT:=add.SUBJECT
3. LAST_BORROWER:=check_out.BORROWER
4. LAST_BORROWER:=UNDEFINED

5. LEND_DATE:=check_out.DATE_OF_LEND
6. LEND_DATE:=UNDEFINED
7. RETURN_DATE:=return.DATE_OF_RETURN
8. RETURN_DATE:=UNDEFINED
9. INLIB:=TRUE
10. INLIB:=FALSE
11. ONLOAN:=TRUE
12. ONLOAN:=FLASE

13. REMOVE_DATE:=remove.DATE_OF_REMOVE
14. REMOVE_DATE:=UNDEFINED

(a)



1. NAME:= join.NAME
2. LIMIT:= join.LIMIT
3. JOINING_DATE:= join.DATE_OF_JOIN
4. LOAN_COUNT:= 0
5. LOAN_COUNT:= LOAN_COUNT+1
6. LOAN_COUNT:= LOAN_COUNT-1

7. DATE_OF_LAST_ACTION:= join.DATE_OF_JOINING
8. DATE_OF_LAST_ACTION:= check_out.DATE_OF_LEND
9. DATE_OF_LAST_ACTION:= return.DATE_OF_RETURN
10. DATE_OF_LAST_ACTION:= leave.DATE_OF_LEAVE
11. Add BOOKID to BOOKLIST
12. Remove BOOKID from BOOKLIST

(b)

Figure 3. Structure diagrams for the library system: (a) the structure diagram of book, (b) the structure diagram of user.

Notice that the first action associated with an entity in the structure diagram plays a role of creating instances of the associated entity; whereas, the last action is in charge of destroying all the instances of the entity.

Having built a model process, we need to define data items and basic operations to describe the detailed 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 structure text. Figure 3 shows the result of the modeling phase, namely, structured diagrams that include basic operations for the book and user entities.

2.2. 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., *Book* in Fig. 4) to build the network of potentially concurrent processes. The network is described in a system specification diagram (SSD). The SSD notations are shown in Figure 5. In this phase, three

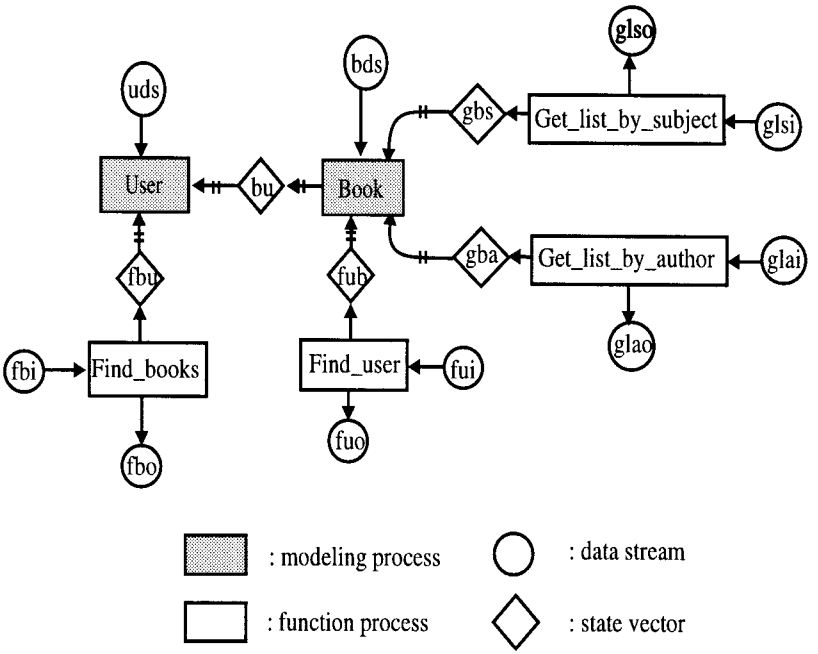


Figure 4. SSD for the library systems.

different types of function process can be included into the network: input function process, information function process, and interactive function process.

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, reject them by producing an error message otherwise. All the input processes together constitute the input subsystem.

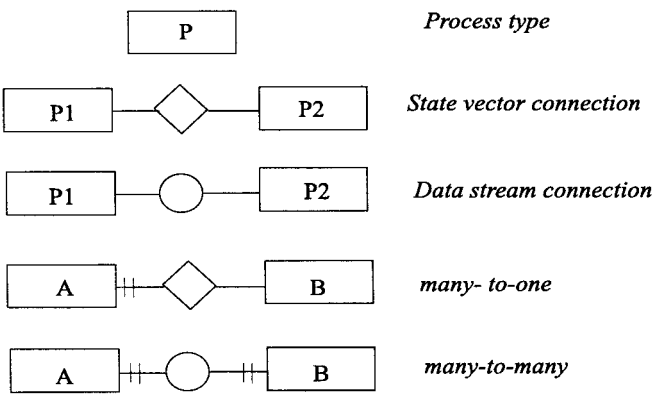


Figure 5. SSD notations.

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, actions, action attributes, and entity attributes. If this is not possible, the model must be elaborated by defining new actions and entity attributes. In Figure 4, the *Find_books*, *Find_user*, *Get_list_by_subject*, and *Get_list_by_author* are information function processes. *Find_books* receives a user identity and outputs the books list of books that the user borrowed. *Find_user* receives a book identity to find a particular user who has borrowed the book. *Get_list_by_author* gets all the books that were written by a particular author. *Get_list_by_subject* gets all the books that are associated with a particular subject.

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 model processes to generate the system actions. Interactive function processes are like information function processes, except that they produce inputs into the model processes instead of system outputs. There are two communication mechanisms between processes in JSD.

- State vector: A state vector (SV) connection constitutes a read-only access from a process to the local data of another process.
- Data stream: A data stream (DS) is a first-in-first-out data queue with infinite capacity.

In Figure 4, the *fbu*, *gbs*, etc. are state vectors, and the *fbi*, *fbo*, etc. are data streams. For example, a user identity is stored in the *fbi* data stream, and read in by the *Find_books*. The *Find_books* process inspects the data in *User* to retrieve the user's borrowed books list, and puts the list on the *fbo* data stream.

The internals of a function process (called a process structure) are developed with JSP (see Fig. 6). The process structure is deduced from its in and out data (time) structure. All the operations required to produce the output are first

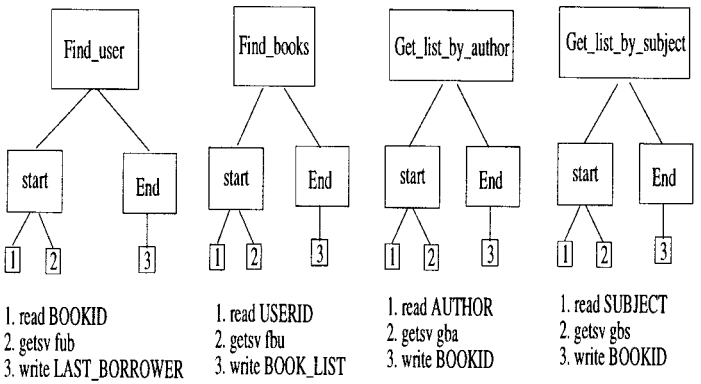


Figure 6. Process structures in the library system.

defined in an operation list, and allocated to the appropriate components in the process structure. Conditions have to be specified for each iteration and selection component. The cardinality relationship between processes could be either one-to-one, one-to-many, or many-to-many (see Fig. 5), which is to capture the notion of multiplicity of a process (i.e., multiple instances). After completing all function processes and communicating them with modeling processes using SV and DS, an SSD is established (see Fig. 4 for a complete SSD of the library system).

In the next section, the proposed approach JSDZ is fully discussed and is illustrated step-by-step using the library system.

3. EXPRESSING JACKSON SYSTEM DEVELOPMENT SPECIFICATIONS IN Z NOTATIONS

In JSDZ, a model process in the modeling phase 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 transformed into a state scheme, and its related operations are converted into operation schemas with an instances set that can be modified by the operations. State invariants are captured in the entity-action list as constraints, which will then be described as state invariants in the state schema. The time-ordering relationship in a structure diagram is transformed into a collection of transition constraints and then grouped under an axiomatic description. In the network phase, a function process in JSD specifications is manifested by an operation schema. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merges.

JSDZ is implemented using JAVA and CLIPS, JAVA is used to build a uniform user interface. CLIPS is used to implement the proposed rule base to transform JSD artifacts into Z specifications. The inputs to the rule base are a collection of initial knowledge, represented as facts in CLIPS, based on the entity-action list, the structure diagram and the system specification diagram from the JSD artifacts. The outputs of JSDZ are Z specifications in LaTeX format. Windows of the JSDZ tool are shown in Figures 7 and 8.

3.1. Transferring the Jackson System Development Artifacts into CLIPS Facts

The rule base in JSDZ encompasses two main modules: modeling and networking module, each of which consists of a collection of heuristics (described in Sections 3.2 and 3.3) to facilitate the transformation. The JSD artifacts are represented as facts to form the input for the JSDZ rule base. In JSDZ, the related information in an entity, such as the entity name, attributes, actions, basic operations, etc., are captured using a *deffacts* in CLIPS. Referring to our example, part of the information in the *BOOK* entity can be represented

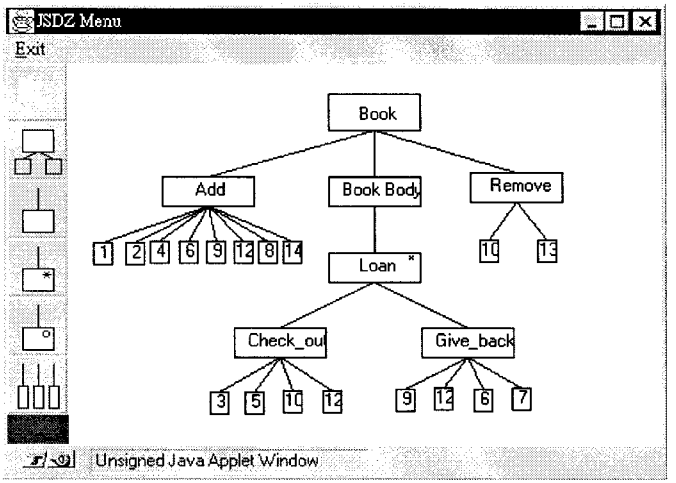


Figure 7. A window of modeling phase in JSDZ tool.

in the *Book* deffacts as follows§:

```
(deffacts MAIN::Book
   EntityActions (EntityName Book)
      (ActionNames add check\_out give\_out give\_back
       remove))
   (EntityAttributes (EntityName Book)
      (Attributes ID DATE\_ADDED TITLE SUBJECT AUTHOR
       LAST\_BORROWER
                 LEND\_DATE RETURN\_DATE INLIB ONLOAN
                 REMOVE\_DATE)
      (Types BOOK\_ID DATE TEXT TEXT SET-TEXT USER\_ID DATE
       DATE BOOLEAN BOOLEAN DATE))
(Action (EntityName Book) (ActionName add)
       (Attributes ID DATE\_OF\_ADDED TITLE SUBJECT AUTHOR)
       (Types BOOK\_ID DATE TEXT TEXT SET-TEXT))
      (BasicOperation (EntityName Book) (ActionName add)
       (Op DATE\_ADDED "=" add.DATE\_OF\_ADDED))
      (BasicOperation (Entity Name Book) (ActionName add)
       (Op TITLE "=" TITLE))
                 :
                 :
)
```

§For details about the CLIPS rules and facts in JSDZ, see Appendix A.

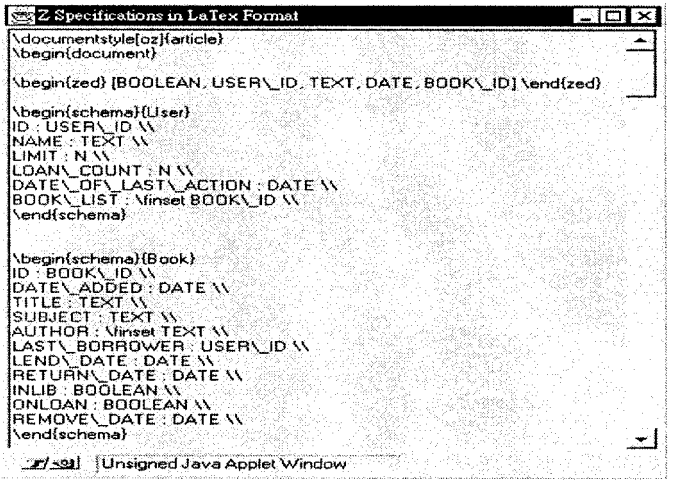


Figure 8. An output of Z specifications.

3.2. Modeling Phase

1. Define given sets for all types of attributes based on the entity-action list. Consider the library system, the types needed in the system are described as follows:

$$[BOOLEAN, BOOK\_ID, USER\_ID, TEXT, DATE]$$

$$REPORT ::= "remove\_date\ is\ unknown" \mid "last\_borrower\ is\ unknown" \mid "lend\_date\ is\ unknown" \mid "return\_date\ is\ unknown"$$

2. Define a state schema, treated as a type for each entity where its signature is depicted using the attributes in the entity-action list. From the same example, the *Book* state schema is defined for the entity *Book*.

Book

id: $BOOK\_ID$
$date\_added$: $DATE$
$title$: $TEXT$
$subject$: $\mathbb{F}\ TEXT$
$author$: $\mathbb{F}\ TEXT$
$last\_borrower$: $USER\_ID$
$date\_lent$: $DATE$
$date\_returned$: $DATE$
$inlib$: $BOOLEAN$
$onloan$: $BOOLEAN$
$date\_removed$: $DATE$

3. Create a data-store state schema to represent an instances set for each entity. In each data store, the schema defined for each entity in the previous step is treated as a type, and an identity is required to uniquely identify each instance. In the same example, there is a data-store state schema *Book_SET*, where *books* is a subset of the type *BOOK* and *idbook* is an identifier. Thus,

$$
\begin{array}{|l}
\hline
\textit{Book\_SET} \\
\textit{books} : \mathbb{F}\ \textit{Book} \\
\textit{idbook} : \textit{BOOK\_ID} \rightarrowtail\!\!\!\!\!\!\!\!\;+\;\textit{Book} \\
\hline
\forall b : \textit{Book} \bullet b \in \textit{books} \quad \Rightarrow \quad b.\textit{id} = \textit{idbook}^{-}1(b) \\
\text{ran } \textit{idbook} = \textit{books} \\
\hline
\end{array}
$$

4. Specify the state invariant in the data-store state schema. In JSD, the state invariant is not considered at all. Based on our enhanced version of the entity-action list, constraints in the list are utilized to specify the state invariants explicitly. In our example, the (c_2) constraint is converted into the formula $\forall b$: $Book \bullet b \in books \Rightarrow ((b.inlib = true \wedge b.onloan = false) \vee (b.inlib = false \wedge b.onloan = true))$. Therefore,

$$
\begin{array}{|l}
\hline
\textit{Book\_SET} \\
\textit{books} : \mathbb{F}\ \textit{Book} \\
\textit{idbook} : \textit{BOOK\_ID} \rightarrowtail\!\!\!\!\!\!\!\!\;+\;\textit{Book} \\
\hline
\forall b : \textit{Book} \circ b \in \textit{books} \quad \Rightarrow \quad b.\ \textit{id} = \textit{idbook}^{-}1(b) \\
\text{ran } \textit{idbook} = \textit{books} \\
\textit{books} = \bigcup\{b : \textit{Book} \mid b \in \textit{books} \bullet b.\textit{inlib} = \textit{true}\} \\
\qquad\quad \cup \bigcup\{b : \textit{Book} \mid b \in \textit{books} \bullet b.\textit{onloan} = \textit{true}\} \\
\forall b : \textit{Book} \bullet b \in \textit{books} \quad \Rightarrow \\
((b.\textit{inlib} = \textit{true} \wedge b.\textit{onloan} = \textit{false}) \vee (b.\textit{inlib} = \textit{false} \wedge b.\textit{onloan} = \textit{true})) \\
\hline
\end{array}
$$

5. Initialize the data-store schema for each entity to specify the initial state. The initialization of a system can be regarded as a special kind of operation that creates a state out of nothing; there is no before state, simply an after state with its variables decorated. The initial state is described by means of the schema *Init_Book_SET′*, where the schema *Init_Book_SET* is defined as follows,

$$
\begin{array}{|l}
\hline
\textit{Init\_Book\_SET} \\
\textit{books} : \mathbb{F}\ \textit{Book} \\
\textit{idbook} : \textit{BOOK\_ID} \rightarrowtail\!\!\!\!\!\!\!\!\;+\;\textit{Book} \\
\hline
\textit{books} = \varnothing \\
\textit{idbook} = \varnothing \\
\hline
\end{array}
$$

6. Define an operation schema for each action of an entity, and its signature according to the entity-action list. If an action changes its associated data store, attach the Δ 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 signature portions of the operation schemas for *add* and *remove* actions of the book entity

are described below, respectively,

add

$\Delta Book\_SET$
$book : Book$
$rep\_remove\_date! : Report$
$rep\_return\_date! : Report$
$rep\_lend\_date! : Report$
$rep\_last\_borrower! : Report$
$id? : BOOK\_ID$
$date\_of\_added? : DATE$
$title? : TEXT$
$subject? : TEXT$
$author? : \mathbb{F}\ TEXT$

remove

ΔBo_SET
$book : Book$
$id? : BOOK\_ID$
$date\_of\_remove? : DATE$

7. Include the basic operations of the action in the predicate part of the operation schema to illustrate the postconditions of the operation. To reflect the semantics of the first and last actions of an entity, the creation and destruction of instances of the entity also need to be specified. For example, the *add* action is treated as an operation that can create new instances to be included into the existing instances set, *BOOK_SET*, namely, $books' = books \cup \{book\}$; while, the *remove* action will delete an instance from the instances set, that is, $books' = books \setminus \{book\}$ and $idbook' = \{id?\} \lhd idbook$. Hence,

add

$\Delta Book\_SET$
$book : Book$
$rep\_remove\_date! : Report$
$rep\_return\_date! : Report$
$rep\_lend\_date! : Report$
$rep\_last\_borrower! : Report$
$id? : BOOK\_ID$
$date\_of\_added? : DATE$
$title? : TEXT$
$subject? : TEXT$
$author? : \mathbb{F}\ TEXT$

$id? \notin \mathrm{dom}\ idbook$
$book.id' = id?$
$book.date\_added' = date\_of\_added?$
$book.title' = title?$
$book.subject' = subject?$
$book.author' = author?$
$book.inlib' = true$
$book.onloan' = false$
$idbook' = idbook \oplus \{id? \mapsto book\}$
$books' = books \cup \{book\}$
$rep\_last\_borrower! = "last\_borrower\ is\ unknown"$
$rep\_lend\_date! = "lend\_date\ is\ unknown"$
$rep\_return\_date! = "return\_date\ is\ unknown"$
$rep\_remove\_date! = "remove\_date\ is\ unknown"$

remove

$\Delta Book\_SET$
$book : Book$
$id? : BOOK\_ID$
$date\_of\_remove? : DATE$

$id? \in \text{dom } idbook$
$book = idbook(id?)$
$book.inlib' = false$
$book.remove\_date' = date\_of\_remove?$
$books' = books \setminus \{book\}$
$idbook' = \{id?\} \lhd idbook$

8. Convert the time-ordering relationship in each structure diagram into a collection of transition constraints‖ and include these constraints in an axiomatic description to serve as a global constraint for restricting actions involved in each entity. For example, the time-ordering relationship for the book entity in Figure 3 can be represented using regular expressions as *Add*,(*Check_out*, *Give_back*)*, *Remove*), which can then be converted into three transition constraints. That is, one constraint is an *add* action that performs the *checkout* or *remove* actions that happened; another constraint is a *checkout* action that performs the *remove* action that happened; the other constraint depicts a *return* action that performs the *remove* or *checkout* actions that occurred. A basic type [STATE] and a free type *Book_actions* are defined first. In the axiomatic description, three formulas are specified in the predicate part to show the constraints on transitions of operations of the *Book* entity. Thus

$[STATE]$
$Book\_actions ::= Add \langle\langle add \rangle\rangle \mid Checkout \langle\langle checkout \rangle\rangle$
$\mid Return \langle\langle return \rangle\rangle \mid Remove \langle\langle remove \rangle\rangle$

$R : \mathbb{P}(STATE \times Book\_actions \times STATE)$

$\forall s0, s1 : STATE; \exists s2, s3 : STATE; (s0, add, s1), (s1, checkout, s2), (s1, remove, s3) \in R$
$\bullet (s0, add, s1) \Rightarrow (s1, checkout, s2) \lor (s1, remove, s3)$
$\forall s1, s2 : STATE; (s1, checkout, s2), (s2, return, s1) \in R$
$\bullet (s1, checkout, s2) \Rightarrow (s2, return, s1)$
$\forall s1, s2 : STATE; \exists s3 : STATE; (s2, return, s1), (s1, remove, s3), (s1, checkout, s2) \in R$
$\bullet (s2, return, s1) \Rightarrow (s1, remove, s3) \lor (s1, checkout, s2)$

3.3. Network Phase

Data Stream

A data stream is a first-in-first-out data buffer with infinite capacity, and two operations are defined in a data stream: *read* and *write*. To express the data stream in Z, we have distinguished two types of data stream: (1) if a data stream

‖This technique is adopted from Ref. 29.

is connected to a function process, then the input–output variables are declared in the schema of the function process; and (2) if it is connected to a model process (an entity), then the information in the entity is included in the schema. In Figure 4, for example, the data stream *glai* is connected to the function process *get_list_by_author*, therefore, the schema for the function process would include the input variable *author*? (see below).

State Vector

A state vector connection implies that a read only access from a process to the local data of another process is performed, which does not change any of the state variables; and therefore, any process (either model or function) that performs the operation needs to attach the Ξ notation to the related data stores. In addition, a *getsv* operation is performed implicitly using the identity function in the function process that is connected to the state vector. In our example, *gba* is an SV that inspects data from the *Book* entity for the function process *get_list_by_author*. The *get_list_by_author* operation schema includes $\Xi Book\_SET$ in its signature part, and uses *idbook*(*bookid*) to retrieve all the information about the *Book* (see below).

Function Process

The transformation of a function process into an operation schema will consider its connected data streams and state vectors, as well as its process structure. Input and information function processes will not change the related state space, while interactive function processes will. Therefore, we usually attach Ξ notation to the related data store for input and information function processes, and Δ for the interactive function process. Refer to the same example (see Fig. 6), the *get_list_by_author* function process receives the user's query from the data stream *glai*, and answers the list of books by a particular author as an output. Thus

get_list_by_author

$$\Xi Book\_SET$$
$$author? : TEXT$$
$$rep! : \mathbb{F}\ BOOK\_ID$$

$$author? \in \bigcup \{bookid : BOOK\_ID \bullet idbook(bookid).author\}$$
$$rep! = \{bookid : BOOK\_ID \mid author? \in idbook(bookid).author\}$$

Merge

In a data stream connection the initiative for the communication comes from the writing process. The reading process must consume the complete stream of data records. If a process reads data streams from different processes, the developer must specify how the data streams should be merged. The merge

strategy defines the sequence in which the receiving process reads the data records.

There are two types of merge: 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 merges, we use a set as an internal buffer in the reading process to take in the data from each data stream. If all data streams are empty, the reading process is suspended, otherwise the set collects the data from all data streams, and clears each data stream. The output sequence of the data in the set is not determined until the 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 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 being suspended or not depends on the problem domain.

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) can assume different types as well.

Cardinality

In JSD, if a process communicates with several instances of another process, the notion of cardinality is applied. In Figure 5, the data streams from the processes of type A are rough merged to form a single input DS in the case of the many-to-one situation, or a multiple inputs DS for many-to-many into process B.

4. RELATED WORK

A number of researchers have reported progress toward the successful integration of formal and informal methods (i.e., methods integration). Semmens et al. have conducted a comparative study of the related work.[26] However, their study only concentrated on approaches in combining structured analysts and formal specification techniques. To provide a more comprehensive view of the state-of-the-art in this line of research, we have classified three categories for methods integration based upon the different informal analysis approaches used.

- From structured analysis (SA) to formal specifications: for example, Polack,[23] Mander and Polack,[20] and Semmens and Allen's[25] work on integrating SA and Z; Randell translated DFD into Z[24]; France combined SA and algebraic specifications[7]; Fraser et al. integrated SA with VDM[8]; Liu combined extended DFD and VDM[19]; and Moulding and Smith combined CORE with VDM and CSP.[21]
- From object-oriented analysis and design (OOA/D) to formal specifications: typical examples are Giovanni and Iachini's work on including Z in HOOD,[10] Hammond integrated Shaler Mellor's OOA with Z,[12] Bourdeau and Cheng

provided a formal semantics for OMT models,[2] and Lee et al. combined Bailin's OOS with Z.[18]

- From JSD to formal notations: Sridhar and Hoare transformed structure diagrams to CSP.[28]

4.1. Structured Analysis to Formal Specifications

In Semmens and Allen's work,[25] there are two phases involved in the analysis. First, a data model of the system is developed, expressed first as an entity relationship diagram (ERD) and then as a Z state schema which is systematically derived from the ERD. In the second phase, the process model is built. A semiformal model is expressed using DFD, then the semantics of each of the processes in the DFD is specified using Z operation schemas. An entity is represented in an entity type schema and an entity instance set schema. Attributes of an entity can be referred to in an entity dictionary. A relationship can be represented as a relationship schema. Partial function, partial injection, or relation are used to specify the cardinality and connectivity relationships between two entities. These schemas build the state space of the system in Z. Data flows between processes in DFD can be shown as input and output variables in an operation schema.

Mander and Polack took a different view, seeing the systems analysis as an important but partially independent precursor to the formal definition.[20] 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 being exclusively derived from it. The basic components of the Z definition: state and process specifications will be 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, operations are formed from function definitions which are derived from function description, I/O structure, effect correspondence diagram, and entity life histories.

Randell focused on how to translate a data flow diagram into Z.[24] 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 processes in DFD. 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, which must be obtained from elsewhere.

France provided a semantically extended DFD as a control-extended DFD (C-DFD) associated with an algebraic specification technique.[7] The syntactic aspect is concerned with its pictorial representation of C-DFD, and the semantic aspect is about the behavioral interpretation of its C-DFD. A C-DFD's data domain is defined by specifiers using a data description language. State specifications and label specifications are defined to describe the data flow in DFD,

and DataStore state specifications are used to describe the data stores in DFD. Operators in accessing these data stores are 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 external entity and the system behavior.

Fraser et al. integrated VDM and structured analysis.[8] Two approaches were described in the paper: (1) structured analysis as a cognitive guide to 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 and the later is automated. The information from the entity dictionary is used to build the information domain. Operators rd and wr indicate the data flow between processes. Precondition and postcondition are used to specify the process's functionality.

Moulding and Smith focus on the use of the VDM with CORE and express the role of CORE in CSP.[21] That is, a VDM specification to define the processing semantics of the actions, and a CSP specification to identify the real-time collective behavior of those actions. The data flows between viewpoints which identified in a CORE model form 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 the VDM-SL modules, in which the actions of the viewpoints are modeled as VDM operations. Internal data flows within a viewpoint were 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 was specified in single viewpoint modeling (SVM) and combined viewpoint modeling (CVM) stages of CORE. Each CORE action, channel, and pool was represented by a CSP process. Processes were combined using the CSP parallel composition operator (||) to specify the overall behavior of the SVM. A CVM was a process composed from the channels, pools, and actions. Similarly, the compositional features of CSP were used to describe a CVM.

4.2. Object-Oriented Analysis to Formal Specifications

Giovanni and Iachini combined HOOD and Z, in which HOOD was used as a structuring mechanism to guide the construction of Z specifications.[10] In this article, WHAT, WithWHAT, and HOW specifications are used to build the formal specification. A WHAT specification may consist of more than one schema and is used to describe an object. These schemas can be partitioned into state schemas and operation schemas. A state schema is used to specify an object's data model. However, information in specifying an object is insufficient. Each method of an object is described in a separated operation schema.

Hammond integrated Shaler-Mellor's OOA and Z.[12] There are three phases in the OOA: (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, an entity

instance set schema, and a relationship schema. In the state transition modeling, a schema event is 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 precondition of an input event and the initial state, and the postcondition of components' values of the after state and output events. An active object is defined to be an object that requires an operation on its data store to add a new instance to the collection of the existing instances. In the process modeling, processes accessors and event generators are identified from Z transition specifications.

Bourdeau and Cheng presented a formal semantics for the OMT object model notations.[2] Object models and instance diagrams in OMT are formalized as algebraic specifications and algebras, respectively. Instance diagrams can be used to provide the 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) to compute the algebraic specifications of the object model and look at a set of algebras that satisfy this specification, and (2) to 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) was chosen as the algebraic specification language. A trait in LSL was used to represent an abstract data type in the object model, and an Σ-algebra was generated by this trait. Traits can be used to depict classes and associations. The related classes within an association was 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.[18] proposed an integration of Bailin's object-oriented specification and Z, called OOSZ. In OOSZ, OOS was 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, and instance creation operation schema and an abstract operation schema, meanwhile, a function process is converted into an operation schema in Z specifications.

4.3. Jackson System Development to Formal Notations

Sridhar and Hoare demonstrated how to express JSD in CSP through several examples in Ref. 28. In their approach, only the structure diagram (SD) is considered in the transformation process. An entity modeled in the SD was described by its life history, and can be defined by two formulas in CSP: the first one to depict the creation of an entity, and the second one to describe the remainder of the history. Adding new actions into the second formula later is permitted. The iterative mechanism between CSP and JSD is provided. That is,

a new entity process can be defined in CSP, and then backward to modify the SD. No systematic transformation heuristics are provided.

Compared with all these approaches, JSDZ offers two important advantages: (1) artifacts 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.

5. DISCUSSION

5.1. Structuring Mechanism

One of the major criticisms of formal methods is that they are not so much "methods" as formal systems. They fail to support many of the methodological aspects of the more traditional structured-development methods. However, as a result, Woodcock[32] proposes the use of schema language to structure Z specifications. Diller[6] proposes a step-by-step procedure in writing Z specifications. However in these two approaches, little emphasis is placed on the underlying development model and little guidance is provided as to how development should proceed. Recently, D. Jackson and M. Jackson[13] proposed the use of views for structuring Z specifications. The basic idea is to decompose a problem in parallel into several views (subproblems) to fit into their applicable problem frames, which are then used to guide the derivation of Z specifications.

Bowen and Hinchey[3] claim that the integration of formal and informal methods leads to a "true" development method that fully supports the software life cycle and allows developers to use more formal techniques in the specification and design phase, supporting refinement to executable code and proof-of properties. JSDZ can be considered as one of the attempts to provide a systematic approach to supporting the structuring of Z specifications through JSD.

5.2. Criteria for Comparison

In Ref. 1, Bicarregui and Ritchie propose three criteria: invariants, frames, postconditions, for comparing VDM and B notations. Essentially, their criteria are for the comparison of different kinds of formal specification. To compare the same type of specification (in our case, Z), we have identified four criteria; state space, state invariants, operation definition, and operation interaction.

- State space: Comparing the representation of state space in both Z and JSDZ, the main difference is that: a state schema of an entity is treated as a type and a data-store state schema as an instances set, both of which constitute the state space in JSDZ.
- State invariants: To express state invariants explicitly in JSDZ, JSD is extended by incorporating constraints into the entity-action list to specify state invariants in the state schema.

- Operation definition: To define an operation schema in JSDZ, artifacts generated by JSD are very useful. For example, basic operations in SD and process structures form the postconditions and outputs for the operation schema. In Z, error handling schemas are generated based on the information gained from preconditions of each operation, which are included in the total specification to make complete the specification of each operation. However, the error handling schema is not part of the product of JSDZ, which is a limitation of JSDZ for future improvement.
- Operation interaction: Z does not provide any mechanism for describing the interactions among operations. In JSDZ, a global transition constraint, specified using the axiomatic description, is used to represent the time ordering relationship in each SD.

5.3. Representing an Object in Z Notations

It is of interest to note that there are a number of ways to represent the concept of an entity–object in Z notations. Hall,[11] Hammond[12] and JSDZ all provide a self identity (an identifier) for an entity–object identification. Data stores are generally used to indicate the connection between an entity–object and its operations. No distinction is made between active or passive entities–objects in Hall's approach. Both JSDZ and Hammond's approach distinguish active from passive entities–objects based on the creation operation.

6. CONCLUDING REMARKS

In this paper, we proposed an integration of an informal method (JSD) with a formal notation (Z). In JSDZ, 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 structure diagram is transformed into an axiomatic description in Z specifications which is used to depict the global transition constraint. Cardinality relationships between processes are translated into Z notations based upon the notion of rough merges. Z specifications are automatically generated using a rule-based approach. In the network phase, a function process in JSD specifications is manifested through an operation schema.

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 helps in deriving Z specifications and as a preliminary that assists in ascertaining the clients requirements, and (2) Z specifications make it easier to identify omissions or errors.

We would like to thank C. K. Chang and J. Y. Kuo for their early work on this project, and Professor W. T. Huang for his invaluable comments on an earlier draft of this paper. This research was supported by National Science Council (Taiwan, R.O.C.) under grant NSC85-2213-E-008-005.

References

1. J. Bicarregui and B. Ritchie, "Invariants, frames and postconditions: A comparison of the vdm and b notations," *IEEE Transactions on Software Engineering*, **21**(2), 79–89 (1995).
2. R.H. Bourdeau and B.H.C. Cheng, "A formal semantics for object model diagrams," *IEEE Transactions on Software Engineering*, **21**(10), 799–821 (1995).
3. J.P. Bowen and M.G. Hinchey, "Seven more myths of formal methods," *IEEE Software*, **12**(4), 34–41 (1995).
4. J.R. Cameron, "An overview of jsd," *IEEE Transactions on Software Engineering*, **SE-12**(2), 222–240 (1986).
5. C. Culber et al., *CLIPS Reference Manual*, *Volume I–III*, Software Technology Branch, L. B. Johnson Space Center, NASA, June 1995.
6. A. Diller, *Z*: *An Introduction to Formal Methods*, 2nd edition Wiley, Chichester, 1994.
7. R.B. France, "Semantically extended data flow diagrams: A formal specification tool," *IEEE Transactions on Software Engineering*, **18**(4), 329–346 (1992).
8. M.D. Fraser, K. Kumar, and V.K. Vaishnavi, "Informal and formal requirements specification languages: Bridging the gap," *IEEE Transactions on Software Engineering*, **17**(5), 454–466 (1991).
9. N. Gehani, "Specifications: Formal and informal—a case study," in *Software Specification Techniques*, N. Gehani and A.D. McGettrick, Eds., Addison-Wesley, Reading, MA, 1986, pp. 173–185.
10. R. Di Giovanni and P.L. Iachini, "Hood and z for the development of complex software systems," in *VDM and Z—Formal Methods in Software Development*, D. Bjorner, C.A.R. Hoare, and H. Langmaack, Eds., Springer-Verlag, Berlin, Germany, 1990, pp. 262–289.
11. A. Hall, "Specifying and interpreting class hierarchies in z," *Proceedings of the Eighth Z User Meeting*, Cambridge, 1994, pp. 120–138.
12. J.A.R. Hammond, "Producing z specifications from object-oriented analysis," *Proceedings of the Eighth Z User Meeting*, 1994, pp. 316–336.
13. D. Jackson and M. Jackson, "Problem decomposition for reuse," *Software Engineering Journal*, Jan. 19–30 (1996).
14. M.A. Jackson, *System Development*, Prentice-Hall, Englewood Cliffs, NJ, 1983.
15. K. Kronlof. Ed., *Method Integration*: *Concepts and Case Studies*, Wiley, Chichester, 1993.
16. J. Lee and L.F. Lai, "Verifying task-based specifications in conceptual graphs," *Information and Software Technology*, **39**(14–15): 913–923 (1998).
17. J. Lee, L.F. Lai, and W.T. Huang, "Task-based specifications through conceptual graphs," *IEEE Expert*, **11**(4), 60–70 (1996).
18. J. Lee, J.I. Pan, and W.T. Huang, "Oosz: An integration of Bailin's object-oriented analysis and formal specifications," *Journal of Information Science and Engineering*, **13** (Dec.), 517–542 (1997).
19. S. Liu, "A formal requirements specification method based on data flow analysis," *Journal of Systems and Software*, **21**, 141–149 (1993).
20. K.C. Mander and F.A. Polack, "Rigorous specifications using structured systems analysis and z," *Information and Software Technology*, **37**(5–6), 285–291 (1995).
21. M. Moulding and L. Smith, "Combining formal specification and core: an experimental investigation," *Software Engineering Journal*, Mar. 31–42 (1995).
22. J.-M. Nerson, "Applying object-oriented analysis and design," *Communications of the ACM*, **35**(9), 63–74 (1992).
23. F. Polack, "Integrating formal notations and systems analysis: Using entity relationship diagrams," *Software Engineering Journal* (Sept.), 363–371 (1992).
24. G. Randell, "Data flow diagrams in z," *Proceedings of the Fifth Annual Z User Meeting*, *Oxford*, *1991*, Springer-Verlag, Berlin, Germany, 1991, pp. 216–227.

25. L.T. Semmens and P. Allen, "Using yourdon and z: an approach to formal specification," *Proceedings of the Fifth Annual Z User Meeting*, *Oxford*, *1990*, Springer-Verlag, Berlin, Germany, pp. 228–253.
26. L.T. Semmens, R.B. France, and T.W. Docker, "Integrating structured analysis and formal specification techniques, *The Computer Journal*, **35**(6), 600–610 (1992).
27. J.M. Spivey, *The Z-Notation*: *a Reference Manual*, 2nd edition Prentice-Hall, Englewood Cliffs, NJ, 1992.
28. K.T. Sridhar and C.A.R. Hoare, "Jsd expressed in csp," in *JSP and JSD*: *the Jackson Approach to Software Development*, J. Cameron, Ed., 2nd edition IEEE Comput. Soc., Los Alamitos, CA, 1989, pp. 334–363.
29. K. Taguchi and K. Araki, "Extending z with state transition constraints," *Proceedings of COMPSAC'96*, Seoul, 1996, pp. 254–260.
30. J.M. Wing, "A study of 12 specifications of the library problem," *IEEE Software* (July), 66–76 (1988).
31. J.M. Wing, "A specifier's introduction to formal methods," *Computer* (Sept.), 8–21 (1990).
32. J.C.P. Woodcock, "Structuring specifications in z," *Software Engineering Journal* (Jan.), 51–66 (1990).

APPENDIX: JSDZ RULE BASE

```
(deffacts MAIN::Book
   (EntityActions (EntityName Book)
      (ActionNames add check\_out give\_ back remove))
   (EntityAttributes (EntityName Book)
      (Attributes ID DATE\_ADDED TITLE SUBJECT AUTHOR
                  LAST\_BORROWER
                  LEND\_DATE RETURN\_DATE INLIB ONLOAN
                  REMOVE\_DATE)
      (Types BOOK\_ID DATE TEXT SET-TEXT USER\_ID DATE DATE
       BOOLEAN BOOLEAN DATE))
   (Action (EntityName Book) (ActionName add)
      (Attributes ID DATE\_OF\_ADDED TITLE SUBJECT
       AUTHOR)(Types BOOK\_ID DATE TEXT TEXT SET-TEXT))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       DATE\_ADDED "=" add. DATE\_OF\_ADDED))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       TITLE "=" add.TITLE))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       AUTHOR "=" add.AUTHOR))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       SUBJECT "=" add.SUBJECT))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       LAST_BORROWER "=" UNDEFINED))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       LEND\_DATE "=" UNDEFINED))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       INLIB "=" TRUE))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       ONLOAN "=" TRUE))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       RETURN\_DATE "=" UNDEFINED))
      (BasicOperation (EntityName Book) (ActionName add) (Op
       REMOVE\_DATE "=" UNDEFINED))
```

```
   (Action (EntityName Book) (ActionName check\_out)
     (Attributes ID BORROWER DATE\_OF\_LEND)(Types BOOK\_ID
      USER\_ID DATE))
     (BasicOperation (EntityName Book) (ActionName
      check\_out) (Op LAST\_BORROWER "="
      check\_out.BORROWER))
     (BasicOperation (EntityName Book) (ActionName
      check\_out) (Op LEND\_DATE "="
      check\_out.DATE\_OF\_LEND))
     (BasicOperation (EntityName Book) (ActionName
      check\_out) (Op INLIB "=" FALSE))
     (BasicOperation (EntityName Book) (ActionName
      check\_out) (Op ONLOAN "=" FALSE))
   (Action (EntityName Book) (ActionName give\_back)
     (Attributes ID DATE\_OF\_RETURN)(Types BOOK\_ID DATE))
     (BasicOperation (EntityName Book) (ActionName
      give\_back) (Op INLIB "=" TRUE))
     (BasicOperation (EntityName Book) (ActionName
      give\_back) (Op ONLOAN "=" FALSE))
     (BasicOperation (EntityName Book) (ActionName
      give\_back) (Op LEND\_DATE "=" UNDEFINE))
     (BasicOperation (EntityName Book) (ActionName
      give\_back) (Op RETURN\_DATE "="
      give\_back.DATE\_OF\_RETURN))
   (Action (EntityName Book) (ActionName remove)
     (Attributes ID DATE\_OF\_REMOVE)(Types BOOK\_ID DATE))
     (BasicOperation (EntityName Book) (ActionName remove)
     (Op INLIB "=" FALSE))
     (BasicOperation (EntityName Book) (ActionName remove)
     (Op REMOVE\_DATE "=" remove.DATE\_OF\_REMOVE))
)

(defrule MODELLING::DefineStateSchema
   (phase DefineStateSchema)
   ?att ← (AttributesTemp ?af $?ar)
   ?typ ← (TypesTemp ?tf $?tr)
   ?sch ← (StateSchema $?n ?r1 ?r2 ?r3)
⇒
   (retract ?att ?typ ?sch)
   (bind ?p :)
   (assert (AttributeTemp $?ar))
   (assert (TypesTemp $?tr))
   (if (eq (sub-string 1 3 ?tf) "SET") then
      (assert (StateSchema $?n ?af ?p
         (str-cat "\\finset" (sub-string 5 (str-length
          ?tf) ?tf))
          ?r1 ?r2 ?r3))
   else (assert (StateSchema $?n ?af ?p ?tf ?r1 ?r2 ?r3))))
(defrule MODELLING::CreatDataStoreStateScheman
   (phase PrintStateSchema)
   (EntityAttributes (EntityName ?en) (Attributes ?af $?ar)
    (Types ?tf $?tr))
⇒
   (bind ?sen (lowcase ?en))
   (bind ?osen (sub-string 1 1 ?sen))
```

```
    (printout DataFile crlf (str-cat \begin{schema} { ?en
     "\\_SET") } crlf)
    (printout DataFile (str-cat ?sen s) ": \\finset " ?en "
     \\\\crlf)
    (printout DataFile (str-cat id ?sen) ":" ?tf " \\fun
     ?en crlf)
    (printout DataFile "\\where" crlf)
    (printout DataFile "\\forall " ?osen ":" ?en "\\spot "
     ?osen " \\in "
               (str-cat ?sen s) " \\wedge " (str-cat ?osen .
                ?af) "="
               (str-cat id ?sen {^-1}) "(" ?osen ") \\\\crlf)
    (printout DataFile "\\ran " (str-cat id ?sen) "="
     (str-cat ?sen s) crlf)
    (printout DataFile "\\end{schema}" crlf crlf))
(defrule MODELLING::DefineOperationSchemaSignature
    (phase DefineOperationSchema)
     ?ata ← (ActionAttributesTemp ?atf $?atrest)
     ?tta ← (ActionTypesTemp ?ttf $?ttrest)
     ?osa ← (OperationsSchema $?os ?r1 ?r2 ?r3)
⇒
    (retract ?ata ?tta ?osa)
    (bind ?p ":")
    (bind ?q "?")
    (bind ?s "")
    (assert (ActionAttributesTemp $?atrest))
    (assert (ActionTypesTemp $?ttrest))
    (if (eq ?atf " - ") then
       (assert (OperationSchema $?os \where ?s ?s ?r1 ?r2
        ?r3))
    else (if (eq (sub-string 1 3 ?ttf) "SET") then
         (assert (OperationsSchema $?os (str-cat "atf ?q) ?p
           (str-cat "\\finset " (sub-string 5 (str-length
            ?ttf) ?ttf)) ?r1 ?r2 ?r3))
       else (assert (OperationSchema $?os (str-cat ?atf ?q)
             ?p ?ttf ?r1 ?r2 ?r3)))))
(defrule MODELLING::DefineOperationSchemaPredicate
    (phase DefineOperationSchema)
    ?boa ← (BasicOperation (EntityName ?a9) (ActionName ?a2)
           (Op ?bo1 ?bo2 ?bor))
    ?osa ← (OperationSchema ?a1 ?a2 ?a3 ?a4 ?a5 ?a6 ?a7 ?a8
            ?a9 $?os ?r1 ?r2 ?r3)
    (test (neq ?bor UNDEFINED))
⇒
    (retract ?boa ?osa)
    (if (lexemep ?bor) then
       (assert (OperationSchema ?a1 ?a2 ?a3 ?a4 ?a5 ?a6 ?a7
        ?a8 ?a9 $?os
          (str-cat ?a7 . ?bo1 ') ?bo2 (filter ?bor (str-cat
           ?a2 .)) ?r1 ?r2 ?r3))
    else (assert (OperationSchema ?a1 ?a2 ?a3 ?a4 ?a5 ?a6 ?a7
          ?a8 ?a9 $?os
       (str-cat ?a7 . ?bo1 ') ?bo2 ?bor ?r1 ?r2 ?r3))))
```

```
(defrule NETWORK::DefineDataStream
   ?eata ← (EntityAttributesTemp ?eatf $?eatr)
   ?eatta ← (EntityAttributesTypesTemp ?eattf $?eattr)
   ?dsa ← (DataStream $?ds ?a1 ?a2 ?a3)
   ?wdsa ← (WDataStream $?wds ?wa1 ?wa2 ?wa3 ?r1 ?r2 ?r3
            ?fa)
⇒
   (retract ?eata ?eatta ?dsa ?wdsa)
   (assert (EntityAttributesTemp $?eatr))
   (assert (EntityAttributesTypesTemp $?eattr))
   (if (eq (sub-string 1 3 ?eattf)"SET") then
      (assert (DataStream $?ds ?eatf ": \\finset "
         (sub-string 5 (str-length ?eattf) ?eattf) ?a1 ?a2
          ?a3))
   else
      (assert (DataStream $?ds ?eatf ":" ?eattf ?a1 ?a2
       ?a3)))
   (assert (WDataStream $?wds (str-cat entry? . ?eatf) "="
      (str-cat ?fa . ?eatf "?") ?wa1 ?wa2 ?wa3 ?r1 ?r2 ?r3
       ?fa)))
```